ADSP-BF7xx Blackfin+ Processor

Programming Reference

Revision 1.0, October 2016

Part Number
82-100123-01

Analog Devices, Inc.
Three Technology Way
Norwood MA , 02062

Notices

Copyright Information

Disclaimer

Trademark and Service Mark Notice

Contents

Preface

Thank you for purchasing and developing systems using an ADSP-BF70x Blackfin+ processor from Analog Devices, Inc.

Purpose of This Manual

The *ADSP-BF70x Blackfin+ Processor Programming Reference* provides core architecture and programming information about the ADSP-BF70x processors. This programming reference provides the main architectural information about the core of these processors. The architectural descriptions cover computational units, the control units, the address arithmetic unit, and control unit, including all features and processes that they support. For hardware (system design) information, see the *Blackfin+ Processor Hardware Reference*. For timing, electrical, and package specifications, see the *ADSP-BF70x Blackfin+ Processor Data Sheet*.

Intended Audience

The primary audience for this manual is a programmer who is familiar with Analog Devices processors. The manual assumes the audience has a working knowledge of the appropriate processor architecture and instruction set. Programmers who are unfamiliar with Analog Devices processors can use this manual, but should supplement it with other texts, such as programming reference books and data sheets, that describe their target architecture.

Manual Contents

This manual consists of the following chapters:

- ***Introduction*** - provides a general description of the processor core architecture, memory architecture, instruction syntax, and notation conventions.
- ***Computational Units*** - describes the arithmetic/logic units (ALUs), multiplier/accumulator units (MACs), shifter, and the set of video ALUs. The chapter also discusses data formats, data types, and register files.
- ***Operating Modes and States*** - describes the operating modes of the processor, as well as the Idle and Reset states.
- ***Program Sequencer*** - describes the operation of the program sequencer, which controls program flow by providing the address of the next instruction to be executed. The chapter also discusses loops, subroutines, jumps, interrupts, and exceptions.
- ***Address Arithmetic Unit*** - describes the Address Arithmetic Unit (AAU), including Data Address Generators (DAGs), addressing modes, how to modify DAG and pointer registers, memory address alignment, and DAG instructions.

- ***Memory*** - describes L1 memories, particularly memory architecture, memory model, memory transaction model, and memory-mapped registers (MMRs). It also discusses instruction, data, and scratchpad memory, which are part of the Blackfin+ processor core.
- ***Instruction Set Reference*** - describes each assembly instruction and its execution, as well as any impact to the Arithmetic Status (`ASTAT`) register. The reference is broken into the following sections:
    - Arithmetic Instructions
    - Sequencer Instructions
    - Memory/Pointer Instructions
    - Specialized Compute Instructions
    - Arithmetic Status Register
    - Instruction Page Tables (including instruction opcode information)
    - Issuing Parallel Instructions
- ***Debug*** - provides a description of the processor's debug functionality, which is used for software debugging and complements some services often found in operating system (OS) kernels.
- ***Numeric Formats*** - describes various aspects of the 16-bit data format and describes how to implement a block floating-point format in software.

What's New in This Manual

This revision (1.0) is the first publicly released version revision of the *ADSP-BF70x Blackfin+ Processor Programming Reference*. It includes corrected errata and updated register diagrams associated with this processor.

Technical or Customer Support

You can reach customer and technical support for processors from Analog Devices in the following ways:

- Post your questions in the processors and DSP support community at ***EngineerZone***:

  http://ez.analog.com/community/dsp
- Submit your questions to technical support at ***Connect with ADI Specialists***:

  http://www.analog.com/support
- E-mail your questions about software/hardware development tools to:

  processor.tools.support@analog.com
- E-mail your questions about processors and DSPs to:

  processor.support@analog.com (world wide support) processor.china@analog.com (China support)

- Phone questions to *1-800-ANALOGD* (USA only)
- Contact your Analog Devices sales office or authorized distributor. Locate one at:

  http://www.analog.com/adi-sales
- Send questions by mail to:

  Analog Devices, Inc.

  Three Technology Way

  P.O. Box 9106

  Norwood, MA 02062-9106 USA

Supported Processors

The following is the list of Analog Devices, Inc. processors supported by the CrossCore Embedded Studio® development tools suite.

Blackfin+® (ADSP-BF7xx) Processors

The name *Blackfin+* refers to the enhanced fixed-point Blackfin core architecture featured by the ADSP-BF70x processor product line, which is a family of 16-bit embedded processors.

Blackfin® (ADSP-BF6xx/BF5xx) Processors

The name *Blackfin* refers to the fixed-point core architecture featured on the following processors: ADSP-BF5xx and ADSP-BF6xx.

SHARC® (ADSP-21xxx) Processors

The name *SHARC* refers to the high-performance, 32-bit, floating-point core architecture featured on the following processors: ADSP-2106x, ADSP-2116x, ADSP-2126x, ADSP-213xx, and ADSP-214xx. These processors can be used in speech, sound, graphics, and imaging applications.

SHARC+® (ADSP-SC5xx, ADSP-215xx) Processors

The name *SHARC+* refers to the enhanced high-performance, 32-bit, floating-point core architecture featured on the following processors: ADSP-215xx/ADSP-SC5xx. The connected SHARC+ ADSP-SC5xx processors also contain an ARM® Cortex-A5® core. These products can be used in speech, sound, graphics, and imaging applications.

The following is the list of Analog Devices, Inc. processors supported by the IAR Embedded WorkBench® development tools. For information about the IAR Embedded WorkBench product and software download, go to http://www.iar.com/en/Products/IAR-Embedded-Workbench .

Mixed-Signal Control Processors

The ADSP-CM40x processors are based on the ARM Cortex®-M4 core and are designed for motor control and industrial applications.

The ADSP-CM41x processors are based on the ARM Cortex-M4 and ARM Cortex-M0 cores and are designed for motor control and industrial applications.

Product Information

Product information can be obtained from the Analog Devices Web site and CrossCore Embedded Studio online Help system.

Analog Devices Web Site

The Analog Devices Web site, http://www.analog.com, provides information about a broad range of products—analog integrated circuits, amplifiers, converters, and digital signal processors.

To access a complete technical library for each processor family, go to: http://www.analog.com/processors/technical_library The manuals selection opens a list of current manuals related to the product as well as a link to the previous revisions of the manuals. When locating your manual title, note a possible errata check mark next to the title that leads to the current correction report against the manual.

Also note, MyAnalog.com is a free feature of the Analog Devices Web site that allows customization of a Web page to display only the latest information about products you are interested in. You can choose to receive weekly e-mail notifications containing updates to the Web pages that meet your interests, including documentation errata against all manuals. MyAnalog.com provides access to books, application notes, data sheets, code examples, and more.

Visit MyAnalog.com to sign up. If you are a registered user, just log on. Your user name is your e-mail address.

EngineerZone

EngineerZone is a technical support forum from Analog Devices. It allows you direct access to ADI technical support engineers. You can search FAQs and technical information to get quick answers to your embedded processing and DSP design questions.

Use EngineerZone to connect with other DSP developers who face similar design challenges. You can also use this open forum to share knowledge and collaborate with the ADI support team and your peers. Visit http://ez.analog.com to sign up.

Notation Conventions

Text conventions used in this manual are identified and described as follows. Additional conventions, which apply only to specific chapters, may appear throughout this document.

Example	Description
File > ***Close***	Titles in reference sections indicate the location of an item within the CrossCore Embedded Studio IDE's menu system (for example, the ***Close*** command appears on the ***File*** menu).
`{this | that}`	Alternative required items in syntax descriptions appear within curly brackets and separated by vertical bars; read the example as `this` or `that`. One or the other is required.
`[this | that]`	Optional items in syntax descriptions appear within brackets and separated by vertical bars; read the example as an optional `this` or `that`.
`[this, …]`	Optional item lists in syntax descriptions appear within brackets delimited by commas and terminated with an ellipse; read the example as an optional comma-separated list of `this`.
`.SECTION`	Commands, directives, keywords, and feature names are in text with `Letter Gothic` font.
`filename`	Non-keyword placeholders appear in text with italic style format.
NOTE:	*NOTE:* For correct operation, ... A note provides supplementary information on a related topic. In the online version of this book, the word *NOTE:* appears instead of this symbol.
CAUTION:	*CAUTION:* Incorrect device operation may result if ... *CAUTION:* Device damage may result if ... A caution identifies conditions or inappropriate usage of the product that could lead to undesirable results or product damage. In the online version of this book, the word *CAUTION:* appears instead of this symbol.
ATTENTION:	*ATTENTION:* Injury to device users may result if ... A warning identifies conditions or inappropriate usage of the product that could lead to conditions that are potentially hazardous for devices users. In the online version of this book, the word *ATTENTION:* appears instead of this symbol.

Register Documentation Conventions

Register diagrams use the following conventions:

- The descriptive name of the register appears at the top with the short form of the name.
- If a bit has a short name, the short name appears first in the bit description, followed by the long name.
- The reset value appears in binary in the individual bits and in hexadecimal to the left of the register.
- Bits marked *X* have an unknown reset value. Consequently, the reset value of registers that contain such bits is undefined or dependent on pin values at reset.

- Shaded bits are reserved

NOTE: To ensure upward compatibility with future implementations, write back the value that is read for reserved bits in a register, unless otherwise specified.

Register description tables use the following conventions:

- Each bit's or bit field's access type appears beneath the bit number in the table in the form (read-access/write-access). The access types include:
  - R = read, RC = read clear, RS = read set, R0 = read zero, R1 = read one, Rx = read undefined
  - W = write, NW = no write, W1C = write one to clear, W1S = write one to set, W0C = write zero to clear, W0S = write zero to set, WS = write to set, WC = write to clear, W1A = write one action
- Many bit and bit field descriptions include enumerations, identifying bit values and related functionality. Unless otherwise indicated (with a prefix), these enumerations are decimal values.

1 Introduction

This *Blackfin+ Processor Programming Reference* provides details on the assembly language instructions used by Blackfin+ processors. The Blackfin+ architecture extends the Micro Signal Architecture (MSA) core developed jointly by Analog Devices, Inc. and Intel Corporation. This manual applies to all ADSP-BF7xx processor derivatives. All devices provide an identical core architecture and instruction set. Additional architectural features are only supported by some devices and are identified in the manual as being optional features. A read-only memory-mapped register, FEATURE0, enables run-time software to query the optional features implemented in a particular derivative. Some details of the implementation may vary between derivatives. This is generally not visible to software, but system and test code may depend on very specific aspects of the memory microarchitecture. Differences and commonalities at a global level are discussed in the Memory chapter. For a full description of the system architecture beyond the Blackfin+ core, refer to the specific hardware reference manual for your derivative. This section points out some of the conventions used in this document.

The Blackfin+ processor combines a dual-MAC signal processing engine, an orthogonal RISC-like microprocessor instruction set, flexible Single Instruction, Multiple Data (SIMD) capabilities, and multimedia features into a single instruction set architecture.

Core Architecture

The Blackfin+ processor core contains two 16-bit multipliers (MACs), one 32-bit MAC, two 40-bit accumulators, one 72-bit accumulator, two 40-bit Arithmetic Logic Units (ALUs), four 8-bit video ALUs, and a 40-bit shifter, shown in the ***Processor Core Architecture*** figure. The Blackfin+ processors work on 8-, 16-, or 32-bit data from the register file.

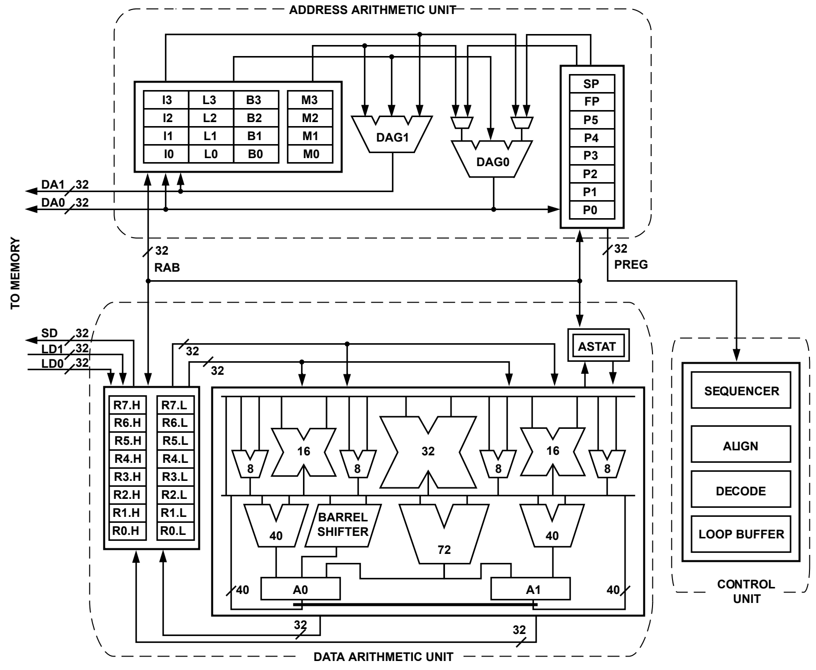


Figure 1-1: Processor Core Architecture

The compute register file contains eight 32-bit registers. When performing compute operations on 16-bit operand data, the register file operates as 16 independent 16-bit registers. All operands for compute operations come from the multi-ported register file and instruction constant fields.

Each 16-bit MAC can perform a 16- by 16-bit multiply per cycle, with accumulation to a 40-bit result. The 32-bit MAC can perform a 32- by 32-bit multiply, with accumulation to 72-bits, or a 16-bit complex multiplication. Signed and unsigned formats, rounding, and saturation are supported.

The ALUs perform a traditional set of arithmetic and logical operations on 16-bit or 32-bit data. Many special instructions are included to accelerate various signal processing tasks, including bit operations such as field extract and population count, divide primitives, saturation and rounding, and sign/exponent detection. The set of video instructions include byte-alignment and packing operations, 16-bit and 8-bit adds with clipping, 8-bit average operations, and 8-bit Subtract/Absolute value/Accumulate (SAA) operations. Also provided are the compare/select and vector search instructions. For some instructions, two 16-bit ALU operations can be performed simultaneously on register pairs (a 16-bit high half and 16-bit low half of a compute register). By also using the second ALU, quad-16-bit operations are possible.

The 40-bit shifter can deposit data and perform shifting, rotating, normalization, and extraction operations.

A program sequencer controls the instruction execution flow, including instruction alignment and decoding. For program flow control, the sequencer supports PC-relative and indirect conditional jumps (with static branch prediction) and subroutine calls. Hardware is provided to support zero-overhead looping. The architecture is fully interlocked, meaning there are no visible pipeline effects when executing instructions with data dependencies.

The address arithmetic unit provides two addresses for simultaneous dual fetches from memory. It contains a multi-ported register file consisting of four sets of 32-bit index, modify, length, and base registers (for circular buffering) and eight additional 32-bit pointer registers (for C-style indexed stack manipulation).

Blackfin+ processors support a modified Harvard architecture in combination with a hierarchical memory structure. Level 1 (L1) memories typically operate at the full processor speed with little or no latency. At the L1 level, the instruction memory holds instructions only. The two data memories hold data, and a dedicated scratchpad data memory can be used to store stack and local variable information.

In addition, multiple L1 memory blocks are provided, which may be configured as a mix of SRAM and cache. The Memory Management Unit (MMU) provides memory protection for individual tasks that may be operating on the core.

The architecture provides three modes of operation: User, Supervisor, and Emulation. User mode has restricted access to a subset of system resources, thus providing a protected software environment. Supervisor and Emulation modes have unrestricted access to the system and core resources.

The Blackfin+ processor instruction set is optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multi-function instructions, and some instructions with very large immediate values are encoded into 64-bit opcodes. Blackfin+ products support a limited multi-issue capability, where a 32-bit instruction can be issued in parallel with two 16-bit instructions. This allows the programmer to use many of the core resources in a single instruction cycle.

The Blackfin+ processor assembly language uses an algebraic syntax. The architecture is optimized for use with the C compiler.

Memory Architecture

The Blackfin+ processor architecture structures memory as a single, unified 4 GB address space using 32-bit addresses. All resources, including internal memory, external memory, and I/O control registers, occupy separate sections of this common address space. The memory portions of this address space are arranged in a hierarchical structure to provide a good cost/performance balance of some very fast, low-latency on-chip memory (as cache or SRAM) and larger, lower-cost and lower-performance off-chip memory systems.

The memory DMA controller provides high-bandwidth data movement capabilities. It can perform block transfers of code or data between the internal and external memory spaces.

Internal Memory

The L1 memory system is the primary, highest-performance memory available to the core. At a minimum, each Blackfin+ processor has two blocks of on-chip memory that provide high-bandwidth access to the core:

- L1 instruction memory, consisting of SRAM and/or an instruction cache. This memory is accessed at the full core clock rate.
- L1 data memory, consisting of SRAM and/or a data cache. This memory block is also accessed at the full core clock rate.

On-chip Level 2 (L2) memory forms an on-chip memory hierarchy with L1 memory and provides much more capacity, but the latency is higher. The on-chip L2 memory may be made cacheable in L1 and is capable of storing both instructions and data.

External Memory

External (off-chip) memory is accessed via on-chip memory peripherals such as DDR controllers.

I/O Memory Space

Blackfin+ processors do not define a separate I/O space. All resources are mapped through the flat 32-bit address space. Control registers for on-chip I/O devices are mapped into memory-mapped registers (MMRs) at addresses in a reserved part of the 4 GB address space. These are separated into two smaller blocks, one containing the control MMRs for all core functions (core MMRs) and the other containing the registers needed for setup and control of the on-chip peripherals outside of the core (system MMRs). All MMRs are accessible only in Supervisor mode.

Event Handling

The event controller on the Blackfin+ processor handles all asynchronous and synchronous events in the system. It supports both nesting and prioritization. Nesting allows multiple event service routines to be active simultaneously, and prioritization ensures that servicing a higher-priority event takes precedence over servicing a lower-priority event. The controller provides support for five different types of events:

- *Emulation* - causes the processor to enter Emulation mode, allowing command and control of the processor via the JTAG interface.
- *Reset* - resets the processor.
- *Non-Maskable Interrupt (NMI)* - the software watchdog timer or the $\overline{\text{NMI}}$ input signal to the processor can generate this event. The NMI event is frequently used as a power-down indicator to initiate an orderly shutdown of the system.
- *Exceptions* - synchronous to program flow, an exception is taken before the instruction is allowed to complete. Conditions such as data alignment violations and undefined instructions cause exceptions.
- *Interrupts* - asynchronous to program flow. These events can be caused by input pins, timers, other peripherals, and software.

Each event has an associated register to hold the return address and an associated return-from-event instruction. When an event is triggered, the state of the processor is saved on the supervisor stack.

The processor event controller consists of two stages, the Core Event Controller (CEC) and the System Event Controller (SEC). The CEC works with the SEC to prioritize and control all system events. Conceptually, interrupts from the peripherals arrive at the SEC and are routed directly into a general-purpose interrupt of the CEC.

Syntax Conventions

The Blackfin+ processor instruction set supports several syntactical conventions that appear throughout this document. These conventions relate to case sensitivity, free format, instruction delimiting, and comments.

Case Sensitivity

The instruction syntax is case insensitive. The assembler treats register names and instruction keywords in a case-insensitive manner (i.e., `R3.l`, `R3.L`, `r3.l`, and `r3.L` are all valid, equivalent input to the assembler).

In explanations and descriptions throughout this manual, upper case is used to help the register names and keywords stand out among normal text.

Free Format

Assembler input is free format and may appear anywhere on the line. One instruction may extend across multiple lines, or more than one instruction may appear on the same line, and white space (e.g., space, tab, or a new line) may appear anywhere between tokens. A token must not have embedded spaces. Tokens include numbers, register names, keywords, user identifiers, and also some multi-character special symbols like "+=", "`/*`", or "`||`".

Instruction Delimiting

A semicolon must terminate every instruction. Several instructions can be placed together on a single line at the programmer's discretion, provided each instruction ends with a semicolon.

Each complete instruction must end with a semicolon. Sometimes, a complete instruction will consist of more than one operation. There are two cases where this occurs.

- Two general operations are combined to be issued across multiple computation units. In this case, a comma separates the different parts:

  ```
  a0 = r3.h * r2.l , a1 = r3.l * r2.h ;
  ```

- A general instruction is combined with one or two memory accesses as a multi-issue instruction. The latter portions of instructions like these are separated by the parallel-issue "`||`" token. For example:

  ```
  a0 = r3.h * r2.l || r1 = [p3++] || r4 = [i2++] ;
  ```

Comments

The assembler supports various kinds of comments, including:

- End of line: A double forward slash token ("`//`") indicates the beginning of a comment that concludes at the next new line character.
- General comment: A general comment begins with the "`/*`" token and ends with the "`*/`" token. It may contain any characters and extend over multiple lines.

Comments are not recursive; if the assembler sees a "`/*`" within a general comment, it issues an assembler warning at build-time.

Notation Conventions

This manual and the assembler use the following conventions:

- Register names are alphabetical, followed by a number in cases where there are more than one register in a logical group. Thus, examples include `ASTAT`, `FP`, `M2`, and `R3`.
- Register names are reserved and may not be used as program identifiers.
- Some operations (such as the *Move Register* instruction) require a register pair. Register pairs are always Data Registers or Accumulators and are denoted using a colon, for example, `R3:2`. The larger number must be written first. Note that the hardware supports only odd-even pairs, for example, `R7:6`, `R5:4`, `R3:2`, `R1:0` and `A1:0.`
- Some instructions (such as the *--SP (Push Multiple)* instruction) require a group of adjacent registers. Adjacent registers are syntactically denoted with the range enclosed in parentheses and separated by a colon. For example, the range of data registers comprised of `R3`, `R4`, `R5`, `R6`, and `R7` is written in this instruction as `R7:3`. Again, the larger number appears first.
- Portions of a particular register may be individually specified by using the dot (".") syntax following the register name, followed by a letter denoting the desired portion. For 32-bit registers, "`.H`" denotes the most-significant ("High") 16 bits, whereas "`.L`" denotes the least-significant 16 bits. Similar access control is available for the 40-bit accumulator registers, which is discussed later.

This manual uses the following conventions.

- When there is a choice of any one register within a register group, this manual shows the register set using a dash ("`-`"). For example, "`R7-0`" in text means that any one of the eight data registers (`R7`, `R6`, `R5`, `R4`, `R3`, `R2`, `R1`, or `R0`) can be used in the syntax for the instruction.
- Immediate values are designated as "`imm`" with the following modifiers:
  - "`imm`" indicates a signed integer and is followed by an integer value indicating how many bits are required to represent it in binary (e.g., *`imm7`* is a signed 7-bit value).
  - The "`u`" prefix indicates an unsigned value (e.g., *`uimm4`* is an unsigned 4-bit value).
  - The "`m`" suffix followed by a number can be appended to provide added alignment requirements. For example, *`uimm16m2`* is an unsigned, 16-bit integer that must be an even number, and *`imm7m4`* is a signed, 7-bit integer that must be a multiple of four.
  - PC-relative, signed values are designated as "*`pcrel`*" and have the following modifiers:
    - the decimal number indicates how many bits are required to represent the value in bianry (e.g., *`pcrel5`* is a 5-bit value).
    - any alignment requirements are designated by an optional "`m`" suffix followed by a number (e.g., *`pcrel13m2`* is a 13-bit integer that must be an even number).
  - Loop PC-relative, signed values are designated as "*`lppcrel`*" with the following modifiers:

- the decimal number indicates how many bits are required to represent the value in binary (e.g., *`lppcrel5`* is a 5-bit value).
- any alignment requirements are designated by an optional "`m`" suffix followed by a number (e.g., *`lppcrel11m2`* is an 11-bit integer that must be an even number).

Glossary

The following terms appear throughout this document. Without trying to explain the Blackfin+ processor, here are the terms used along with their definitions.

Register Names

The architecture includes the registers shown in the ***Registers*** table.

Table 1-1: Registers

Register	**Description**
Accumulators	The two 40-bit `A1` and `A0` registers that normally contain data that is being manipulated. Each accumulator can be accessed in five ways: as one 40-bit register, as one 32-bit register (designated as `A1.W` or `A0.W`), as two 16-bit registers (designated as `A1.H` and `A1.L` or `A0.H` and `A0.L`) and as one 8-bit register (designated as `A1.X` or `A0.X`) for the overflow bits that extend beyond bit 31. The accumulators may be used as a pair (designated as `A1:0`) to hold the 40-bit real and imaginary parts of a complex fixed-point number or a single 72-bit real fixed-point number.
Data Registers	The set of eight 32-bit registers (`R0`, `R1`, `R2`, `R3`, `R4`, `R5`, `R6`, and `R7`) that normally contain data for manipulation (abbreviated to `Dreg` when referenced in this manual). Data registers can be accessed as 32-bit registers or as two independent 16-bit registers. The least significant 16 bits of each register is called the "low" half and is designated with "`.L`" following the register name. The most significant 16 bits is called the "high" half and is designated with "`.H`" following the name (e.g., `R7.L`, `r2.h`, `r4.L`, and `R0.h`).
Pointer Registers	The set of six 32-bit registers (`P0`, `P1`, `P2`, `P3`, `P4`, and `P5`) that normally contain byte addresses of data structures (abbreviated to `Preg` when referenced in this manual). The pointer registers can be accessed as 16-bit halves, similar to the data registers above.
Stack Pointer	`SP` is a part of the `Preg` register group and contains the 32-bit address of the last occupied byte location in the stack. The stack grows downward in memory (i.e., by pre-decrementing `SP` during stack push operations).
Frame Pointer	`FP` is a part of the `Preg` register group and contains the 32-bit address of the previous Frame Pointer on the stack, located at the top of the frame.
Loop Top	`LT0` and `LT1` contain the 32-bit address of the instruction at the top of a zero-overhead loop.
Loop Count	`LC0` and `LC1` contain the 32-bit counter for the zero-overhead loop iterations. These registers are initialized during loop setup and decrement when the loop bottom is reached. The loop exits when the count reaches 0.
Loop Bottom	`LB0` and `LB1` contain the 32-bit address of the instruction at the bottom of a zero-overhead loop.
Index Register	The set of four 32-bit registers (`I0`, `I1`, `I2`, and `I3`) that normally contain byte addresses of data structures (abbreviated to `Ireg` when referenced in this manual).
Modify Registers	The set of four 32-bit registers (`M0`, `M1`, `M2`, and `M3`) that normally contain offset values that modify (add to or subtract from) one of the `Iregs` (abbreviated to `Mreg` when referenced in this manual).

Table 1-1: Registers (Continued)

Register	Description
Length Registers	The set of four 32-bit registers (`L0`, `L1`, `L2`, and `L3`) that normally contain the length (in bytes) of a circular buffer (abbreviated to `Lreg` when referenced in this manual). When a `Lreg` is 0, circular addressing for the corresponding `Ireg` is disabled (e.g., if `L3=0`, circular addressing for the `I3` index is disabled).
Base Registers	The set of four 32-bit registers (`B0`, `B1`, `B2`, and `B3`) that normally contain the base address of a buffer in memory (abbreviated as `Breg` when referenced in this manual).

Functional Units

The architecture includes the three processor sections shown in the ***Units*** table.

Table 1-2: Units

Processor	Description
Data Address Generator (DAG)	Calculates the effective address for indirect and indexed memory accesses. Consists of two blocks, DAG0 and DAG1.
Multiply and Accumulate Unit (MAC)	Performs arithmetic functions on data. Consists of three blocks: MAC0 and MAC1, each associated with an accumulator (`A0` and `A1`, respectively) and executing on 16-bit input operands, and MAC10 associated with the accumulator pair `A1:0` when handling 32-bit input operands.
Arithmetic Logical Unit (ALU)	Performs arithmetic computations and binary shifts on data. Operates on the data registers and accumulators. Consists of three blocks (ALU0, ALU1, and ALU10), each associated with an accumulator (`A0`, `A1`, and the `A1:0` pair, respectively). Each ALU operates in conjunction with a MAC unit.

Arithmetic Status Bits

The Blackfin+ architecture includes 12 Arithmetic Status register (`ASTAT`) bits that indicate specific results from the instruction that just executed. A summary of the status bits appears in the ***ASTAT Bits*** table. All status bits are active high and are set as the result of an operation storing its results to a data or accumulator register. Instructions targeting pointer or DAG registers do not affect these status bits.

Table 1-3: ASTAT Bits

Bit	Description
AC0	ALU0 Carry
AC0_COPY	Copy of ALU0 Carry
AC1	ALU1 Carry
AN	Negative
AQ	Quotient

Table 1-3: ASTAT Bits (Continued)

Bit	Description
AV0	Accumulator 0 Overflow
AVS0	Accumulator 0 Sticky Overflow Set when AV0 is set, but remains set until explicitly cleared by software.
AV1	Accumulator 1 Overflow
AVS1	Accumulator 1 Sticky Overflow Set when AV1 is set, but remains set until explicitly cleared by software.
AZ	Zero
CC	Control Code bit Multipurpose bit set, cleared and tested by specific instructions.
V	Overflow (for data register results)
V_COPY	Copy of Overflow (for data register results)
VS	Sticky Overflow (for data register results) Set when V is set, but remains set until explicitly cleared by software.

Fractional Convention

Fractional numbers include subinteger components less than 1. Whereas decimal fractions appear to the right of a decimal point, binary fractions appear to the right of a binal point.

In instructions that assume placement of a binal point (e.g., in computing sign bits for normalization or alignment purposes), the binal point convention depends on the size of the register being used, as shown in the ***Fractional Notation*** table and the ***Conventional Placement of Binal Point*** figure.

Table 1-4: Fractional Notation

Register Width	Format	Notation	# of Sign Bits	# of Extension Bits	# of Fractional Bits
80-bit accumulator pair (8 bits unused)	Signed Fractional	9.63	1	8	63
	Unsigned Fractional	8.64	0	8	64
64-bit register pair	Signed Fractional	1.63	1	0	63
	Unsigned Fractional	0.64	0	0	64
40-bit registers	Signed Fractional	9.31	1	8	31
	Unsigned Fractional	8.32	0	8	32
32-bit registers	Signed Fractional	1.31	1	0	31
	Unsigned Fractional	0.32	0	0	32
16-bit registers	Signed Fractional	1.15	1	0	15
	Unsigned Fractional	0.16	0	0	16

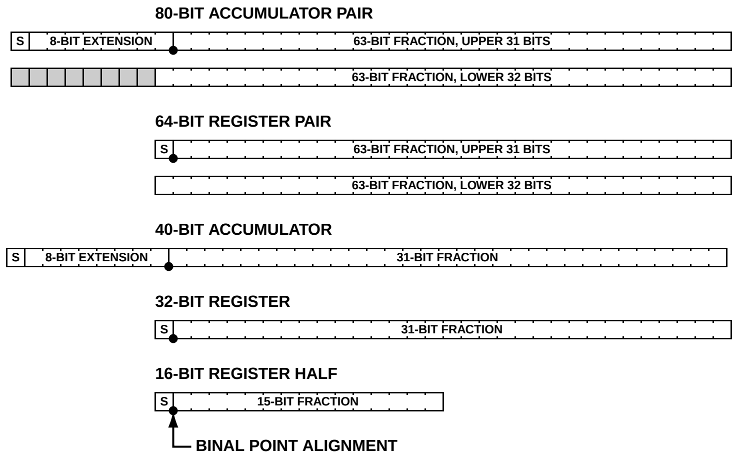


Figure 1-2: Conventional Placement of Binal Point

Saturation

When the result of an arithmetic operation exceeds the range of the destination register, important information can be lost.

Saturation is a technique used to contain the quantity within the values that the destination register can represent. When a value is computed that exceeds the capacity of the destination register, then the value written to the register is the largest value that the register can hold with the same sign as the original value.

- If an operation would otherwise cause a positive value to overflow and become negative, the saturation instead limits the result to the maximum positive value for the register size being used.
- Conversely, if an operation would otherwise cause a negative value to overflow and become positive, saturation limits the result to the maximum negative value for the register's size.

The maximum positive value in a 16-bit register is 0x7FFF, whereas the maximum negative value is 0x8000. For signed two's-complement fractional data in 1.15 format, the range of values that can be represented is -1 through ($1-2^{-15}$).

The maximum positive value in a 32-bit register is 0x7FFF_FFFF, whereas the maximum negative value is 0x8000_0000. For signed two's-complement fractional data in 1.31 format, the range of values that can be represented is -1 through ($1-2^{-31}$).

The maximum positive value in a 40-bit register is 0x7F_FFFF_FFFF, whereas the maximum negative value is 0x80_0000_0000. For signed two's-complement fractional data in 9.31 format, the range of values that can be represented is -256 through ($256-2^{-31}$).

The maximum positive value in a 64-bit register pair is 0x7FFF_FFFF_FFFF_FFFF, whereas the maximum negative value is 0x8000_0000_0000_0000. For signed two's-complement fractional data in 1.63 format, the range of values that can be represented is -1 through ($1-2^{-63}$).

A real value held in the 80-bit accumulator pair, `A1:0`, only has 72 bits of useful data; therefore, the maximum positive value in the 80-bit accumulator pair is 0x7F_FFFF_FFFF_FFFF_FFFF, and the maximum negative value is 0x80_0000_0000_0000_0000. For signed two's-complement fractional data in 9.63 format, the range of values that can be represented is -256 through (256-2^{-63}).

For example, if a 16-bit register containing 0x1000 (decimal integer +4096) was shifted left 3 places without saturation, it would overflow to 0x8000 (decimal -32,768). With saturation, however, a left shift of 3 or more places would always produce the largest positive 16-bit number, 0x7FFF (decimal +32,767).

Another common example is copying the lower half of a 32-bit register into a 16-bit register. If the 32-bit register contains 0xFEED_0ACE and the lower half of this negative number is copied into a 16-bit register without saturation, the result is 0x0ACE, which changes the sign to represent a positive number. With saturation, however, the 16-bit result maintains its negative sign and becomes 0x8000.

The Blackfin+ architecture implements 40-bit saturation for all arithmetic operations that write a single accumulator destination register except as noted in the individual instruction descriptions when an optional 32-bit saturation mode can constrain a 40-bit accumulator to the 32-bit register range. The Blackfin+ architecture performs 32-bit saturation for 32-bit destination registers only where noted in the instruction descriptions.

Overflow is the alternative to saturation. The number is allowed to simply exceed its bounds and lose its most significant bit(s), retaining only the lowest (least-significant) portion of the number. Overflow can occur when a 40-bit value is written to a 32-bit destination or when a 72-bit value is written to a 64-bit or 32-bit destination. If there was any useful information in the upper eight bits of the 40-bit value, then information is lost in the process. Some processor instructions report overflow conditions in the Arithmetic Status (`ASTAT`) register bits, as noted in the instruction descriptions.

Rounding and Truncating

Rounding is a means of reducing the precision of a number by removing lower-order bits from that number's representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, a number has N bits of precision, but the new number will have only M bits of precision (where N>M). In this case, N-M bits of precision are removed from the number in the process of rounding.

The *round-to-nearest* method returns the closest number to the original. An original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the three-bit, two's-complement fraction 0.25 (binary 0.01) to the nearest two-bit two's-complement fractional notation, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called *biased rounding*.

The *convergent rounding* method also returns the closest number to the original. However, in cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number (with the least significant bit being a 0). Taking the example above, the result would be 0.0 because that is the even value among the two options, 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called *unbiased rounding*.

Some instructions for this processor support biased and unbiased rounding, as governed by the Rounding Mode bit in bit in the Arithmetic Status register (`ASTAT.RND_MOD`).

Another common way to reduce the significant bits representing a number is to simply mask off the N-M lower bits. This process is known as *truncation* and results in a relatively large bias.

The ***8-Bit Number Reduced to 4 Bits of Precision*** figure shows other examples of rounding and truncation methods.

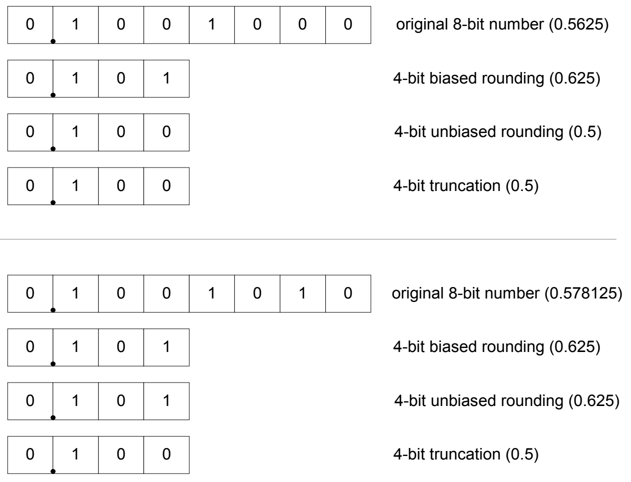

0 1 0 0 1 0 0 0
original 8-bit number (0.5625)
0 1 0 1
4-bit biased rounding (0.625)
0 1 0 0
4-bit unbiased rounding (0.5)
0 1 0 0
4-bit truncation (0.5)
0 1 0 0 1 0 1 0
original 8-bit number (0.578125)
0 1 0 1
4-bit biased rounding (0.625)
0 1 0 1
4-bit unbiased rounding (0.625)
0 1 0 0
4-bit truncation (0.5)

Figure 1-3: 8-Bit Number Reduced to 4 Bits of Precision

Automatic Circular Addressing

The Blackfin+ processor provides an optional circular (or "modulo") addressing feature that increments an index register (*`Ireg`*) through a pre-defined address range, then automatically resets the *`Ireg`* to repeat that range. This feature improves input/output loop performance by eliminating the need to manually re-initialize the address index pointer each time. Circular addressing is useful, for instance, when repetitively loading or storing a string of fixed-sized data blocks.

The circular buffer must meet the following criteria:

- The maximum length of a circular buffer (the value held in any *`Lreg`*) must be an unsigned number less than 2^{31}.
- The magnitude of the modifier (the value held in the *`Mreg`* used in the instruction) must be less than the length of the circular buffer (the value in the *`Lreg`* that corresponds to the *`Ireg`* used in the instruction).
- The initial location pointed to by the *`Ireg`* must be within the circular buffer defined by the base-length register pair (the *`Breg`* and *`Lreg`* register pair. E.g., *`B2`* and *`L2`*).

If any of these conditions is not satisfied, then processor behavior is not specified.

There are two elements of automatic circular addressing:

- Indexed address instructions

- Four sets of circular buffer addressing registers, comprised of one of each of the *Ireg*, *Breg*, and *Lreg* register groups (specifically, I0/B0/L0, I1/B1/L1, I2/B2/L2, and I3/B3/L3)

To qualify for circular addressing, the indexed address instruction must explicitly modify an index register. Some indexed address instructions use a modify register (*Mreg*) to increment the *Ireg* value. In that case, any *Mreg* can be used to increment any *Ireg*. The *Ireg* used in the instruction specifies which of the four circular buffer sets to use.

The circular buffer registers define the length (*Lreg*) of the data block in bytes and the base (*Breg*) address to re-initialize the *Ireg* when a wrap condition is encountered at the end of the buffer.

Some instructions, such as *Add Immediate* and *Modify–Decrement*, modify an index register without using it for addressing; however, even these instructions are still affected by circular addressing, if enabled.

Disable circular addressing for an *Ireg* by setting the corresponding *Lreg* to 0. For example, set L2 = 0 to disable circular addressing for register I2. Any non-zero value in an *Lreg* enables circular addressing for its corresponding DAG buffer registers.

2 Computational Units

The processor's computational units perform numeric processing for DSP and general control algorithms. The seven computational units are two arithmetic/logic units (ALUs), three multiplier/accumulator (MAC) units, a shifter, and a set of video ALUs, all of which get data from registers in the data register file. Computational instructions for these units provide fixed-point operations, and each computational instruction can execute every cycle.

The computational units handle different types of operations. The ALUs perform arithmetic and logic operations. The MACs perform multiplication and execute multiply/add and multiply/subtract operations. The shifter executes logical and arithmetic shifts and performs bit packing and extraction. The video ALUs perform Single Instruction, Multiple Data (SIMD) logical operations on specific 8-bit data operands.

Data moving into and out of the computational units goes through the data register file, which consists of eight 32-bit registers. In operations requiring 16-bit operands, the registers are paired, providing sixteen possible 16-bit registers.

The processor's assembly language provides access to the data register file. The syntax allows programs to move data to and from these registers while simultaneously specifying a computation's data format.

The ***Processor Core Architecture*** figure provides a graphical guide to the other topics in this chapter. An examination of each computational unit provides details about its operation and is followed by a summary of computational instructions. Studying the details of the computational units, register files, and data buses leads to a better understanding of proper data flow for computations. Next, details about the processor's advanced parallelism reveal how to take advantage of multifunction instructions.

The ***Processor Core Architecture*** figure also shows the relationship between the data register file and the computational units (multipliers, ALUs, and shifter).

Single function MAC, ALU, and shifter instructions have unrestricted access to the data registers in the data register file. Multifunction operations may have restrictions that are described in the section for that particular operation.

Two additional 40-bit registers, `A0` and `A1`, provide accumulator results. These registers are dedicated to the ALUs and are used primarily for multiply-and-accumulate functions.

The traditional modes of arithmetic operations, such as fractional and integer, are specified directly in the instruction. Rounding modes are set from the `ASTAT` register, which also records status and conditions for the results of the computational operations.

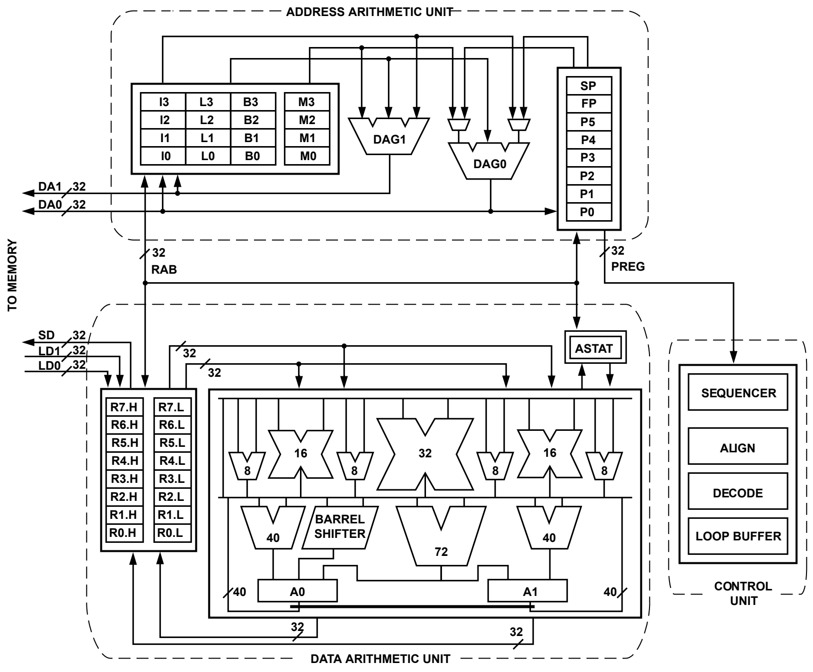


Figure 2-1: Processor Core Architecture

Using Data Formats

Blackfin+ processors are primarily 32-/16-bit fixed-point machines. Most operations assume a two's-complement number representation, while others assume unsigned numbers or straight binary data. Other instructions support 32-bit complex fixed-point, 8-bit arithmetic, and block floating point operations. For detailed information about each number format, see the numeric formats appendix.

In the Blackfin+ processor architecture, signed data is always in two's-complement format. These processors do not use signed-magnitude, one's-complement, binary-coded decimal (BCD), or excess-n formats.

Binary String

The binary string format is the least complex binary notation, within which 16- or 32-bit data is treated as a bit pattern. Examples of computations using this format are the logical operations `NOT`, `AND`, `OR`, and `XOR`. These ALU operations treat their operands as binary strings with no provision for sign bit or binal point placement.

Unsigned Numbers

Unsigned binary numbers may be thought of as positive and having nearly twice the magnitude of a signed number of the same length. The processor treats the least significant words of multiple precision numbers as unsigned numbers.

Signed Numbers: Two's-Complement

In the Blackfin+ processor architecture, the word *signed* refers to two's-complement numbers. Blackfin+ processor operations presume two's-complement arithmetic is being performed.

Fractional Representation: 1.15 and 1.31

The Blackfin+ processor is optimized for operation on data that is in fractional binary format, denoted by either the 1.15 ("one dot fifteen") or 1.31 ("one dot thirty one") nomenclature. In the 1.15 format, the Most Significant Bit (MSB) is the sign bit, which is followed by the 15 fractional bits that together represent data values from -1 to 0.999969. Similarly, in 1.31 format, the sign bit and 31 fractional bits represent values from -1 to 0.99999999953.

The ***Bit Weighting for 1.15 Numbers and 1.31 Numbers*** figure shows the bit weighting for fractional data. This figure also includes examples of 1.15 numbers and 1.31 numbers with their decimal equivalents.

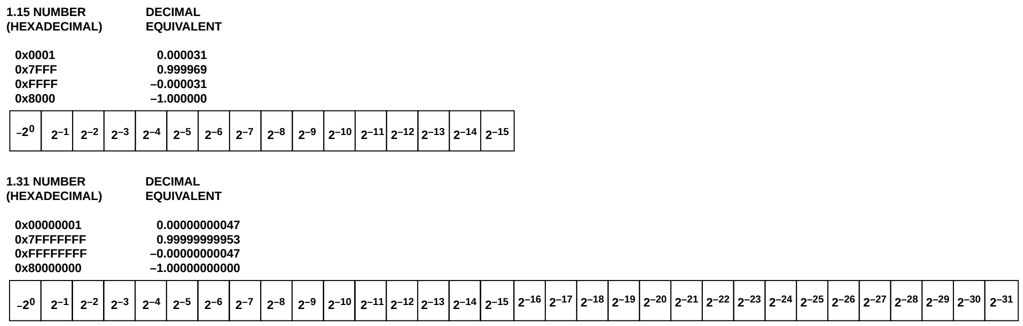


Figure 2-2: Bit Weighting for 1.15 Numbers and 1.31 Numbers

Complex Numbers

Complex numbers are represented as two 16-bit fixed-point numbers, in either fractional 1.15 or 16-bit signed integer format. A complex operand is stored in a 32-bit data register with the real part stored in the least significant bits (`Rx.L`) and the imaginary part in the most significant bits (`Rx.H`).

Register Files

The processor's computational units have three defined register groups, a data register file, a pointer register file, and a set of Data Address Generation (DAG) registers.

- The data register file receives operands from the data buses for the computational units and stores computational results.
- The pointer register file has pointers for addressing operations.
- The DAG registers are dedicated registers that manage zero-overhead circular buffers for DSP operations.

The ***AAU Register Files*** figure provides more information about the pointer and DAG registers.

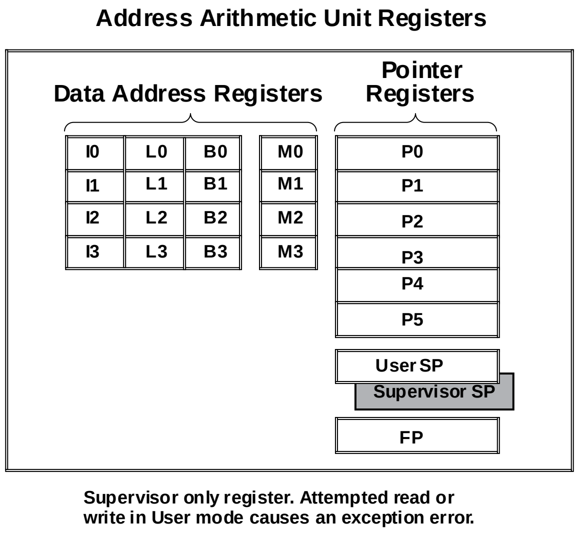

Address Arithmetic Unit Registers
Data Address Registers
Pointer Registers
I0
L0
B0
M0
I1
L1
B1
M1
I2
L2
B2
M2
I3
L3
B3
M3
P0
P1
P2
P3
P4
P5
User SP
Supervisor SP
FP
Supervisor only register. Attempted read or write in User mode causes an exception error.

Figure 2-3: AAU Register Files

Data Register File

The data register file consists of eight 32-bit registers, `R7` through `R0`. Each register may be viewed as a pair of independent 16-bit registers, denoted independently as the low half (`R[n].L`) or the high half (`R[n].H`) of the 32-bit register. For more information, see Data Registers .

For example, these instructions represent a 32-bit and a 16-bit operation:

```
R2 = R1 + R2;        /* 32-bit addition */
R2 = R1.H * R0.L;    /* 16-bit multiplication */
```

Three separate 32-bit buses (two load, one store) connect the register file to L1 data memory. Transfers between the data register file and data memory can move up to two 32-bit words of valid data per core clock cycle. Often, these represent four 16-bit words.

Accumulator Registers

In addition to the data register file, the processor has two dedicated, 40-bit accumulator registers, `A0` and `A1`. Each can be referred to by its 16-bit low half (`An.L`) or 16-bit high half (`An.H`) or its 8-bit extension (`An.X`). Each can also be referred to as a 32-bit register (`An.W`) consisting of the lower 32 bits, or as a complete 40-bit result register (`An`). For more information, see the following registers:

- Accumulator 0 Register
- Accumulator 0 Extension Register
- Accumulator 1 Register
- Accumulator 1 Extension Register

These examples illustrate this convention:

```
A0 = A1;     /* 40-bit move */
```

```
A1.W = R7;   /* 32-bit move */
A0.H = R5.H; /* 16-bit move */
R6.H = A0.X; /* read 8-bit value, sign extend to 16 bits */
```

The accumulator registers may be used together to hold an 80-bit complex result or a 72-bit fixed-point result. The combined accumulator register is called `A1:0`. A 72-bit fixed-point result is held in the combined register with the least significant bits in `A0.W`, the middle bits in `A1.W`, and the most significant bits in `A1.X`.

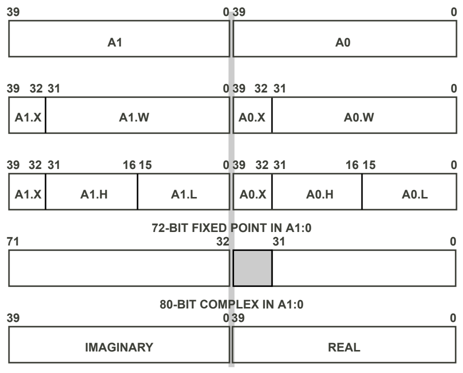


Figure 2-4: 40-Bit Accumulator Registers

Register File Instruction Summary

The ***Register File Instructions and Status*** table lists the register file instructions. In this table, note the meaning of these symbols:

- Allreg denotes: `R[7:0]`, `P[5:0]`, `SP`, `USP`, `FP`, `I[3:0]`, `M[3:0]`, `B[3:0]`, `L[3:0]`, `A0.X`, `A0.W`, `A1.X`, `A1.W`, `ASTAT`, `RETS`, `RETI`, `RETX`, `RETN`, `RETE`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `SEQSTAT`, `SYSCFG`, `EMUDAT`, `CYCLES`, and `CYCLES2`.
- Ax denotes either ALU result register, `A0` or `A1`.
- Dreg denotes any data register file register.
- Sysreg denotes the system registers: `ASTAT`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, or `RETS`, `LC[1:0]`, `LT[1:0]`, `LB[1:0]`, `EMUDAT`, `CYCLES`, and `CYCLES2`.
- Preg denotes any pointer (`P[5:0]`), `FP`, or `SP` register.
- Dreg_even denotes `R0, R2, R4,` or `R6.`
- Dreg_odd denotes `R1, R3, R5,` or `R7.`
- DPreg denotes any data register file register (`R[7:0]`) or any pointer (`P[5:0]`), `FP`, or `SP` register.
- Dreg_lo denotes the lower 16 bits of any data register file (`R[7:0].L`) register.
- Dreg_hi denotes the upper 16 bits of any data register file (`R[7:0].H`) register.
- Ax.L denotes the lower 16 bits of either accumulator (`A0.W` or `A1.W`).
- Ax.H denotes the upper 16 bits of either accumulator (`A0.W` or `A1.W`).

- Dreg_byte denotes the least significant byte of the data register file register (`R[7:0]`).
- Option (X) denotes sign-extended data into the uppermost bits of the destination register.
- Option (Z) denotes zero-extended data into the uppermost bits of the destination register.
- * indicates the status bit may be set or cleared, depending on the result of the instruction.
- ** indicates the status bit is cleared.
- - indicates no effect.

Table 2-1: Register File Instructions and Status

	ASTAT Status Bits						
Instruction	**AZ**	**AN**	**AC0 AC0_COPY AC1**	**AV0 AVS**	**AV1 AV1S**	**CC**	**V V_COPY VS**
allreg = allreg ; [*1]	-	-	-	-	-	-	-
Ax = Ax ;	-	-	-	-	-	-	-
Ax = Dreg ;	-	-	-	-	-	-	-
Ax = Dreg (X) ;	-	-	-	-	-	-	-
Ax = Dreg (Z) ;	-	-	-	-	-	-	-
A1 = Dreg (X), A0 = Dreg (X) ;	-	-	-	-	-	-	-
A1 = Dreg (Z), A0 = Dreg (Z) ;	-	-	-	-	-	-	-
A1 = Dreg (X), A0 = Dreg (Z) ;	-	-	-	-	-	-	-
A1 = Dreg (Z), A0 = Dreg (X) ;	-	-	-	-	-	-	-
Dreg_even = A0 ;	*	*	-	-	-	-	*
Dreg_odd = A1 ;	*	*	-	-	-	-	*
Dreg_even = A0, Dreg_odd = A1 ;	*	*	-	-	-	-	*
Dreg_odd = A1, Dreg_even = A0 ;	*	*	-	-	-	-	*
IF CC DPreg = DPreg ;	-	-	-	-	-	-	-
IF ! CC DPreg = DPreg ;	-	-	-	-	-	-	-
Dreg = Dreg_lo (Z) ;	*	**	**	-	-	-	**/-
Dreg = Dreg_lo (X) ;	*	*	**	-	-	-	**/-
Ax.X = Dreg_lo ;	-	-	-	-	-	-	-
Dreg_lo = Ax.X ;	-	-	-	-	-	-	-

Table 2-1: Register File Instructions and Status (Continued)

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AVS	AV1 AV1S	CC	V V_COPY VS
Ax.L = Dreg_lo ;	-	-	-	-	-	-	-
Ax.H = Dreg_hi ;	-	-	-	-	-	-	-
Dreg_lo = A0 ;	*	*	-	-	-	-	*
Dreg_hi = A1 ;	*	*	-	-	-	-	*
Dreg_hi = A1 ; Dreg_lo = A0 ;	*	*	-	-	-	-	*
Dreg_lo = A0 ; Dreg_hi = A1 ;	*	*	-	-	-	-	*
Dreg = Dreg_byte (Z) ;	*	**	**	-	-	-	**/-
Dreg = Dreg_byte (X) ;	*	*	**	-	-	-	**/-

*1 Warning: Not all register combinations are allowed. For details, see the functional description of the Move Register instruction.

Data Types

The Blackfin+ processor supports 32-bit words, 16-bit half-words, and bytes. The 32- and 16-bit words can be integer or fractional, and bytes are always integers. Integer data types can be signed or unsigned, whereas fractional data types are always signed. 32-bit words can also be complex numbers comprised of 16-bit real and imaginary parts, with the real part in the least significant bits and the imaginary part in the most significant bits of the data register.

The ***Data Types and Representation in Memory/Register*** table illustrates the formats for data that resides in memory, the register file, and the accumulators. In the table, the letter d represents one bit, and the letter s represents one signed bit.

Some instructions manipulate data in the registers by sign-extending or zero-extending the data to 32 bits:

- Instructions zero-extend unsigned data
- Instructions sign-extend signed 16-bit half-words and 8-bit bytes

Other instructions manipulate data as 32-bit numbers. In addition, two 16-bit half words or four 8-bit bytes can be manipulated as 32-bit values.

In the table, note the meaning of these symbols:

- s = sign bit(s)

- d = data bit(s)
- "." = binary point in the format column. Bits to the left are the whole part of the data, and bits to the right are the fractional part of the data. Where applicable, it is also inserted in the representation columns, though the binary point itself is not part of the data.

Table 2-2: Data Types and Representation in Memory/Register

Format	Representation in Memory	Representation in 32-bit Register
32.0 Unsigned Word	dddd dddd dddd dddd dddd dddd dddd dddd	dddd dddd dddd dddd dddd dddd dddd dddd
32.0 Signed Word	sddd dddd dddd dddd dddd dddd dddd dddd	sddd dddd dddd dddd dddd dddd dddd dddd
16.0 Unsigned Half Word	dddd dddd dddd dddd	0000 0000 0000 0000 dddd dddd dddd dddd
16.0 Signed Half Word	sddd dddd dddd dddd	ssss ssss ssss ssss sddd dddd dddd dddd
8.0 Unsigned Byte	dddd dddd	0000 0000 0000 0000 0000 0000 dddd dddd
8.0 Signed Byte	sddd dddd	ssss ssss ssss ssss ssss ssss sddd dddd
1.15 Signed Fraction	s.ddd dddd dddd dddd	ssss ssss ssss ssss s.ddd dddd dddd dddd
1.31 Signed Fraction	s.ddd dddd dddd dddd dddd dddd dddd dddd	s.ddd dddd dddd dddd dddd dddd dddd dddd
Fractional Complex	s.ddd dddd dddd dddd s.ddd dddd dddd dddd	s.ddd dddd dddd dddd s.ddd dddd dddd dddd
Integer Complex	sddd dddd dddd dddd sddd dddd dddd dddd	sddd dddd dddd dddd sddd dddd dddd dddd
Packed 8.0 Unsigned Byte	dddd dddd dddd dddd dddd dddd dddd dddd	dddd dddd dddd dddd dddd dddd dddd dddd
Packed 1.15 Signed Fraction	s.ddd dddd dddd dddd s.ddd dddd dddd dddd	s.ddd dddd dddd dddd s.ddd dddd dddd dddd

Endianness

Both internal and external memory are accessed in little endian byte order. For more information, see the memory transaction model.

ALU Data Types

With the exception of the signed division primitive (`DIVS`), ALU operations treat operands and results as either 16- or 32-bit binary strings. ALU result status bits treat the results as signed, indicating status with the overflow status bits (`AV0`, `AV1`) and the negative status bit (`AN`). Each ALU has its own sticky overflow status bit, `AV0S` and `AV1S`, respectively. Once set, these bits remain set until cleared by writing directly to the `ASTAT` register. An additional `V` status bit is set or cleared depending on the transfer of the result from both accumulators to the register file. Furthermore, the sticky `VS` bit is set with the `V` bit and remains set until explicitly cleared by software.

The logic of the overflow bits (`V`, `VS`, `AV0`, `AV0S`, `AV1`, and `AV1S`) is based on two's-complement arithmetic. A bit or set of bits is set if the Most Significant Bit (MSB) changes in a manner not predicted by the signs of the operands and the nature of the operation. For example, adding two positive numbers must generate a positive result. A change in the sign bit signifies an overflow and sets the corresponmding overflow status bit (`AVx`). Adding a negative and a positive number may result in either a negative or positive result, so this cannot cause an overflow.

The logic of the carry bits (`AC0` and `AC1`) is based on unsigned magnitude arithmetic. The bit is set if a carry is generated from bit 16 (the MSB). The carry bits are most useful for the lower word portions of a multiword operation.

ALU results generate status information. For more information about using ALU status, see Using Computational Status.

MAC Data Types

Each MAC produces results that are binary strings. The inputs are interpreted according to the information given in the instruction itself (whether it is a signed value multiplied by a signed value, an unsigned value multiplied by an unsigned value, a mixture of signed/unsigned, or a rounding operation). The MAC results are either signed or unsigned, depending on the signs of the operands, and is accordingly zero- or sign-extended to the width of the accumulator.

The 32-bit MAC multiplies 32-bit operands to produce a 64-bit result. This result is zero- or sign-extended to 72 bits and added to the 72-bit value in the `A1:0` register pair. Two 32-bit complex operands can be multiplied to produce a 64-bit result consisting of a 32-bit real part and a 32-bit imaginary part. Both parts are sign extended to 40-bits, and the real part is added to the value in `A0` while the imaginary part is added to the value in `A1`.

The 16-bit MACs multiply 16-bit operands to produce a 32-bit result which is zero- or sign-extended up to the full 40-bit width of the `A0` or `A1` register.

The processor supports two modes of format adjustment, the fractional mode for signed fractional operands (1.31 format with one sign bit and 31 fractional bits or 1.15 format with one sign bit and 15 fractional bits) and the regular mode for operands with any other format combination.

When the processor multiplies two 1.15 operands, the result is a 2.30 (two sign bits and 30 fractional bits) number. In the fractional mode, the MAC automatically shifts the product left one bit before transferring the result to the result register. This shift of the redundant sign bit causes the MAC result to be in 1.31 format, which can be rounded (truncated) to 1.15 format. Similarly, when the processor multiplies two 1.31 operands, the result is a 2.62 number which is automatically shifted to produce a 1.63 result before transferring to the result register.

In other modes, the left shift does not occur. For example, if the operands are in 16.0 format, the 32-bit MAC result would be in 32.0 format. A left shift is not needed and would change the numerical representation.

The MAC result may be added to or subtracted from the value in an accumulator register (`A0`, `A1`, or `A1:0`).

For 16-bit signed fractional operands, the 32-bit product output is format-adjusted (sign-extended and shifted one bit to the left) before being applied to either accumulator (`A0` or `A1`). For example, bit 31 of the product lines up with bit 32 of `A0` (which is bit 0 of `A0.X`), and bit 0 of the product lines up with bit 1 of `A0` (which is bit 1 of `A0.W`). The Least Significant Bit (LSB) is zero-filled, and the fractional MAC result format appears in the ***16-bit Signed Fractional Multiplier Results Format*** figure.

For integer and unsigned fractional 16-bit operands, the 32-bit product register is not shifted before being applied to `A0` or `A1`. The ***Other 16-bit Multiplier Results Format*** figure shows the integer mode result placement.

For 32-bit signed fractional operands, the 64-bit product output is format adjusted (sign-extended and shifted one bit to the left-before being applied to the `A1:0` accumulator pair). Bit 63 of the product lines up with bit 64 of `A1:0` (which is bit 0 of `A1.X`), and bit 0 of the product lines up with bit 1 of `A1:0` (which is bit 1 of `A0.W`). The Least Significant Bit (LSB) is zero-filled. Note `A0.X` is not used when the combined register `A1:0` holds a 72-bit accumulation result of 32-bit operands.

For other 32-bit integer and unsigned fractional multiplier operands, the 64-bit product is not shifted before being applied to `A1:0`.

The result of multiplying 32-bit complex operands consisting of a 16-bit imaginary part and a 16-bit real part is a pair of 40-bit signed fractions or signed integers. The accumulation proceeds as for fractional or integer multiplication with the real part of the accumulation performed in `A0` and the imaginary part in `A1`.

MAC results generate status information when they update accumulators or when they are transferred to a destination register in the register file. For more information, see Using Computational Status.

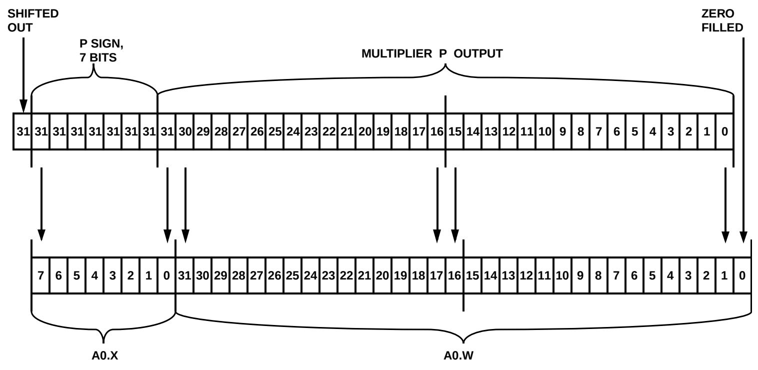


Figure 2-5: 16-bit Signed Fractional Multiplier Results Format

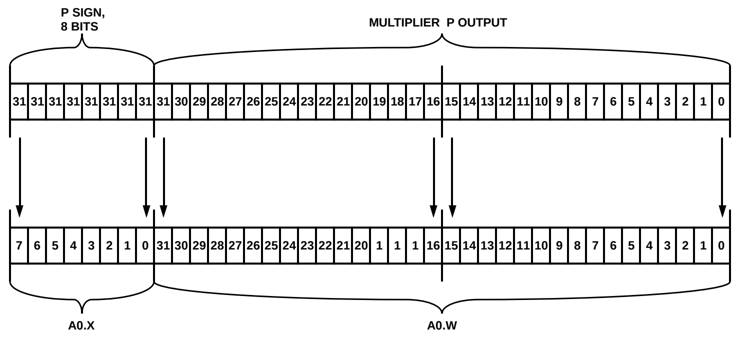


Figure 2-6: Other 16-bit Multiplier Results Format

Shifter Data Types

Many operations in the shifter are explicitly geared to signed (two's-complement) or unsigned values. Logical shifts assume unsigned magnitude or binary string values, and arithmetic shifts assume two's-complement values.

The exponent logic assumes two's-complement numbers. The exponent logic supports block floating point, which is also based on two's-complement fractions.

Shifter results generate status information. For more information about using shifter status, see Using Computational Status.

Arithmetic Formats Summary

The *ALU Arithmetic Formats* table, *MAC Fractional Modes Formats* table, *MAC Arithmetic Integer Modes Formats* table, and *Shifter Arithmetic Formats* table summarize some of the arithmetic characteristics of computational operations.

Table 2-3: ALU Arithmetic Formats

Operation	Operand Formats	Result Formats
Addition	Signed or unsigned	Interpret status bits
Subtraction	Signed or unsigned	Interpret status bits
Logical	Binary string	Same as operands
Division	Explicitly signed or unsigned	Same as operands

Table 2-4: MAC Fractional Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	1.15 signed fractional 1.31 signed fractional 0.16 unsigned fractional 0.32 unsigned fractional 2x 1.15 signed fractional complex	2.30, shifted to 1.31 2.62, shifted to 1.63 0.32, not shifted 0.64, not shifted 2x 2.30, shifted to 2x 1.31
Multiplication/Addition	1.15 and 8.32 signed fractional 1.31 and 8.64 signed fractional 0.16 and 8.32 unsigned fractional 0.32 and 8.64 unsigned fractional 2x 1.15 and 2x 8.32 signed fractional complex	8.32 signed 8.64 signed 8.32 unsigned 8.64 unsigned 2x 8.32
Multiplication/Subtraction	1.15 and 8.32 signed fractional 1.31 and 8.64 signed fractional 0.16 and 8.32 unsigned fractional 0.32 and 8.64 unsigned fractional 2x 1.15 and 2x 8.32 signed fractional complex	8.32 signed 8.64 signed 8.32 unsigned 8.64 unsigned 2x 8.32

Table 2-5: MAC Arithmetic Integer Modes Formats

Operation	Operand Formats	Result Formats
Multiplication	16.0 signed or unsigned	32.0, not shifted

Table 2-5: MAC Arithmetic Integer Modes Formats (Continued)

Operation	Operand Formats	Result Formats
	32.0 signed or unsigned 2x 16.0 signed integer complex	64.0, not shifted 2x 64.0, not shifted
Multiplication/Addition	16.0 signed or unsigned and 40.0 32.0 signed or unsigned and 72.0 2x 16.0 and 2x 40.0 signed integer complex	40.0 72.0 2x 40.0
Multiplication/Subtraction	16.0 signed or unsigned and 40.0 32.0 signed or unsigned and 72.0 2x 16.0 and 2x 40.0 signed integer complex	40.0 72.0 2x 40.0

Table 2-6: Shifter Arithmetic Formats

Operation	Operand Formats	Result Formats
Logical Shift	Unsigned binary string	Same as operands
Arithmetic Shift	Signed	Same as operands
Exponent Detect	Signed	Same as operands

Rounding MAC Results

On many MAC operations, the processor supports rounding (`RND` option) of the results. Rounding is a means of reducing the precision of a number by removing the lower bits from that number's representation and possibly modifying the remaining portion of the number to more accurately represent its former value. For example, if an original number has N bits of precision and the desired number has only M bits of precision (where N > M), the process of rounding removes N - M bits of precision from the number.

The `RND_MOD` bit in the `ASTAT` register determines whether the `RND` option provides biased or unbiased rounding. For unbiased rounding, set the `RND_MOD` bit = 0. For biased rounding, set the `RND_MOD` bit = 1.

Unbiased Rounding

The convergent rounding method returns the number closest to the original number. In cases where the original number lies exactly halfway between two numbers, this method returns the nearest even number (the one containing an LSB of 0). For example, when rounding the three-bit, two's-complement fraction 0.25 (binary 0.01) to the nearest two-bit, two's-complement fraction, the result would be 0.0 because that is the even-numbered choice between 0.5 and 0.0. Since it rounds up and down based on the surrounding values, this method is called unbiased rounding.

Unbiased rounding uses the ALU's capability of rounding 72-bit results at the boundary between bit 31 and bit 32 and 40-bit results at the boundary between bit 15 and bit 16. Rounding can be specified as part of the instruction syntax. When rounding is selected, the output register contains the rounded 16-bit result. The accumulator is never rounded.

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding adds a one into bit position 31 or 15 of the adder chain. This method causes a net positive bias because the midway value is always rounded upward.

The accumulator eliminates this bias by forcing bit 32 or bit 16 in the result output to 0 when it detects this mid-way point. Forcing this bit to 0 has the effect of rounding odd values in the discarded part of the result upward and even values downward, yielding a large sample bias of 0, assuming uniformly distributed values.

The following examples use x to represent any bit pattern (not all zeros). The example in the ***Unbiased Multiplier Rounding*** figure shows a typical rounding operation for `A0`, but the example also applies to `A1`.

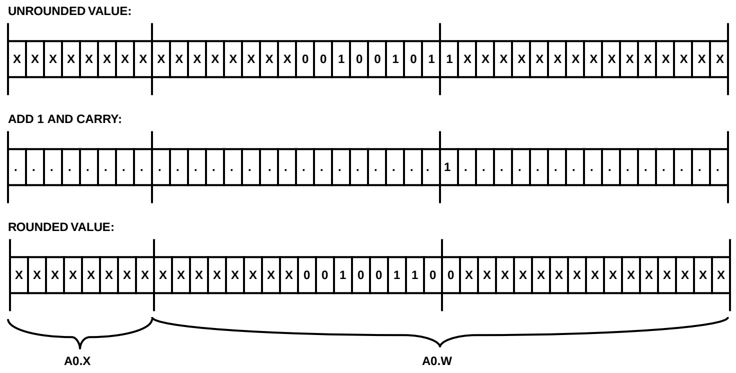


Figure 2-7: Typical Unbiased Multiplier Rounding

The compensation to avoid net bias becomes visible when all of the lower 15 bits are 0 and bit 15 is 1 (the midpoint value), as shown in the ***Avoiding Net Bias in Unbiased Multiplier Rounding*** figure. In this figure, bit 16 of `A0` is forced to 0. This algorithm is employed on every rounding operation, but it is evident only when the bit patterns shown in the lower 16 bits of the next example are present. When a 72-bit value is rounded, the bias becomes visible when all of the lower 31 bits are 0 and bit 31 is 1. In this case, net bias is avoided by forcing bit 32 to 0.

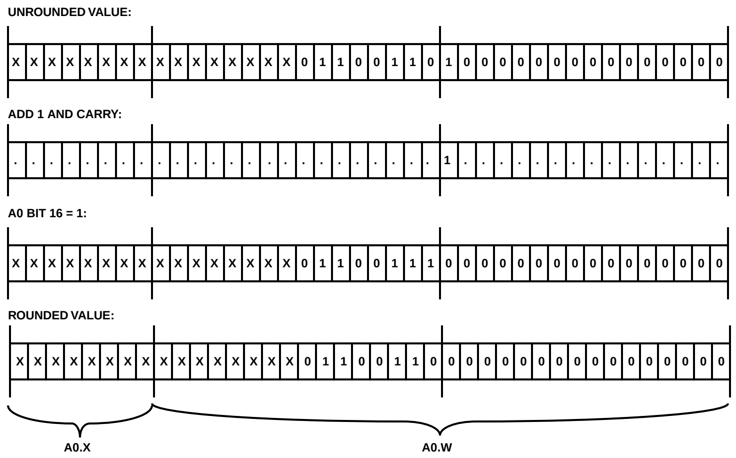


Figure 2-8: Avoiding Net Bias in Unbiased Multiplier Rounding

Biased Rounding

The round-to-nearest method also returns the number closest to the original. However, by convention, an original number lying exactly halfway between two numbers always rounds up to the larger of the two. For example, when rounding the three-bit, two's-complement fraction 0.25 (binary 0.01) to the nearest two-bit, two's-complement fraction, this method returns 0.5 (binary 0.1). The original fraction lies exactly midway between 0.5 and 0.0 (binary 0.0), so this method rounds up. Because it always rounds up, this method is called biased rounding.

The RND_MOD bit in the ASTAT register enables biased rounding. When the RND_MOD bit is cleared, the RND option in multiplier instructions uses the normal, unbiased rounding operation, as discussed in Unbiased Rounding.

When the RND_MOD bit is set (=1), the processor uses biased rounding instead of unbiased rounding. When operating in biased rounding mode, all rounding operations performed on 72-bit results with A0.W set to 0x80000000 round up, rather than only rounding odd values up. Similary, all rounding operations performed on 40-bit results with A0.L/A1.L set to 0x8000 round up. For an example of biased rounding, see Table 2-7 Biased Rounding in Multiplier Operation.

Table 2-7: Biased Rounding in Multiplier Operation

A0/A1 Before RND	Biased RND Result	Unbiased RND Result
0x00 0000 8000	0x00 0001 8000	0x00 0000 0000
0x00 0001 8000	0x00 0002 0000	0x00 0002 0000
0x00 0000 8001	0x00 0001 0001	0x00 0001 0001
0x00 0001 8001	0x00 0002 0001	0x00 0002 0001
0x00 0000 7FFF	0x00 0000 FFFF	0x00 0000 FFFF
0x00 0001 7FFF	0x00 0001 FFFF	0x00 0001 FFFF

Biased rounding affects 40-bit results only when the A0.L/A1.L register contains 0x8000 and 72-bit results only when A0.W contains 0x80000000. All other rounding operations work normally. This mode allows more efficient implementation of bit-specified algorithms that use biased rounding, such as the Global System for Mobile Communications (GSM) speech compression routines.

Truncation

Another common way to reduce the significant bits representing a number is to simply mask off the N - M lower bits. This process is known as *truncation* and results in a relatively large bias. Instructions that do not support rounding revert to truncation. The ASTAT.RND_MOD bit has no effect on truncation.

Special Rounding Instructions

The ALU provides the ability to round the arithmetic results directly into a data register with biased or unbiased rounding, as described previously. It also provides the ability to round on different bit boundaries. The options RND12, RND, and RND20 round at bit 12, bit 16, and bit 20, respectively, regardless of the state of the ASTAT.RND_MOD bit. For example:

```
R3.L = R4 (RND) ;
```

performs biased rounding at bit 16, depositing the result in a half word (`R3.L`).

```
R3.L = R4 + R5 (RND12) ;
```

performs an addition of two 32-bit numbers, does biased rounding at bit 12, and deposits the result in a half word (`R3.L`).

```
R3.L = R4 + R5 (RND20) ;
```

performs an addition of two 32-bit numbers, does biased rounding at bit 20, and deposits the result in a half word (`R3.L`).

Using Computational Status

The MAC, ALU, and shifter update the overflow and other status bits in the processor's Arithmetic Status (`ASTAT`) register. To use status conditions from computations in program sequencing, use conditional instructions to test the `CC` status bit in the `ASTAT` register after the instruction executes. This method permits monitoring each instruction's outcome. The `ASTAT` register is a 32-bit register, with some bits reserved. To ensure compatibility with future implementations, writes to this register should write back the values read from these reserved bits.

ASTAT Register

The ***Arithmetic Status register*** (`ASTAT`) provides information about the result of an operation. The processor updates the status bits in `ASTAT`, indicating the status of the most recent ALU, MAC, or shifter operation. For more information, see Arithmetic Status Register .

Arithmetic Logic Unit (ALU)

The two ALUs perform arithmetic and logical operations on fixed-point data. ALU fixed-point instructions operate on 16-, 32-, and 40-bit fixed-point operands and output 16-, 32-, or 40-bit fixed-point results. ALU instructions include:

- Fixed-point addition and subtraction of registers
- Addition and subtraction of immediate values
- Accumulation and subtraction of multiplier results
- Logical `AND`, `OR`, `NOT`, `XOR`, bitwise XOR (`BXOR`), and Negate operations
- Functions:
  - Absolute Value (`ABS`)
  - Maximum (`MAX`) and Minimum (`MIN`)
  - Rounding (`(RND)`)

- Division primitives (`DIVS`and `DIVQ`)

ALU Operations

Primary ALU operations occur on ALU0, while parallel operations occur on ALU1, which performs a subset of ALU0 operations.

The ***Inputs and Outputs of Each ALU*** table describes the possible inputs and outputs of each ALU.

Table 2-8: Inputs and Outputs of Each ALU

Input	Output
Two or four 16-bit operands	One or two 16-bit results
Two 32-bit operands	One 32-bit result
32-bit result from the multiplier	Combination of 32-bit result from the multiplier with a 40-bit accumulation result

Combining operations in both ALUs can result in four 16-bit results, two 32-bit results, or two 40-bit results generated in a single instruction.

Single 16-Bit Operations

In single 16-bit operations, any two 16-bit register halves may be used as the input to the ALU. An addition, subtraction, or logical operation produces a 16-bit result that is deposited into an arbitrary destination register half. ALU0 is used for this operation because it is the primary resource for ALU operations. For example:

```
R3.H = R1.H + R2.L (NS) ;
```

adds the 16-bit contents of `R1.H` (`R1` high half) to the contents of `R2.L` (`R2` low half) and deposits the result in `R3.H` (`R3` high half) with no saturation (`(NS)`).

Dual 16-Bit Operations

In dual 16-bit operations, any two 32-bit registers may be used as the input to the ALU, considered as pairs of 16-bit operands. An addition, subtraction, or logical operation produces two 16-bit results that are deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation because it is the primary resource for ALU operations. For example:

```
R3 = R1 +|- R2 (S) ;
```

adds the 16-bit contents of `R2.H` (`R2` high half) to the contents of `R1.H` (`R1` high half) and deposits the result in `R3.H` (`R3` high half) with saturation (`(S)`). The instruction also subtracts the 16-bit contents of `R2.L` (`R2` low half) from the contents of `R1.L` (`R1` low half) and deposits the result in `R3.L` (`R3` low half) with saturation. For more information, see 16-bit MAC Data Flow Details.

Quad 16-Bit Operations

In quad 16-bit operations, any two 32-bit registers may be used as the inputs to ALU0 and ALU1, considered as pairs of 16-bit operands. A small number of addition or subtraction operations produces four 16-bit results that are deposited into two arbitrary 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because there are only two 32-bit data paths from the data register file to the arithmetic units, the same two pairs of 16-bit inputs must be presented to both ALU1 and ALU0. The instruction construct is identical to that of a dual 16-bit operation, except it is duplicated for both ALUs and identifies unique destination registers for the ALU operation to store results to. For example:

```
R3 = R0 +|+ R1, R2 = R0 -|- R1 (S) ;
```

performs four operations:

- Adds the 16-bit contents of `R1.H` (`R1` high half) to the 16-bit contents of `R0.H` (`R0` high half) and deposits the result in `R3.H` with saturation (`(S)`).
- Adds `R1.L` to `R0.L` and deposits the result in `R3.L`, also with saturation.
- Subtracts the 16-bit contents of `R1.H` (`R1` high half) from the 16-bit contents of `R0.H` (`R0` high half) and deposits the result in `R2.H`, also with saturation.
- Subtracts `R1.L` from `R0.L` and deposits the result in `R2.L`, also with saturation.

This single-cycle instruction is the equivalent of the following four-cycle sequence of instructions:

```
R3.H = R0.H + R1.H (S) ;
R3.L = R0.L + R1.L (S) ;
R2.H = R0.H - R1.H (S) ;
R2.L = R0.L - R1.L (S) ;
```

Single 32-Bit Operations

In single 32-bit operations, any two 32-bit registers may be used as the input to the ALU, considered to be 32-bit operands. An addition, subtraction, or logical operation produces a 32-bit result that is deposited into an arbitrary 32-bit destination register. ALU0 is used for this operation because it is the primary resource for ALU operations.

In addition to the 32-bit input operands coming from the data register file, operands may also be sourced from and deposited into the pointer register file, consisting of the eight registers `P[5:0]`, `SP`, and `FP`. However, the pointer register file and data register file cannot be used interchangeably in the same instruction. For example:

```
R3 = R1 + R2 (NS) ;
```

adds the 32-bit contents of `R2` to the 32-bit contents of `R1` and deposits the result in `R3` with no saturation.

```
P3 = P1 + SP ;
```

adds the 32-bit contents of `P1` to the 32-bit contents of `SP` and deposits the result in `P3`. Notice that the saturation qualifier is not supported when using the pointer register file, as `ASTAT` bits are not affected by ALU operations on pointer register file registers.

```
P3 = R1 + R2 (NS) ;
```

is an illegal instruction, with or without the saturation qualifier, as it attempts to source the data register file for an operation that is writing to the pointer register file.

Dual 32-Bit Operations

In dual 32-bit operations, any two 32-bit registers may be used as the input to ALU0 and ALU1, considered to be a pair of 32-bit operands. An addition or subtraction produces two 32-bit results that are deposited into two 32-bit destination registers. Both ALU0 and ALU1 are used for this operation. Because only two 32-bit data paths go from the data register file to the arithmetic units, the same two 32-bit input registers must be presented to both ALU0 and ALU1. For example:

```
R3 = R1 + R2, R4 = R1 - R2 (NS);
```

adds the 32-bit contents of `R2` to the 32-bit contents of `R1` and deposits the result in `R3` with no saturation (`(NS)`). It also subtracts the 32-bit contents of `R2` from that of `R1` and deposits the result in `R4` with no saturation.

A specialized form of this instruction uses the 40-bit ALU result registers as input operands, creating the sum and difference of the `A0` and `A1` registers. For example:

```
R3 = A0 + A1, R4 = A0 - A1 (S);
```

transfers the saturated 32-bit sum of the accumulators to the `R3` register and the saturated 32-bit difference between the accumulators to the `R4` register.

ALU Division Support Features

The ALU supports division with two special divide primitives. These instructions (`DIVS`, `DIVQ`) let programs implement a non-restoring, conditional (error checking) addition/subtraction/division algorithm.

The division can be either signed or unsigned, but both the dividend and divisor must be of the same type. Details about using division and programming examples are available in the arithmetic operation chapter.

Special SIMD Video ALU Operations

Four 8-bit video ALUs enable the processor to process video information with high efficiency. Each video ALU instruction may take from one to four pairs of 8-bit inputs and return one to four 8-bit results. The inputs are presented to the video ALUs in two 32-bit words from the data register file. The possible operations include:

- Quad 8-Bit Add or Subtract
- Quad 8-Bit Average
- Quad 8-Bit Pack or Unpack
- Quad 8-Bit Subtract-Absolute-Accumulate
- Byte Align

For more information about the operation of these instructions, see the video/pixel operation instructions chapter.

ALU Instruction Summary

The Table 2-9 ALU Instructions and Status table lists the ALU instructions and how they affect the ASTAT status bits. In the table, note the meaning of these symbols:.

- Dreg denotes any data register file register.
- Dreg_lo_hi denotes any 16-bit register half in any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- imm7 denotes a signed, 7-bit wide, immediate value.
- A*x* denotes either ALU result register A0 or A1 .
- DIVS denotes a divide sign primitive.
- DIVQ denotes a divide quotient primitive.
- MAX denotes the maximum, or most positive, value of the source registers.
- MIN denotes the minimum value of the source registers.
- ABS denotes the absolute value of the upper and lower halves of a single 32-bit register.
- RND denotes rounding a half word.
- RND12 denotes saturating the result of an addition or subtraction and rounding the result at bit 12.
- RND20 denotes saturating the result of an addition or subtraction and rounding the result at bit 20.
- SIGNBITS denotes the number of sign bits in a number minus one.
- EXPADJ denotes the lesser of the number of sign bits in a number minus one and a threshold value.
- *indicates the status bit may be set or cleared, depending on the results of the instruction.
- ** indicates the status bit is cleared.
- - indicates no effect.
- *d* indicates AQ contains the dividend MSB Exclusive-OR divisor MSB.

Table 2-9: ALU Instructions and Status

	ASTAT Status Bits						
Instruction	**AZ**	**AN**	**AC0 AC0_COPY AC1**	**AV0 AV0S**	**AV1 AV1S**	**V V_COPY VS**	**AQ**
Dreg = Dreg + Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg - Dreg (S) ;	*	*	*	-	-	*	-
Dreg = Dreg + Dreg,	*	*	*	-	-	*	-

Table 2-9: ALU Instructions and Status (Continued)

Instruction	ASTAT Status Bits						
	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
Dreg = Dreg - Dreg ;							
Dreg_lo_hi = Dreg_lo_hi + Dreg_lo_hi ;	*	*	*	-	-	*	-
Dreg_lo_hi = Dreg_lo_hi - Dreg_lo_hi (S) ;	*	*	*	-	-	*	-
Dreg = Dreg +|+ Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg +|- Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg -|+ Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg -|- Dreg ;	*	*	*	-	-	*	-
Dreg = Dreg +|+Dreg, Dreg = Dreg -|- Dreg ;	*	*	-	-	-	*	-
Dreg = Dreg +|- Dreg, Dreg = Dreg -|+ Dreg ;	*	*	-	-	-	*	-
Dreg = A*x* + A*x*, Dreg = A*x* - A*x* ;	*	*	*	-	-	*	-
Dreg += imm7 ;	*	*	*	-	-	*	-
Dreg = (A0 += A1) ;	*	*	*	*	-	*	-
Dreg_lo_hi = (A0 += A1) ;	*	*	*	*	-	*	-
A0 += A1 ;	*	*	*	*	-	-	-
A0 -= A1 ;	*	*	*	*	-	-	-
DIVS (Dreg, Dreg) ;	*	*	*	*	-	-	d
DIVQ (Dreg, Dreg) ;	*	*	*	*	-	-	d
Dreg = MAX (Dreg, Dreg) (V) ;	*	*	-	-	-	**/-	-
Dreg = MIN (Dreg, Dreg) (V) ;	*	*	-	-	-	**/-	-
Dreg = ABS Dreg (V) ;	*	**	-	-	-	*	-
A*x* = ABS A*x* ;	*	**	-	*	*	*	-
A*x* = ABS A*x*, A*x* = ABS A*x* ;	*	**	-	*	*	*	-
A*x* = -A*x* ;	*	*	*	*	*	*	-
A*x* = -A*x*, A*x* =- A*x* ;	*	*	*	*	*	*	-

Table 2-9: ALU Instructions and Status (Continued)

	ASTAT Status Bits						
Instruction	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	V V_COPY VS	AQ
A*x* = A*x* (S) ;	*	*	-	*	*	-	-
A*x* = A*x* (S), A*x* = A*x* (S) ;	*	*	-	*	*	-	-
Dreg_lo_hi = Dreg (RND) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg + Dreg (RND12) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg - Dreg (RND12) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg + Dreg (RND20) ;	*	*	-	-	-	*	-
Dreg_lo_hi = Dreg - Dreg (RND20) ;	*	*	-	-	-	*	-
Dreg_lo = SIGNBITS Dreg ;	-	-	-	-	-	-	-
Dreg_lo = SIGNBITS Dreg_lo_hi ;	-	-	-	-	-	-	-
Dreg_lo = SIGNBITS An ;	-	-	-	-	-	-	-
Dreg_lo = EXPADJ (Dreg, Dreg_lo) (V) ;	-	-	-	-	-	-	-
Dreg_lo = EXPADJ (Dreg_lo_hi, Dreg_lo);	-	-	-	-	-	-	-
Dreg = Dreg & Dreg ;	*	*	**	-	-	**/-	-
Dreg = ~ Dreg ;	*	*	**	-	-	**/-	-
Dreg = Dreg | Dreg ;	*	*	**	-	-	**/-	-
Dreg = Dreg ^ Dreg ;	*	*	**	-	-	**/-	-
Dreg =- Dreg ;	*	*	*	-	-	*	-

Multiply Accumulators (MACs)

The three MACs (MAC10, MAC0 and MAC1) perform fixed-point multiplication and multiply-accumulate operations. Multiply-accumulate operations are available with either cumulative addition or cumulative subtraction.

MAC10 executes fixed-point instructions which operate on 32-bit fixed-point data and produce 64-bit results that may be added to or subtracted from a 72-bit accumulator. It also executes instructions which operate on 32-bit complex fixed-point data and produce 64-bit results that may be added to or subtracted from an 80-bit accumulator.

MAC0 and MAC1 execute fixed-point instructions which operate on 16-bit fixed-point data and produce 32-bit results that may be added to or subtracted from a 40-bit accumulator.

Inputs are treated as fractional, fractional complex, integer, or integer complex unsigned or two's-complement data. Multiplier instructions include:

- Multiplication
- Multiply-accumulate with addition (with optional rounding)
- Multiply-accumulate with subtraction (with optional rounding)
- Dual versions of the above operations using 16-bit operands

MAC Operation

Each of the 16-bit multipliers, MAC0 and MAC1, has two 32-bit inputs from which it derives the two 16-bit operands. For single multiply-accumulate instructions, these operands can be any of the `R[n]` data registers. Each multiplier accumulates results in its accumulator register, `A1` or `A0`, which can be saturated to 32 or 40 bits. The multiplier result can also be written directly to a 16- or 32-bit destination register with optional rounding.

The 32-bit multiplier, MAC10, has two 32-bit operands which can be any of the `R[n]` data registers. The multiplier accumulates results in the register pair `A1:0`, which are saturated to 72 bits. The 64-bit multiplier result can also be written directly to a data register pair or it can be rounded or truncated to 32-bits and written to an individual data register.

Each multiplier instruction has options that specify whether the inputs are in integer or fractional format, with the format of the result matching that of the inputs. In MAC0, the inputs are either both signed or both unsigned, whereas MAC10 and MAC1 each support a mixed-mode option.

Complex multiplication instructions executed by MAC10 also have options that specify whether the format of both of the inputs is complex signed integer or complex signed fractional. For signed fractional inputs, the multiplier automatically left shifts the result by one bit to remove the redundant sign bit. Unsigned fractional, integer, and mixed modes do not perform a shift for sign bit correction.

For more information regarding multiplier instructions, see MAC Instruction Options.

Placing MAC Results in Accumulator Registers

As shown in 16-bit MAC Data Flow Details, each MAC may write results to dedicated accumulator registers `A0` or `A1`. MAC0 writes to `A0`, MAC1 writes to `A1`, and MAC10 writes to both `A0` and `A1`. Each accumulator register is divided into three parts:

- `A0.L/A1.L` contain the lowermost 16 bits (bits 15:0)
- `A0.H/A1.H` contain the next 16 bits (bits 31:16)
- `A0.X/A1.X` contain the uppermost 8 bits (bits 39:32)

When MAC0 or MAC1 write to its respective accumulator register, the 32-bit result is deposited into the lower bits of the combined accumulator register (`A0.H`/`A0.L` and `A1.H`/`A1.L`), and the MSB is sign-extended into the upper eight bits of the register (`A0.X`/`A1.X`).

The results of 32-bit fixed-point and complex multiplication instructions utilize the `A1:0` accumulator register pair as a 64-bit meta-register. These MAC10 operations write the lower 32 bits of the 64-bit fixed-point result to `A0.W` and the upper 32 bits to `A1.W`. Because MAC1 supplies the most significant 32 bits of the result, the resulting value can be either sign- or zero-extended into `A1.X`. However, because MAC0 is providing only the lower 32 bits of the result, `A0.X` is not used and is always set to zero. MAC10 writes the real part of complex results to `A0` and the imaginary part to `A1`.

The accumulator pair can be initialized by transferring data from a data register pair using a dual-register move. For example, these instructions transfer 64-bit values into the 72-bit accumulator:

```
A1 = R1 (X), A0 = R0 (X); /* sign-extend R1:0 into A1:0 */
A1 = R1 (Z), A0 = R0 (Z); /* zero-extend R1:0 into A1:0 */
A1 = A0 = 0;              /* A1:0 = 0 */
```

Rounding or Saturating MAC Results

On a multiply-accumulate operation, the accumulator data can be saturated and, optionally, rounded for extraction to a register pair, register, or register half. When a multiply deposits a result only in a register pair, register or register half, the saturation and rounding works the same way. The rounding and saturation operations work as follows.

- Rounding is applied only to fractional results, with the exception of the `IH` option, which applies rounding and high half extraction to an integer result. For the `IH` option, the rounded result is obtained by adding 0x8000 to the accumulator (for MAC) or the multiplication result register and then saturating it to 32 bits. For more information, see MAC Instruction Options. Rounding cannot be combined with a multiply-accumulate into the 72-bit accumulator, `A1:0`, but it can be performed by an instruction which only extracts the value from `A1:0` into a data register.
- If an overflow or underflow occurs during the operation, the saturate operation sets the specified result register to the maximum positive or negative value. For more information, see Saturating MAC Results on Overflow.
- The `NS` option prevents saturation. When an overflow or underflow has occurred, the specified result register is set to the low-order bits of the full result. The `NS` option is only supported for integer multiplications of 32-bit operands.

Saturating MAC Results on Overflow

The following bits in `ASTAT` indicate multiplier overflow status:

- `ASTAT.AV0` (bit 16) and `ASTAT.AV1` (bit 18) record an overflow condition for the `A0` and `A1` accumulators, respectively. For MAC10 operations, `ASTAT.AV0` also records the overflow condition for `A1:0`. If the bit is cleared (=0), no overflow or underflow has occurred. If the bit is set (=1), an overflow or underflow has occurred. The `ASTAT.AV0S` (bit 17) and `ASTAT.AV1S` (bit 19) bits are sticky bits which must be explicitly cleared by application code.

- `ASTAT.V` (bit 24) and `ASTAT.VS` (bit 25) are set if the overflow occurs in extracting the accumulator result to a register, with `ASTAT.VS` being the sticky version of the `ASTAT.V` bit which must be cleared explicitly by application code.

32-bit MAC Data Flow Details

The 32-bit MAC has two 32-bit inputs, performs a 32-bit fixed-point or complex multiplication, and either stores the result to the 72-bit meta-register comprised of the accumulator register pair `A1:0` or extracts to a 32-bit register or to a 64-bit register pair.

For complex calculations, the 32-bit real and imaginary results are passed to 40-bit adder/subtracters, which may be used to modify the values in the accumulator registers, with the real part in `A0` and the imaginary part in `A1`. Alternatively, the result may be written directly to a `R[n]` data register.

For fixed-point calculations, the 64-bit product is passed to a 72-bit adder/subtracter, which may be used to modify the values in the 72-bit meta-register comprised of the accumulator register pair `A1:0`, which is a concatenation of two 32-bit registers (`A0.W` and `A1.W`) and an 8-bit register (`A1.X`). For example:

```
A1:0 += R2 * R3 ;
```

In this instruction, the multiply is performed, and the results are added to the previous value in the 72-bit `A1:0` accumulator register pair. Alternatively, the new product could also be passed directly to any of the `R[n]` data registers.

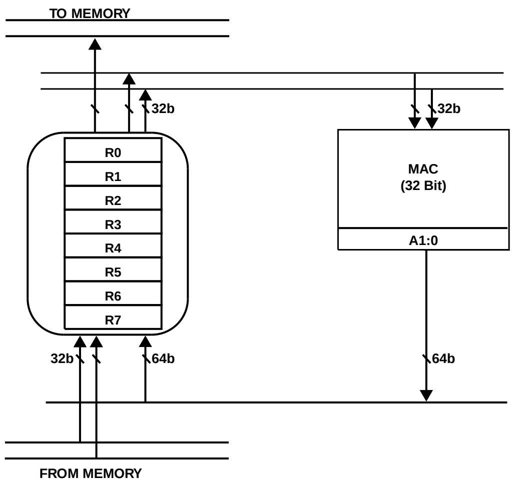


Figure 2-9: 32-bit Multiplier/Accumulators

32-bit Multiply Without Accumulate

The MAC may operate without the accumulation function. If accumulation is not used, the result can be directly stored to any of the `R[n]` data registers or to an accumulator register. The destination can be an individual 32-bit register or a 64-bit register pair. If the destination register is 32 bits, then the data that is extracted from the multiplier is the most useful information, which is dependent on the data type of the input:

- Fractional operands - the upper half of the result, which contains the sign information and the most significant bits of the fractional data, is extracted and stored in the 32-bit destination register.
- Integer operands - the lower half of the result is extracted and stored in the 32-bit destination register.
- Complex fractional operands - the upper half of the real result is extracted and stored in the lower half of the 32-bit destination register, and the upper half of the imaginary result is extracted and stored in the upper half of the 32-bit destination register.
- Complex integer operands - the lower half of the real result is extracted and stored in the lower half of the 32-bit destination register, and the lower half of the imaginary result is extracted and stored in the upper half of the 32-bit destination register.

For example:

```
R0 = R1 * R2 (FU) ;
```

The `(FU)` qualifier indicates that the inputs are unsigned fractional data. This instruction deposits the upper 32 bits of the multiplication result (by default, with rounding and saturation) into `R0`.

This instruction uses unsigned integer operands, as designated by the `(IU)` qualifier:

```
R1 = R2 * R3 (IU, NS) ;
```

The lower 32 bits of the multiplication result are deposited into `R1` (without saturation, as designated by the `(NS)` qualifier).

This instruction is an example of a multiply being stored to a 64-bit register pair:

```
R1:0 = R1 * R2 ;
```

Regardless of the operand type, this instruction computes a 64-bit multiplication result (by default, with saturation) and deposits the upper 32 bits into `R1` and the lower 32 bits into `R0`.

This instruction uses complex fractional operands:

```
R1 = cmul(R2, R3) ;
```

The upper 16 bits of the real part of the multiplication result are stored to `R1.L,` and the upper 16 bits of the imaginary part of the result go to `R1.H.`

This instruction uses complex signed integer operands, as designated by the `(IS)` qualifier:

```
R1 = cmul(R2, R3)(IS);
```

The lower 16 bits of the real part of the multiplication result are stored to `R1.L,` and the lower 16 bits of the imaginary part of the result go to `R1.H.`

This instruction is an example of a complex multiply being stored to a 64-bit register pair:

```
R1:0 = cmul(R2, R3) ;
```

The full 32 bits of the real part of the multiplication result are stored to `R0`, and the full 32 bits of the imaginary part of the result go to `R1`.

For backward compatibility, the Blackfin+ processor also supports a two-operand version of the 32-bit multiply instruction:

```
R0 *= R1 ;
```

This is equivalent to:

```
R0 = R0 * R1 (IS, NS) ;
```

Note that the assumptions are that the input operands are signed integers (`(IS)`) and that the result will not be saturated (`(NS)`).

16-bit MAC Data Flow Details

The ***16-bit Multiplier/Accumulators*** figure shows the register file along with the 16-bit multiplier/accumulators.

Each 16-bit MAC has two 16-bit inputs, performs a 16-bit multiplication, and either stores the result to a 40-bit accumulator or extracts it to a 16-bit or 32-bit register. Two 32-bit words are available at the MAC inputs, providing four 16-bit operands to chose from.

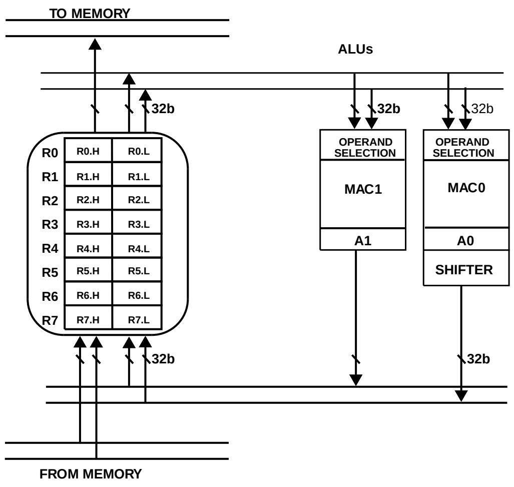


Figure 2-10: 16-bit Multiplier/Accumulators

One of the operands must be from one of the halves of a 32-bit register, while the other operand comes from another half-register. In this fashion, each MAC can process four possible input operand combinations. The ***Four Possible Combinations of 16-bit MAC Operands*** figure shows these possible combinations.

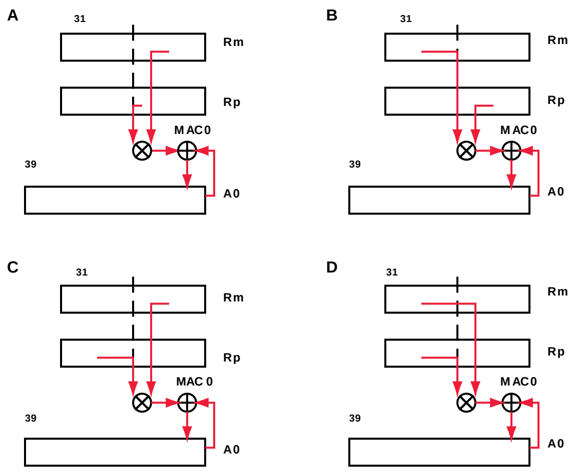


Figure 2-11: Four Possible Combinations of 16-bit MAC Operands

NOTE: As shown in the figure, the inputs to the MAC must come from two different registers. If the two 32-bit registers contain the same data, the equivalent of squaring a half-register or multiplying the upper half and lower half of the same register becomes possible.

The 32-bit product is passed to a 40-bit adder/subtracter, which can modify the contents of the accumulator register by the computed result or pass the product directly to the data register file. The 40-bit `A0` and `A1` accumulator registers are each comprised of a 32-bit register (`A0.W` and `A1.W`, respectively) and an 8-bit register (`A0.X` and `A1.X`, respectively). For example:

```
A1 += R3.H * R4.H ;
```

In this instruction, `MAC1` performs a multiply of two 16-bit inputs and modifies the current 40-bit `A1` accumulator content by the computed product.

16-bit Multiply Without Accumulate

The 16-bit MAC may operate without the accumulation function. If accumulation is not used, the result can be directly stored to a data register file register or to an accumulator register. The destination register may be 16 bits or 32 bits. If the 16-bit destination register is a low half-register (e.g., `R[n].L`), then MAC0 is used. Conversely, if it is a high half-register (e.g., `R[n].H`), then MAC1 is used. For a 32-bit destination register, either MAC0 or MAC1 can be used.

For 16-bit destination registers, the data that is extracted from the multiplier is the most useful data, which is dependent on the format of the input data:

- Fractional operands (or when the `(IH)` qualifier is used) - the upper half of the result is extracted and stored to the 16-bit destination register (see the ***Multiplication of 16-bit Fractional Operands*** figure).
- Integer operands - the lower half of the result is extracted and stored to the 16-bit destination register (see the ***Multiplication of 16-bit Integer Operands*** figure).

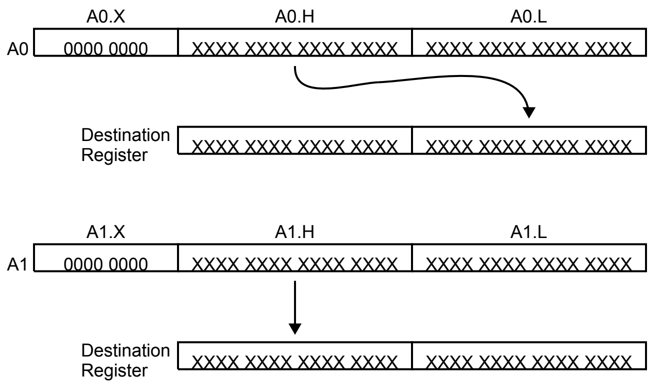


Figure 2-12: Multiplication of 16-bit Fractional Operands

For example, this instruction uses unsigned fractional input operands, as designated by the `(FU)` qualifier:

```
R0.L = R1.L * R2.L (FU) ;
```

The upper 16 bits of the MAC0 multiplication result (by default, with rounding and saturation) are deposited into the lower half of `R0`.

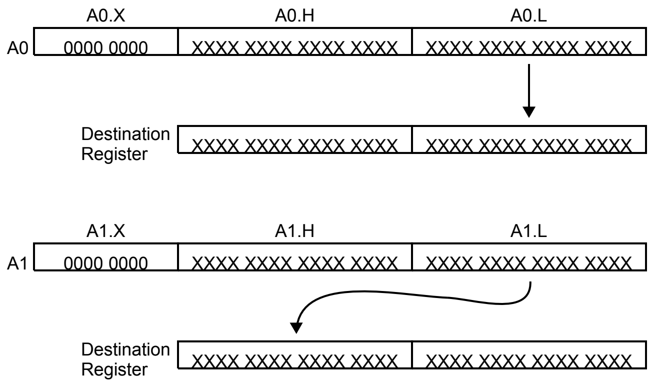


Figure 2-13: Multiplication of 16-bit Integer Operands

For example, this instruction uses unsigned integer input operands, as designated by the `(IU)` qualifier:

```
R0.H = R2.H * R3.H (IU) ;
```

The lower 16 bits of the MAC1 multiplication result (with saturation) are deposited into the upper half of `R0`.

Finally, for a 16-bit multiply storing to a 32-bit register:

```
R0 = R1.L * R2.L ;
```

Regardless of the input operand type, 32 bits of the MAC0 multiplication result (with saturation) are stored to `R0`, using MAC0.

Dual 16-bit MAC Operations

The processor has two 16-bit MACs, which can be used simultaneously to double the MAC throughput. The same two 32-bit input registers are input to each MAC unit, providing each with four possible combinations of 16-bit input operands. Dual-MAC operations are frequently referred to as vector operations because data can be arranged and stored to registers such that vector computations are possible.

An example of a dual multiply-accumulate instruction is:

```
A1 += R1.H * R2.L, A0 += R1.L * R2.H ;
```

This instruction represents two multiply-accumulate operations performed simultaneously by the core:

- In the first operation, the `A1` accumulator denotes use of MAC1. The high half of `R1` is multiplied by the low half of `R2`, and the product is then used to modify the previous content of the `A1` accumulator.
- In the second operation, the `A0` accumulator denotes use of MAC0. The low half of `R1` is multiplied by the high half of `R2`, and the product is then used to modify the previous content of the `A0` accumulator.

The results of the MAC operations may be written to registers in a number of ways:

- as a pair of 16-bit halves
- as a pair of 32-bit registers
- as an independent 16-bit register half
- as an independent 32-bit register

For example, consider the case of writing to a pair of 16-bit destination register halves:

```
R3.H = (A1 += R1.H * R2.L), R3.L = (A0 += R1.L * R2.L) ;
```

Each 40-bit accumulator is packed into a 16-bit register half. The result from MAC1 (`A1`) must be transferred to a high half of a destination register (`R3.H`), and the result from MAC0 (`A0`) must be transferred to the low half of the same destination register (`R3.L`).

The data format of the input operands determines the correct bits to extract from the accumulator to deposit into the 16-bit destination register. See 16-bit Multiply Without Accumulate.

When writing to a pair of 32-bit destination registers:

```
R3 = (A1 += R1.H * R2.L), R2 = (A0 += R1.L * R2.L) ;
```

In this case, the same multiply-accumulate results in the 40-bit accumulators are instead packed into two 32-bit data registers, with the MAC1 results going to `R3` and the MAC0 results going to `R2`). The destination registers must be in defined pairs (`R[1:0]`, `R[3:2]`, `R[5:4]`, or `R[7:6]`), with the MAC1 result targeting the higher register in the pair and the MAC0 result targeting the lower register in the pair.

Mixed modes are also supported. For example:

```
R3.H = (A1 += R1.H * R2.L), A0 += R1.L * R2.L ;
```

This instruction is an example of one accumulator being transferred to a data register while the other is used to compute the multiply-accumulate without being transferred to a data register. Either a 16- or a 32-bit register may be specified as the destination register.

MAC Instruction Summary

The Table 2-10 MAC Instructions and Status table lists the MAC instructions and how they affect the `ASTAT` status bits. In the table, note the meaning of these symbols:

- Dreg denotes any data register file register.
- Dreg_lo_hi denotes any 16-bit register half in any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- Dreg_pair denotes a pair of adjacent data register file registers. The lower half of the 64-bit value is stored in the lower-numbered even register, while the upper half is stored in the higher-numbered odd register.
- x denotes either the `A0` or `A1` MAC accumulator register.
- * indicates the status bit may be set or cleared, depending on the result of the computation.
- - indicates no effect.

MAC instruction options are described in MAC Instruction Options.

Table 2-10: MAC Instructions and Status

	ASTAT Status Bits		
Instruction	**AV0 AV0S**	**AV1 AV1S**	**V V_COPY VS**
Dreg = Dreg * Dreg ;	-	-	*
Dreg_pair = Dreg * Dreg ;	-	-	*
A1:0 = Dreg * Dreg ;	*	-	-
A1:0 += Dreg * Dreg ;	*	-	-
A1:0 -= Dreg * Dreg ;	*	-	-
Dreg = (A1:0 = Dreg * Dreg) ;	*	-	*
Dreg = (A1:0 += Dreg * Dreg) ;	*	-	*
Dreg = (A1:0 -= Dreg * Dreg) ;	*	-	*
Dreg_pair = (A1:0 = Dreg * Dreg) ;	*	-	*
Dreg_pair = (A1:0 += Dreg * Dreg) ;	*	-	*
Dreg_pair = (A1:0 -= Dreg * Dreg) ;	*	-	*
Dreg = cmul(Dreg, Dreg) ;	-	-	*
Dreg_pair = cmul(Dreg, Dreg) ;	-	-	*
A1:0 = cmul(Dreg, Dreg) ;	*	*	-

Table 2-10: MAC Instructions and Status (Continued)

Instruction	ASTAT Status Bits		
	AV0 AV0S	AV1 AV1S	V V_COPY VS
A1:0 += cmul(Dreg, Dreg) ;	*	*	-
A1:0 -= cmul(Dreg, Dreg) ;	*	*	-
Dreg = (A1:0 = cmul(Dreg, Dreg)) ;	*	*	*
Dreg = (A1:0 += cmul(Dreg, Dreg)) ;	*	*	*
Dreg = (A1:0 -= cmul(Dreg, Dreg)) ;	*	*	*
Dreg_pair = (A1:0 = cmul(Dreg, Dreg)) ;	*	*	*
Dreg_pair = (A1:0 += cmul(Dreg, Dreg)) ;	*	*	*
Dreg_pair = (A1:0 -= cmul(Dreg, Dreg)) ;	*	*	*
Dreg_lo = Dreg_lo_hi * Dreg_lo_hi ;	-	-	*
Dreg_hi = Dreg_lo_hi * Dreg_lo_hi ;	-	-	*
Dreg = Dreg_lo_hi * Dreg_lo_hi ;	-	-	*
Ax = Dreg_lo_hi * Dreg_lo_hi ;	*	*	-
Ax += Dreg_lo_hi * Dreg_lo_hi ;	*	*	-
Ax -= Dreg_lo_hi * Dreg_lo_hi ;	*	*	-
Dreg_lo = (A0 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_lo = (A0 -= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg_hi = (A1 -= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (Ax = Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (Ax += Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg = (Ax -= Dreg_lo_hi * Dreg_lo_hi) ;	*	*	*
Dreg *= Dreg ;	-	-	-

MAC Instruction Options

The following is a summary of all the MAC instruction options, as not all options are available for all instructions. For information regarding how to use specific options with individual instructions, see the arithmetic operation instructions chapter.

default

No option; input data format is signed fractional.

`(IS)`

Input data operand format is signed integer. No shift correction is made.

`(FU)`

Input data operand format is unsigned fractional. No shift correction is made.

`(IU)`

Input data operand format is unsigned integer. No shift correction is made.

`(T)`

Input data operand format is signed fractional. When copying to the destination register half, the lower 16 bits of the accumulator content is truncated.

`(TFU)`

Input data operand format is unsigned fractional. When copying to the destination register half, the lower 16 bits of the accumulator content is truncated.

`(ISS2)`

Input data operand format is signed integer:

- When a multiply-accumulate is performed to a 32-bit register, the accumulator contents are scaled (left-shifted by one) when the results are copied to the destination register. If scaling produces a signed value that requires more than 32 bits to be properly represented, the number is saturated to its maximum positive or negative value.
- When a multiply-accumulate is performed to a 16-bit register half, the accumulator contents are scaled (as above) when copying the lower 16 bits to the destination half register. If scaling produces a signed value that requires more than 16 bits to be properly represented, the number is saturated to its maximum positive or negative value.

`(IH)`

This option indicates integer multiplication with high half-word extraction. The accumulator is saturated at 32 bits, and bits 31:16 of the accumulator are rounded and copied into the destination register half.

(W32)

Input data operand format is signed fractional with no extension bits in the accumulators at 32 bits. Left-shift correction of the product is performed, as required. This option is used for legacy GSM speech vocoder algorithms written for 32-bit accumulators. For this option only, this special case applies:

```
0x8000 x 0x8000 = 0x7FFF
```

(M)

Operation uses mixed-multiply mode. Valid only for MAC1 versions of the instruction. Multiplies a signed fractional operand by an unsigned fractional operand with no left-shift correction, where the first operand is signed and the second is unsigned. MAC0 performs an unmixed multiplication of signed fractional operands unless another format as specified (i.e., MAC0 executes the specified signed/signed or unsigned/unsigned multiplication). The (`M`) option can be used alone or be paired with another format option.

(NS)

Operation is non-saturating. When copying the accumulator contents to a destination register, the low-order bits are copied if the whole value will not fit in the destination. The (`NS`) option can only be used in conjunction with integer format options.

Barrel Shifter (Shifter)

The shifter provides bitwise shifting functions for 16-, 32-, or 40-bit inputs, yielding a 16-, 32-, or 40-bit output. These functions include arithmetic shift, logical shift, rotate, and various bit test, set, pack, unpack, and exponent detection functions. These shift functions can be combined to implement numerical format control, including full floating-point representation.

Shifter Operations

The shifter instructions (>>>, >>, <<, `ASHIFT`, `LSHIFT`, `ROT`) can be used various ways, depending on the underlying arithmetic requirements. The `ASHIFT` and >>> instructions represent an arithmetic shift, while the `LSHIFT`, <<, and >> instructions represent a logical shift.

The arithmetic shift and logical shift operations can be further broken into subsections. Instructions that are intended to operate on 16-bit single or paired numeric values (as would occur in many DSP algorithms) can use the instructions `ASHIFT` and `LSHIFT`. These are typically three-operand instructions.

Instructions that are intended to operate on a 32-bit register value and use two operands, such as those instructions frequently generated by the compiler, use the >>> and >> instructions.

Arithmetic shift, logical shift, and rotate instructions can obtain the shift argument from a register or directly from an immediate value in the instruction. For details about shifter instructions, see Shifter Instruction Summary.

Two-Operand Shifts

Two-operand shift instructions shift an input register and deposit the result into the same register.

Immediate Shifts

An immediate shift instruction shifts the input bit pattern to the right (downshift) or left (upshift) by a given number of bits. Immediate shift instructions use the data value in the instruction itself to control the magnitude and direction of the shift operation.

For example, consider the case where `R0` contains the value 0x0000_B6A3. A 4-bit downshift operation could be as follows:

```
R0 >>= 0x04 ;
```

The 4-bit downshifted result of 0x0000_0B6A is stored in `R0`.

If the same 0x0000_B6A3 value were 4-bit upshifted:

```
R0 <<= 0x04 ;
```

The 4-bit upshifted result of 0x000B_6A30 would be stored in `R0`.

Register Shifts

Register-based shifts use a register to hold the magnitude of the shift. The entire 32-bit register is used as the shift magnitude, and if this value exceeds 31, the shift result is 0.

For example, the following sequence performs a 4-bit upshift:

```
R0 = 0x0000B6A3 ;   // Load value to be shifted
R2 = 0x4 ;          // Set shift magnitude to 4
R0 <<= R2 ;         // Perform left shift by 4
```

As a result of this sequence, the 0x0000B6A3 value in `R0` is upshifted 4 bits and stored back to `R0` as 0x000B_6A30.

Three-Operand Shifts

Three-operand shifter instructions shift an input register and deposit the result into a destination register.

Immediate Shifts

Immediate shift instructions use the data value in the instruction itself to control the magnitude and direction of the shifting operation.

The following is an example of a 4-bit downshift applied to a 32-bit value:

```
R0 = 0x0000B6A3 ;       // Load value to be shifted
R1 = R0 >> 0x04 ;       // Perform right shift by 4
```

As a result of this sequence, the 32-bit 0x0000_B6A3 value in `R0` is right-shifted by four and stored to `R1` as 0x0000_0B6A.

Similarly, a 4-bit upshift applied to a 16-bit value could be implemented as follows:

```
R0.L = 0xB6A3 ;        // Load value to be shifted
R1.H = R0.L << 0x04 ;   // Perform left shift by 4
```

As a result of this sequence, the 16-bit 0xB6A3 value in `R0.L` is left-shifted by four and stored to `R1.H` as 0x6A30.

Register Shifts

Register-based shifts use a data register to hold the shift magnitude. For `ASHIFT`, `LSHIFT` and `ROT` instructions performing register-based shifts, the shift magnitude must be in the lowest six bits of a low data register half (`R[n].L`). The upper 10 bits of `R[n].L` are masked off and ignored.

The following is an example of a register-based 4-bit logical upshift:

```
R0   = 0x0000B6A3 ;         // Load value to be shifted
R2.L = 0x4 ;               // Load shift magnitude
R1   = LSHIFT R0 by R2.L ; // Perform shift
```

As the shift magnitude is positive, this sequence results in a logical left shift of the 0x0000_B6A3 input data by four, zero-filling the vacated lower four bits and storing the 0x000B_6A30 result to `R1`.

For a register-based 4-bit arithmetic downshift, an example sequence is:

```
R0   = 0xB6A30000 ;         // Load value to be shifted
R2.L = -0x4 ;              // Load shift magnitude
R1   = ASHIFT R0 by R2.L ; // Perform shift
```

As the shift magnitude is negative, this sequence results in an arithmetic right shift of the 0xB6A3_0000 input data by four, sign-extending through the vacated upper four bits and storing the 0xFB6A_3000 result to `R1`.

The `ROT` instruction uses the shifter to shift the input operand through the Arithmetic Status register's Condition Code bit (`ASTAT.CC`). When the shift is performed, the `ASTAT.CC` bit in inserted into the data between bit 0 and bit 31 of the original data. For example:

```
R0   = 0xABCDEF12 ;         // Load value to rotate through CC
R2.L = 0x4 ;               // Set rotate magnitude
R1   = ROT R0 by R2.L ;    // Perform rotation
```

Assuming that `ASTAT.CC` was 0 entering the above sequence, the positive magnitude in `R2` results in a 4-bit left rotation being applied to the 0xABCD_EF12 input data, with the `ASTAT.CC` bit value of 0 appearing between bit 0 and bit 31 of the input data. The resulting data pattern of 0xBCDE_F125 is stored to `R1`. Note that the `ASTAT.CC` bit is included in the result at bit 3, followed by the b#101 from bits 31:29 of the input data. The input data bit 28 (which is 0) is now in `ASTAT.CC`.

Bit Test, Set, Clear, and Toggle

The shifter provides the method to test, set, clear, and toggle specific bits of a data register. All instructions have two arguments, the source register and the bit location. While the set, clear, and toggle instructions modify the value in the source data register, the test instruction does not change it; rather, it impacts the `ASTAT.CC` bit.

The following examples show a variety of operations.

```
BITCLR ( R0, 6 ) ;        // Clears bit 6 of R0
```

```
BITSET ( R2, 9 ) ;        // Sets bit 9 of R2
BITTGL ( R3, 2 ) ;        // Toggles bit 2 of R3
CC = BITTST ( R3, 0 ) ;  // Puts bit 0 of R3 in ASTAT.CC
```

When programming, header files containing `#define` statements provide constant definitions for specific bits in memory-mapped registers. It is important to examine the definition techniques used in these header files because the constant definitions do not contain the position of the bit; rather, these header files define bit masks. A constant definition in a header file working with bit masks might be set to 0x20 to describe bit five of a register. The `BITPOS` macro provided by the Blackfin processor assembler helps when working with bit mask definitions and bit manipulation instructions. For example, the following assembly code uses a `BITPOS` macro with a `BITTST` instruction:

```
#define BITFIVE 0x20
CC = BITTST ( R5, BITPOS ( BITFIVE ) ) ;
```

The `BITPOS` macro parses the `BITFIVE` definition at program build-time, identifying the lowermost set bit to be the fifth bit and changing the instruction passed to the assembler to:

```
CC = BITTST ( R5, 5 ) ;
```

This will result in the `ASTAT.CC` bit being set to the value of bit 5 of the `R5` register. For detailed information about `BITPOS`, see the *CrossCore Embedded Studio Assembler and Preprocessor Manual*.

Field Extract and Field Deposit

A bit-field with a width ranging from 1- to 16-bit may be extracted from or deposited to anywhere within a 32-bit data element using the `EXTRACT` and `DEPOSIT` instructions, respectively. Two register arguments are associated with these instructions, the first containing the 32-bit source data for the operation and the second containing field calibration and, specific to the `DEPOSIT` instruction, the data needed for the instruction to perform its function.

Both the `EXTRACT` and `DEPOSIT` instructions use the data contained in the first `R[n]` data register argument and apply the calibration details defined in the second `R[n]` data register argument to it before placing the result in a 32-bit `R[n]` destination data register (without modifying either of the argument registers). The first argument for both instructions is always a 32-bit `R[n]` data register; however, the second argument is not consistent:

- `DEPOSIT` - the upper half of the calibration register contains the data that needs to be deposited into the source data contained in the first argument before storing the result to the destination register; therefore, the second argument must be a full 32-bit `R[n]` data register.
- `EXTRACT` - as the extract operation doesn't require data to be changed in the source data in the first argument, the upper half of the calibration register is meaningless; therefore, the second argument must be the lower half of a `R[n].L` data register.

For example, consider the scenario where `R0` is the data argument and `R1` is the calibration argument:

- `R0` = 0xAABBCCDD
- `R1` = 0x0333100C

R0 contains the source data for the operation. Some field within this register will be the source or destination of the operation prior to the result being transferred to the instruction's 32-bit destination register, as governed by the R1 calibration register, which is structured as follows:

- R1[7:0] - length of the bit-field of interest. In this case, the field is 12 bits wide (0x0C).
- R1[15:8] - bit location where the field of interest begins. In this case, the 12-bit field begins in the R0 source data register at bit 16 (0x10), thus defining the field of interest to be R0[27:16].
- R1[31:16] - up to a 16-bit right-justified data field, required by the DEPOSIT operation (0x0333).

The EXTRACT instruction simply reads the field of interest from the source data and places it into the destination register with a mandatory zero-extension ((Z)) or sign-extension ((X)) applied to it. One of these qualifiers must always be associated with the operation, as follows:

```
R3 = EXTRACT ( R0 , R1.L ) ( Z ) ;  // R3 = 0x00000ABB
R3 = EXTRACT ( R0 , R1.L ) ( X ) ;  // R3 = 0xFFFFFABB
```

As described, the syntax includes the 32-bit destination (R3), the 32-bit source data argument (R0), and the 16-bit calibration argument (R1.L). In both cases, the 12-bit 0xABB bit-field of interest contained in R0[27:16] is read from the R0 source register and is then either zero-extended (0x00000ABB) or sign-extended (0xFFFFFABB) before being stored to the R3 destination register.

The DEPOSIT instruction takes the R0 source register data and replaces the field of interest defined by the lower portion of the calibration register (R1[15:0]) with the data in the upper portion of the calibration register (R1[31:16]) before storing the result into the destination register. There is both a non-extending and a sign-extending version of the instruction, as follows:

```
R3 = DEPOSIT ( R0 , R1 ) ;         // R3 = 0xA333CCDD
R3 = DEPOSIT ( R0 , R1 ) ( X ) ;   // R3 = 0x0333CCDD
```

As described, the syntax includes the 32-bit destination (R3), the 32-bit source data argument (R0), and the 32-bit calibration argument (R1). In both cases, the 12-bit 0x333 bit-field defined in the upper half of the R1 calibration register replaces the bit-field of interest in R0[27:16]. The field is deposited in place without extension (0xA333CCDD) or zero-extended (0x0333CCDD) before being stored to the R3 destination register.

Packing Operation

The shifter also supports a series of packing and unpacking instructions. Consider the case where:

- R0 contains 0x11223344
- R1 contains 0x55667788

Packing operations return:

```
R2 = PACK(R0.L, R0.H); /* R2 = 0x33441122 */
R3 = PACK(R1.L, R0.H); /* R3 = 0x77881122 */
R4 = BYTEPACK(R0, R1); /* R4 = 0x66882244 */
```

The BYTEUNPACK instruction is silently controlled by the *Ix* registers. For example:

```
(R6, R7) = BYTEUNPACK R1:0;
```

The value of the I0 register determines what is returned to the R6 and R7 destination registers, as follows:

- I0 = 0: R6 = 0x00110022, R7 = 0x00330044
- I0 = 1: R6 = 0x00880011, R7 = 0x00220033
- I0 = 2: R6 = 0x00770088, R7 = 0x00110022
- I0 = 3: R6 = 0x00660077, R7 = 0x00880011

For more details, see the Spread 8-Bit to 16-Bit (ByteUnPack) instruction description and the Pack 8-Bit to 32-Bit (BytePack) instruction description.

Shifter Instruction Summary

The Table 2-11 Shifter Instructions and Status table lists the shifter instructions and how they affect the ASTAT register status bits. In the table, note the meaning of these symbols:

- Dreg denotes any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- *indicates the status bit may be set or cleared, depending on the results of the instruction.
- * 0 indicates versions of the instruction that send results to accumulator A0 set or clear AV0.
- * 1 indicates versions of the instruction that send results to accumulator A1 set or clear AV1.
- ** indicates the status bit is cleared.
- *** indicates ASTAT.CC contains the latest value shifted into it.
- - indicates no effect.

Table 2-11: Shifter Instructions and Status

	ASTAT Status Bits						
Instruction	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
BITCLR (Dreg, uimm5) ;	*	*	**	-	-	-	**/-
BITSET (Dreg, uimm5) ;	**	*	**	-	-	-	**/-
BITTGL (Dreg, uimm5) ;	*	*	**	-	-	-	**/-
CC = BITTST (Dreg, uimm5) ;	-	-	-	-	-	*	-

Table 2-11: Shifter Instructions and Status (Continued)

	ASTAT Status Bits						
Instruction	**AZ**	**AN**	**AC0 AC0_COPY AC1**	**AV0 AV0S**	**AV1 AV1S**	**CC**	**V V_COPY VS**
CC = !BITTST (Dreg, uimm5) ;	-	-	-	-	-	*	-
Dreg = DEPOSIT (Dreg, Dreg) ;	*	*	**	-	-	-	**/-
Dreg = EXTRACT (Dreg, Dreg) ;	*	*	**	-	-	-	**/-
BITMUX (Dreg, Dreg, A0) ;	-	-	-	-	-	-	-
Dreg_lo = ONES Dreg ;	-	-	-	-	-	-	-
Dreg = PACK (Dreg_lo_hi, Dreg_lo_hi);	-	-	-	-	-	-	-
Dreg >>>= uimm5 ;	*	*	-	-	-	-	**/-
Dreg >>= uimm5 ;	*	*	-	-	-	-	**/-
Dreg <<= uimm5 ;	*	*	-	-	-	-	**/-
Dreg = Dreg >>> uimm5 ;	*	*	-	-	-	-	**/-
Dreg = Dreg >> uimm5 ;	*	*	-	-	-	-	**/-
Dreg = Dreg << uimm5 ;	*	*	-	-	-	-	*
Dreg = Dreg >>> uimm4 (V) ;	*	*	-	-	-	-	**/-
Dreg = Dreg >> uimm4 (V) ;	*	*	-	-	-	-	**/-
Dreg = Dreg << uimm4 (V) ;	*	*	-	-	-	-	*
A*x* = A*x* >>> uimm5 ;	*	*	-	** 0/-	** 1/-	-	-
A*x* = A*x* >> uimm5 ;	*	*	-	** 0/-	** 1/-	-	-
A*x* = A*x* << uimm5 ;	*	*	-	* 0	* 1	-	-
Dreg_lo_hi = Dreg_lo_hi >>> uimm4 ;	*	*	-	-	-	-	**/-
Dreg_lo_hi = Dreg_lo_hi >> uimm4 ;	*	*	-	-	-	-	**/-
Dreg_lo_hi = Dreg_lo_hi << uimm4 ;	*	*	-	-	-	-	*
Dreg >>>= Dreg ;	*	*	-	-	-	-	**/-

Table 2-11: Shifter Instructions and Status (Continued)

	ASTAT Status Bits						
Instruction	AZ	AN	AC0 AC0_COPY AC1	AV0 AV0S	AV1 AV1S	CC	V V_COPY VS
Dreg >>= Dreg ;	*	*	-	-	-	-	**/-
Dreg <<= Dreg ;	*	*	-	-	-	-	**/-
Dreg = ASHIFT Dreg BY Dreg_lo ;	*	*	-	-	-	-	*
Dreg = LSHIFT Dreg BY Dreg_lo ;	*	*	-	-	-	-	**/-
Dreg = ROT Dreg BY imm6 ;	-	-	-	-	-	***	-
Dreg = ASHIFT Dreg BY Dreg_lo (V) ;	*	*	-	-	-	-	*
Dreg = LSHIFT Dreg BY Dreg_lo (V) ;	*	*	-	-	-	-	**/-
Dreg_lo_hi = ASHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	-	-	-	-	*
Dreg_lo_hi = LSHIFT Dreg_lo_hi BY Dreg_lo ;	*	*	-	-	-	-	**/-
A*x* = A*x* ASHIFT BY Dreg _lo ;	*	*	-	* 0	* 1	-	-
A*x* = A*x* ROT BY imm6 ;	-	-	-	-	-	***	-
Dreg = (Dreg + Dreg) << 1 ;	*	*	*	-	-	-	*
Dreg = (Dreg + Dreg) << 2 ;	*	*	*	-	-	-	*

ADSP-BF70x Computational Unit Register Descriptions

The Computational Unit Register File contains the following registers.

Table 2-12: ADSP-BF70x Computational Unit Register List

Name	Description
`R[n]`	Data Registers
`A0X`	Accumulator 0 Extension Register
`A0`	Accumulator 0 Register
`A1X`	Accumulator 1 Extension Register
`A1`	Accumulator 1 Register

Table 2-12: ADSP-BF70x Computational Unit Register List (Continued)

Name	Description
ASTAT	Arithmetic Status Register

Data Registers

There are eight 32-bit R[n] data registers for use in computations and data moves. Each may be accessed as a 32-bit entity or as a pair of independent 16-bit registers, denoted as the low register half (R[n].L) or the high register half (R[n].H). See the appropriate instruction reference pages for details.

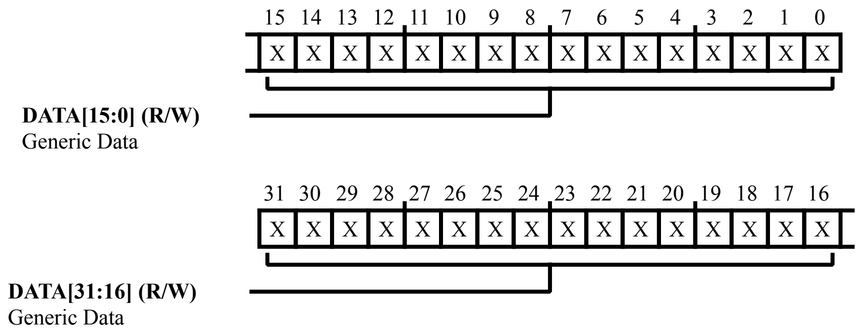


Figure 2-14: R[n] Register Diagram

Table 2-13: R[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Generic Data. The R[n].DATA bit fields, whether the low half (bits 15:0), high half (bits 31:16), or full register (bits 31:0), hold data for arithmetic operations or data moves.

Accumulator 0 Register

The processor has two dedicated, 40-bit accumulator registers, A0 and A1. A0 may be accessed via its 16-bit low half (A0.L), its 16-bit high half (A0.H), or its 8-bit extension (A0.X) register (see the associated A0X register documentation for details). A0 can also be accessed as a 32-bit register (A0.W), which extracts the lower 32 bits of the accumulator.

The A0 and A1 accumulator registers may be combined to hold an 80-bit complex result or a 72-bit fixed-point result. The combined accumulator register is defined to be A1:0, where the least significant 32 bits are in A0.W, the next 32 bits are in A1.W, and the 8-bit overflow from a 32-bit fixed-point operation or the most significant eight bits from a complex operation are in A1.X (see the associated A1X register documentation for details).

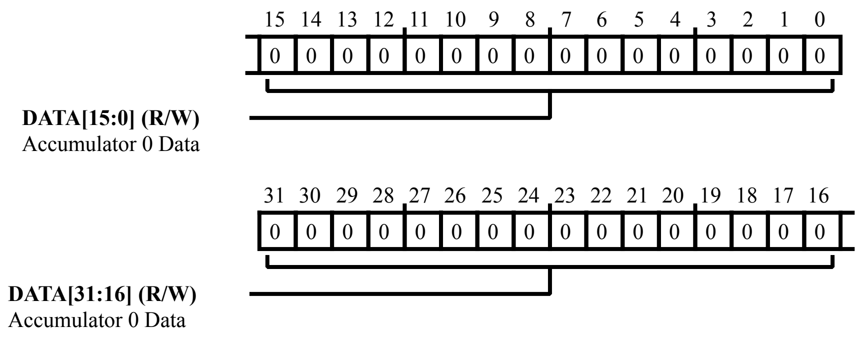


Figure 2-15: A0 Register Diagram

Table 2-14: A0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Accumulator 0 Data. The A0.DATA bits hold accumulator 0 data.

Accumulator 1 Register

The processor has two dedicated, 40-bit accumulator registers, A0 and A1. A1 may be accessed via its 16-bit low half (A1.L), its 16-bit high half (A1.H), or its 8-bit extension (A1X) register (see the associated Accumulator 1 Extension register documentation for details). A1 can also be accessed as a 32-bit register (A1.W), which extracts the lower 32 bits of the accumulator.

The accumulator registers may be combined to hold an 80-bit complex result or a 72-bit fixed-point result. The combined accumulator register is defined to be A1:0, where the least significant 32 bits are in A0.W, the next 32 bits are in A1.W, and the 8-bit overflow from a 32-bit fixed-point operation or the most significant eight bits from a complex operation are in A1X.

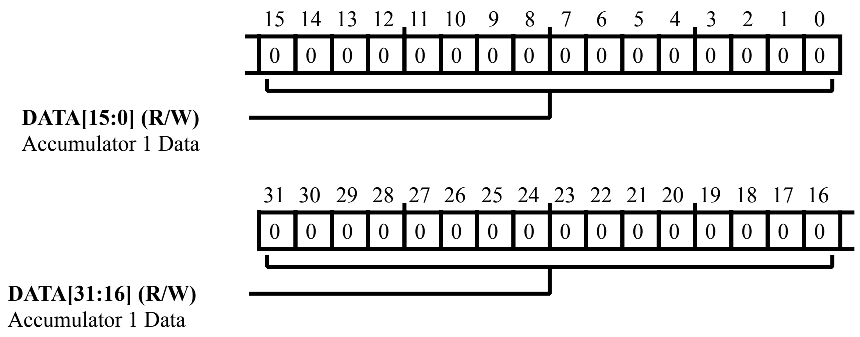


Figure 2-16: A1 Register Diagram

Table 2-15: A1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	DATA	Accumulator 1 Data. The A1.DATA bits hold accumulator 1 data.

Accumulator 0 Extension Register

For 16-bit MAC0 and 32-bit ALU0 operations, the `A0X` register contains eight bits of overflow information from the operation. When the `A0` register is used in combination with the `A1` register to form an 80-bit accumulator A1:0, the `A0X` register is not used.

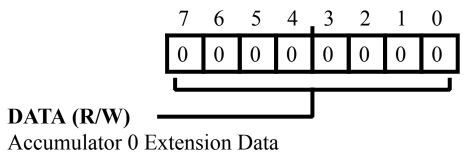


Figure 2-17: A0X Register Diagram

Table 2-16: A0X Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	DATA	Accumulator 0 Extension Data. The `A0X.DATA` bits hold overflow data for results from computation unit 0.

Accumulator 1 Extension Register

For 16-bit MAC1 and 32-bit ALU1 operations, the A1.X register contains eight bits of overflow information from the operation. When the A1 register is used in combination with the A0 register to form an 80-bit accumulator A1:0, the A1.X register contains eight bits of overflow information for 72-bit fixed-point results or the most significant eight bits of an 80-bit complex result.

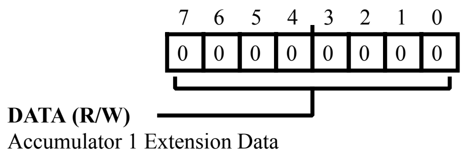


Figure 2-18: A1X Register Diagram

Table 2-17: A1X Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	DATA	Accumulator 1 Extension Data. The `A1X.DATA` bits hold overflow data for results from computation unit 1 or the most significant byte of 80-bit complex results.

Arithmetic Status Register

The ASTAT register contains both status bits and control bits. Status bits are updated by the processor for each ALU, MAC, or shifter instruction executed, as documented in the respective Instruction Summary tables in each of the ALU, MAC, and Barrel Shifter sections of the Computational Units chapter, as well as in many of the specific Instruction Reference Pages.

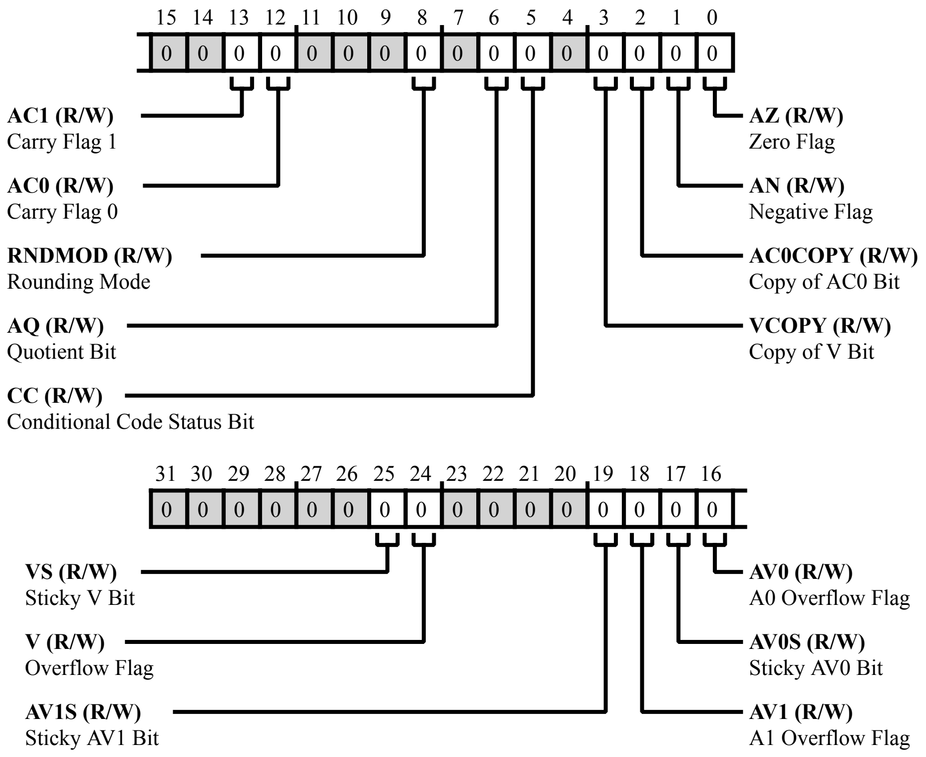


Figure 2-19: ASTAT Register Diagram

Table 2-18: ASTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
25 (R/W)	VS	Sticky V Bit. Set when `ASTAT.V` is set. Remains set until specifically cleared by software.
24 (R/W)	V	Overflow Flag. Set when the most recent ALU0 or MAC0 operation result targeting a non-accumulator register overflowed, otherwise it is cleared.
19 (R/W)	AV1S	Sticky AV1 Bit. Set when `ASTAT.AV1` is set. Remains set until explicitly cleared by software.
18 (R/W)	AV1	A1 Overflow Flag.

Table 2-18: ASTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
		Set when the most recent ALU1 or MAC1 operation result targeting `A1` overflowed, otherwise it is cleared.
17 (R/W)	AV0S	Sticky AV0 Bit. Set when `ASTAT.AV0` is set. Remains set until explicitly cleared by software.
16 (R/W)	AV0	A0 Overflow Flag. Set when the most recent ALU0 or MAC0 operation result targeting `A0` overflowed, otherwise it is cleared.
13 (R/W)	AC1	Carry Flag 1. Set when the most recent ALU1 operation targeting `A1` generated a carry, otherwise it is cleared.
12 (R/W)	AC0	Carry Flag 0. Set when the most recent ALU0 operation targeting `A0` generated a carry, otherwise it is cleared.
8 (R/W)	RNDMOD	Rounding Mode. The `ASTAT.RNDMOD` bit is a control bit used to select the rounding mode for arithmetic instructions that support rounding.
		0 Unbiased rounding
		1 Biased rounding
6 (R/W)	AQ	Quotient Bit. The `ASTAT.AQ` bit is the XOR of the 32-bit dividend MSB with the 16-bit divisor MSB. This bit is affected by only the DIVS and DIVQ instructions.
5 (R/W)	CC	Conditional Code Status Bit. The `ASTAT.CC` status bit is set or cleared based on how it is used in assembly instructions and is used for conditional program sequencing. See the Conditional Code Status Bit section of the Program Sequencer chapter for details.
3 (R/W)	VCOPY	Copy of V Bit. Set if `ASTAT.V` is set, and cleared if `ASTAT.V` is cleared.
2 (R/W)	AC0COPY	Copy of AC0 Bit. Set if `ASTAT.AC0` is set, and cleared if `ASTAT.AC0` is cleared.
1 (R/W)	AN	Negative Flag. Set when the most recent ALU or shifter operation result was negative, otherwise it is cleared.
0 (R/W)	AZ	Zero Flag. Set when the most recent ALU or shifter operation result was zero, otherwise it is cleared.

3 Operating Modes and States

The processor supports the following three processor modes:

- User mode
- Supervisor mode
- Emulation mode

Emulation and Supervisor modes have unrestricted access to the core resources. User mode has restricted access to certain system resources, thus providing a protected software environment.

User mode is considered the domain of application programs. Supervisor mode and Emulation mode are usually reserved for the kernel code of an operating system. The processor mode is determined by the Supervisor Access (`SACC`) bit of the `SYSCFG` register and the Event Controller. When the `SACC` bit is set, or when servicing an interrupt, a nonmaskable interrupt (NMI), or an exception, the processor is in Supervisor mode. When servicing an emulation event, the processor is in Emulation mode. When not servicing any events and the `SACC` bit is cleared, the processor is in User mode.

The current processor mode may be identified by interrogating the `IPEND` memory-mapped register (MMR) and the `SACC` bit of the `SYSCFG` register, as shown in the Identifying the ***Current Processor Mode*** table.

Table 3-1: Identifying the Current Processor Mode

Event	Mode	IPEND	SACC Bit
Interrupt	Supervisor	0x10 The core is processing an interrupt event if `IPEND[0]`, `IPEND[1]`, `IPEND[2]`, and `IPEND[3] = 0`.	x
Exception	Supervisor	0x08 The core is processing an exception event if `IPEND[0] = 0`, `IPEND[1] = 0`, `IPEND[2] = 0`,	x

Table 3-1: Identifying the Current Processor Mode (Continued)

Event	Mode	IPEND	SACC Bit
		`IPEND[3] = 1`, and `IPEND[15:4]` are 0s or 1s.	
NMI	Supervisor	0x04 The core is processing an NMI event if `IPEND[0] = 0`, `IPEND[1] = 0`, `IPEND[2] = 1`, and `IPEND[15:2]` are 0s or 1s.	x
Reset	Supervisor	= 0x02 As the reset state is exited, `IPEND` is set to 0x02, and the reset vector runs in Supervisor mode.	x
Emulation	Emulator	= 0x01 The processor is in Emulation mode if `IPEND[0] = 1`, regardless of the state of the `IPEND[15:1]` bits.	x
None None	Supervisor User	= 0x00 = 0x00	1 0

In addition, the processor supports the following two non-processing states:

- Idle state
- Reset state

The ***Processor Modes and States*** figure illustrates the processor modes and states as well as the transition conditions between them when the `SACC` and `STRICT` bits in the `SYSCFG` register are in their default (zero) state.

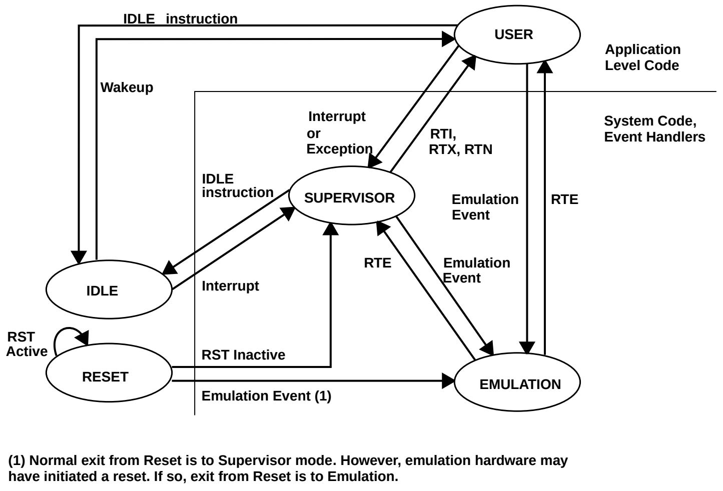


Figure 3-1: Processor Modes and States

User Mode

The processor is in User mode when it is not in Reset or Idle state, when it is not servicing an interrupt, NMI, exception, or emulation event, and when the `SACC` bit of the `SYSCFG` register is zero. User mode is used to process application level code that does not require explicit access to system registers. Any attempt to access restricted system registers causes an exception event. The ***Registers Accessible in User Mode*** table lists the registers that may be accessed in User mode.

Table 3-2: Registers Accessible in User Mode

Processor Registers	*Register Names*
Data Registers	`R[7:0], A[1:0]`
Pointer and DAG Registers	`P[5:0], SP, FP, I[3:0], M[3:0], L[3:0], B[3:0]`
Sequencer and Status Registers	`RETS, LC[1:0], LT[1:0], LB[1:0], ASTAT, CYCLES, CYCLES2`

Protected Resources and Instructions

System resources consist of a subset of processor registers, all MMRs, and a set of protected instructions.

The system and core MMRs are located in a reserved region of memory which is protected from User mode access. Any attempt to access MMR space in User mode causes an exception. Refer to the specific Blackfin+ Processor Hardware Reference for the location of the MMR region in your system.

A list of protected instructions appears in the ***Protected Instructions*** table. Any attempt to issue any of the protected instructions from User mode causes an exception event.

The Strict Supervisor Access (`STRICT`) bit in the `SYSCFG` register causes the `IDLE` instruction to be a protected instruction. When the `STRICT` bit is set, an attempt to issue the `IDLE` instruction from User mode causes an exception event.

Table 3-3: Protected Instructions

Instruction	Description
RTI	Return from Interrupt
RTX	Return from Exception
RTN	Return from NMI
CLI	Disable Interrupts
STI	Enable Interrupts
RAISE	Force Interrupt/Reset
STI IDLE	Enable Interrupts and Idle
IDLE	Idle Causes an exception only if the `SYSCFG.STRICT` bit is set.
RTE	Return from Emulation Always causes an exception if executed outside of Emulation mode.

Protected Memory

Additional memory locations can be protected from User mode access. A Cacheability Protection Lookaside Buffer (CPLB) entry can be created and enabled. See ***Memory Management Unit*** in the ***Memory*** chapter for further information.

Entering User Mode

When coming out of reset, the processor is in Supervisor mode because it is servicing a reset event. To enter User mode from the Reset state, two steps must be performed:

1. A return address must be loaded into the `RETI` register.
2. An `RTI` instruction must be executed.

The `SACC` bit of the `SYSCFG` register is zero at reset, so this value does not need to be changed to enter User mode upon executing the above `RTI` instruction. The following example code shows how to enter User mode upon reset.

Example Code to Enter User Mode Upon Reset

The ***Entering User Mode from Reset*** example provides code for entering User mode from reset:

```
/* Entering User Mode from Reset */
    P1.L = lo(START) ;   /* Point to start of user code */
    P1.H = hi(START) ;
    RETI = P1 ;
    RTI ;    /* Return from Reset Event */
START :  /* Place user code here */
```

Return Instructions That Invoke User Mode

The ***Return Instructions That Can Invoke User Mode*** table provides a summary of return instructions that can be used to invoke User mode from various processor event service routines. When these instructions are used in service routines, the value of the return address must first be stored to the appropriate event register (`RETx`). For interrupt service routines, the return address (`RETI`) can be stored to the stack if the service routine itself is interruptible, which enables interrupt nesting. For this case, the address must be popped from the stack into `RETI` prior to executing the `RTI` instruction.

NOTE: Return instruction will only cause the processor to enter User mode if the `SACC` bit of the `SYSCFG` register is set to zero. When this bit is set to one, the processor remains in Supervisor mode after all event handlers have been exited.

The processor remains in User mode until one of these events occurs:

- An interrupt, NMI, or exception event invokes Supervisor mode.
- An emulation event invokes Emulation mode.
- A reset event invokes the Reset state.

Table 3-4: Return Instructions That Can Invoke User Mode

Current Process Activity	Return Instruction to Use	Execution Resumes at Address in This Register
Interrupt Service Routine	RTI	RETI
Exception Service Routine	RTX	RETX
Non-Maskable Interrupt Service Routine	RTN	RETN
Emulation Service Routine	RTE	RETE

Supervisor Mode

Supervisor mode has full, unrestricted access to all processor system resources, including all emulation resources, unless a CPLB has been configured to prevent it and is enabled. See ***Memory Management Unit*** in the ***Memory*** chapter for further details.

The processor services all interrupt, NMI, and exception events in Supervisor mode and remains in Supervisor mode on return from all event handlers if the `SACC` bit of the `SYSCFG` register is set to one.

The stack pointer referenced by the `SP` register alias and modified by stack push/pop instructions is the Supervisor Stack Pointer while an event is being serviced (`IPEND` is non-zero), and the User Stack Pointer when executing at task level (`IPEND` is zero). This is the case irrespective of whether the processor is running in Supervisor or User mode.

Only Supervisor mode can use the register alias `USP`, which always references the User Stack Pointer. There is no unique alias for the Supervisor Stack Pointer, so this register can only be be referenced within an event handler using the general stack pointer alias, `SP`.

Normal processing begins in Supervisor mode from the Reset state. The processor transitions from the Reset state to Supervisor mode, servicing the reset event, where it remains until an emulation event or return instruction occurs to change the mode. Before the return instruction is issued, the `RETI` register must be loaded with a valid return address.

Non-OS Environments

For non-OS environments, application code should remain in Supervisor mode so that it can access all core and system resources. On leaving the Reset state, the processor initiates operation by servicing the reset event. Emulation is the only event that can preempt this activity; therefore, lower priority events cannot be processed.

The simplest method of keeping the processor in Supervisor mode and allowing lower priority events to be processed is to set the `SACC` bit of the `SYSCFG` register before returning from the reset event with an `RTI` instruction. Prior to executing the `RTI` instruction, `RETI` must be loaded with the address of the code to be executed after leaving all event handlers.

Earlier Blackfin processors did not have a `SACC` bit, so it was necessary to execute all code in the lowest priority interrupt (`IVG15`). Events and interrupts are described further in the ***Events and Interrupts*** section of the ***Program Sequencer*** chapter. The interrupt handler for `IVG15` is set to the application code's starting address, and then the low-priority interrupt is forced using the `RAISE 15` instruction. The `IVG15` interrupt is not serviced until the return from the reset event and any pending interupts with intermediate priorities have been serviced. Therefore, before executing the `RTI` instruction to return from the reset event, `RETI` is loaded with the address of a loop that executes in User mode until the `IVG15` interrupt is serviced.

Example Code for Supervisor Mode Coming Out of Reset

To remain in Supervisor mode when coming out of the Reset state, use code as shown in the ***Staying in Supervisor Mode Coming Out of Reset*** example.

```
/* Staying in Supervisor Mode Coming Out of Reset */
    R0 = SYSCFG ;
    BITSET (R0, BITP_SYSCFG_SACC) ;
    SYSCFG = R0 ; /* Set SACC bit */
    RETI = START ; /* Set return address to START */
    RTI ; /* Return from Reset Event */

START:
    /* Task level code executes in Supervisor mode */
```

Code written for older Blackfin processors must remain at the lowest interrupt level (`IVG15`) in order to stay in Supervisor mode as shown in the***Staying in Supervisor Mode Coming Out of Reset (Legacy)*** example.

```
/* Staying in Supervisor Mode Coming Out of Reset (Legacy) */
    P0.L = lo(EVT15) ; /* Point to IVG15 in Event Vector Table */
    P0.H = hi(EVT15) ;
    P1.L = lo(START) ;   /* Point to start of User code */
    P1.H = hi(START) ;
    [P0] = P1 ; /* Place the address of START in IVG15 of EVT */
    P0.L = lo(IMASK) ;
    R0 = [P0] ;
    R1.L = lo(EVT_IVG15) ;
    R0 = R0 | R1 ;
    [P0] = R0 ; /* Set (enable)IVG15 bit in IMASK register */
    RAISE 15 ;   /* Invoke IVG15 interrupt */
    P0.L = lo(WAIT_HERE) ;
    P0.H = hi(WAIT_HERE) ;
    RETI = P0 ;   /* RETI loaded with return address */
    RTI ;   /* Return from Reset Event */

WAIT_HERE :  /* Execute in User mode till IVG15 is serviced */
    JUMP WAIT_HERE ;

START:   /* IVG15 vectors here */
    /* Clears IPEND bit 4 to enable interrupts globally. */
    [--SP] = RETI ;
```

Emulation Mode

The processor enters Emulation mode if Emulation mode is enabled and either of these conditions is met:

- An external emulation event occurs.
- The EMUEXCPT instruction is issued.

The processor remains in Emulation mode until the emulation service routine executes an RTE instruction. If the SACC bit of the SYSCFG register is zero and no interrupts are pending when the RTE instruction executes, the processor switches to User mode. Otherwise, the processor switches to Supervisor mode.

Idle State

Idle state stops all processor activity at the user's discretion, usually to conserve power during lulls in activity. No processing occurs during the Idle state. The Idle state is invoked by an IDLE instruction or an STI IDLE instruction. The IDLE instruction notifies the processor hardware that the Idle state is requested, whereas STI IDLE also enables interrupts in a manner that avoids race conditions.

The processor remains in the Idle state until a peripheral or external device, such as a SPORT or the Real-Time Clock (RTC), generates an interrupt that requires servicing.

In Example Code for Transition to Idle State, core interrupts are disabled before the device intended to wake the core from Idle is programmed, and the STI IDLE instruction is executed. When all the pending processes have completed, the core re-enables interrupts and disables its clocks. The use of the combined STI IDLE instruction to enter Idle state and enable interrupts ensures that any interrupt will bring the core out of the Idle state and terminate the idle instruction, rather than interrupting before the idle instruction has begun execution.

Example Code for Transition to Idle State

To transition to the Idle state, use the code shown in the ***Transitioning to Idle State*** example:

```
/* Transitioning to Idle State */
CLI R0 ;                        /* disable interrupts */

/* program wakeup device */

BITSET (R0, BITP_IMASK_IVG11) ; /* ensure device can interrupt */
STI IDLE R0 ;                   /* drain pipeline, enter Idle, enable interrupts */
```

Reset State

Reset state initializes the core logic. During Reset state, application programs and the operating system do not execute, and clocks are stopped.

The core remains in the Reset state as long as system logic asserts the $\overline{\text{RESET}}$ signal. Upon deassertion, the core completes the reset sequence and switches to Supervisor mode with event system priority 1, where it executes code found at an address supplied by the system. Refer to theReset Control Unit (RCU) chapter in the Hardware Reference Manual for details.

The only method to enter the Reset state is by recieving a $\overline{\text{RESET}}$ signal from the system. The `RAISE 1` instruction will execute the code addressed by `EVT1` at event system priority 1, but it does not actually reset the core. In both cases, an `RTI` instruction will exit the priority 1 event. In neither case is a return address automatically saved in `RETI`, so the register must be explicitly loaded within the event handler prior to executing the `RTI` instruction.

The ***Core State Upon Reset*** table summarizes the state of the core upon reset.

Table 3-5: Core State Upon Reset

Item	Description of Reset State
Operating Mode	Supervisor mode in reset event, clocks stopped
Rounding Mode	Unbiased rounding
Cycle Counters	Disabled, zero
DAG Registers (I, L, B, M)	Random values (must be cleared at initialization)
Data and Address Registers	Random values (must be cleared at initialization)
IPEND, IMASK, ILAT	Cleared, interrupts globally disabled with IPEND bit 4
CPLBs	Disabled
L1 Instruction Memory	SRAM (cache disabled)
L1 Data Memory	SRAM (cache disabled)
Cache Validity Bits	Invalid

System Reset and Power Up

For processor-specific system reset and power up information, see the Processor Hardware Reference manual.

ADSP-BF70x Mode-Related Register Descriptions

The Mode-Related Register File contains the following registers.

Table 3-6: ADSP-BF70x REGFILE Register List

Name	Description
`SYSCFG`	System Configuration Register

System Configuration Register

The SYSCFG register controls the configuration of the processor. This register is accessible only from Supervisor mode.

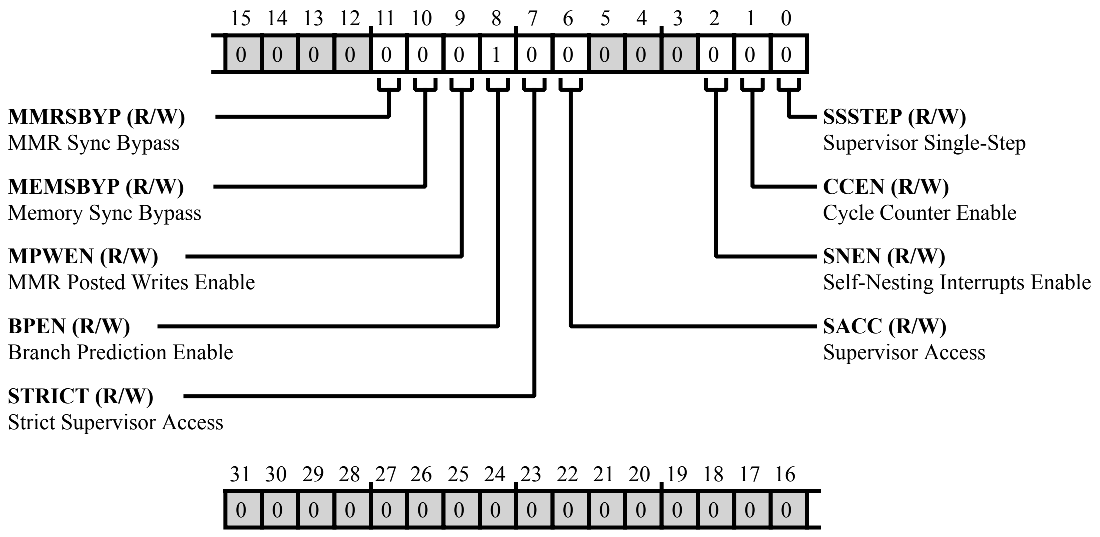


Figure 3-2: SYSCFG Register Diagram

Table 3-7: SYSCFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
11 (R/W)	MMRSBYP	MMR Sync Bypass. The SYSCFG.MMRSBYP bit enables bypass mode for clock domain synchronization between the core and the Memory-Mapped Register interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass
10 (R/W)	MEMSBYP	Memory Sync Bypass. The SYSCFG.MEMSBYP bit enables bypass mode for clock domain synchronization between the core and the System Memory Bus interface. Enabling this feature reduces read latency.	
		0	No bypass
		1	Bypass

Table 3-7: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
9 (R/W)	MPWEN	MMR Posted Writes Enable. The `SYSCFG.MPWEN` bit enables support for posting consecutive system MMR writes to the system fabric. When disabled, the processor waits for each response before allowing a new write to go into the system. When modifying this bit, an `SSYNC` instruction must then be executed to allow the system to finish any outstanding writes before changing the MMR write policy.	
		0	Disable
		1	Enable
8 (R/W)	BPEN	Branch Prediction Enable. The `SYSCFG.BPEN` bit selects whether the program sequencer uses dynamic or static branch prediction operation.	
		0	Use static prediction
		1	Use dynamic prediction
7 (R/W)	STRICT	Strict Supervisor Access. The `SYSCFG.STRICT` bit restricts additional resources to require Supervisor mode access. When enabled, accessing any of these additional resources in User mode (e.g., executing an `IDLE` instruction) causes an Illegal Supervisor Access exception.	
		0	Disable (normal/previous Blackfin access operation)
		1	Enable (strict supervisor access operation)
6 (R/W)	SACC	Supervisor Access. The `SYSCFG.SACC` bit selects whether or not Supervisor mode access is permitted when the processor is not servicing any events.	
		0	Disable (access only when servicing an event)
		1	Enable (access whether or not servicing any events)
2 (R/W)	SNEN	Self-Nesting Interrupts Enable. The `SYSCFG.SNEN` bit enables self-nesting interrupt operation. While nesting allows a lower-priority interrupt handler to be interrupted by higher-priority interrupts, self-nesting allows for interrupts of the same priority to also be responded to immediately as they occur in the system.	
		0	Disable (normal interrupt operation)
		1	Enable (self-nesting interrupt operation)

Table 3-7: SYSCFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
1 (R/W)	CCEN	Cycle Counter Enable. The SYSCFG.CCEN bit enables cycle counter operation. When the cycle counter is enabled, it is incremented every core clock (CCLK) cycle (including wait states) in both User and Supervisor modes. The cycle counter stops counting while the processor is in Emulator mode. The 64-bit cycle counter is comprised of two registers, with the least significant 32 bits in the CYCLES register and the most significant 32 bits in the CYCLES2 register.	
		0	Disable cycle counter
		1	Enable cycle counter
0 (R/W)	SSSTEP	Supervisor Single-Step. The SYSCFG.SSSTEP bit enables single-step operation, in which a Supervisor exception occurs after the processor executes each instruction. This bit only applies to executing instructions in User mode or to processing interrupts in Supervisor mode. The SYSCFG.SSSTEP bit is ignored if the core is processing an exception or a higher-priority event. If precise exception timing is required, a CSYNC instruction must be executed immediately after setting this bit.	
		0	Disable (normal operation)
		1	Enable (single step operation)

4 Program Sequencer

This chapter describes the Blackfin+ processor program sequencing and interrupt processing modules. For information about instructions that control program flow, see the program flow control instruction reference pages. For information about instructions that control interrupt processing, see the external event management chapter. Discussion of derivative-specific interrupt sources can be found in the hardware reference for the specific part.

Introduction

In the processor, the program sequencer controls program flow, constantly providing the address of the next instruction to be executed. Program flow in the chip is mostly linear, with the processor executing program instructions sequentially.

The linear flow varies occasionally when the program uses non-sequential program structures, such as those illustrated in the program flow variations figure. Non-sequential structures direct the processor to execute an instruction that is not at the next sequential address. These structures include:

- ***Loops*** - one sequence of instructions executes several times with zero overhead (no latency between the loop bottom instruction and the loop top instruction).
- ***Subroutines*** - an intentional vector from sequential flow to execute instructions from another part of memory before resuming from where the vector was placed.
- ***Jumps*** - an intentional vector from sequential flow to execute instructions from another part of memory that does not return to where the vector was placed.
- ***Interrupts and Exceptions*** - run-time events that trigger a vector to a specified subroutine that gets executed before returning flow to the application at the point at which the vector occurred.
- ***Idle*** - this instruction causes the core to stop executing the application and hold its current state until an interrupt occurs, at which point the programmed vector for that interrupt is taken to service the interrupt before returning flow to after the `IDLE;` instruction and resuming execution of the application code.

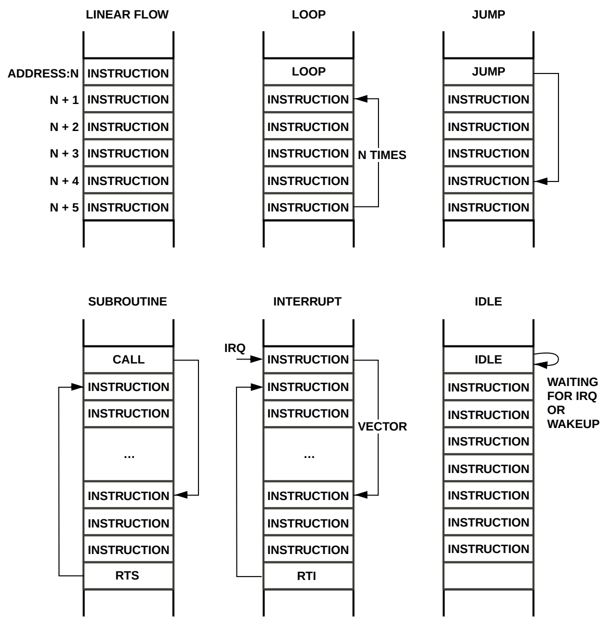


Figure 4-1: Program Flow Variations

The sequencer manages execution of these program structures by selecting the address of the next instruction to execute. Typically, the fetches are to contiguous memory after the current instruction, and fetches are made well in advance of actual execution in order to keep the pipeline full. When non-sequential code is involved, pre-fetched instructions are invalidated once the change of flow occurs, and the new instructions are fetched from the new memory that is being executed from. Dynamic branch prediction helps with this need to invalidate and re-fetch by running ahead of the sequencer and anticipating the next instruction address that the sequencer will select as a result of an upcoming branch. When the branch prediction is successful, the pre-fetched instructions do not need to be invalidated.

The fetched instruction address enters the pipeline, with the program counter (`PC`) pointing to the address of the instruction that is about to execute. The 10-stage pipeline contains the 32-bit addresses of the instructions currently being fetched, decoded, and executed. The `PC` couples with the `RETx` return registers, which store the `PC` when an event occurs such that the processor can return to this address and resume execution after the code associated with handling that event is run. All addresses generated by the sequencer are 32-bit instruction addresses in memory.

To manage events, the event controller handles interrupt and event processing, determines whether an interrupt is masked, and generates the appropriate event vector address. This aspect of code flow is described in greater detail in the Interrupt Processing section of this manual.

In addition to providing data addresses, the data address generators (DAGs) can provide instruction addresses for the sequencer's indirect branches.

The sequencer evaluates conditional instructions and loop termination conditions. The loop registers support nested loops. The memory-mapped registers (MMRs) store information used to implement interrupt service routines.

The block diagram shows the core Program Sequencer module and how it connects to the Core Event Controller (CEC).

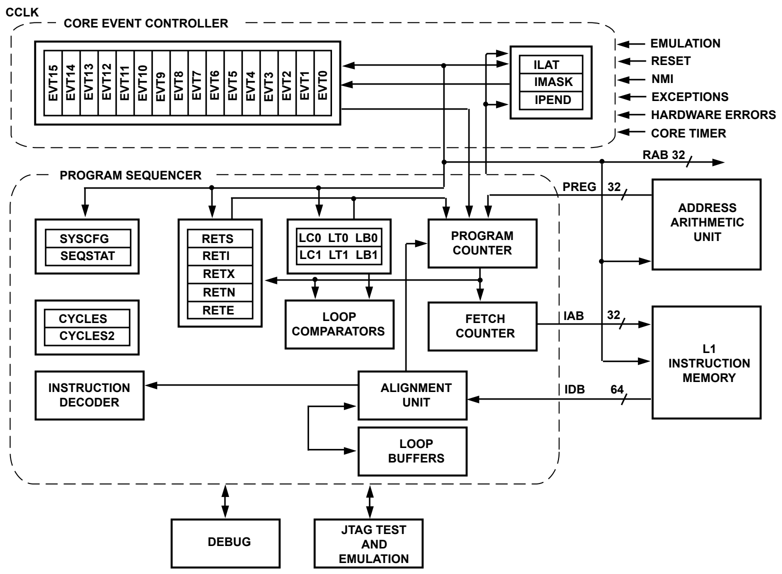


Figure 4-2: Program Sequencing and Interrupt Processing Block Diagram

Sequencer-Related Registers

The Non-Memory-Mapped Sequencer Registers table lists core registers associated with the sequencer. Except for the `PC` register, all sequencer-related registers are directly readable and writable by move instructions, for example:

```
SYSCFG = R0 ;
```

```
P0 = RETI ;
```

Manually pushing or popping registers to or from the stack is done using the explicit instructions:

```
[--SP] = Rn ; /* for push */
```

```
Rn = [SP++] ; /* for pop */
```

Similarly, all non-memory-mapped sequencer registers can be pushed to and popped from the system stack:

```
[--SP] = CYCLES ; /* for push */
```

```
SYSCFG = [SP++] ; /* for pop */
```

However, load/store operations and immediate loads are not supported.

Table 4-1: Non-Memory-Mapped Sequencer Registers

Register Name	Description
SEQSTAT	Sequencer Status
Return Address Registers: RETX RETN	Return Address registers: Return from Exception Return from NMI

Table 4-1: Non-Memory-Mapped Sequencer Registers (Continued)

Register Name	Description
RETI RETE RETS	Return from Interrupt Return from Emulation Return from Subroutine
Zero-Overhead Loop Registers: LC0, LC1 LT0, LT1 LB0, LB1	Zero-Overhead Loop registers: Loop Counters Loop Tops Loop Bottoms
FP, SP	Frame Pointer and Stack Pointer
SYSCFG	System Configuration
CYCLES, CYCLES2	Cycle Counters
PC	Program Counter The `PC` is an embedded register. It is not directly accessible with program instructions.

In addition to these core sequencer registers, there is a set of memory-mapped registers that interact closely with the program sequencer, such as the Event Vector Table (`EVTx`) registers. For information about these interrupt control registers, see Events and Interrupts. Although the registers of the Core Event Controller are memory-mapped, they still connect to the same 32-bit Register Access Bus (RAB) and perform in the same way; however, the System Event Controller (SEC) registers reside in the `SYSCLK` domain. For debug and test registers, see the processor hardware reference manual.

Instruction Pipeline

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the processor executes instructions from memory in sequential order by incrementing the look-ahead address.

The processor has a 10-stage instruction pipeline, shown in the Stages of Instruction Pipeline table.

Table 4-2: Stages of Instruction Pipeline

Pipeline Stage	Description
Instruction Fetch 1 (IF1)	Issue instruction address to IAB bus, start compare tag of instruction cache
Instruction Fetch 2 (IF2)	Wait for instruction data
Instruction Fetch 3 (IF3)	Read from IDB bus and align instruction
Instruction Decode (DEC)	Decode instructions
Address Calculation (AC)	Calculation of data addresses and branch target address

Table 4-2: Stages of Instruction Pipeline (Continued)

Pipeline Stage	Description
Data Fetch 1 (DF1)	Issue data address to DA0 and DA1 bus, start compare tag of data cache
Data Fetch 2 (DF2)	Read register files
Execute 1 (EX1)	Read data from LD0 and LD1 bus, start multiply and video instructions
Execute 2 (EX2)	Execute/Complete instructions (shift, add, logic, etc.)
Write Back (WB)	Writes back to register files, SD bus, and pointer updates (also referred to as the "commit" stage)

The Processor Pipeline figure shows a diagram of the pipeline.

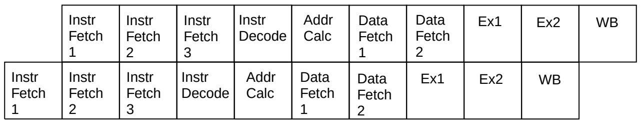


Figure 4-3: Processor Pipeline

The instruction fetch and branch logic generates 32-bit fetch addresses for the Instruction Memory Unit. If the dynamic branch predictor is enabled, the fetch address may be generated based upon a dynamic prediction of decisions made by the sequencer later in the pipeline. The Instruction Alignment Unit (IAU) returns instructions and their width information at the end of the IF3 stage.

For each instruction type (16-, 32-, or 64-bit), the IAU ensures that the alignment buffers have enough valid instructions to be able to provide an instruction every cycle. Since the instructions can be 16, 32, or 64 bits wide, the IAU may not need to fetch an instruction from the cache every cycle. For example, for a series of 16-bit instructions, the IAU gets an instruction from the Instruction Memory Unit once in four cycles. The alignment logic requests the next instruction address based on the status of the alignment buffers. The sequencer responds by generating the next fetch address in the next cycle, provided there is no change of flow.

The sequencer holds the fetch address until it receives a request from the alignment logic or until a mis-predict is detected. If no change of flow is predicted, the sequencer increments the previous fetch address by 8 (the next 8 bytes). A mis-predict occurs when the dynamic branch predictor is disabled and a change of flow occurs, such as a branch or an interrupt, or the predictor is enabled but failed to correctly anticipate the next fetch address. When a mis-predict occurs, data in the IAU is invalidated. The sequencer decodes and distributes instruction data to the appropriate locations, such as the register file and data memory.

Data register file reads occur in the DF2 pipeline stage (for operands).

Data register file writes occur in the WB stage (for stores). The MACs and the video units are active in the EX1 stage, and the ALUs and shifter are active in the EX2 stage. The accumulators are written at the end of the EX2 stage.

The program sequencer also controls stalling and invalidating the instructions in the pipeline. Multi-cycle instruction stalls occur between the IF3 and DEC stages. DAG and sequencer stalls occur between the DEC and AC stages. Computation and register file stalls occur between the DF2 and EX1 stages. Data memory stalls occur between the EX1 and EX2 stages.

NOTE: The sequencer ensures that the pipeline is fully interlocked and that all the data hazards are hidden from the programmer.

Multi-cycle instructions behave as multiple single-cycle instructions being issued from the decoder over several clock cycles. For example, the Push Multiple or Pop Multiple instruction can push or pop from 1 to 14 `Dregs` and/or `Pregs`, and the instruction remains in the decode stage for a number of clock cycles equal to the number of registers being accessed.

Multi-issue instructions are 64 bits wide and consist of one 32-bit instruction and two 16-bit instructions. All three instructions execute in the same number of cycles as the slowest of the three.

Any non-sequential program flow can potentially decrease the processor's instruction throughput. Non-sequential program operations include:

- Jumps
- Subroutine calls and returns
- Interrupts and returns
- Loops

Branches

One type of non-sequential program flow that the sequencer supports is branching. A branch occurs when a `JUMP` or `CALL` instruction begins execution at a new location other than the next sequential address. For descriptions of how to use the `JUMP` and `CALL` instructions, see the program flow control instruction reference pages. Briefly:

- A `JUMP` or a `CALL` instruction transfers program flow to another memory location. The difference between a `JUMP` and a `CALL` is that a `CALL` automatically loads the return address into the `RETS` register. The return address is the next sequential address after the `CALL` instruction. This load makes the address available for the `CALL` instruction's matching return instruction (`RTS;`), allowing easy return from the subroutine.
- A return instruction causes the sequencer to fetch the instruction at the return address, which is stored in one of the `RETx` registers. The types of return instructions include:
  - return from subroutine (`RTS`), associated with the `RETS` register
  - return from interrupt (`RTI`), associated with the `RETI` register
  - return from exception (`RTX`), associated with the `RETX` register
  - return from emulation (`RTE`), associated with the `RETE` register
  - return from non-maskable interrupt (`RTN`), associated with the `RETN` register

- A `JUMP` instruction can be conditional, depending on the status of the `ASTAT.CC` bit. These instructions are immediate and may not be delayed. The program sequencer can evaluate the `ASTAT.CC` bit to decide whether or not to execute a branch. If no condition is specified, the branch is always taken. Conditional `JUMP` instructions use static branch prediction to reduce the branch latency caused by the length of the pipeline, when dynamic branch prediction is not enabled.

Branches can be direct or indirect. A direct branch address is embedded in the instruction itself (e.g., `JUMP 0x30`), whereas an indirect branch gets its address from the contents of a `Preg` (e.g., `JUMP(P3)`). Both direct and indirect branches can be PC-relative or absolute.

Direct Jumps (Short, Long and Extra-Long)

The sequencer supports three lengths of PC-relative direct jumps - short, long, and extra long. In all cases, the target of the branch may be a PC-relative address from the location of the instruction plus or minus an offset. The PC-relative offset for the short jump is a 13-bit immediate value that must be a multiple of two (`imm13m2`), thus providing an effective dynamic range of -4096 to +4094 bytes.

The PC-relative offset for the long jump is a 25-bit immediate value that must also be a multiple of two (`imm25m2`), thus providing an effective dynamic range of -16,777,216 to +16,777,214 bytes.

The PC-relative offset for the extra long jump is a 32-bit immediate value that must also be a multiple of two (`imm32m2`), thus providing an effective dynamic range covering the entire address space of the processor.

When application code is being developed, the offset to the jump destination is usually unknown. The development tools will evaluate the offset during the build process and select the appropriate jump length when the `JUMP 0xnnnnnnnn` instruction is generated by the compiler or identified by the assembler.

Each of the supported jump lengths has its own version of the `JUMP` instruction, the exact form of which is differentiated by appending the appropriate modifier and argument:

- Short jumps (requires 13-bit offset): `JUMP.S 0xnnnn;`
- Long jumps (requires 25-bit offset): `JUMP.L 0xnnnnnnn;`
- Extra-long jumps (requires a 32-bit offset): `JUMP.XL 0xnnnnnnnn;`

The assembler will replace `JUMP` instructions with the appropriate `JUMP.S`, `JUMP.L` or `JUMP.XL` instructions.

Rather than hard-coding jump target addresses, symbolic addresses can be used in assembly source files. Symbolic addresses are called labels and are marked by a trailing colon. See the *CrossCore Embedded Studio Assembler and Preprocessor Manual* for details.

```
JUMP mylabel ;
/* skip any code placed here */
mylabel:
/* continue to fetch and execute instructions beginning here */
```

Direct jumps to an absolute address are also supported. The target address is taken directly from the 32-bit immediate value in the instruction. For an absolute jump, use `JUMP.A 0xnnnnnnnn`.

Direct Call (Long and Extra-Long)

A call instruction is a branch instruction that copies the address of the instruction which would have executed next (had the call instruction not executed) into the `RETS` register. The development tools will evaluate the offset and select the appropriate PC-relative call instruction when the `CALL` instruction syntax is used.

The long direct call instruction has a 25-bit PC-relative offset that must be a multiple of two (`imm25m2`), thus providing an effective dynamic range of –16,777,216 to +16,777,214 bytes. A long direct call can be specified explicitly by using a `CALL.L` instruction.

The extra-long direct call instruction has a 32-bit PC-relative offset that must be a multiple of two (`imm32m2`), thus providing an effective dynamic range covering the entire address space of the processor. An extra-long PC-relative direct call can be specified explicitly by using a `CALL.XL` instruction.

An absolute direct call is achieved using the `CALL.A` instruction.

Indirect Jump and Call (Absolute)

The indirect `JUMP` and `CALL` instructions get their destination absolute address (branch target) from a data address generator (DAG) pointer register. For the `CALL` instruction, the `RETS` register is loaded with the address of the instruction which would have executed next in the absence of the `CALL` instruction. For example:

```
JUMP (P3) ;
CALL (P0) ;
```

A pointer register can be loaded directly with a 32-bit address:

```
P4 = mytarget;
JUMP (P4);
/* sequential code that is jumped over */
mytarget:
/* continue here */
```

Indirect Jump and Call (PC-Relative)

The PC-relative indirect `JUMP` and `CALL` instructions use the contents of a pointer register as the offset to the branch target. For the `CALL` instruction, the `RETS` register is loaded with the address of the instruction which would have executed next (had the `CALL` instruction not executed). For example:

```
JUMP (PC + P3) ;
CALL (PC + P0) ;
NOP;              // RETS points to this instruction
```

Subroutines

Subroutines are code sequences that are invoked by a `CALL` instruction. Assuming the stack pointer `SP` has been initialized properly, a typical scenario could look like the following:

```
/* parent function */
R0 = 0x1234 (Z);           /* CCES compiler passes 1st argument in R0 */
CALL my_function;
```

```
/* continue here after the call */
[P0] = R0;                  /* save return value (CCES compiler places it in R0) */
JUMP somewhere_else;

my_function:                /* subroutine label */
   [--SP] = (R7:7, P5:5);   /* push (save)  2 used registers to stack */
   P5.H = hi(myregister);   /* P5 used locally */
   P5.L = lo(myregister);
   R7 = [P5];               /* R7 used locally */
   R0 = R0 + R7;            /* R0 used for parameter passing and return value */
   (R7:7, P5:5) = [SP++];   /* pop (restore) saved registers from stack */
   RTS;                     /* return from subroutine */
my_function.end:            /* closing subroutine label */
```

Due to the syntax of the push-multiple and pop-multiple instructions requiring that the higher registers appear first, the CCES compiler uses the upper data and pointer registers for local purposes and the lower registers to pass arguments and store return values. See the address arithmetic unit chapter for more details on stack management, as well as the *CrossCore Embedded Studio Compiler and Library Manual* for register usage.

The `CALL` instruction not only redirects the program flow to the *my_function* routine, it also writes the address of the instruction following the `CALL` instruction into the `RETS` register such that the `RETS` register holds the address where program execution resumes after the `RTS` instruction executes. In the above example, this is the address of the `[P0]=R0;` instruction. The return address is not passed to any stack in the background; rather, the `RETS` register functions as a single-entry hardware stack. This scheme enables " leaf functions" (subroutines that do not contain further `CALL` instructions) to execute with less overhead, as no bus transfers need to be performed. If a subroutine calls other functions, it must save the content of the `RETS` register explicitly, most likely via stack operations:

```
/* parent function */
CALL function_a;
/* continue here after the call */
JUMP somewhere_else;


function_a:         /* subroutine label */
   [--SP] = RETS;   /* save RETS onto stack */
   CALL function_b; /* call further subroutines */
   CALL function_c;
   RETS = [SP++];   /* restore RETS */
   RTS;             /* return from subroutine */
function_a.end:     /* closing subroutine label */

function_b:         /* subroutine label */
   /* do something */
   RTS;             /* return from subroutine */
function_b.end:     /* closing subroutine label */

function_c:          /* subroutine label */
  /* do something else */
   RTS;             /* return from subroutine */
function_c.end:     /* closing subroutine label */
```

Stack Variables and Parameter Passing

Many subroutines require input arguments from the calling function and need to return their results. Certain core registers are used for passing arguments, while others return the result. See the *CrossCore Embedded Studio Compiler and Library Manual* for details.

NOTE: It is recommended that assembly programs meet the same conventions used by the C/C++ compiler.

The CCES compiler passes up to three arguments in registers and utilizes the stack to pass any arguments beyond three that are defined in the function prototype . The following assembly example shows how to pass and return arguments using the stack:

```
_parent:
   R0 = 1;            /* load argument 1 */
   R1 = 3;            /* load argument 2 */
   [--SP] = R0;       /* push argument 1 to stack */
   [--SP] = R1;       /* push argument 2 to stack */
   CALL _sub;         /* call subroutine */

   R1 = [SP++];       /* R1 = 4 */
   R0 = [SP++];       /* R0 = 2 */
_parent.end:

_sub:
   [--SP] = FP;      /* save frame pointer */
   FP = SP;          /* create new frame */
   [--SP] = (R7:5);  /* save clobbered registers R7, R6, and R5 */

   R6 = [FP+4];      /* R6 = 3, from the stack push of R1 in _parent */
   R7 = [FP+8];      /* R7 = 1, from the stack push of R0 in _parent */
   R5 = R6 + R7;     /* processing */
   R6 = R6 - R7;

   [FP+4] = R5;      /* R5 = 4, place on stack where R1 was saved */
   [FP+8] = R6;      /* R6 = 2, place on stack where R0 was saved */

   (R7:5) = [SP++];  /* restore preserved registers */
   FP = [SP++];      /* restore frame pointer */
   RTS;
_sub.end:
```

Since the stack pointer `SP` is modified inside the subroutine for local stack operations, the frame pointer `FP` is used to save the original state of `SP`. Because the 32-bit frame pointer itself must be pushed onto the stack first, the `FP` is four bytes beyond the original `SP` address.

The Blackfin+ instruction set features a pair of instructions that provides cleaner and more efficient functionality than the above example, `LINK` and `UNLINK`. These multi-cycle instructions perform multiple operations that can be best explained by the equivalent code sequences shown in the table.

Table 4-3: Link and Unlink Code Sequence Equivalents

LINK n;	UNLINK;
`[--SP] = RETS;`	`SP = FP;`
`[--SP] = FP;`	`FP = [SP++];`
`FP = SP;`	`RETS = [SP++];`
`SP += -n;`	

The following subroutine is similar to the previous example, except the LINK and UNLINK instructions are used. This means that the RETS register is also saved to the stack to enable nested subroutine calls, and the SP can optionally be adjusted to allow for local information to be stored on the stack. Because of this, another 32-bit value is being pushed to the stack, which means the value stored to FP is now eight bytes from the original SP address instead of four. Additionally, since no local frame is required to accommodate local variables on the stack, the LINK instruction gets the parameter "0", as the SP does not get adjusted:

```
_sub2:
   LINK 0;           /* creates new frame, saves RETS and SP */
   [--SP] = (R7:5);  /* save clobbered registers R7, R6, and R5 */
   R6 = [FP+8];      /* R6 = 3, from the stack push of R1 in _parent */
   R7 = [FP+12];     /* R7 = 1, from the stack push of R0 in _parent */

   R5 = R6 + R7;     /* processing */
   R6 = R6 - R7;

   [FP+8] = R5;      /* R5 = 4, place on stack where R1 was saved */
   [FP+12] = R6;     /* R6 = 2, place on stack where R0 was saved */

   (R7:5) = [SP++];  /* restore preserved registers */
   UNLINK;           /* restore SP, FP, and RETS */
   RTS;
_sub2.end:
```

If subroutines require local, private, and/or temporary variables, the stack can be used. The LINK instruction takes a parameter that specifies the size of the stack memory required for this. The following example provides two local 32-bit variables and initializes them to zero when the routine is entered:

```
_sub3:
   LINK 8;                /* save FP/RETS, allocate 8 stack bytes for local data */
   [--SP] = (R7:0, P5:0); /* save all potentially clobbered registers */
   R7 = 0 (Z);            /* set initialization value to 0 */
   [FP-4] = R7;           /* initialize 1st local variable on stack */
   [FP-8] = R7;           /* initialize 2nd local variable on stack */
      /* code goes here */
   (R7:0, P5:0) = [SP++]; /* restore preserved registers */
   UNLINK;                /* restore SP, FP, and RETS */
   RTS;
_sub3.end:
```

For more information, see the `LINK` and `UNLINK` instruction reference pages.

Conditional Processing

The Blackfin+ processors support conditional processing through conditional jump and move instructions. Conditional processing is described in the following sections:

- Conditional Code Status Bit
- Conditional Branches
- Branch Prediction
- Speculative Instruction Fetches
- Conditional Register Move

Conditional Code Status Bit

The processor supports a Conditional Code status bit (`ASTAT.CC`), which is used to resolve the direction of a branch. This status bit may be accessed multiple ways (note that the assembler syntax for referencing this bit is `CC`):

- By the sequencer - a conditional branch is determined based on its value
- Load/store interaction with a `Dreg`:

  ```
  R0 = CC; /* R0 becomes either 0 or 1 */
  ```

  ```
  CC = R1; /* CC set to 0 only if R0 is 0, otherwise 1 */
  ```

- Bit test (`BITTST`) instruction:

  ```
  CC = BITTST (R0, 31) ; /* CC set to value of bit 31 in R0 */
  ```

- Reading/writing other `ASTAT` status bits:

  ```
  CC = AV0;
  ```

  ```
  AV0 = CC;
  ```

- Result of a `Preg` comparison:

  ```
  CC = P0 < P1 ; /* CC set to 1 if P0 < P1 */
  ```

- Result of a `Dreg` comparison:

  ```
  CC = R5 == R7; /* CC set to 1 only if R5 = R7 */
  ```

- Rotate (`ROT`) instruction shifts through the `ASTAT.CC` bit:

  ```
  R0 = ROT R1 by R2.L ; /* CC set to value of R1[[R2.L]] bit */
  ```

- Store-exclusive and `SYNCEXCL` instructions:

  ```
  CC = ([P5] = R0) (P5) ;
  ```

- Test and set instruction (`TESTSET`):

  ```
  TESTSET (P5) ;
  ```

These nine ways of accessing the `ASTAT.CC` bit are used to control program flow. The branch is explicitly separated from the instruction that sets the arithmetic status bits. A single bit resides in the instruction encoding that specifies the interpretation for the value of `ASTAT.CC`. The interpretation is to "branch on true" or "branch on false".

The comparison operations have the form `CC = expr`, where *expr* involves a pair of registers of the same type (e.g., `Dreg` or `Preg`) or a single register and a small immediate constant. The small immediate constant is a 3-bit signed number (-4 through 3) for signed comparisons and a 3-bit unsigned number (0 through 7) for unsigned comparisons.

The sense of `ASTAT.CC` is determined by equal (==), less than (<), and less than or equal to (<=) operators. There are also bit test operations that test whether or not a bit in a 32-bit data register is set.

Conditional Branches

The sequencer supports conditional branches. Conditional branches are `JUMP` instructions whose execution branches or continues linearly, depending on the value of the `CC` bit. The target of the branch is a PC-relative address from the location of the instruction, plus or minus an offset. The PC-relative offset is an 11-bit immediate value that must be a multiple of two (`imm11m2`), thus providing an effective dynamic range of -1024 to +1022 bytes.

For example, the following instruction tests the `CC` status bit and, if it is positive, jumps to a location identified by the label `dest_address`:

```
IF CC JUMP dest_address ;
```

Similarly, a branch can also be taken when the `CC` bit is not set:

```
IF !CC JUMP other_addr ;
```

NOTE: Take care when conditional branches are followed by load operations. For more information, see the Load/Store instruction reference pages.

Branch Prediction

Branches can be accelerated if the processor predicts the target of an upcoming branch instruction before it has been committed and fetches instructions from the target sooner, thus reducing the number of instructions that need to be aborted. Prediction can be performed dynamically based upon the location and direction of the branches the processor has previously executed, or prediction may be static based upon information supplied by the programmer.

Blackfin+ processors support dynamic branch prediction. Software can check for this capability by testing the `BPRED` bit of the `FEATURE0` MMR. Older Blackfin implementations which do not have a `FEATURE0` MMR do not support dynamic branch prediction.

Dynamic branch prediction is enabled by setting the `SYSCFG.BPEN` bit. Once enabled, the branch latency of all branches, whether taken or not taken, is significantly reduced. The latency is never longer than the latency of the branch had it been statically mispredicted.

The dynamic predictor comes up in a state in which it can immediately start executing. There may be some delay before it can improve the latency of branches, but software does not need to do anything other than set `SYSCFG.BPEN` to enable it. The static prediction bit in conditional branch instructions is used to initialize the dynamic prediction. Prediction is validated before the instruction at the branch address is committed, so self-modifying code and code overlays will work as expected with the predictor enabled.

Additional control registers are provided for test, verification, and tuning of the the dynamic branch predictor. Software should not need to use these registers to achieve reasonable performance in the general case. For more information on the dynamic branch predictor see Dynamic Branch Prediction.

Statistically, dynamic branch prediction is far more successful than static prediction and can lead to significant improvements in program performance without code modification. However, the predictor consumes power and is a feature that is a candidate to be disabled in low-power applications.

When dynamic branch prediction is disabled or not available, the sequencer supports static branch prediction to accelerate execution of conditional branches. These branches are executed based on the state of the `ASTAT.CC` bit.

In the EX2 stage, the sequencer compares the actual `ASTAT.CC` bit value to the predicted value. If the value was mispredicted, the branch is corrected, and the correct address is available for the WB stage of the pipeline.

The branch latency for statically predicted conditional branches is as follows:

- Correctly predicts a non-taken branch: 0 `CCLK` cycles
- Mispredicts a non-taken branch: 8 `CCLK` cycles
- Correctly predicts a taken branch: 4 `CCLK` cycles.
- Mispredicts a taken branch: 8 `CCLK` cycles.

For all unconditional branches, the branch target address computed in the AC stage of the pipeline is sent to the Instruction Fetch Address (IFA) bus at the beginning of the DF1 stage. All statically predicted unconditional branches have a latency of 4 `CCLK` cycles. Consider the example in the Branch Prediction table:

Table 4-4: Branch Prediction

Instruction	**Description**
If CC JUMP dest (bp)	This instruction tests the `ASTAT.CC` status bit. If it is set, it jumps to the location identified by the "dest" label, otherwise it continues to the next sequential instruction. When dynamic branch prediction is disabled or not available, `(bp)` indicates the branch is statically predicted taken. If `ASTAT.CC` is set, the branch is correctly predicted and the branch latency is reduced. Otherwise, the branch is incorrectly predicted and the branch latency increases. When dynamic branch prediction is enabled, `(bp)` is simply used to initialize the predictor. Branches that are always taken or always not taken will tend to be correctly predicted and have a greatly reduced branch latency.

Dynamic Branch Prediction

Dynamic branch prediction is performed by a unit in the Blackfin+ processor known as the Branch Predictor (BP). The default configuration of the BP should be suitable for most programs, so further configuration apart from enabling it should not be necessary. However, certain applications may benefit from tuning the BP unit, and this section provides a detailed description so that programmers can gain an insight into how the performance of their programs is affected.

Branch Predictor Overview

To improve branch execution performance, the Blackfin+ BP learns the type, source address, and target address of most static and conditional branches. For conditional branches, it will also learn and update a four-value prediction code that indicates which direction the branch is most likely to take based on recent execution history. This information is stored in a 2 KB RAM referred to as the BP Table. Each time an instruction memory fetch is made, the memory address is used to access the BP Table to see if the BP has learned a branch associated with that address. If there is a static branch or a conditional branch that is predicted taken in the table, the BP provides the type of instruction and target address to the fetch block, which then immediately fetches from the target address. Branch predictions made by the BP result in a savings of five cycles for static branches and eight cycles for conditional branches (as compared to non-predicted branches).

The BP performs two kinds of accesses to the BP Table:

- Fetch accesses - triggered every time the sequencer fetches an instruction from memory. The BP uses the fetch address to read from the table and check for possible branch hits. If one is found, the branch data is evaluated, and a prediction is sent to the instruction fetch block, which then fetches the target address provided by the BP.
- Management accesses - add new entries to the table or modify existing entries. Management accesses are only performed in cycles when there are no instruction fetches. This ensures that BP predictions have a higher priority, even at the expense of additional delay in adding or updating branch entries to the table.

The BP has two 32-bit memory-mapped registers (MMRs), `BP_CFG` and `BP_STAT`, to control and monitor its operation.

BP RAM

The Blackfin+ fetch unit operates on instruction data that is 64 bits wide and 64-bit-aligned in memory. Each 64-bit segment is referred to as a line, with one line being fetched for each instruction fetch. The BP is designed to predict two possible branches for each line.

The BP Table RAM can be viewed as two-way set associative. Each row will hold the data for two branches or one line. The first branch learned for a line will be associated with Set 0 and the second branch with Set 1. Additional branches will be added by alternating and overwriting Set 0 or Set 1. The data in each set can be divided into two 32-bit parts, the TAG and the TARG. The TAG section contains the branch source information, and the TARG contains the branch destination (target address).

The BP uses 64 bits for each table entry in Set 1 or Set 0. The LRU bit points to the oldest branch table entry that was accessed for each line. This data is used to determine which set to overwrite when a new branch is to be learned.

It is written to both the TAG0 and TAG1 portions of the row whenever either Set 0 or Set 1 is written. The other bit assignments for the branch TAG and TARG portions of table entries can be seen in the ***BP Table Entry Structure*** table.

Table 4-5: BP Table Entry Structure

Field	Name (Description)
TAG[31:10]	SOURCE_ADDR (21 most significant bits of the branch source address [31:10]).
TAG[9]	Reserved
TAG[8:6]	TYPE (Branch type).
	b#000 (BRCC - Conditional Jump)
	b#001 (JUMP - Unconditional Jump)
	b#010 (RTS - Return from Subroutine)
	b#011 (Reserved)
	b#100 (Reserved)
	b#101 (CALL32 - Short Call)
	b#110 (CALL64 - Long Call)
	b#111 (CALL128 - Long Call Multi-Instruction)
TAG[5:4]	PREDICTION (Prediction Strength).
	b#00 (Strongly Not Taken)
	b#01 (Weakly Not Taken)
	b#10 (Weakly Taken)
	b#11 (Strongly Taken)
TAG[3]	VALID (Valid indicator). Can be set to 0 to remove entry from prediction process.
TAG[2:1]	SOF (Byte offset of the branch in the line, from source address[2:1]).
TAG[0]	LRU (Least recently used indicator). Policy bit to determine which BP Table entries to remove.
TARG[31:0]	TARG_ADDR (32-bit branch target address).

Configuring The Branch Predictor

BP support is controlled by the `SYSCFG.BPEN` bit. When set, the BP unit is enabled, and specific features of the BP unit can be individually controlled via the `BP_CFG` register. To disable the enabled BP unit, a `CSYNC` instruction is first needed before clearing the `SYSCFG.BPEN` bit. When the BP is disabled in this fashion, it finishes any fetch or management table access that is in progress but does not begin a new access after the ongoing transaction completes. If the BP unit is to be re-enabled again, also remember to set the `BP_CFG.CLRBP` bit to flush the BP table.

NOTE: Once the `BP_CFG.CLRBP` bit is set, flushing the BP Table requires 150 core clock (CCLK) cycles, which must be accounted for in software before re-enabling branch prediction (i.e., setting the `SYSCFG.BPEN` bit again).

The `BP_CFG` MMR also provides the enable bits for each of the branch types that is supported. When the enable bits are set to 1, the BP will execute requests to learn branches of that specific type and add them to the BP Table. When set to 0, new branches for the specified type will not be learned; however, branches of this type that are already in the BP Table will continue to be predicted and updated.

The ***Branch Instructions Supported by BP*** table shows the branch instructions supported by the BP. The BP type code which is stored with the branch TAG data is shown following each instruction type.

Table 4-6: Branch Instructions Supported by BP

Instruction	BP_CFG Enable Bit
JUMP pcrel13	JUMPEN
JUMP.L pcrel24	JUMPEN
JUMP Imm32(opt pcrel)	JUMPEN
IF CC JUMP pcrel10	JUMPCCEN
IF CC JUMP pcrel10 (bp)	JUMPCCEN
IF !CC JUMP pcrel10	JUMPCCEN
IF !CC JUMP pcrel10 (bp)	JUMPCCEN
CALL pcrel24	CALL32EN
CALL Imm32(opt pcrel)	CALL64EN
RTS	RTSEN

The following branch instructions are not supported by the BP:

- JUMP (Px)
- JUMP (PC + Px)
- CALL (Px)
- CALL (PC + Px)
- RTE
- RTI
- RTX
- RTN
- LSETUP

  NOTE: Hardware loops are zero-latency without the BP.

Prediction for all branch types should be enabled for best average performance. However, prediction for individual branch types may be disabled to fine tune BP operation for a specific application. Empirical testing may identifty a more optimal configuration for specific applications, but experimentation towards that goal should begin by measuring program performance with all the enable bits set.

BP Store Buffers

The BP receives information from the sequencer pipeline control logic which tells it when to learn and update data about branches that it is executing. This information comes to the BP at several places in the pipe and on different phases of the clock. This requires the BP to align and store data in order to enter it into the BP Table RAM in a single access. The strategy of executing management table operations between instruction fetches, which are not easily predictable, also requires the BP to buffer data before it is entered into the table.

To meet these requirements, the BP has two data store buffers. These buffers store the data coming from the sequencer that is used to load the TAG and TARG portions of the BP Table entry. They also store information such as whether to learn or update, as well as data required for updating prediction states.

Each store buffer is managed by a three-value state machine: idle, waiting for additional data, or full. The store buffer enters the full state once it has all of the data that it needs to complete each type of table access operation. When the buffer is full, it generates a request to the table state control machine to move its data to the table. It waits in the full state until the table state control machine accepts its data and begins writing it to the table. The store buffer machine can then move to idle, waiting, or full with a new buffer full of data.

The BP has additional store buffer logic which controls the next buffer to load and the order that buffer requests are fed to the table state control machine.

BP Table Control

The table control state machine manages four types of table accesses that can be requested by the store buffers:

- Learn access - creates a new entry in the table and writes TAG and TARG data to either Set 0 or Set 1 of the table row indicated by bits 9:3 of the branch source address found in the TAG.
- Update access - changes the prediction values in the TAG fields of Set 0 or Set 1 based on the branch source address and information provided by the sequencer when the predicted branch is executed and an update is requested.
- Instruction mispredict access - occurs when the type of a prediction does not match the type expected by the sequencer. When this occurs, the sequencer requests an instruction mispredict access and provides the offending source address. The table control state machine then executes an instruction mispredict access, which sets the Valid bit to 0 in the TAG field of the appropriate entry in the table. This prevents further predictions from this entry.
- Address mispredict access - occurs when the type of prediction matches what the sequencer expects, but the target address does not. When this occurs, the sequencer requests an address mispredict access and provides the

offending source and correct target address. The table control state machine then executes an address mispredict access, which updates the TARG field of the appropriate entry in the table with the correct target address. This insures that future predictions to this source address will point to the correct target address.

NOTE: While the learn and update accesses are all that should be necessary for static code, self-modifying code and code overlays can cause the BP Table to contain outdated branch entries which will cause the BP to send incorrect predictions to the sequencer. Supporting these use cases requires the instruction and address mispredict accesses.

NOTE: The LRU bit is toggled and updated for each of the four types of table accesses.

To support the four types of table accesses, the table control state machine has seven states:

- Idle - occurs when there are no access requests from the store buffers.
- Check - moves the data from a requesting store buffer to a local table control buffer and executes a RAM read using bits 9:3 of the source address from the local buffer for the RAM address. The state machine may remain in the check state for several cycles until a non-fetch cycle is available for the read to execute in.
- Process - entered once the check state read completes. Data from the check state read is used during this state to determine if the branch in the local buffer was found in the table, calculate new prediction values if the access is an update, set the next valid bit to 0 if the access is an instruction mispredict, and set the next value for the LRU bit. The process state always requires only one cycle, and instruction fetches and predictions can be performed while it is executing.

Following the process state, the state machine will move into one of the remaining four write states (one for each of the learn, update, instruction mispredict, and address mispredict access types contained in the local buffer, as detailed above). Each of these access types will execute a RAM write using data obtained from the local buffer or calculated during the process cycle. The four different states enable different RAM write control lines, depending on the type of data which needs to be written for each access type. The state machine may remain in any of these four write states for several cycles until a non-fetch cycle is available for the write to execute in. Upon completion of the write, the state machine will return to idle or can move to the check state and begin processing the next store buffer request immediately.

The local control buffer managed by the table control state machine stores the same data as a store buffer. The requesting store buffer is freed as soon as the check state is entered, therefore three table access requests may be in-flight at any given time.

The table control state machine waits in the check and write states until a non-fetch cycle is available. To ensure this wait is not indefinite, the number of sequential fetch cycles are counted. If a threshold is exceeded, the sequencer is requested to hold off an instruction fetch for one cycle. The `STMOUTVAL` field of the `BP_CFG` register holds the threshold value, and the `STMOUTCNTR` field of the `BP_STAT` register holds the current value of the counter.

All types of table accesses perform a table read when in the check state. If the access is a learn request, a matching entry is not expected to be found in the table. If a matching entry is found, the `DFL` bit in the `BP_STAT` register is set, and the new data is not written to the table. This condition can occur when multiple learn requests are issued by the sequencer as a result of a delayed entry of the first request into the branch table. If the access is an update, instruction mispredict, or address mispredict and the entry is not found, the NFL bit in the BP_STAT register is set, and no data is written to the table. This can occur when a table entry is overwritten with a new branch entry just after it is used to make a prediction. Both these conditions can occur during normal BP operation. The DFL and NFL bits are sticky and are reset by writing-1-to-clear the CLRDFL and CLRNFL bits of the BP_CFG register.

Table Initialization

The BP Table RAM contains LRU bits, valid bits, and prediction value bits that are used for control purposes. These bits must be in a known state before the BP Table can be used for predictions, so the entire table is initialized whenever the core is reset. The initialization process for the BP Table is triggered whenever $\overline{\text{RESET}}$ is asserted. All of the entries in the BP Table are written with 0s, one row at a time, which requires approximately 150 core clock cycles. During this interval, all other types of accesses to the table are blocked; hence, no prediction, learning, or register table access is possible.

During initialization, the BP Table is unavailable, but the store buffers are allowed to operate, which means that learning and update requests will be loaded to the store buffers. They will remain in the buffers until the BP Table is done initializing and is ready to accept store buffer requests, at which point the last two BP operations requested by the sequencer during the initialization period will be loaded to the BP Table. The operations requested by the sequencer prior to the last two requests are lost.

The BP also provides the capability to re-initialize the BP Table while the core is not in the Reset state. This feature may be useful in situations where code overlays are being swapped, where it may be more cycle-efficient to remove stale branches from the table before learning branches associated with the new code block. While this would prevents the need to individually correct stale branches through instruction or address mispredict operations, the usefulness of this approach must be evaluated on a case-by-case basis.

To reinitialize the BP Table, set the self-clearing BP_CFG.CLRBPbit. Writing this BP Table Clear bit will trigger the same initialization process that occurs when $\overline{\text{RESET}}$ is asserted, and the BP will begin predicting and learning with a clean BP Table after the required delay.

NOTE: Software must accommodate the required delay such that no branch instruction associated with any of the enabled BP_CFG bits gets executed while the BP Table is resetting.

Sequencer BP Requests

The sequencer produces four types of requests which are loaded to the store buffers and then used by the BP to modify the BP Table: learn, update, instruction mispredict, and address mispredict. The sequencer provides the data for these requests at two different points during pipe execution, which is described next.

The BP receives the earliest requests in pipe stage "E", which are referred to as mid-pipe requests, and the targets for these branches are received in pipe stages "F" and "G". The second group of requests occurs in pipe stage “J”, which are referred to as late-pipe requests, and the targets for these branches are received at the same time (in pipe stage “J”).

Mid-pipe requests include learns for most static branches, including jumps, calls, and returns from subroutines. Dynamic branches or conditional branches (BRCCs) which are predicted taken (BP argument) are also learned at the mid-pipe point. BRCCs which are not predicted taken (no BP argument) are not learned mid-pipe. The Instruction Mispredict request is also asserted at mid-pipe.

Update requests for all branch types occur late-pipe. BRCCs which are not predicted taken (no BP argument) and are taken (mispredicted) are learned at this point. BRCCs which are predicted taken (BP argument) and are not taken (mispredicted) were learned at mid-pipe and are not re-learned at this point. The first prediction for these branches will be incorrect but will be updated to the proper value in the update following the first prediction. Address Mispredict requests are asserted by the sequencer in pipe stage “H”. The new target value for address mispredicts isn’t asserted until pipe stage “J”, so the BP treats Address Mispredicts as late-pipe requests.

The sequencer performs update requests to modify the prediction values in the table based on whether the BP dynamic branch predictions were correct or not. Updates are also used to monitor the number of predictions made and to modify the `LRU` bit in the BP Table entry so that older entries are overwritten first. By default, the sequencer will generate an update for every BP prediction it receives, including predictions for static branches such as unconditional jumps, calls and returns from subroutine. The BP has two alternative modes in which the table is updated less frequently, which are selected by setting the `SKUPD` or `SKUPDLRU` bits in the `BP_CFG` register. Only one of these bits should be set at a time.

- Skip Update mode is enabled when the `SKUPD` bit of the `BP_CFG` register is set. This mode causes updates to be skipped if the prediction code for a predicted branch is strongly taken or strongly not taken. Since the prediction code is set to strongly taken for all static branches, updates will be eliminated for all static branches. Updates for BRCCs will be eliminated if the prediction code is strongly taken or strongly not taken and the prediction was not mispredicted. Mispredicted BRCCs generate an update request regardless of the update mode so the prediction value can be updated.
- Skip Update LRU mode is enabled when the `SKUPDLRU` bit of the `BP_CFG` register is set. This mode causes updates to be skipped if the prediction code for a predicted branch is strongly taken or strongly not taken and the predicted branch is not the oldest (LRU) in the table for a specific line. The additional LRU qualification results in the newest accessed branch in the table being kept longer. This should increase the frequency of predictions for branches located near the current PC.

The Skip Update and Skip Update LRU modes sacrifice the accuracy of prediction to reduce the frequency of updates. An excessive number of updates might increase the delay before branches are actually learned and impact the frequency of instruction fetches, so this can be a net benefit; however, the default settings are expected to perform best in general.

BP Store Buffer State Machine

When a mid-pipe request is assigned to one of the store buffers in pipe stage “E”, its state moves from Idle to Wait. The wait is required to allow the pipe to progress to the “F/G” stages to receive the target address. Once the target is received, the BP has all of the required data, and the state moves from Wait to Full. A request is then sent to the BP Table Control state machine to transfer the requested operation and data to the BP Table Control local buffer so it can be used to modify the BP Table. Once the request is accepted and the data is moved, the store buffer state machine can return to Idle. When a late-pipe request is assigned to a store buffer, the state machine moves directly from

Idle to Full because all of the required data is available in the same cycle. When the state moves to Full, the interaction with the BP Table Control state machine is the same as in the mid-pipe request case. The store buffer data is moved, and the store buffer state machine can return to Idle.

There are several exceptions to this basic operation due to sequencer operation or to reduce the number of cycles that it takes for a request to move through the store buffers and be incorporated into the BP Table.

1. When a late-pipe request is sent to the BP before or at the same time as the target for a mid-pipe request - the state machine would have moved from Idle to Wait when the mid-pipe request was received. If a late-pipe request is received at this point, it is given a higher priority because late-pipe requests will typically cause a change of flow (COF), which means the mid-pipe request will be killed and its target never fetched or learned. When the late-pipe request is received, the store buffer loads the data as in a normal late-pipe request, and the state machine moves to Full. The transfer of the late-pipe request then moves to the BP Table in the normal fashion. The mid-pipe data is overwritten, and the mid-pipe request is dropped.
2. Mid-pipe transactions can be lost if another branch which is not supported by the BP causes a COF to occur while the state machine is in Wait - interrupts are a good example of this. When a mid-pipe operation starts, the state machine moves to Wait. If an interrupt COF occurs before the mid-pipe target is fetched and sent to the BP, the correct target will never be sent to the BP to complete the mid-pipe learn. In this case, the non-learned COF causes the mid-pipe request to be dropped, and the state returns to Idle. This scenario does not occur for late-pipe requests because all of the data is presented in the same cycle and a late-pipe learn will not occur in the same cycle as any other non-supported branch COF. This means that the BP has everything it needs to complete the late-pipe request, so it executes without a problem.
3. If requests from the sequencer come close together, the store buffer state machine does not have to return to Idle before starting a new request - it is possible for the state machine to move from Full to Wait or from Full to Full again with the appropriate new mid- or late-pipe requests. If the state machine is Full when an new request is started, and the request to move the current data to the BP Table Control local buffer has not been accepted, the current data will be overwritten. The number of times that data is overwritten is an important measure of store buffer performance, so overwrites are included as one of the BP event monitoring parameters.
4. Mid-pipe instruction mispredicts are treated differently than other mid-pipe requests - instruction mispredicts occur when a branch which is no longer valid is predicted. It is important to correct this problem quickly to prevent the branch from being predicted a second time and incurring the unneeded fetch stalls again. To speed up these requests, the instruction mispredict request is treated as a late-pipe request even though it occurs at mid-pipe. When the request is received, it causes the store buffer state machine to move from Idle to Full. This is possible since the target address normally associated with mid-pipe requests is not needed to execute the request. Once the state machine is Full, the request moves to the BP Table Control buffer in the usual fashion. The special handling of these requests allows them to move through the store buffers in one cycle.

The final aspect of store buffer operation which needs to be discussed is the interaction among them. Store buffers interact in two ways:

1. The order that requests are sent to the BP Table Control buffer for execution - the most recently loaded or newest store buffer request is sent to the BP Table Control buffer first. If a request has already been asserted but has not been accepted by the BP Table Control state machine, it will be de-asserted, and the newer request will

be asserted. The older request may be re-asserted when the newer request is accepted by the BP Table Control state machine. This policy results in getting branches close to the current PC into the BP Table as quickly as possible, thus increasing the probability of these branches being predicted in tight loops and increasing BP efficiency.

2. The order in which sequencer requests are loaded to the buffers, which is governed by several policies:
   a. The first buffer loaded after reset is always Store Buffer 0.
   b. When a store buffer is written and its state goes to Full, the other buffer is selected as the next store buffer to be written to, which results in loads being alternated between the store buffers. The contents of the oldest store buffer will be overwritten if all of the requests have not been executed to the BP Table when a new request is received. This behavior is seen most frequently and occurs when requests are not close together.
   c. Store buffers which have just had their data accepted by the BP Table Control state machine will now be considered empty and will be loaded next. This policy is in effect for only one cycle per request, when the BP Table Control state machine informs the store buffers that it has completed a request and can accept data for the next request. This cycle is typically the cycle after a BP Table write, which is the last cycle of the completed BP Table request. Once this single cycle is completed, control of buffer loading reverts to the policy of alternating buffers. This policy reduces the amount of older store buffer data which is overwritten and dropped, thus increasing BP efficiency. It occurs when branches are spaced close together and a new request has been allowed to move to the BP Table ahead of a request that is waiting. Several new requests may be loaded and executed to the table before an older request has an opportunity to be loaded. Due to this policy, the order and timing of store buffer loads depends on the exact cycle when requests are completed by the BP Table Control state machine. The execution of requests by the BP Table Control state machine is also strongly a function of when instructions are fetched. Since the timing of instruction fetches is highly variable, the net effect is that timing and loading of store buffer requests is also highly variable. This policy can also cause older requests to be delayed a long time before actually being entered into the table.

There is one other case where store buffer loading is affected. This is the first exception case described above, which occurs when a late-pipe request is sent to the BP before or at the same time as the target for a mid-pipe request. As previously noted, the late-pipe request will override the mid-pipe request, which will change the data in the store buffer, move the buffer state to Full, and cause this buffer to be marked as the newest buffer. This will also change the order of buffer requests executed to the BP Table even though it did not change the next buffer which was selected to be loaded.

Loading the most recent requests to the BP Table first is also important in achieving quick table loads when handling instruction mispredict requests. When an instruction mispredict request is received, it is the newest request and is thus sent to the table immediately, which allows the BP to guarantee a one-cycle throughput through the store buffers for this type of request.

The order and timing for loading sequencer requests to the store buffers affects when branch entries enter the BP Table, thus affecting BP predictions and program performance. As has been discussed, the order, timing, and potential overwriting of requests to the store buffers are highly variable. To provide some visibility and debug capability for this process, the Store Buffer Full (ST0FULL and ST1FULL) state bits for each of the store buffer state machines have been brought out to the BP Status register (BP_STAT). These bits are updated every cycle and can be used to observe store buffer operation.

BP Predictions

The BP Table is checked for branch hits each time an instruction fetch is executed. Checks are triggered within the BP in pipe stage “A”, and predictions are available approximately 1 ½ cycles after the instruction fetch is started.

The first step in generating a prediction is to determine if there are branch entries found in the line that is being fetched, and the BP supports predictions for up to two branch entries per line. This requires that the BP also determine where each branch instruction is located within the line. Since each line is 64 bits (eight bytes) and the minimum instruction size is two bytes, the maximum number of instructions per line is four, thus requiring two bits to locate the start of an instruction or instruction offset (SOF) within a line. These bits correspond to bits 2:1 of the instruction address, as the two-byte minimum requirement for an op-code makes bit 0 irrelevant.

When an instruction fetch is evaluated for a prediction, the BP compares the 22-bit TAGs and the two-bit SOFs in both Sets 0 and 1 of the appropriate BP Table row with the corresponding bits in the fetch address. If the TAGS match, the SOF of the fetch address is less than or equal to the SOF of the entry, and the Valid bit is true, then the entry is a branch hit. If the TAGSs match, and the Valid bit is set, but the SOF of the fetch address is greater than the SOF of the entry, then there is no hit.

If there are no branch hits for a given fetch address, then there will be no predictions for that instruction fetch.

If there is a hit in either Set 0 or Set 1 (but not both), then the data from the set which produced the hit will be used to generate the prediction. In this case, a mux on the BP outputs will be switched to send the required SOF, type, and target address data from the set which hit to the sequencer. The final prediction evaluation, however, depends on the prediction value found in the BP Table for the branch entry. If the prediction value is strongly or weakly taken, the hit creates a Taken prediction, and the sequencer is signaled to fetch the TARG value provided by the BP. If the prediction value is strongly or weakly not taken, the hit creates a Not Taken prediction, and the sequencer is not signaled to fetch the target. Not Taken predictions are useful to monitor BP performance but are not sent to the sequencer, so no COF or BP Table updates occur.

A more complex prediction process happens when two hits are found in the BP Table for a given fetch address. When this occurs, the prediction is determined using the SOFs and prediction values for each branch entry. The cases which are possible are discussed below. In each case, the TAGs are assumed to match the fetch address, and the Valid bits are set for both BP Table entries. Branch A may be in Set 0 or 1 with Branch B located in the other set.

1. Fetch SOF <= Branch A SOF < Branch B SOF - the prediction value for Branch A is Strongly or Weakly Taken, and the value for Branch B is a "don’t care". In this case, a Taken prediction will be issued for Branch A, which has the lower offset and will be first in the pipe. Since it is predicted Taken, it will be fetched to avoid the fetch overhead caused by its COF. The presence of and prediction value for Branch B are not factors because Branch B comes after Branch A in the pipe and will be killed when Branch A is executed.
2. Fetch SOF <= Branch A SOF < Branch B SOF - the prediction value for Branch A is Strongly or Weakly Not Taken, and the value for Branch B is Strongly or Weakly Taken. In this case, a Taken prediction will be issued

for Branch B. Branch A has the lower offset, so it will be first in the pipe, but it is not taken; therefore, a prediction and a fetch are not necessary. Branch B will be executed because Branch A is not taken. Since Branch B will be executed and is predicted taken, issuing the prediction for it will be useful in avoiding the fetch overhead caused by its COF.

3. Branch A SOF < Fetch SOF <= Branch B SOF - the prediction value for Branch A is a "don't care", and the value for Branch B is Strongly or Weakly Taken. In this case, a Taken prediction will be issued for Branch B. This happens when a branch with a target address offset that is between that of Branch A and B is executed. Branch A has a lower offset, but it will not be in the pipe and will never be executed, so there is no need to predict it. Branch B is predicted taken and will be executed, so a prediction for Branch B will be useful in avoiding the fetch overhead caused by its COF.
4. Fetch SOF <= Branch A SOF < Branch B SOF - the prediction value for Branches A and B are both Strongly or Weakly Not Taken. In this case, a Not Taken prediction will be issued for Branch A because it has the lower SOF. A prediction will not be created for Branch B because it will be fetched when Branch A is fetched; and, since it is not taken, a fetch and prediction will not be required to load its target.
5. Branch A SOF < Fetch SOF <= Branch B SOF - the prediction values for Branches A and B are both Strongly or Weakly Not Taken. In this case, a Not Taken prediction will be issued for Branch B. The prediction will not be issued for Branch A because the SOF of the target address is greater than the SOF for Branch A. The SOF of the target address is less than or equal to the SOF for Branch B, so Branch B is a valid hit, and a Not Taken prediction is issued for it.
6. Branch A SOF < Branch B SOF < Fetch SOF - the prediction values for Branches A and B are both "don't care". In this situation, no predictions are issued because no branches exist in the line beyond Branch B.

Once the correct set and prediction type has been identified, the BP prediction mux is switched to the appropriate set, and the prediction signals and data are sent to the sequencer, as previously described for the single-hit case.

As presented in the BP RAM Design section, there are four values for the Prediction Code which can be assigned to each Branch Table entry: Strongly Taken, Weakly Taken, Weakly Not Taken, and Strongly Not Taken. When a static branch such as a `CALL` or `RTS` is learned, its prediction value is assigned as Strongly Taken. This value does not change when updates are performed for these entries. When a conditional branch (BRCC) is learned, its initial prediction value is assigned as Weakly Taken. This value is recomputed each time the branch is predicted, and an update is executed for the entry.

Predictions which are predicted Taken and are Taken (not mispredicted) cause the value to move one prediction value towards Strongly Taken. Once the Prediction Code is set to Strongly Taken, it will remain in that state through taken updates until a Mispredict update occurs, which causes the state to move to Weakly Taken.

Predictions which are predicted Taken and are Not Taken (mispredicted) cause the value to move one prediction value towards Strongly Not Taken. Once the Prediction Code is set to Strongly Not Taken, it will remain in that state until a Mispredict update occurs, which causes the state to move to Weakly Not Taken.

Branch predictions are not directly affected by memory stalls. The BP will check for a branch hit each time an instruction is fetched. If a hit is found and a prediction made, the BP will hold the prediction signals and data on its outputs until the next instruction fetch. If the new fetch produces a hit, the prediction signals and data will be changed to reflect the new prediction. If the new fetch does not produce a hit, the control signals are de-asserted to indicate that there is no prediction. The data, however, is a "don't care" and may or may not change depending on the previous and current data fetched from the BP Table. When an instruction fetch occurs and is stalled, the BP will check for a prediction in the first cycle of the fetch. The results of this check are held on the BP outputs through the stall until the next fetch, thus allowing the BP data to be used at the end of the stall when it is required by the sequencer. The only effect that memory stalls have on the BP is that they change the timing of when fetches occur. As discussed, BP efficiency is affected by the number and timing of instruction fetches, so it is possible to see a change in BP operation when there is a high density of memory stalls.

In general, BP predictions will occur whenever there is an instruction fetch, with one notable exception. When the BP makes a prediction, the target address of the branch is fetched in the next cycle. It is possible that the branch fetched may span into the next address, so the consequent address must also be fetched. This additional fetch is referred to as the trailer; and, when this added fetch is required, the policy described above is changed. In this case, when a Taken prediction is made, the target address and the trailer address will always be pre-fetched in the pipeline. Since the trailer address will only be used for branches which span the two addresses and the branch is taken, branches which occur later in the trailer will never be executed. If they will never be executed, there is no need to predict them; therefore, the BP does not check nor make predictions for trailer addresses which are fetched.

The BP is responsible for learning branches and detecting when the branches that it has learned are being fetched, but it is not always responsible for providing the target address of the branch. For `RTS` branches, the target address can come from either an internally-maintained eight-deep call return stack or directly from the `RETS` register.

Speculative Instruction Fetches

The pipeline architecture requires the program sequencer to speculatively fetch instructions that may have to be discarded. A useful example of this operation is the sequence:

```
P1 = 0x80000000;
CC = P0 == P1;
if CC JUMP skip;
CSYNC;
CALL (P0);
skip:
```

Even without the shown `CSYNC` instruction, the sequence is fully functional, but the internal behavior of the processor changes if it is omitted. For example, if `P0` were 0x8000_0000 entering this sequence, the `CALL` instruction would not execute. The presence of the `CSYNC` instruction before it guarantees that the pipe doesn't advance beyond the `CSYNC` instuction, thus no fetches are performed to satisfy the `CALL` instruction's dependency on the `P0` content. However, if the `CSYNC` instruction were removed from this sequence, the instruction fetch from `P0` would still happen to satisfy the `CALL` instruction. Since address 0x8000_0000 resides in DDR memory space, the sequence would attempt to trigger an instruction fetch from that location. If the DDR controller were not yet initialized properly, the conditional instruction fetch could trigger a hardware error; thus, the `CSYNC` instruction is recommended. See the load/store instruction reference pages for details on related data load topics.

Conditional Register Move

Register moves can be performed depending on whether the value of the `ASTAT.CC` status bit is true or false (1 or 0, respectively). In some cases, using this instruction instead of a branch eliminates the cycles lost because of the branch. These conditional moves can be done between any data registers or pointer registers (including `SP` and `FP`). For example:

```
IF CC R0 = P0 ;  /* do register move if CC is TRUE  */
IF !CC P1 = P2 ; /* do register move if CC is FALSE */
```

Hardware Loops

The sequencer supports a mechanism of zero-overhead looping, meaning that there are no cycle penalties when wrapping from the loop bottom to the loop top. The sequencer contains two loop units, each containing three registers. Each loop unit has a Loop Top register (`LT0`, `LT1`), a Loop Bottom register (`LB0`, `LB1`), and a Loop Count register (`LC0`, `LC1`).

Zero-overhead loops are most conveniently written with the `LOOP/LOOP_END` construct. The loop start is marked with a `LOOP`, `LOOPZ` or `LOOPLEZ` instruction, and the end of the loop is marked with a `LOOP_END` pseudo-instruction. The following code example shows a loop that contains two instructions and iterates 32 times.

```
  LOOP LC0 = 32 ;
    R5 = R0 + R1(ns) || R2 = [P2++] || R3 = [I1++] ;
    R5 = R5 + R2 ;
  LOOP_END ;
```

Loops that begin with the `LOOP` instruction decrement and test the counter at the end of the loop, exiting the loop if the decrement results in a count of zero. At least one iteration of the loop is always executed.

The `LOOPZ` and `LOOPLEZ` instructions test the counter before the first iteration of the loop and only execute the first iteration if the counter is initially within range. The `LOOPZ` instruction jumps to the instruction after the `LOOP_END` when the counter is initially zero. The `LOOPLEZ` instruction jumps to the instruction after the `LOOP_END` when the counter is initially less than or equal to zero. When the counter is initially in range, then the `LOOPZ` and `LOOPLEZ` instructions operate in the same way as the `LOOP` instruction. See the `LSETUP` and `LOOP` instruction reference pages for operation details.

The following code shows two loops with an unknown iteration count. In the first loop, the `LOOP` instruction is used, so at least one iteration is executed. In the second loop, the `LOOPZ` instruction is used, so the number of iterations that are executed will match whatever is retrieved from the address pointed to by `P4`.

```
P5 = [P4];          /* Get loop count value from memory location in P4 */
LOOP LC1 = P5;
  /* loop body executed at least once */
LOOP_END;
LOOPZ LC0 = P5;
  /* loop body only executed if count is initially not 0 */
LOOP_END;
```

The assembler translates LOOP, LOOPZ, and LOOPLEZ instructions to LSETUP, LSETUPZ, and LSETUPLEZ instructions, respectively, which contain the PC-relative address of the final instruction in the loop. The LOOP_END pseudo-instruction is simply there to locate the end of the loop, so it does not get translated to any instruction at all. Upon disassembly, the replacement LSETUP-type instruction is seen.

Two sets of zero-overhead loop registers implement loops, using hardware counters instead of software instructions to evaluate loop conditions. After evaluation, processing branches to a new target address. Both sets of registers include the Loop Counter (LC), Loop Top (LT), and Loop Bottom (LB) registers. The Loop Registers table describes the 32-bit loop register sets.

Table 4-7: Loop Registers

Registers	Description	Function
LC0, LC1	Loop Counter	Maintains a count of the remaining iterations of the loop
LT0, LT1	Loop Top	Holds the address of the first instruction within a loop
LB0, LB1	Loop Bottom	Holds the address of the last instruction of the loop

When an instruction at address X is executed, and X matches the contents of LB0, then the next instruction executed will be from the address in LT0. In other words, when PC == LB0, then an implicit jump to LT0 is executed.

The LC0 and LC1 registers are unsigned 32-bit registers, each supporting 2^{32} -1 iterations through the loop.

A loopback only occurs when the count is greater than or equal to two. If the count is non-zero, then the count is decremented by one. For example, consider the case of a loop with two iterations. At the beginning, the count is two. On reaching the first loop end, the count is decremented to one, and the program flow jumps back to the top of the loop (to execute a second time). On reaching the end of the loop again, the count is decremented to zero, but no loopback occurs because the body of the loop has already been executed twice.

The LSETUP, LSETUPZ, or LSETUPLEZ instructions can be used to load all three registers of a loop unit at once. When executing one of these loop setup instructions, the program sequencer loads the address of the loop's last instruction into LBx and the address of the loop's first instruction into LTx. The bottom address of the loop is computed from a PC-relative offset held in the instruction, which limits the maximum loop size to 2046 bytes. It is recommended that the loop top address is the instruction after the loop setup instruction.

Each loop register can also be loaded individually with a register transfer, but this incurs a significant overhead if the loop count is non-zero (the loop is active) at the time of the transfer.

For compatibility with earlier Blackfin processors, the LSETUP instruction without immediate count may contain a start offset. With this form of the instruction, the loop top can be up to 30 bytes after the LSETUP instruction. However, a four-cycle latency occurs on the first loopback if the LSETUP specifies a non-zero start offset.

A legacy form of the LOOP syntax is also supported. Using this syntax, a loop gets assigned a name. All loop instructions are enclosed between the LOOP_BEGIN and LOOP_END brackets.

```
  LC0 = R0;
  LOOP myloop LC0;
/* instructions between setup and body - NOT RECOMMENDED */
  LOOP_BEGIN myloop;
    /* loop body */;
  LOOP_END myloop;
```

The processor supports a four-location instruction loop buffer that reduces instruction fetches while in loops. If the loop code contains four or fewer instructions, then no fetches to instruction memory are necessary for any number of loop iterations because the instructions are stored locally. The loop buffer effectively eliminates the instruction fetch time in loops with more than four instructions by allowing fetches to take place while instructions in the loop buffer are being executed.

The processor has no restrictions regarding which instructions can occur in a loop end position. All instructions, including branches and calls, are allowed at the loop bottom location.

Two-Dimensional Loops

The processor features two Loop Units, each providing its own set of loop registers:

- `LC[1:0]` – the Loop Count registers
- `LT[1:0]` – the Loop Top address registers
- `LB[1:0]` – the Loop Bottom address registers

Therefore, two-dimensional loops are supported directly in hardware, consisting of an outer loop and a nested inner loop.

NOTE: The outer loop is always represented by Loop Unit 0 (`LC0`, `LT0`, `LB0`), while Loop Unit 1 (`LC1`, `LT1`, `LB1`) manages the inner loop.

To enable the two nested loops to end at the same instruction (`LB1` equals `LB0`), Loop Unit 1 is assigned higher priority than Loop Unit 0. A loopback caused by Loop Unit 1 on a particular instruction (`PC==LB1`, `LC1>=2`) will prevent Loop Unit 0 from looping back on that same instruction, even if the address matches. Loop Unit 0 is allowed to loop back only after `LC1` is exhausted. Consequently, when no instructions appear after the inner loop within the outer loop body, the outer loop must use `LC0` while the inner loop uses `LC1`. The following example shows a two-dimensional loop:

```
#define M 32
#define N 1024
  P4 = M (Z);
  P5 = N-1 (Z);
  LOOP LCO = P4;
    R7 = 0 ;
    MNOP || R2 = [I0++] || R3 = [I1++] ;
    LOOP LC1 = P5;
      R5 = R2 + R3 (NS) || R2 = [I0] || R3 = [I1++] ;
      R7 = R5 + R7 (NS) || [I0++] = R5;
    LOOP_END ;
    R5 = R2 + R3 ;
    R7 = R5 + R7 (NS) || [I0++] = R5 ;
    [I2++] = R7 ;
  LOOP_END ;
```

The above example processes an MxN data structure. The inner loop is unrolled and executes N-1 times. The outer loop is not unrolled and still provides room for optimization.

Loop Unrolling

DSP algorithms are typically optimized for speed rather than for small code size. When fetching data from circular buffers, loops are often unrolled in order to pass only N-1 times. The initial data fetch is executed before the loop is entered. Similarly, the final calculations are done after the loop terminates, for example:

```
#define N 1024
global_setup:
/* Initialize DAG registers for 2 circular buffers, 1 in each of Banks A/B */
  I0.H = 0x1180; I0.L = 0x0000; B0 = I0; L0 = N*2 (Z);
  I1.H = 0x1190; I1.L = 0x0000; B1 = I1; L1 = N*2 (Z);
  P5 = N-1 (Z);

algorithm:
  A0 = 0 || R0.H = W[I0++] || R1.L = W[I1++];
  LOOP LC0 = P5;
    A0+= R0.H * R1.L || R0.H = W[I0++] || R1.L = W[I1++];
  LOOP_END;
  A0+= R0.H * R1.L;
```

As shown, the accumulator register is cleared while the first data elements are prefetched before the loop is iterated, then the loop body performs the accumulation while fetching the next elements in a single instruction. This is iterated for the length of the buffer minus one such that the last accumulation occurs after the loop completes. This technique optimizes data fetching to exactly N times, and the `Iregs` are reset to their initial values when processing is complete. As such, the algorithm can subsequently be executed multiple times without any need to re-initialize the DAG registers.

Saving and Resuming Loops

Normally, loops can process and terminate without regard to system-level concepts. Even in the presence of interrupts and exceptions, no special care is needed. There are, however, a few situations that require special attention when a loop gets interrupted by events that require the loop resources themselves:

- If the loop is interrupted by an interrupt service routine that also contains a hardware loop and requires the same loop unit
- If the loop is interrupted by a pre-emptive task switch
- If the loop contains a `CALL` instruction that invokes an unknown subroutine that may have local loops

In scenarios like these, the loop environment can be saved and restored by pushing and popping the loop registers. For example, to save Loop Unit 0 onto the system stack in a function prolog, use this code:

```
[--SP] = LC0;
[--SP] = LB0;
[--SP] = LT0;
```

To pop the loop registers back off the stack, thus restoring them to the state they were in upon executing the above code, the complementary restore code to insert into the function epilog is:

```
LT0 = [SP++];
LB0 = [SP++];
LC0 = [SP++];
```

Writes or pops to the loop registers cause some internal side-effects to re-initialize the loop hardware properly. The hardware does not force the user to save and restore all three loop registers, as there might be cases where saving one or two of them is sufficient. Consequently, every pop instruction in the example above may require the loop hardware to re-initialize again, which takes multiple cycles because the loop buffers must also be pre-filled again.

To avoid unnecessary penalty cycles, the loop hardware follows these rules:

- Restoring `LC0` and `LC1` registers always re-initializes the loop hardware and causes a ten-cycle "replay" penalty.
- Restoring `LT0`, `LT1`, `LB0`, and `LB1` performs in a single cycle, if the corresponding loop counter register is zero.
- If `LCx` is non-zero, every write to the `LTx` and `LBx` registers also attempts to re-initialize the loop hardware and causes a ten-cycle penalty.

In terms of performance, there is a difference depending on the order that the loop registers are popped. For best performance, restore the `LCx` registers last. Furthermore, it is recommended that interrupt service routines and global subroutines that contain hardware loops terminate their local loops cleanly; that is, do not artificially break the loops, and do not execute return instructions within the loops. This guarantees that the `LCx` registers are 0 when `LTx` and `LBx` registers are popped.

Example Code for Using Hardware Loops in an ISR

The following code shows the optimal method of saving and restoring loop registers when using hardware loops in an interrupt service routine:

```
lhandler:
  /* ...Save other registers here... */
  [--SP] = LC0; /* save loop 0 */
  [--SP] = LB0;
  [--SP] = LT0;

  /* ... Handler code here ... */

  LT0 = [SP++];
  LB0 = [SP++];
  LC0 = [SP++];   /* This will cause a "replay" (a ten-cycle refetch) */

  /* ... Restore other registers here ... */

  RTI;
```

If the handler uses Loop Unit 0, it is a good idea to have it leave `LC0=0` at the end. Normally, this happens naturally, as the loop is fully executed. When this is true, then `LT0` and `LB0` restores will not incur additional cycles. If `LC0` is non-zero when these restores occur, each pop will incur the ten-cycle "replay" penalty. Popping or writing `LC0` always incurs this penalty.

Events and Interrupts

The Event Controller of the processor manages five types of events:

- Emulation
- Reset
- Non-Maskable Interrupt (NMI)
- Exception
- Interrupt

NOTE: The word *event* describes all five types of activities shown above that can disrupt application code flow. The Event Controller manages fifteen different events in all, as there are eleven of the Interrupt type that can have dedicated handlers.

An *interrupt* is an event that asynchronously changes normal processor instruction flow. In contrast, an *exception* is a software-initiated event whose effects are synchronous to program flow.

The event system is nested and prioritized. Consequently, several service routines may be active at any time, and a low-priority event may be pre-empted by one of higher priority.

The processor employs a two-level event control mechanism. The Core Event Controller (CEC) works with the System Event Controller (SEC) to prioritize and control all system interrupts. The SEC provides mapping between the many peripheral interrupt sources and the prioritized general-purpose interrupt inputs of the core, which can individually be masked in the SEC. In addition to the dedicated handlers for many events (emulation, reset, NMI, exception, hardware error interrupt, and core timer interrupt), the CEC also supports nine general-purpose interrupts (`IVG7` - `IVG15`). It is recommended that at least the two lowest priority interrupts (`IVG14` and `IVG15`) be reserved for software interrupt handlers, leaving seven prioritized interrupt inputs (`IVG7` - `IVG13`) to support the system.

NOTE: The SEC maps all system events to `IVG11`, thus leaving `IVG7` to `IVG10` and `IVG12` to `IVG15` free for use as software interrupt handlers, with the first group having higher priority than all the system-related interrupts and the second having lower priority. Refer to the Hardware Reference Manual for your processor for a detailed description of the SEC.

The Core Event Mapping table shows the core events and their priority level, as seen by the core, including those controlled by the SEC on `IVG11`. The Core Event Source column is sorted by priority from highest to lowest, such that all the general-purpose interrupts are lower in priority than the rest, and these general-purpose interrupts are also prioritized from `IVG7`(highest) to `IVG15`(lowest).

Table 4-8: Core Event Mapping

Core Event Source	Core Event Name
Emulation (Highest Priority)	EMU
Reset	RST
NMI	NMI
Exception	EVX
Reserved	-
Hardware Error Interrupt	IVHW
Core Timer Interrupt General-Purpose Interrupts	IVTMR IVG7-IVG15 (highest to lowest priority, respectively)

Core Event Controller Registers

The Event Controller uses three core MMRs to coordinate pending event requests. In each of these MMRs, the 16 lower bits correspond to the 16 event levels (for example, bit 0 corresponds to "Emulator mode"). The registers are:

- `IMASK` - interrupt mask
- `ILAT` - interrupt latch
- `IPEND` - interrupts pending

These three registers are accessible in Supervisor mode only.

Interrupt Pending Register (IPEND)

The Core Interrupt Pending register (`IPEND`) keeps track of all currently nested interrupts. When an event is accepted by the core for processing, the corresponding bit in `IPEND` is set, whether the event occurs at the application level or while servicing another event. Because each bit in `IPEND` indicates that the corresponding interrupt is either actively being serviced by the core or is nested at some level awaiting service completion due to a higher-priority event that occurred after it was raised, coupled with the fact that interrupt priority is highest at the LSB of the register, the least significant set bit in `IPEND` indicates the interrupt that is currently being serviced. At any given time, `IPEND` holds the current status of all nested events.

NOTE: The `IPEND[4]` bit is not associated with an event. It is used by the Core Event Controller to temporarily disable interrupts on entry and exit to an interrupt service routine.

This register is read-only and accessible only in Supervisor mode. For more information, see the Interrupt Pending Register .

Interrupt Latch Register (ILAT)

Each set bit in the Core Interrupt Latch register (`ILAT`) indicates that the corresponding event has been latched by the CEC, but it has not yet been accepted into the processor for handling. Once set, the bit will automatically be cleared by hardware before the first instruction in the corresponding ISR is executed. This occurs at the point the interrupt is accepted, where the CEC clears the `ILAT[N]` bit while simultaneously setting the corresponding `IPEND[N]` bit, thus flagging that event to now be pending on the core (either actively being processed or nested at some level).

The `ILAT` register can only be accessed in Supervisor mode. While reads are straightforward, writes to `ILAT` can be used to manually clear latched events for cases where latched interrupt requests need to be cancelled rather than serviced. To clear any `ILAT[N]` bit, first make sure that `IMASK[N] == 0`, then write `ILAT[N] = 1`.

The `RAISE` instruction can be used to set `ILAT[15:5]`and `ILAT[2:1]`.

The `EXCPT` instruction can be used to set `ILAT[3]`.

Only the JTAG $\overline{\text{TRST}}$pin can control `ILAT[0]`.

For more information, see the Interrupt Latch Register .

Interrupt Mask Register (IMASK)

The Core Interrupt Mask register (`IMASK`) controls which interrupt levels are enabled to be serviced by a software handler function. The `IMASK` register may be read and written in Supervisor mode only. While the lowermost bits are hard-coded to always enable servicing of the highest-priority events (emulation, reset, NMI, exception, and hardware error), the upper 11 bits are user-configurable to optionally disable servicing of interrupts from the core timer at `IVG6` to the lowest-priority general-purpose interrupt at `IVG15`. If `IMASK[N] == 1` and `ILAT[N] == 1`, then the vector to service interrupt `N` will be taken if a higher-priority interrupt is not already recognized and in the act of being serviced. If `IMASK[N] == 0` and `ILAT[N]` gets set by interrupt `N`, the interrupt will not be serviced, but `ILAT[N]` will remain set. For more information, see the Interrupt Mask Register .

Event Vector Table (EVT)

The 16-entry Event Vector Table (EVT) is a hardware table comprised of MMRs (`EVT[n]`) containing 32-bit vector addresses to support servicing of core events latched during application run-time. Each `EVT[n]` register can be programmed with the start address of an assigned interrupt service routine associated with a specific event. When the event associated with the `EVT[n]` register occurs, instruction fetches start at the programmed address.

The processor architecture allows for unique addresses to be programmed into each of the interrupt vectors (i.e., interrupt vectors are not determined by a fixed offset from an interrupt vector table base address). This approach minimizes latency by not requiring a long jump from the vector table to the actual ISR code.

The Core Event Vector table lists events by priority, with each event having a corresponding bit in the `ILAT`, `IMASK`, and `IPEND` event state registers.

Table 4-9: Core Event Vector Table

Name	Event Class	Event Vector Register	MMR Location	Notes
EMU	Emulation	EVT0	0x1FE0 2000	Highest priority. Vector address is provided by JTAG.

Table 4-9: Core Event Vector Table (Continued)

Name	Event Class	Event Vector Register	MMR Location	Notes
RST	Reset	EVT1	0x1FE0 2004	RAISE 1 vector. Not used by Reset Control Unit (RCU).
NMI	NMI	EVT2	0x1FE0 2008	
EVX	Exception	EVT3	0x1FE0 200C	
Reserved	Reserved	EVT4	0x1FE0 2010	Reserved.
IVHW	Hardware Error	EVT5	0x1FE0 2014	
IVTMR	Core Timer	EVT6	0x1FE0 2018	
IVG7	GP Interrupt 7	EVT7	0x1FE0 201C	User-Programmable Software/ System Interrupt.
IVG8	GP Interrupt 8	EVT8	0x1FE0 2020	User-Programmable Software/ System Interrupt.
IVG9	GP Interrupt 9	EVT9	0x1FE0 2024	User-Programmable Software/ System Interrupt.
IVG10	GP Interrupt 10	EVT10	0x1FE0 2028	User-Programmable Software/ System Interrupt.
IVG11	GP Interrupt 11	EVT11	0x1FE0 202C	System Interrupt for System Event Controller (SEC).
IVG12	GP Interrupt 12	EVT12	0x1FE0 2030	User-Programmable Software/ System Interrupt.
IVG13	GP Interrupt 13	EVT13	0x1FE0 2034	User-Programmable Software/ System Interrupt.
IVG14	GP Interrupt 14	EVT14	0x1FE0 2038	User-Programmable Software/ System Interrupt.
IVG15	GP Interrupt 15	EVT15	0x1FE0 203C	User-Programmable Software/ System Interrupt.

Return Registers and Instructions

Similar to the RETS register controlled by the CALL and RTS instructions, interrupts and exceptions also use single-entry hardware stack registers. If an interrupt is serviced, the program sequencer saves the return address into the RETI register prior to jumping to the associated event vector contained in the corresponding EVT[n] register. A typical interrupt service routine terminates with an RTI instruction, which causes the sequencer to reload the Program Counter (PC) from the RETI register. The following example shows a simple interrupt service routine:

```
isr:
  [--SP] = (R7:0, P5:0);  /* save core registers to stack */
  [--SP] = ASTAT;         /* save arithmetic status register to stack */

  /* service routine code */
  ASTAT = [SP++];           /* restore arithmetic status register from stack */
  (R7:0, P5:0) = [SP++];    /* restore core registers from stack */
  RTI;                      /* return from interrupt */
isr_end:
```

When interrupt nesting is not enabled, there is no need to manually manage the RETI register, as the application must always return to the application level before recognizing any events that were latched since vectoring to service the interrupt, and the CEC will automatically care for this.

If the service routine must be interruptible by a higher-priority interrupt, nesting of interrupts is required, as is thoughtful management of the RETI register. Reads of the RETI register enable nesting of interrupts, and writes to it disable nesting. Typically, RETI is simply pushed to the stack, which will both save its content and enable nesting of interrupts; similarly, the complementary stack pop operation will disable nesting and restore the content. This scheme enables the service routine to be broken down into both interruptible and non-interruptible sections:

```
isr:
  [--SP] = (R7:0, P5:0);  /* save core registers to stack */
  [--SP] = ASTAT;         /* save arithmetic status register to stack */

  /* critical region (atomic) code */

  [--SP] = RETI;          /* enable nesting */

  /* interruptible service routine code */

  RETI = [SP++];          /* disable nesting */

  /* more critical region (atomic) code */

  ASTAT = [SP++];         /* restore arithmetic status register from stack */
  (R7:0, P5:0) = [SP++]   /* restore core registers from stack */
  RTI;                    /* return from interrupt */
isr.end:
```

NOTE: If there is not a need for non-interruptible code inside the service routine, it is good programming practice to enable nesting immediately by pushing RETI to the stack in the first instruction of the ISR, thus avoiding incurring unnecessary delays before higher-priority interrupt routines can be executed:

See Interrupt Nesting for more details on interrupt nesting.

Emulation events, NMI, and exceptions use a technique similar to that of interrupts described above; however, each has its own return register and return instruction. The Return Registers and Instructions table provides an overview, listing the events in decreasing priority.

NOTE: Unlike the interrupt event that requires manual configuration of interrupt nesting, each of the other event types are enabled by the architecture to pre-empt anything of lower priority (e.g., an NMI event will always interrupt an exception event, an exception event will always interrupt an interrupt event, etc.).

Table 4-10: Return Registers and Instructions

Name	Event Class	Return Register	Return Instruction
EMU	Emulation	RETE	RTE
RST	Reset	-	-
NMI	NMI	RETN	RTN
EVX	Exception	RETX	RTX
Reserved	Reserved	-	-
IVHW	Hardware Error	RETI	RTI
IVTMR	Core Timer	RETI	RTI
IVG7	GP Interrupt 7	RETI	RTI
IVG8	GP Interrupt 8	RETI	RTI
IVG9	GP Interrupt 9	RETI	RTI
IVG10	GP Interrupt 10	RETI	RTI
IVG11	GP Interrupt 11	RETI	RTI
IVG12	GP Interrupt 12	RETI	RTI
IVG13	GP Interrupt 13	RETI	RTI
IVG14	GP Interrupt 14	RETI	RTI
IVG15	GP Interrupt 15	RETI	RTI

Executing RTX, RTN, or RTE in a Lower-Priority Event

The `RTX`, `RTN`, and `RTE` instructions are designed to return from an exception, NMI, or emulator event, respectively. Do not use them to return from a lower-priority event. To return from an interrupt, use the `RTI` instruction. Failure to use the correct instruction will cause incorrect program flow.

Execution of any of these instructions causes the CEC to clear the appropriate `IPEND[n]` bit:

- `RTE` clears `IPEND[0]`
- `RTN`clears `IPEND[2]`
- `RTX` clears `IPEND[3]`
- `RTI` clears the highest-priority set interrupt bit in `IPEND`(IVHW, IVTMR, or IVG[n] bits)

Emulation Interrupt

An emulation event causes the processor to enter Emulation mode, where instructions are read from the JTAG interface. It is the highest priority interrupt to the core.

For detailed information about emulation, see the Debug chapter of the *Blackfin+ Processor Hardware Reference Manual*.

Reset Interrupt

The Reset Control Unit (RCU) controls how the core enters and exits the reset state and supplies the software address to which the core vectors upon exiting it. See the Hardware Reference Manual for a detailed description of the RCU.

Executing the `RAISE 1` instruction does not directly assert a core reset; rather, it simply creates an interrupt with a priority level of one (exceeded only by the emulation interrupt above). The `RAISE 1` instruction also does not save the return address to a register, therefore it is not possible to automatically return from the interrupt vector once it is taken.

Use of the `EVT1` register to provide the vector address for the `RAISE 1` instruction is enabled by clearing bit 15 of the `EVT_OVERRIDE` register. Otherwise, with this bit set, the vector address is supplied directly by the RCU. A reset signalled by the RCU always vectors to the address supplied by the RCU.

In earlier Blackfin processors, the `RAISE 1` instruction caused a software reset. However, it is generally unsafe for a core to transfer control to boot code, which assumes the core and system is coming out of reset, when in fact nothing has been reset. If the former software reset functionality is desired in the Blackfin+ application, the following software control is assumed:

- After booting, the `EVT1` register is programmed with the ISR location for software interrupt level one.
- `EVT_OVERRIDE[15]` is cleared.
- When a `RAISE 1` instruction is executed:
    - The ISR goes through the appropriate mechanisms via the RCU to shut off all core interfaces.
    - The RCU resets the core, seen by the core as an external reset.

Non-Maskable Interrupt (NMI)

The NMI entry at `EVT2` is reserved for non-maskable interrupts, which may be triggered by the system (via the NMIpin or the software watchdog) or by the core (in response to a memory parity error or execution of a `RAISE 2` instruction).

Propagation of an NMI event from sources external to the core is under the control of the System Eevent Controller (SEC, see the Hardware Reference manual for details). The `SEQSTAT.SYSNMI` bit will be set upon vectoring to the NMI handler, if triggered by a source external to the core.

One of the parity error indicator bits in `SEQSTAT` (`PEIC`, `PEDC`, `PEIX` or `PEDX`) will be set on entry to the NMI handler if triggered by a parity error. ***See the Memory chapter for details regarding parity errors.***

If an external NMI event occurs while the processor is already servicing an NMI, reset, or emulation event, an Unhandled NMI Error system interrupt will be triggered, which is handled by the SEC.

If an exception occurs in an event handler that is already servicing an exception, NMI, or reset event, a double-fault system interrupt will be triggered, which is also handled by the SEC. The core stalls until system intervention after a double-fault occurs.

The SEQSTAT.NSPECABT bit will be set upon entry to an NMI, reset, or emulation handler if a non-speculative access such as a system MMR read or a read from I/O device memory was aborted by the event. The read will be attempted again upon returning from the handler, which may not be desired if the read has side-effects (e.g., accessing a FIFO, etc.).

Exceptions

Exceptions are discussed in Hardware Errors and Exception Handling.

Hardware Error Interrupt

Hardware Errors are discussed in Hardware Errors and Exception Handling.

Core Timer Interrupt

The Core Timer Interrupt (IVTMR) is triggered when the core timer value reaches zero. For more information about the core timer, see the Core Timer (TMR) chapter.

General-Purpose Core Interrupts (IVG7-IVG15)

The System Event Controller (SEC) can forward interrupt requests to the core as events IVG15-IVG7, referred to as general-purpose core interrupts. See the Servicing System Interrupts section for more details. By default, the SEC is configured to forward all system interrupts as IVG11, but this is configurable and leaves available eight other general-purpose interrupt vectors for other system or software events.

Software can trigger general-purpose interrupts by using the RAISE instruction. The RAISE instruction can force events for interrupts IVG15-IVG7, IVTMR, and IVHW.

NOTE: It is a useful practice to reserve the two lowest priority interrupts (IVG15 and IVG14) as software interrupt handlers.

Interrupt Processing

The following sections describe interrupt processing.

Globally Enabling/Disabling Interrupts

General-purpose interrupts can be globally disabled using the CLI Dreg instruction and subsequently re-enabled by the STI Dreg instruction, both of which are only available in Supervisor mode. Reset, NMI, emulation, and exception events cannot be globally disabled. Globally disabling interrupts clears the IMASK[15:5] bits after saving IMASK's current state to the desginated register:

```
CLI R5;    /* save IMASK to R5 and disable all maskable interrupts */
/* place critical instructions here */
STI R5;    /* restore IMASK from R5 */
```

When multiple instructions need to be atomic or are too time-critical to be delayed by an interrupt, disable the general-purpose interrupts, but be sure to re-enable them at the conclusion of the code sequence.

Servicing Interrupts

The Core Event Controller (CEC) utilizes the `ILAT` register as a single interrupt queueing element per event. The appropriate `ILAT[n]` bit is set when an interrupt rising edge is detected (which takes two core clock cycles) and cleared when the respective `IPEND[n]` register bit is set. The `IPEND[n]` bit indicates that the event vector has entered the core pipeline. At this point, the CEC recognizes and queues the next rising edge event on the corresponding interrupt input. The minimum latency from the rising edge transition of the general-purpose interrupt to the `IPEND[n]` output assertion is three core clock cycles. However, the latency can be higher, depending on the core's activity level and state.

To determine when to service an interrupt, the controller logically ANDs the three quantities in `ILAT[n]`, `IMASK[n]`, and the current processor priority level.

Servicing the highest priority interrupt involves these actions:

1. The interrupt vector in the Event Vector Table (EVT) becomes the next fetch address. When an interrupt occurs, most instructions currently in the pipeline are aborted. On a Service exception, all instructions after the excepting instruction are aborted. On an Error exception, the excepting instruction and all instructions after it are aborted.
2. The return address is saved in the appropriate return register (`RETI` for interrupts, `RETX` for exceptions, `RETN` for NMIs, and `RETE` for debug emulation). The return address is the address of the instruction after the last instruction executed from normal program flow, except for the case of an Error exception, in which case it is the address of the offending instruction.
3. The processor mode is set to the level of the event taken. If the event is an NMI, exception, or interrupt, the processor mode is Supervisor. If the event is an emulation exception, the processor mode is Emulation.
4. Before the first instruction starts execution, the corresponding interrupt bit in `ILAT` is cleared, and the corresponding bit in `IPEND` is set. Bit `IPEND[4]` is also set to disable all interrupts until the return address in `RETI` is saved.

Interrupt Nesting

If the processor takes a vector to service Event A, Event A becomes "active". In the absence of other events, the processor will complete servicing of the active event and then resume the application at the point the event was serviced. If, however, there are many events to handle and it is desired to handle certain events with higher priority in a timely fashion, nesting of interrupts is required. Interrupt nesting allows the processor to continue to respond to higher-priority events while servicing lower-priority events. In the given example, if the higher-priority Event B occurred while servicing the active Event A, then nesting allows for the processor to immediately respond to Event B. In this case, Event B becomes the active event, and Event A is nested. Several levels of nesting are possible, as described in the Interrupt Pending Register (IPEND) section.

The highest-priority interrupt levels (emulation, reset, NMI, and exception) automatically support interrupt nesting. Each can pre-empt another, provided its priority is higher, and each will pre-empt any of the interrupt events, which are by definition to be lower priority. For example, if an NMI occurs while executing an exception handler, the processor will immediately vector to service the NMI and will pend completion of the exception handler until the NMI event has been fully serviced. However, if an exception were to occur during an NMI handler, the processor would not vector (instead, a double-fault condition will be raised). Conversely, if either an exception or an NMI were to occur while processing an interrupt event, the processor would immediately vector to the appropriate handler, complete servicing of that event, then return to the interrupt handler to complete servicing of the interrupt. But no interrupt event could be configured to interrupt a higher-priority event such as an exception or NMI.

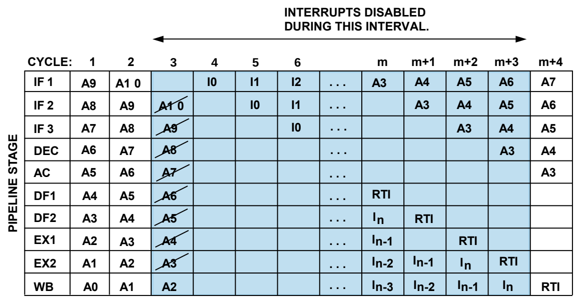
Unlike the higher-priority events that automatically support nesting, the interrupt events themselves can be programmed to optionally support interrupt nesting. For more information, see Return Registers and Instructions.

Non-Nested Interrupts

If interrupts do not require nesting, all interrupts are disabled while the interrupt service routine is executing, thereby gating the servicing of any interrupts that occur after the vector is taken. This restriction does not apply to emulation, NMI, and exception events, which will still be accepted by the system.

When the system does not need to support nested interrupts, there is no need to store the return address held in `RETI`. Only the portion of the machine state used within the interrupt service routine must be saved to the stack. To return from a non-nested interrupt service routine, only the `RTI` instruction must be executed, as the return address is already held in the `RETI` register.

The Non-Nested Interrupt Handling figure shows an example of interrupt handling where interrupts are globally disabled for the entire interrupt service routine.



CYCLE:	1	2	3	4	5	6		m	m+1	m+2	m+3	m+4
IF 1	A9	A1 0		I0	I1	I2	. . .	A3	A4	A5	A6	A7
IF 2	A8	A9	~~A1 0~~		I0	I1	. . .		A3	A4	A5	A6
IF 3	A7	A8	~~A9~~			I0	. . .			A3	A4	A5
DEC	A6	A7	~~A8~~				. . .				A3	A4
AC	A5	A6	~~A7~~				. . .					A3
DF1	A4	A5	~~A6~~				. . .	RTI				
DF2	A3	A4	~~A5~~				. . .	I_n	RTI			
EX1	A2	A3	~~A4~~				. . .	I_{n-1}		RTI		
EX2	A1	A2	~~A3~~				. . .	I_{n-2}	I_{n-1}	I_n	RTI	
WB	A0	A1	A2				. . .	I_{n-3}	I_{n-2}	I_{n-1}	I_n	RTI

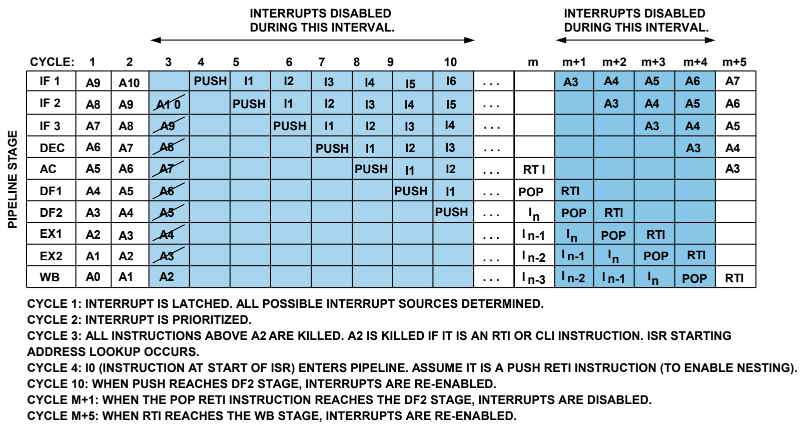
CYCLE 1: INTERRUPT IS LATCHED. ALL POSSIBLE INTERRUPT SOURCES DETERMINED.
CYCLE 2: INTERRUPT IS PRIORITIZED.
CYCLE 3: ALL INSTRUCTIONS ABOVE A2 ARE KILLED. A2 IS KILLED IF IT IS AN RTI OR CLI INSTRUCTION. ISR STARTING ADDRESS LOOKUP OCCURS.
CYCLE 4: I0 (INSTRUCTION AT START OF ISR) ENTERS PIPELINE.
CYCLE M: WHEN THE RTI INSTRUCTION REACHES THE DF1 STAGE, INSTRUCTION A3 IS FETCHED IN PREPARATION FOR RETURNING FROM INTERRUPT.
CYCLE M+4: RTI HAS REACHED WB STAGE, RE-ENABLING INTERRUPTS.

Figure 4-4: Non-Nested Interrupt Handling

Nested Interrupts

If interrupts require nesting, the return address for the currently-being-serviced interrupt (stored to `RETI`when the vector is taken) must be explicitly saved prior to executing the higher-priority ISR and then subsequently restored upon completion of it. Interrupt service routines that support nesting should enable nesting and save the content of the `RETI`register in a single stack push instruction (`[--SP] = RETI`). This clears the global interrupt disable bit `IPEND[4]`, thus enabling interrupts again. After this, all registers that are modified by the interrupt service routine should be saved to the stack. Processor state is stored in the Supervisor stack, not in the User stack; hence, the instructions to push to and pop from the stack use the Supervisor stack.

The Nested Interrupt Handling figure illustrates how pushing `RETI` to the stack re-enables interrupts while in an interrupt service routine, resulting in a short duration where interrupts are globally disabled.


INTERRUPTS DISABLED DURING THIS INTERVAL.
INTERRUPTS DISABLED DURING THIS INTERVAL.
PIPELINE STAGE

CYCLE:	1	2	3	4	5	6	7	8	9	10		m	m+1	m+2	m+3	m+4	m+5
IF 1	A9	A10		PUSH	I1	I2	I3	I4	I5	I6	. . .		A3	A4	A5	A6	A7
IF 2	A8	A9	~~A10~~		PUSH	I1	I2	I3	I4	I5	. . .			A3	A4	A5	A6
IF 3	A7	A8	~~A9~~			PUSH	I1	I2	I3	I4	. . .				A3	A4	A5
DEC	A6	A7	~~A8~~				PUSH	I1	I2	I3	. . .					A3	A4
AC	A5	A6	~~A7~~					PUSH	I1	I2	. . .	RTI					A3
DF1	A4	A5	~~A6~~						PUSH	I1	. . .	POP	RTI				
DF2	A3	A4	~~A5~~							PUSH	. . .	I_n	POP	RTI			
EX1	A2	A3	~~A4~~								. . .	I_{n-1}	I_n	POP	RTI		
EX2	A1	A2	~~A3~~								. . .	I_{n-2}	I_{n-1}	I_n	POP	RTI	
WB	A0	A1	A2								. . .	I_{n-3}	I_{n-2}	I_{n-1}	I_n	POP	RTI

CYCLE 1: INTERRUPT IS LATCHED. ALL POSSIBLE INTERRUPT SOURCES DETERMINED.
CYCLE 2: INTERRUPT IS PRIORITIZED.
CYCLE 3: ALL INSTRUCTIONS ABOVE A2 ARE KILLED. A2 IS KILLED IF IT IS AN RTI OR CLI INSTRUCTION. ISR STARTING ADDRESS LOOKUP OCCURS.
CYCLE 4: I0 (INSTRUCTION AT START OF ISR) ENTERS PIPELINE. ASSUME IT IS A PUSH RETI INSTRUCTION (TO ENABLE NESTING).
CYCLE 10: WHEN PUSH REACHES DF2 STAGE, INTERRUPTS ARE RE-ENABLED.
CYCLE M+1: WHEN THE POP RETI INSTRUCTION REACHES THE DF2 STAGE, INTERRUPTS ARE DISABLED.
CYCLE M+5: WHEN RTI REACHES THE WB STAGE, INTERRUPTS ARE RE-ENABLED.

Figure 4-5: Nested Interrupt Handling

Example Prolog Code for Nested Interrupt Service Routine

```
/* Prolog code for nested interrupt service routine. Push return address in RETI to Supervisor
stack, ensuring that interrupts are back on. When the vector was first taken, interrupts were suspended. */

ISR:
  [--SP] = RETI ; /* Enables interrupts and saves return address to stack */
  [--SP] = ASTAT ;
  [--SP] = (R7:0, P5:0) ; /* Context save */
/* Body of service routine */
```

Example Epilog Code for Nested Interrupt Service Routine

```
/* Epilog code for nested interrupt service routine. Restore ASTAT, Data and Pointer registers.
Popping RETI from Supervisor stack ensures that interrupts are suspended between load of return
address and RTI. */

  (R7:0, P5:0) = [SP++] ;
  ASTAT = [SP++] ;
  RETI  = [SP++] ;
/* Execute RTI, which jumps to return address, re-enables interrupts, and switches to User
mode if this is the last nested interrupt in service. */
```

```
RTI;
```

The `RTI` instruction causes the return from an interrupt. The return address is popped into the `RETI` register from the stack, an action that suspends interrupts from the time that `RETI` is restored until `RTI` finishes executing. The suspension of interrupts prevents a subsequent interrupt from corrupting the `RETI` register.

Next, the `RTI` instruction clears the highest priority bit that is currently set in `IPEND`. The processor then jumps to the address pointed to by the value in the `RETI` register and re-enables interrupts by clearing `IPEND[4]`.

Self-Nesting of Core Interrupts

Interrupts that are "self-nested" can be interrupted by events at the same priority level. When the `SYSCFG.SNEN` bit is set, self-nesting of core interrupts is supported. Self-nesting is supported for any interrupt level generated with the `RAISE` instruction, as well as for core level interrupts.

As an example, assume that the `SNEN` bit is set and the processor is servicing an interrupt generated by the `RAISE 14;` instruction. Once the `RETI` register has been saved to the stack within the service routine, another `RAISE 14;` instruction would force the processor to again vector to the beginning of the `IVG14` interrupt handler. This scheme becomes especially useful and is required when many events can share the same priority level, as is the case with the SEC event being provided by default on `IVG11`. With the SEC monitoring all the system-level events with programmable priority, it has the ability to queue and prioritize the events that are passed to the core, but it continually passes these events at the same `IVG11`priority level. If self-nesting were not supported by the core, each event would have to be serviced to completion before the next could be accepted by the core from the SEC, regardless of the event's programmed priority in the SEC. With self-nesting, the SEC can raise Event A on `IVG11` and continue to monitor for something of a higher priority. And if something with higher priority is raised, it can signal the core to now interrupt servicing of the current `IVG11`event in order to now service the higher-priority system interrupt on the same core interrupt `IVG11`level.

Servicing System Interrupts

System event management is the responsibility of the System Event Controller (SEC). The SEC manages the configuration of each system interrupt or fault source and provides notification and identification of the highest priority active system interrupt request to the core. The SEC must be configured to notify the core of any particular system event. See the hardware reference manual for a detailed description of the SEC.

The SEC prioritizes system interrupts and provides a single interrupt indicator to the core along with an interrupt ID. System interrupts are directed to core interrupt level 11 (`IVG11`). The interrupt ID is latched in the `CEC_SID` register for use in the interrupt service routine. For more information, see Context ID Register .

All the interrupts from the many peripherals and other components of the system are therefore handled by the single ISR for `IVG11`, which interacts with with the SEC, as well as performs the appropriate action to service the source of the interrupt.

The SEC prioritizes system interrupts. It will latch a new `IVG11` interrupt if a higher-priority system interrupt occurs after the current system interrupt has been acknowledged. Self-nesting of core interrupts must be enabled if the higher-priority interrupt detected by the SEC is to be serviced by the core while it is servicing a lower-priority system interrupt. In other words, the `SYSCFG.SNEN` bit must be set.

Device-specific code may be separated from general SEC programming concerns by maintaining a table of routines dedicated to each interrupt source. The `IVG11` ISR is responsible for the following actions:

1. Reading the `SEC_SID` MMR to obtain the interrupt source ID (SID).
2. Writing `SEC_SID` to send the acknowledge signal to the SEC.
3. Calling a device-specific handler.
4. Writing the SEC Global End Register MMR (`SEC_END.SID`) to indicate the interrupt has been serviced.

The following code is an example of an ISR that performs these actions, which illustrates some programming concerns to be aware of:

```
ivg11_sec_isr:
   [--SP] = (P4:P5) ;
   P5 = [CEC_SID] ;
/* Writing any value to CEC_SID sends an ACK to the SEC. After acknowledgement, the value
in CEC_SID will change asynchronously if a higher-priority interrupt becomes active, so the
only reliable record of the current interrupt source is in P5 */

   [CEC_SID] = R0 ;
/* The interrupt source is safely in P5, and the SEC will only interrupt with a higher-
priority notification. So it is safe to re-enable core interrupts by clearing IPEND[4],
which is a side-effect of saving RETI. */

   [--SP] = RETI ;
   [--SP] = RETS ;
/* Use interrupt source to index a table of handlers for each specific interrupt. */

   P4 = specific_handlers;
   P4 = P4 + (P5 << 2);
   P4 = [P4];
   CALL (P4);
/* Assume the handler has preserved the interrupt source in P5. */

   RETS = [SP++] ;
   RETI = [SP++] ;
/* Write interrupt source to SEC Global End register to indicate the core has finished
servicing the interrupt.  The SEC may now raise lower-priority interrupts, so this should
happen with core interrupts disabled via IPEND[4] to prevent the lower-priority interrupt
being serviced before the higher-priority ISR has returned.  Do this after RETI has been
restored, which implicitly sets IPEND[4]. */

   [REG_SEC0_END] = P5 ;
   (P4:5) = [SP++] ;
   RTI ;
```

In practice, use of any device driver or interrupt support routine supplied by a third party will require the `IVG11` ISR code it is designed to work with. Software designed to work with CCES requires the ISR included in the run-time library, which is automatically linked at build-time.

Clearing Interrupt Requests

When the processor services a core event, it automatically clears the requesting bit in the ILAT register and no further action is required by the interrupt service routine.

Interrupts from peripherals are forwarded to the core by the SEC. The first level of the interrupt service routine handles the interaction with the SEC as described in Servicing System Interrupts.

It is important to understand that the SEC does not provide any interrupt acknowledgment feedback to the peripherals. The signalling peripheral does not release its level-sensitive request until it is explicitly instructed by software to do so. If the request is not cleared by software, the peripheral keeps requesting the interrupt, and on receiving the end of interrupt acknowledgement the SEC will immediately re-interrupt the core. This causes the core to vector to the service routine again as soon as the RTI instruction is executed.

Every software routine that services peripheral interrupts must clear the signalling interrupt request in the respective peripheral. The individual peripherals provide customized mechanisms for how to clear interrupt requests. See the hardware reference manual for the details for peripherals in your processor.

Software Interrupts

Software cannot set bits of the ILAT register directly, as writes to ILAT cause a write-1-to-clear (W1C) operation. Instead, use the RAISE instruction to set individual ILAT bits by software. It safely sets any of the ILAT bits without affecting the rest of the register.

```
RAISE 14; /* fire software interrupt request */
```

The RAISE instruction must not be used to fire emulation events or exceptions, which are managed by the related EMUEXCPT and EXCPT instructions. For details, see the external event management chapter.

Often, the RAISE instruction is executed in interrupt service routines to degrade the interrupt priority. This enables less urgent parts of the service routine to be interrupted even by low priority interrupts.

```
isr7:        /* service routine for IVG7 */
...
/* execute high priority instructions here */
/* handshake with signalling peripheral */
RAISE 14;
RTI;
isr7.end:
isr14:     /* service routine for IVG14 */
...
/* further process event initiated by IVG7 */
RTI;
isr14.end:
```

The example above may read data from any receiving interface, post it to a queue, and let the lower priority service routine process the queue after the isr7 routine returns. Since IVG15 is used for normal program execution in non-multi-tasking system, IVG14 is often dedicated to software interrupt purposes.

The code in Example Code for an Exception Handler uses the same principle to handle an exception with normal interrupt priority level.

Latency in Servicing Events

In some processor architectures, if instructions are executed from external memory and an interrupt occurs while the instruction fetch operation is underway, then the interrupt is held off from being serviced until the current fetch operation has completed. Consider a processor operating at 300 MHz and executing code from external memory with 100 ns access times. Depending on when the interrupt occurs in the instruction fetch operation, the interrupt service routine may be held off for around 30 instruction clock cycles. When cache line fill operations are taken into account, the interrupt service routine could be held off for many hundreds of cycles.

In order for high priority interrupts to be serviced with the least latency possible, the processor allows any high latency fill operation to be completed at the system level, while an interrupt service routine executes from L1 memory. See the figure.

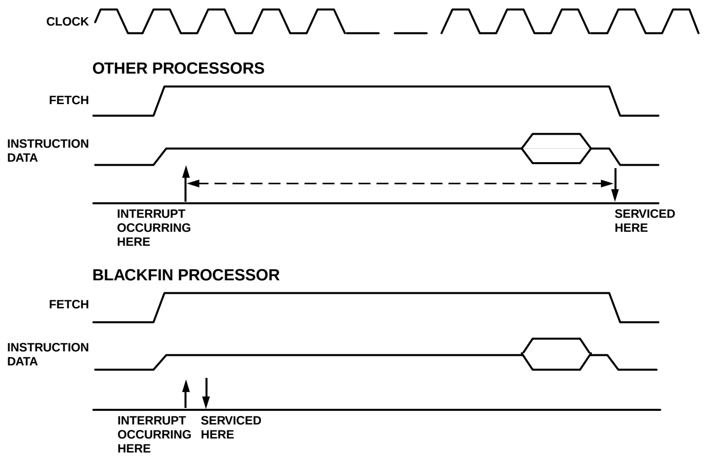


Figure 4-6: Minimizing Latency in Servicing an ISR

If an instruction load operation misses the L1 instruction cache and generates a high latency line fill operation, then when an interrupt occurs, it is not held off until the fill has completed. Instead, the processor executes the interrupt service routine in its new context, and the cache fill operation completes in the background.

Note the interrupt service routine must reside in L1 cache or SRAM memory and must not generate a cache miss, an L2 memory access, or a peripheral access, as the processor is already busy completing the original cache line fill operation. If a load or store operation is executed in the interrupt service routine requiring one of these accesses, then the interrupt service routine is held off while the original external access is completed, before initiating the new load or store.

If the interrupt service routine finishes execution before the load operation has completed, then the processor continues to stall, waiting for the fill to complete.

This same behavior is also exhibited for stalls involving reads of slow data memory or peripherals.

Writes to slow memory generally do not show this behavior, as the writes are deemed to be single cycle, being immediately transferred to the write buffer for subsequent execution.

For detailed information about cache and memory structures, see the memory chapter .

Hardware Errors and Exception Handling

The following sections describe hardware errors and exception handling.

SEQSTAT Register

The Sequencer Status register (SEQSTAT) contains information about the current state of the sequencer as well as diagnostic information from the last event. SEQSTAT is accessible only in Supervisor mode. For more information, see the Sequencer Status Register .

Hardware Error Interrupt

A hardware error interrupt indicates a hardware error or system malfunction. A core-related external hardware errors occurs when logic external to the core, such as a memory bus controller, is unable to complete a data transfer (read or write) initiated by the core and invokes a core hardware error interrupt on the SEC. This interrupt may be serviced by the core as described in the Servicing System Interrupts section.

An internal hardware error is generated by internal error conditions within the core, such as Performance Monitor overflow Such hardware errors invoke the internal hardware error interrupt (interrupt IVHW in the event vector table (EVT) and ILAT, IMASK, and IPEND registers). The hardware error interrupt service routine can then read the cause of the error from the 5-bit HWERRCAUSE field appearing in the sequencer status register (SEQSTAT) and respond accordingly.

The list of supported hardware conditions, with their related HWERRCAUSE codes, appears in the hardware conditions table. The bit code for the most recent error appears in the HWERRCAUSE field. If multiple hardware errors occur simultaneously, only the last one can be recognized and serviced. The core does not support prioritizing, pipelining, or queuing multiple error codes. The Hardware Error Interrupt remains active as long as any of the error conditions remain active.

Note that a hardware error status cannot be cleared by software.

In case of hardware error, the RETI does not store the address of the instruction that caused the hardware error. The error could have been caused by an instruction executed a number of core clock cycles before a hardware error is registered.

Table 4-11: Hardware Conditions Causing Hardware Error Interrupts

Hardware Condition	HWERRCAUSE (Binary)	HWERRCAUSE (Hexadecimal)	Notes / Examples
Performance Monitor Overflow	0b10010	0x12	Refer to the performance monitor unit section of the processor hardware reference.
RAISE 5 instruction	0b11000	0x18	Software issued a RAISE 5 instruction to invoke the Hardware Error Interrupt (IVHW).
Reserved	All other bit combinations.	All other values.	

Exceptions (Events)

Exceptions are synchronous to the instruction stream. In other words, a particular instruction causes an exception when it attempts to finish execution. No instructions after the offending instruction are executed before the exception handler takes effect.

Many of the exceptions are memory related. For example, an exception is given when a cacheability protection lookaside buffer (CPLB) miss or protection violation occurs. Exceptions are also given when illegal instructions or illegal combinations of registers are executed.

An excepting instruction may or may not commit before the exception event is taken, depending on if it is a service type or an error type exception.

An instruction causing a service type event will commit, and the address written to the `RETX` register will be the next instruction after the excepting one. An example of a service type exception is the single step.

An instruction causing an error type event cannot commit, so the address written to the `RETX` register will be the address of the offending instruction. An example of an error type event is a CPLB miss.

NOTE: Usually the `RETX` register contains the correct address to return to. To skip over an excepting instruction, take care in case the next address is not simply the next linear address. This could happen when the excepting instruction is a loop end. In that case, the proper next address would be the loop top.

The `EXCAUSE[5:0]` field in the Sequencer Status register (`SEQSTAT`) is written whenever an exception is taken, and indicates to the exception handler which type of exception occurred. Refer to the events table for a list of events that cause exceptions.

ATTENTION: If an exception occurs in an event handler that is already servicing an exception, `NMI`, reset, or emulation event, this will trigger a double fault condition, and the address of the excepting instruction will be written to `RETX`.

Table 4-12: Events That Cause Exceptions

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service See Note 1.	Notes/Examples
Force Exception instruction EXCPT with 4-bit m field	m field	S	Instruction provides 4 bits of EXCAUSE.
Single step	0x10	S	When the processor is in single step mode, every instruction generates an exception. Primarily used for debugging.
Undefined instruction	0x21	E	May be used to emulate instructions that are not defined for a particular processor implementation.
Illegal instruction combination	0x22	E	See section for multi-issue rules in the *Blackfin Processor Programming Reference*.
Data access CPLB protection violation	0x23	E	Attempted read or write to Supervisor resource, or illegal data memory access. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. (A simultaneous, dual access to two MMRs using the data address generators generates this type of exception.) In addition, this entry is used to signal a protection violation caused by disallowed memory access, and it is defined by the Memory Management Unit (MMU) cacheability protection lookaside buffer (CPLB).
Data access misaligned address violation	0x24	E	Attempted misaligned I/O or test data access.
Unrecoverable event	0x25	E	For example, an exception generated while processing a previous exception.
Data access CPLB miss	0x26	E	Used by the MMU to signal a CPLB miss on a data access.
Data access multiple CPLB hits	0x27	E	More than one CPLB entry matches data fetch address.
Exception caused by an emulation watchpoint match	0x28	E	There is a watchpoint match, and one of the EMUSW bits in the Watchpoint Instruction Address Control register (WPIACTL) is set.
Instruction fetch misaligned address violation	0x2A	E	Attempted misaligned instruction cache fetch. (Note this exception can never be generated from PC-relative branches, only from indirect branches.)

Table 4-12: Events That Cause Exceptions (Continued)

Exception	EXCAUSE [5:0]	Type: (E) Error (S) Service See Note 1.	Notes/Examples
Instruction fetch CPLB protection violation	0x2B	E	Illegal instruction fetch access (memory protection violation).
Instruction fetch CPLB miss	0x2C	E	CPLB miss on an instruction fetch.
Instruction fetch multiple CPLB hits	0x2D	E	More than one CPLB entry matches instruction fetch address.
Illegal use of supervisor resource	0x2E	E	Attempted to use a Supervisor register or instruction from User mode. Supervisor resources are registers and instructions that are reserved for Supervisor use: Supervisor only registers, all MMRs, and Supervisor only instructions. This error code is also used for errors that do not fit into any other category.

NOTE: *(1)* For services (S), the return address is the address of the instruction that follows the exception. For errors (E), the return address is the address of the excepting instruction.

If an instruction causes multiple exception, the exception with the highest priority is first registered in the SEQSTAT. The exception priority is as listed in the exceptions by priority table. If the highest priority exception is handled, the next highest priority exception is registered and can be handled (and so on).

For example, suppose that the following instruction generates an instruction CPLB miss (0x2C) exception and a data CPLB miss (0x26) exception. On execution of this instruction, a instruction CPLB will be first generated. After this instruction exception is handled by the user the core will execute the instruction again and this time it will generate a data CPLB exception.

```
[P0] = R0 ;
/* generates an instruction CPLB miss and a data CPLB miss */
```

Table 4-13: Exceptions by Descending Priority

Priority	Exception	EXCAUSE
1	Unrecoverable Event	0x25
2	I-Fetch Multiple CPLB Hits	0x2D
3	I-Fetch Misaligned Access	0x2A
4	I-Fetch Protection Violation	0x2B
5	I-Fetch CPLB Miss	0x2C
6	I-Fetch Access Exception	0x29

Table 4-13: Exceptions by Descending Priority (Continued)

Priority	Exception	EXCAUSE
7	Watchpoint Match	0x28
8	Undefined Instruction	0x21
9	Illegal Combination	0x22
10	Illegal Use of Protected Resource	0x2E
11	DAG0 Multiple CPLB Hits	0x27
12	DAG0 Misaligned Access	0x24
13	DAG0 Protection Violation	0x23
14	DAG0 CPLB Miss	0x26
15	DAG1 Multiple CPLB Hits	0x27
16	DAG1 Misaligned Access	0x24
17	DAG1 Protection Violation	0x23
18	DAG1 CPLB Miss	0x26
19	EXCPT Instruction	m field
20	Single Step	0x10

Exceptions While Executing an Exception Handler

While executing the exception handler, avoid issuing an instruction that generates another exception. If an exception is caused while executing code within the exception handler, the NMI handler, or the reset vector:

- A double fault interrupt is sent to the SEC.
- The excepting instruction is not committed. All writebacks from the instruction are prevented.
- The `EXCAUSE` field in the `SEQSTAT` register is set to `0x25` (Unrecoverable Event) and `RETX` is updated with the address of the faulting instruction.
- The generated exception is not taken, and `PC` is not advanced.
- The core continues to fetch the same instruction and convert it to a nop until the exception goes away, or a higher priority interrupt takes over, or reset is asserted by the system.

In practice the only sensible means of recovery is to configure the SEC to reset the core on receipt of a double fault interrupt.

The `SEQSTAT` and `RETX` registers can be inspected within a debugger to diagnose the problem.

Allocating the System Stack

The software stack model for processing exceptions implies that the Supervisor stack must never generate an exception while the exception handler is saving its state. However, if the Supervisor stack is in cached or protected memory it may, in fact, cause CPLB miss exceptions.

One way to guarantee that the Supervisor stack never generates an exception is by calculating the maximum space that all interrupt service routines and the exception handler occupy while they are active, and then ensuring active CPLB entries cover this amount of memory. This may not be practical if the space required for the stack is large as only a limited number of CPLB entries may be active at once.

Another option is to provide a separate stack for the exception handler, as it is easier to calculate the total space required for this stack and to ensure it is covered by a single CPLB entry. The following code illustrates switching to a dedicated exception stack.

Switching stack within an exception handler

```
  .SECTION L1_scratchpad;
  .ALIGN 4;
  .VAR except_save_sp, except_stack[EXCEPT_STACK_SIZE];

  .SECTION L1_code;
except_handler:
  [ except_save_sp ] = SP;    /* save stack pointer */
  SP = except_stack+EXCEPT_STACK_SIZE;
  /* now safe to save registers in except_stack */
  [--SP] = (R7:6, P5:4);
  [--SP] = ASTAT;
  /* place core of service routine here */
  ASTAT = [SP++];
  (R7:6, P5:4) = [SP++];
  /* restore stack pointer before return */
  SP = [ except_save_sp ];
  RTX;
except_handler.end:
```

Similar considerations apply to parity error handlers. If the system stack could be in memory that caused the parity error then it is necessary to switch to a stack in a different memory region, such as ECC protected L2 memory, before saving any registers or risk a double parity error fault.

Exceptions and the Pipeline

Interrupts and exceptions treat instructions in the pipeline differently.

- When an interrupt occurs, all instructions in the pipeline are aborted.
- When an exception occurs, all instructions in the pipeline after the excepting instruction are aborted. For error exceptions, the excepting instruction is also aborted.

Because exceptions, NMIs, and emulation events have a dedicated return register, guarding the return address is optional. Consequently, the `PUSH` and `POP` instructions for exceptions, NMIs, and emulation events do not affect the interrupt system.

Note, however, the return instructions for exceptions (`RTX`, `RTN`, and `RTE`) do clear the Least Significant Bit (LSB) currently set in `IPEND`.

Deferring Exception Processing

Exception handlers are usually long routines, because they must discriminate among several exception causes and take corrective action accordingly. The length of the routines may result in long periods during which the interrupt system is, in effect, suspended.

To avoid lengthy suspension of interrupts, write the exception handler to identify the exception cause, but defer the processing to a low priority interrupt. To set up the low priority interrupt handler, use the Force Interrupt / Reset instruction (`RAISE`).

NOTE: When deferring the processing of an exception to lower priority interrupt `IVGx`, the system must guarantee that `IVGx` is entered before returning to the application-level code that issued the exception. If a pending interrupt of higher priority than `IVGx` occurs, it is acceptable to enter the high priority interrupt before `IVGx`.

Example Code for an Exception Handler

The following code is for an exception routine handler with deferred processing.

Exception Routine Handler With Deferred Processing

```
/* Determine exception cause by examining EXCAUSE field in SEQSTAT (first save contents of
R7, P5, P4 and ASTAT in Supervisor SP on a private stack.) */
.SECTION L1_scratchpad ;
#define EXCEPT_STACK_SZ 4
.VAR EXCEPT_SAVED_SP, EXCEPT_STACK[EXCEPT_STAK_SZ] ;
.SECTION L1_code;
except_handler:
[EXCEPT_SAVED_SP] = SP ;
SP = EXCEPT_STACK+EXCEPT_STACK_SZ ;
[--SP] = (R7,P5:4) ;
[--SP] = ASTAT ;
R7 = SEQSTAT ;
/* Mask the contents of SEQSTAT, and leave only EXCAUSE in R0 */
R7 <<= 26 ;
R7 >>= 26 ;
/* Using jump table EVTABLE, jump to the event pointed to by R0 */
P5 = R7 ;
P4 = _EVTABLE ;
P5 = P4 + ( P5 << 1 ) ;
R7 = W [ P5 ] (Z) ;
P4 = R0 ;
JUMP (PC + P4) ;
```

```
/* The entry point for an event is as follows. Here, processing is deferred to low priority
interrupt IVG12. Also, parameter passing would typically be done here. */
_EVENT1:
RAISE 12 ;
JUMP.S _EXIT ;
/* Entry for event at IVG13 */
_EVENT2:
RAISE 13 ;
JUMP.S _EXIT ;
/* Comments for other events */
/* At the end of handler, restore R7, P5, P1 and ASTAT, and return. */
_EXIT:
ASTAT = [SP++] ;
(R7,P5:4) = [SP++] ;
SP = [EXCEPT_SAVED_SP] ;
RTX ;
_EVTABLE:
.byte2 addr_event1;
.byte2 addr_event2;
...
.byte2 addr_eventN;
/* The jump table EVTABLE holds 16-bit address offsets for each event. With offsets, this
code is position independent and the table is small.
+--------------+
| addr_event1  | _EVTABLE
+--------------+
| addr_event2  | _EVTABLE + 2
+--------------+
|     . . .    |
+--------------+
| addr_eventN  | _EVTABLE + 2N
+--------------+
*/
```

Example Code for an Exception Routine

The following code provides an example framework for an interrupt routine jumped to from an exception handler such as that described above.

Interrupt Routine for Handling Exception

```
[--SP] = RETI ;   /* Push return address on stack. */

/* Put body of routine here.*/

RETI = [SP++] ;   /* To return, pop return address and jump. */

RTI ;   /* Return from interrupt. */
```

ADSP-BF70x Sequencer-Related Register Descriptions

The Sequencer-Related Register File contains the following registers.

Table 4-14: ADSP-BF70x Sequencer-Related Register List

Name	**Description**
`RETS`	Return from Subroutine Register
`LC[n]`	Loop Count Register
`LT[n]`	Loop Top Register
`LB[n]`	Loop Bottom Register
`SEQSTAT`	Sequencer Status Register
`RETI`	Return from Interrupt Register
`RETX`	Return from Exception Register
`RETN`	Return from NMI Register
`RETE`	Return from Emulator Register

Sequencer Status Register

The SEQSTAT register contains information about the current state of the sequencer and diagnostic information from the most recent event. This register is read-only (except for the W1C status bits) and accessible only in Supervisor mode.

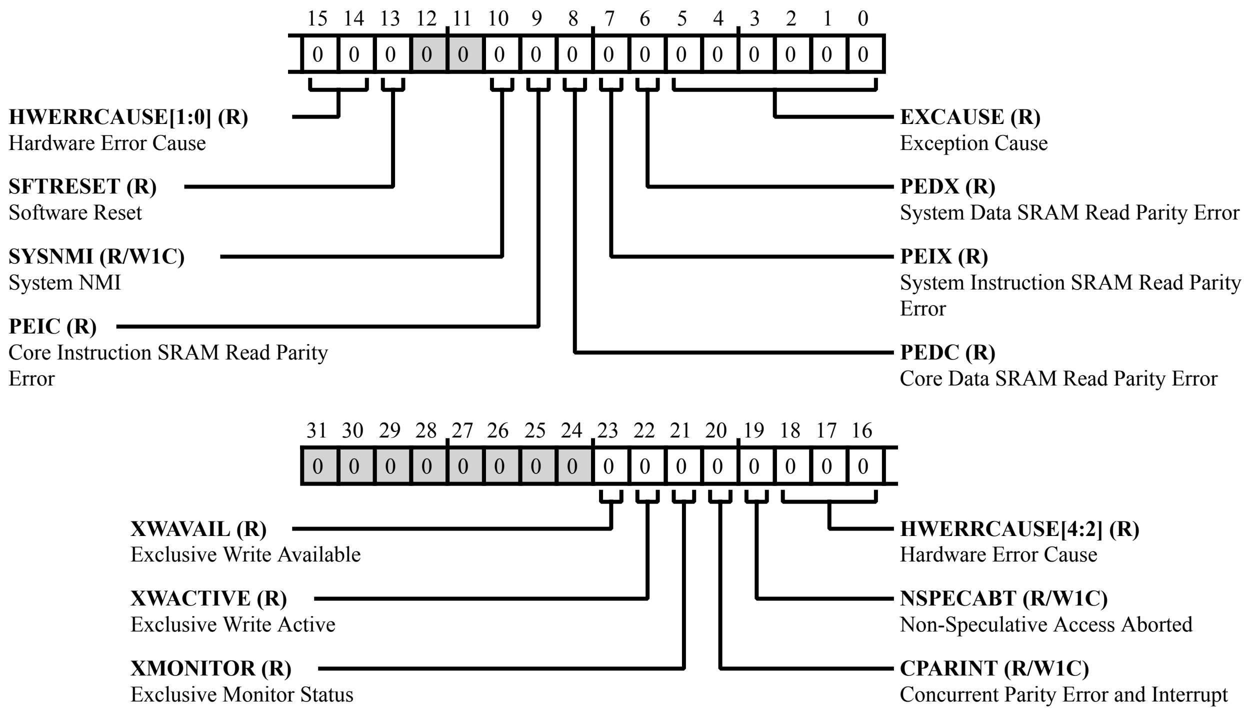


Figure 4-7: SEQSTAT Register Diagram

Table 4-15: SEQSTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
23 (R/NW)	XWAVAIL	Exclusive Write Available. The SEQSTAT.XWAVAIL bit indicates whether an exclusive write response is available.	
		0	No status
		1	Write response available
22 (R/NW)	XWACTIVE	Exclusive Write Active. The SEQSTAT.XWACTIVE bit indicates whether an exclusive write is currently active.	
		0	No status
		1	Write currently active

Table 4-15: SEQSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
21 (R/NW)	XMONITOR	Exclusive Monitor Status. The `SEQSTAT.XMONITOR` bit indicates monitors for exclusive or open access.	
		0	Open access
		1	Exclusive access
20 (R/W1C)	CPARINT	Concurrent Parity Error and Interrupt. The `SEQSTAT.CPARINT` bit indicates that a parity error could not suppress a concurrent interrupt. If the concurrent interrupt was an NMI or had higher priority, the parity error may have occurred on the return address of the current interrupt service routine. If the concurrent interrupt had lower priority than an NMI, the parity error may have occurred on the return address of previous interrupt service routine.	
		0	No status
		1	Error and interrupt occurred
19 (R/W1C)	NSPECABT	Non-Speculative Access Aborted. The `SEQSTAT.NSPECABT` bit indicates indicates that a non-speculative access was interrupted by an NMI or was interrupted by a hardware emulator interrupt. If the access was to a resource that has read side-effects (e.g., a FIFO), data from the access may have been lost.	
		0	No status
		1	Access aborted
18:14 (R/NW)	HWERRCAUSE	Hardware Error Cause. The `SEQSTAT.HWERRCAUSE` bits hold the encoding for the cause of the last hardware error generated by the processor core.	
		18	Performance Monitor count overflowed
		24	Core executed the RAISE 5; instruction
13 (R/NW)	SFTRESET	Software Reset. The `SEQSTAT.SFTRESET` bit indicates whether or not the most recent processor reset was a software reset.	
		0	Not software reset
		1	Software reset occurred
10 (R/W1C)	SYSNMI	System NMI. The `SEQSTAT.SYSNMI` bit indicates whether or not the system NMI input is active.	
		0	No status
		1	NMI active

Table 4-15: SEQSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
9 (R/NW)	PEIC	Core Instruction SRAM Read Parity Error. The `SEQSTAT.PEIC` bit indicates a parity error on an L1 instruction memory read by the processor.	
		0	No status
		1	Parity error occurred
8 (R/NW)	PEDC	Core Data SRAM Read Parity Error. The `SEQSTAT.PEDC` bit indicates a parity error on an L1 data memory read by the processor.	
		0	No status
		1	Parity error occurred
7 (R/NW)	PEIX	System Instruction SRAM Read Parity Error. The `SEQSTAT.PEIX` bit indicates a parity error on an L1 instruction read by the system, such as a DMA transfer out of L1 instruction memory.	
		0	No status
		1	Parity error occurred
6 (R/NW)	PEDX	System Data SRAM Read Parity Error. The `SEQSTAT.PEDX` bit indicates a parity error on an L1 data memory read by the system, such as a cache write back or a DMA transfer out of L1 data memory.	
		0	No status
		1	Parity error occurred
5:0 (R/NW)	EXCAUSE	Exception Cause. The `SEQSTAT.EXCAUSE` bits hold a value, which indicates the cause of the most recent exception.	
		0-15	Core executed EXCPT m; instruction (m = 0-15)
		16	Supervisor single-step
		17	Emulator trace buffer overflow
		33	Undefined instruction
		34	Illegal combination of instructions
		35	DAG protection violation
		36	DAG misaligned access
		37	Unrecoverable event
		38	DAG CPLB miss
		39	DAG multiple CPLB hits

Table 4-15: SEQSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		40	Emulator watchpoint match
		42	I-fetch misaligned access
		43	I-fetch protection violation
		44	I-fetch CPLB miss
		45	I-fetch multiple CPLB hits
		46	Protection violation, illegal use of Supervisor resource

Return from Subroutine Register

The CALL instruction calls a subroutine either indirectly through one of the P[n] registers or directly using a PC-relative offset. When the CALL instruction executes, the address of the next contiguous instruction is saved to the RETS register such that program execution can resume when the subroutine executes its return instruction (RTS).

The RETS register is not a memory-mapped register, but it is directly accessible using move instructions and can be pushed to or popped from the system stack; however, the RETS register cannot be used as the source or destination register for load/store or immediate load operations.

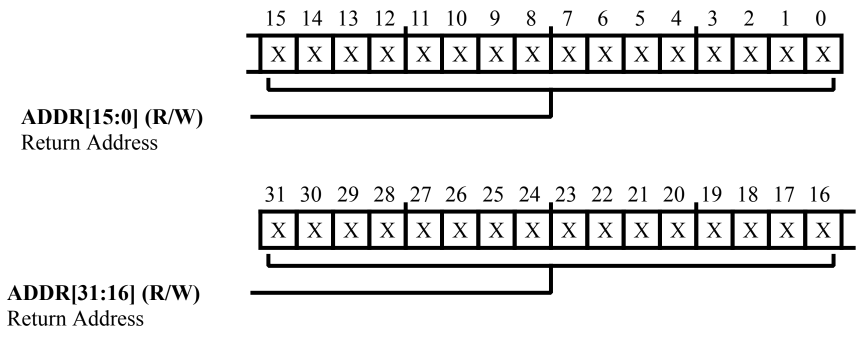


Figure 4-8: RETS Register Diagram

Table 4-16: RETS Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETS.ADDR bits hold the address of the instruction to return to when the RTS instruction is executed.

Return from Interrupt Register

When the processor vectors to a handler to service an interrupt, it saves the address of the last instruction to execute to the RETI register such that program execution can resume when the return from interrupt instruction (RTI) is executed at the end of the interrupt service routine.

The RETI register is not a memory-mapped register, but it is directly accessible using move instructions; however, it cannot be used as the source or destination register for load/store or immediate load operations.

The RETI register can also be pushed to or popped from the system stack. When RETI is pushed to the system stack, interrupt nesting is enabled. When RETI is popped from the system stack, interrupt nesting is disabled.

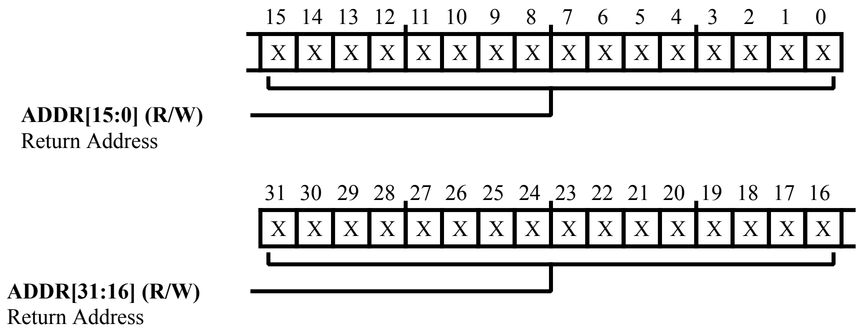


Figure 4-9: RETI Register Diagram

Table 4-17: RETI Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETI.ADDR bits hold the address of the instruction to return to when the RTI instruction is executed.

Return from Exception Register

When the processor vectors to a handler function in response to an exception, it saves either the address of the excepting instruction (for Error exceptions) or the address of the following contiguous instruction after the excepting instruction (for Service exceptions) to the RETX register such that program execution can resume when the return from exception (RTX) instruction is executed at the end of the exception handler routine.

The RETX register is not a memory-mapped register, but it is directly accessible using move instructions and can be pushed to or popped from the system stack; however, the RETX register cannot be used as the source or destination register for load/store or immediate load operations.

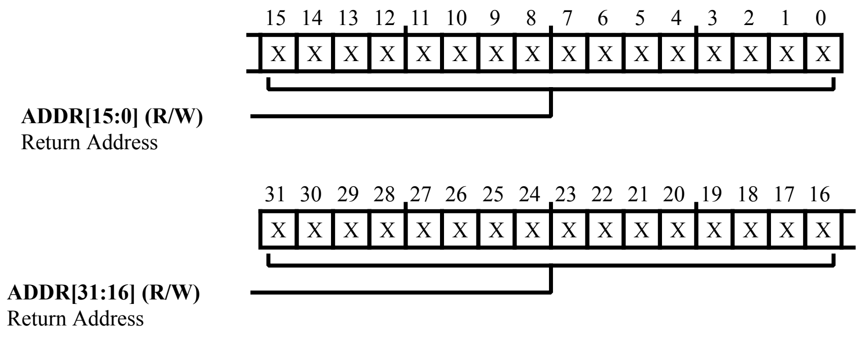


Figure 4-10: RETX Register Diagram

Table 4-18: RETX Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETX.ADDR bits hold the address of the instruction to return to when the RTX instruction is executed.

Return from NMI Register

When the processor vectors to a handler function to service a non-maskable interrupt (NMI) event, it saves the address of the last executed instruction to the `RETN` register such that program execution can resume when the return from NMI (RTN) instruction is executed at the end of the NMI handler routine.

The `RETN` register is not a memory-mapped register, but it is directly accessible using move instructions and can be pushed to or popped from the system stack; however, the `RETN` register cannot be used as the source or destination register for load/store or immediate load operations.

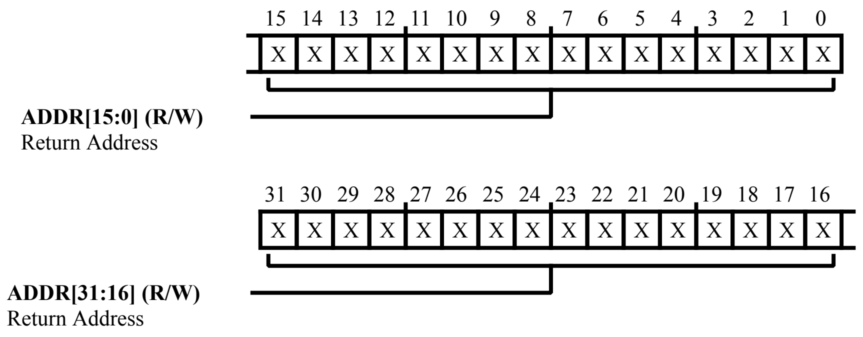


Figure 4-11: RETN Register Diagram

Table 4-19: RETN Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The `RETN.ADDR` bits hold the address of the instruction to return to when the RTN instruction is executed.

Return from Emulator Register

When the processor vectors to handle a hardware emulator event, it saves the address of the last executed instruction to the RETE register such that program execution can resume when the return from emulator instruction (RTE) is executed.

The RETE register is not a memory-mapped register, but it is directly accessible using move instructions and can be pushed to or popped from the system stack; however, the RETE register cannot be used as the source or destination register for load/store or immediate load operations.

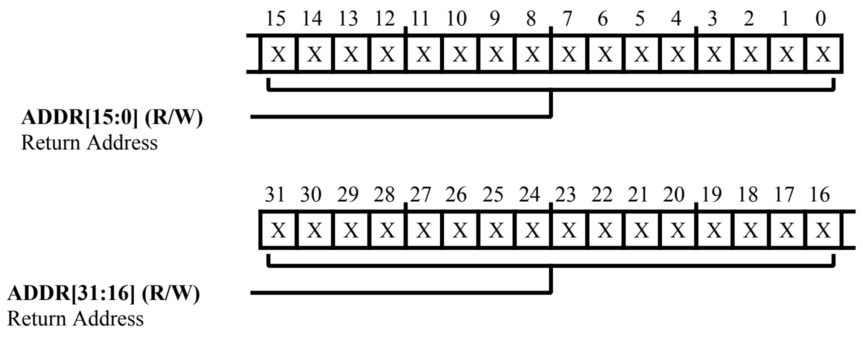


Figure 4-12: RETE Register Diagram

Table 4-20: RETE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Return Address. The RETE.ADDR bits hold the address of the instruction to return to when the RTE instruction is executed.

Loop Top Register

The `LT[n]` register is initialized via the loop setup (LSETUP) instruction and holds the address of the first instruction in a hardware loop. This is the address that program execution automatically continues to when the corresponding loop bottom (`LB[n]`) instruction has executed and the corresponding loop counter (`LC[n]`) has not expired.

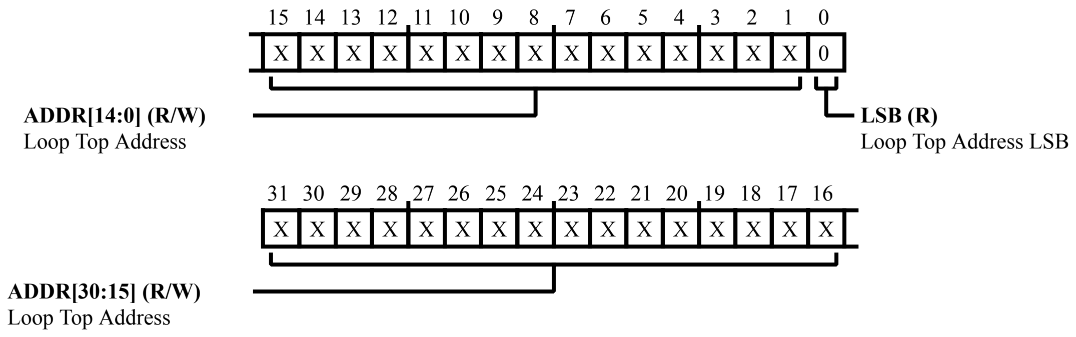


Figure 4-13: LT[n] Register Diagram

Table 4-21: LT[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:1 (R/W)	ADDR	Loop Top Address. The `LT[n].ADDR` bits hold the address of the first instruction in the hardware loop.
0 (R/NW)	LSB	Loop Top Address LSB. The `LT[n].LSB` bit is the LSB of the address of the first instruction in the hardware loop.

Loop Bottom Register

The LB[n] register is initialized via the loop setup (LSETUP) instruction and holds the address of the last instruction in a hardware loop. When the instruction at this address is executed, the corresponding loop counter (LC[n]) is decremented and checked for expiration. If it has not expired, program execution resumes at the address of the corresponding loop top (LT[n]), otherwise the loop is exited and program execution continues to the next contiguous instruction after the loop bottom.

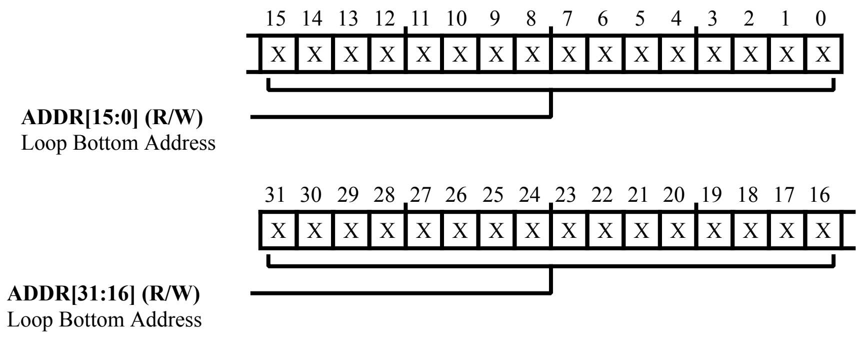


Figure 4-14: LB[n] Register Diagram

Table 4-22: LB[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Loop Bottom Address. The LB[n].ADDR bits hold the address of the last instruction in the hardware loop.

Loop Count Register

The LC[n] register is initialized by the loop setup (LSETUP) instruction and depicts the number of remaining iterations for the currently executing hardware loop. It is decremented after the corresponding loop bottom LB[n] instruction is executed and checked for expiration by hardware. If it is non-zero, program execution resumes at the corresponding loop top LT[n] address. When LC[n] expires, the loop exits and program execution resumes at the next contiguous instruction after the loop's bottom.

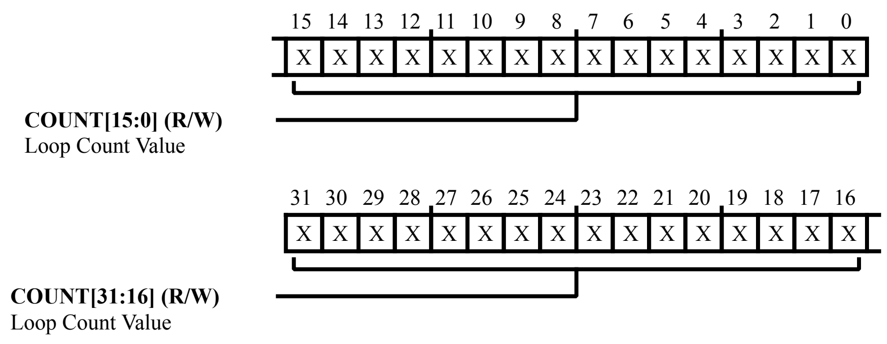


Figure 4-15: LC[n] Register Diagram

Table 4-23: LC[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	COUNT	Loop Count Value. The LC[n].COUNT bits hold the count value of the remaining iterations of the loop.

Blackfin+ ICU Register Descriptions

Interrupt Control Unit (ICU) contains the following registers.

Table 4-24: Blackfin+ ICU Register List

Name	Description
CEC_SID	System ID Register
ICU_CID	Context ID Register
EVT[n]	Event Vector Table Registers
EVT_OVERRIDE	Event Vector Table Override Register
ILAT	Interrupt Latch Register
IMASK	Interrupt Mask Register
IPEND	Interrupt Pending Register

System ID Register

The CEC_SID register contains the system ID of the interrupt that was most recently accepted by the processor core. When the core accepts the interrupt, it sends an interrupt acknowledge to the SCI, causing the SCI to update the value in its SEC_CSID register. For more information about interrupt system ID assignments, see the SCI section of the processor hardware reference manual.

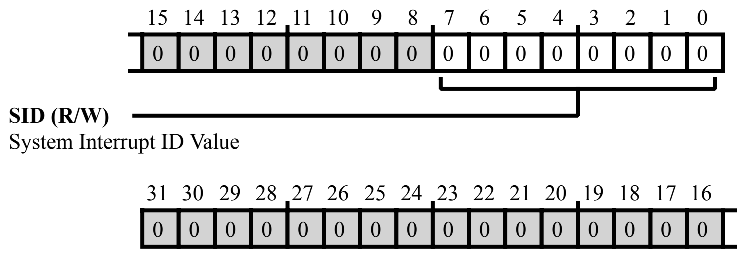


Figure 4-16: CEC_SID Register Diagram

Table 4-25: CEC_SID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	SID	System Interrupt ID Value. The CEC_SID.SID bits hold the system ID assignment for the interrupt that was most recently accepted by the processor core.

Context ID Register

The ICU_CID register is defined as part of the debug trace specification and holds a software-specified 32-bit context ID. This context ID is captured by the program flow trace block for comparison (trace filtering) and may be included in the trace packets.

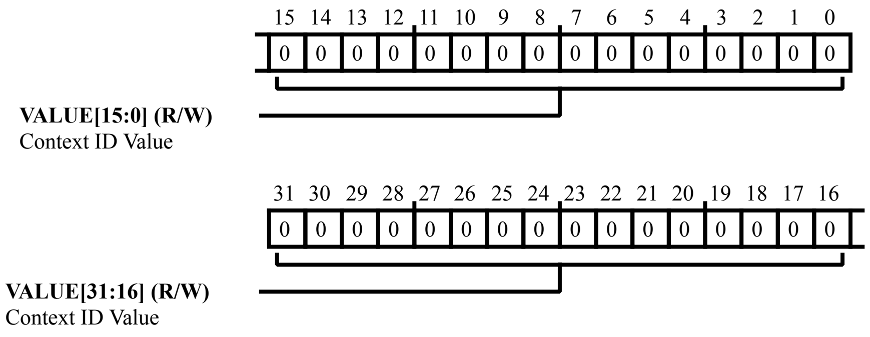


Figure 4-17: ICU_CID Register Diagram

Table 4-26: ICU_CID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	VALUE	Context ID Value. The ICU_CID.VALUE bits hold a software-defined value specifying the software context ID.

Event Vector Table Registers

The memory-mapped EVT[n] registers are a set of registers used to store interrupt service routine vector addresses, providing an entry for each possible core event. Each of these registers can be programmed at reset with the vector address for the corresponding interrupt service routine. When an event occurs, instructions are fetched starting at the address in the EVT[n] register associated with that event.

The processor architecture allows unique addresses to be programmed into each of the EVT[n] registers such that interrupt vectors are not determined from a fixed offset from an interrupt vector table base address. This approach minimizes latency by not requiring a long jump from the vector table to the actual interrupt service routine.

See the IMASK, IPEND, and ILAT register descriptions for a list of interrupt vectors.

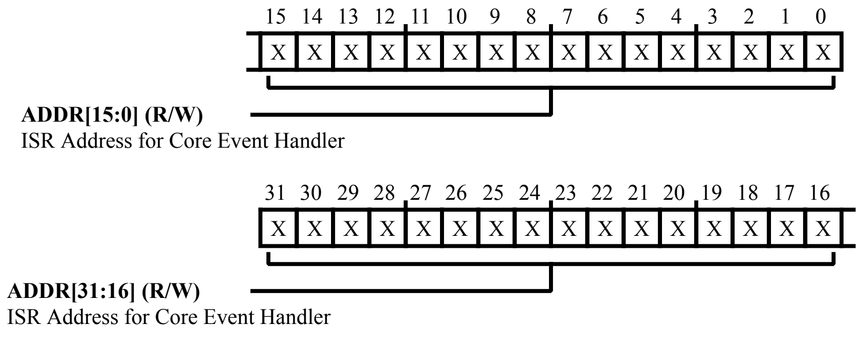

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
X X X X X X X X X X X X X X X X
ADDR[15:0] (R/W)
ISR Address for Core Event Handler
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
X X X X X X X X X X X X X X X X
ADDR[31:16] (R/W)
ISR Address for Core Event Handler

Figure 4-18: EVT[n] Register Diagram

Table 4-27: EVT[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	ISR Address for Core Event Handler. The EVT[n].ADDR bits hold the address to vector to when a specific core event occurs.

Event Vector Table Override Register

The EVT_OVERRIDE register determines whether an event causes the core to vector to the address associated with that event in the corresponding EVT[n] register or to the address on the external bus at reset (as stored in the EVT1 register to handle the reset event). For example, if EVT_OVERRIDE.IVG7 is set, the IVG7 interrupt causes a vector to the address in the EVT1 register rather than to the address in the EVT7 register.

This feature eliminates the need for double-indirection to direct an interrupt vector to an externally supplied address.

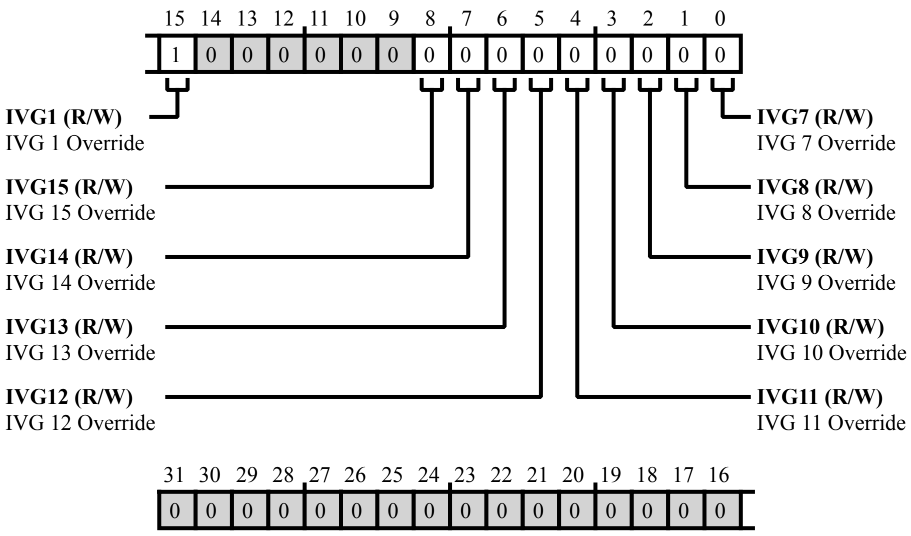


Figure 4-19: EVT_OVERRIDE Register Diagram

Table 4-28: EVT_OVERRIDE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/W)	IVG1	IVG 1 Override. When set, the EVT_OVERRIDE.IVG1 bit overrides the corresponding EVT1 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG1 event occurs.
8 (R/W)	IVG15	IVG 15 Override. When set, the EVT_OVERRIDE.IVG15 bit overrides the corresponding EVT15 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG15 event occurs.
7 (R/W)	IVG14	IVG 14 Override. When set, the EVT_OVERRIDE.IVG14 bit overrides the corresponding EVT14 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG14 event occurs.
6	IVG13	IVG 13 Override.

Table 4-28: EVT_OVERRIDE Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
(R/W)		When set, the `EVT_OVERRIDE.IVG13` bit overrides the corresponding EVT13 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG13 event occurs.
5 (R/W)	IVG12	IVG 12 Override. When set, the `EVT_OVERRIDE.IVG12` bit overrides the corresponding EVT12 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG12 event occurs.
4 (R/W)	IVG11	IVG 11 Override. When set, the `EVT_OVERRIDE.IVG11` bit overrides the corresponding EVT11 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG11 event occurs.
3 (R/W)	IVG10	IVG 10 Override. When set, the `EVT_OVERRIDE.IVG10` bit overrides the corresponding EVT10 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG10 event occurs.
2 (R/W)	IVG9	IVG 9 Override. When set, the `EVT_OVERRIDE.IVG9` bit overrides the corresponding EVT9 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG9 event occurs.
1 (R/W)	IVG8	IVG 8 Override. When set, the `EVT_OVERRIDE.IVG8` bit overrides the corresponding EVT8 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG8 event occurs.
0 (R/W)	IVG7	IVG 7 Override. When set, the `EVT_OVERRIDE.IVG7` bit overrides the corresponding EVT7 address for this interrupt, directing the processor to instead take the vector address from the EVT1 register when the IVG7 event occurs.

Interrupt Latch Register

Each bit in the ILAT register is set when the corresponding event is latched for servicing by the interrupt controller, but not yet accepted into the core for processing. A set bit is then reset by hardware before the first instruction in the corresponding interrupt service routing is executed, at which point the corresponding interrupt pending bit is set in the IPEND register.

Accesses to the ILAT register are limited to Supervisor mode. Supervisor mode writes to this register are write-1-to-clear (W1C) any set bit, but care must be taken to first ensure that the corresponding IMASK bit is cleared before doing so. This write functionality to ILAT is provided for cases where latched interrupt requests need to be cleared (canceled) rather than serviced.

The RAISE instruction can be used to set any of the defined bits except for `ILAT.EMU` and `ILAT.EVX`, but the EXCPT instruction can be used to set `ILAT.EVX`.

Only the JTAG /TRST pin can clear `ILAT.EMU`.

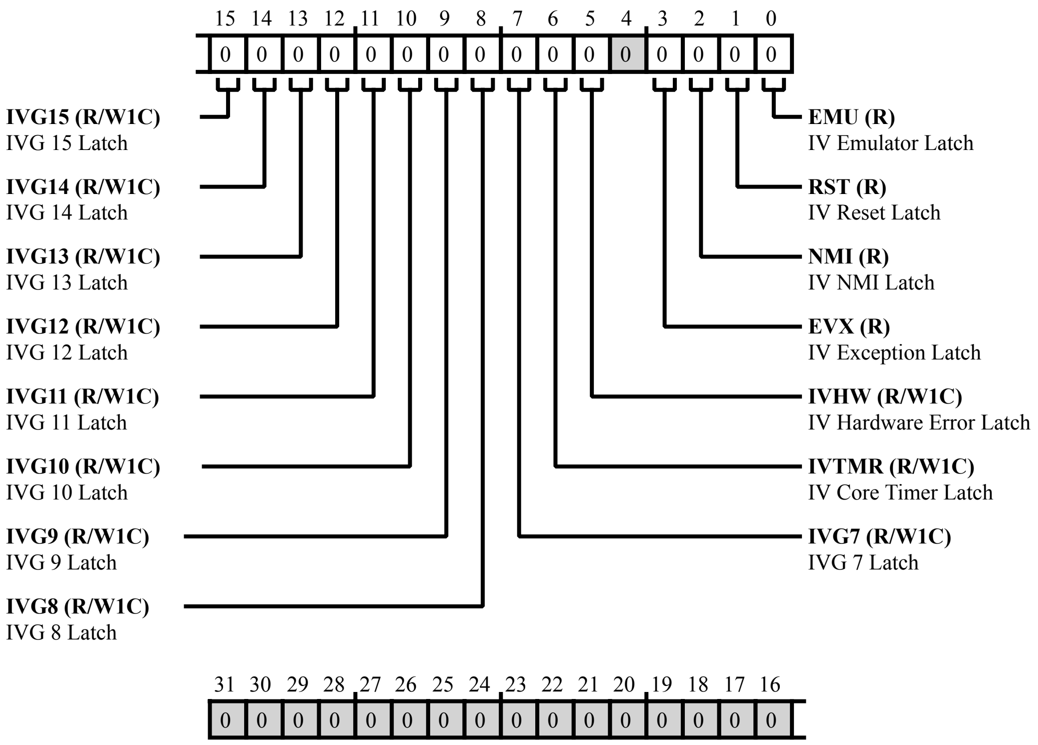


Figure 4-20: ILAT Register Diagram

Table 4-29: ILAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/W1C)	IVG15	IVG 15 Latch.

Table 4-29: ILAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
		The ILAT.IVG15 bit if set when the IVG15 event is latched, but not yet accepted into the core for processing.
14 (R/W1C)	IVG14	IVG 14 Latch. The ILAT.IVG14 bit if set when the IVG14 event is latched, but not yet accepted into the core for processing.
13 (R/W1C)	IVG13	IVG 13 Latch. The ILAT.IVG13 bit if set when the IVG13 event is latched, but not yet accepted into the core for processing.
12 (R/W1C)	IVG12	IVG 12 Latch. The ILAT.IVG12 bit if set when the IVG12 event is latched, but not yet accepted into the core for processing.
11 (R/W1C)	IVG11	IVG 11 Latch. The ILAT.IVG11 bit if set when the IVG11 event is latched, but not yet accepted into the core for processing.
10 (R/W1C)	IVG10	IVG 10 Latch. The ILAT.IVG10 bit if set when the IVG10 event is latched, but not yet accepted into the core for processing.
9 (R/W1C)	IVG9	IVG 9 Latch. The ILAT.IVG9 bit if set when the IVG9 event is latched, but not yet accepted into the core for processing.
8 (R/W1C)	IVG8	IVG 8 Latch. The ILAT.IVG8 bit if set when the IVG8 event is latched, but not yet accepted into the core for processing.
7 (R/W1C)	IVG7	IVG 7 Latch. The ILAT.IVG7 bit if set when the IVG7 event is latched, but not yet accepted into the core for processing.
6 (R/W1C)	IVTMR	IV Core Timer Latch. The ILAT.IVTMR bit if set when the Core Timer event is latched, but not yet accepted into the core for processing.
5 (R/W1C)	IVHW	IV Hardware Error Latch. The ILAT.IVHW bit if set when the Hardware Error event is latched, but not yet accepted into the core for processing.
3 (R/NW)	EVX	IV Exception Latch. The ILAT.EVX bit if set when an Exception event is latched, but not yet accepted into the core for processing.
2	NMI	IV NMI Latch.

Table 4-29: ILAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
(R/NW)		The `ILAT.NMI` bit if set when an NMI event is latched, but not yet accepted into the core for processing.
1 (R/NW)	RST	IV Reset Latch. The `ILAT.RST` bit if set when the Reset event is latched, but not yet accepted into the core for processing.
0 (R/NW)	EMU	IV Emulator Latch. The `ILAT.EMU` bit if set when the Emulator event is latched, but not yet accepted into the core for processing.

Interrupt Mask Register

The IMASK register indicates which interrupts are allowed to be accepted into the core for processing. Accesses to the IMASK register are limited to Supervisor mode. Each of the bits associated with programmable interrupt vectors (IMASK.IVG15 through IMASK.IVG7) can be set to enable that interrupt level for servicing by the core. The lower order bits in the IMASK.UNMASKABLE field are associated with vectors that cannot be disabled and and are therefore read-only.

When an IMASK bit is set and the corresponding event is latched in the ILAT register, the interrupt vector is taken unless preempted by a higher-priority interrupt. When a lower-priority interrupt is preempted, the core will not service the interrupt until it becomes the highest-priority interrupt request and the IMASK bit is still set.

If the IMASK bit is cleared and the associated interrupt is latched in the ILAT register, the interrupt request will not propagate to the core for servicing, and the associated ILAT bit remains set.

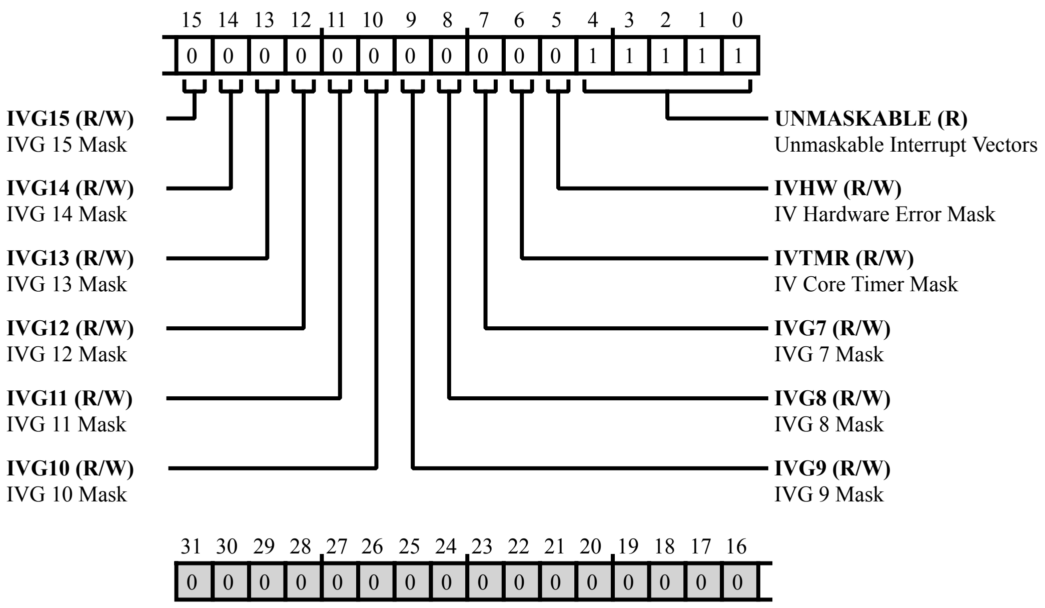


Figure 4-21: IMASK Register Diagram

Table 4-30: IMASK Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/W)	IVG15	IVG 15 Mask. When set, the IMASK.IVG15 bit enables propagation of the IVG15 interrupt request to the processor core for servicing.
14 (R/W)	IVG14	IVG 14 Mask. When set, the IMASK.IVG14 bit enables propagation of the IVG14 interrupt request to the processor core for servicing.
13	IVG13	IVG 13 Mask.

Table 4-30: IMASK Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
(R/W)		When set, the `IMASK.IVG13` bit enables propagation of the IVG13 interrupt request to the processor core for servicing.
12 (R/W)	IVG12	IVG 12 Mask. When set, the `IMASK.IVG12` bit enables propagation of the IVG12 interrupt request to the processor core for servicing.
11 (R/W)	IVG11	IVG 11 Mask. When set, the `IMASK.IVG11` bit enables propagation of the IVG11 interrupt request to the processor core for servicing.
10 (R/W)	IVG10	IVG 10 Mask. When set, the `IMASK.IVG10` bit enables propagation of the IVG10 interrupt request to the processor core for servicing.
9 (R/W)	IVG9	IVG 9 Mask. When set, the `IMASK.IVG9` bit enables propagation of the IVG9 interrupt request to the processor core for servicing.
8 (R/W)	IVG8	IVG 8 Mask. When set, the `IMASK.IVG8` bit enables propagation of the IVG8 interrupt request to the processor core for servicing.
7 (R/W)	IVG7	IVG 7 Mask. When set, the `IMASK.IVG7` bit enables propagation of the IVG7 interrupt request to the processor core for servicing.
6 (R/W)	IVTMR	IV Core Timer Mask. When set, the `IMASK.IVTMR` bit enables propagation of the Core Timer interrupt request to the processor core for servicing.
5 (R/W)	IVHW	IV Hardware Error Mask. When set, the `IMASK.IVHW` bit enables propagation of the Hardware Error interrupt request to the processor core for servicing.
4:0 (R/NW)	UNMASKABLE	Unmaskable Interrupt Vectors. The `IMASK.UNMASKABLE` bits are associated with events that cannot be disabled (Emulator, Reset, NMI, and Exception events), therefore these bits are always set.

Interrupt Pending Register

With the exception of the IPEND.IRPTEN bit, a set bit in the IPEND register indicates that the associated event has been accepted into the core for processing and is either actively being serviced or is nested at some level due to higher-priority interrupts and is awaiting service completion. The IPEND register is limited to read-only accesses from Supervisor mode.

The IPEND.IRPTEN bit is used by the core event controller to temporarily disable interrupts upon entry to and exit from an interrupt service routine.

When an event is latched in ILAT with the associated IMASK bit also set (to allow propagation of the interrupt request to the core), the vector is taken and the event is considered processed. At this point, the corresponding bit in IPEND is set by hardware. If a higher-priority event is latched in ILAT (with the associated IMASK bit also set) while servicing the current event, the core will vector to service that higher-priority event and set its corresponding IPEND bit. In this fashion, the least significant set bit in this register indicates the interrupt that is currently being serviced, and any other set bit indicates that the associated event was accepted into the core but is nested at some level (has not yet been fully serviced).

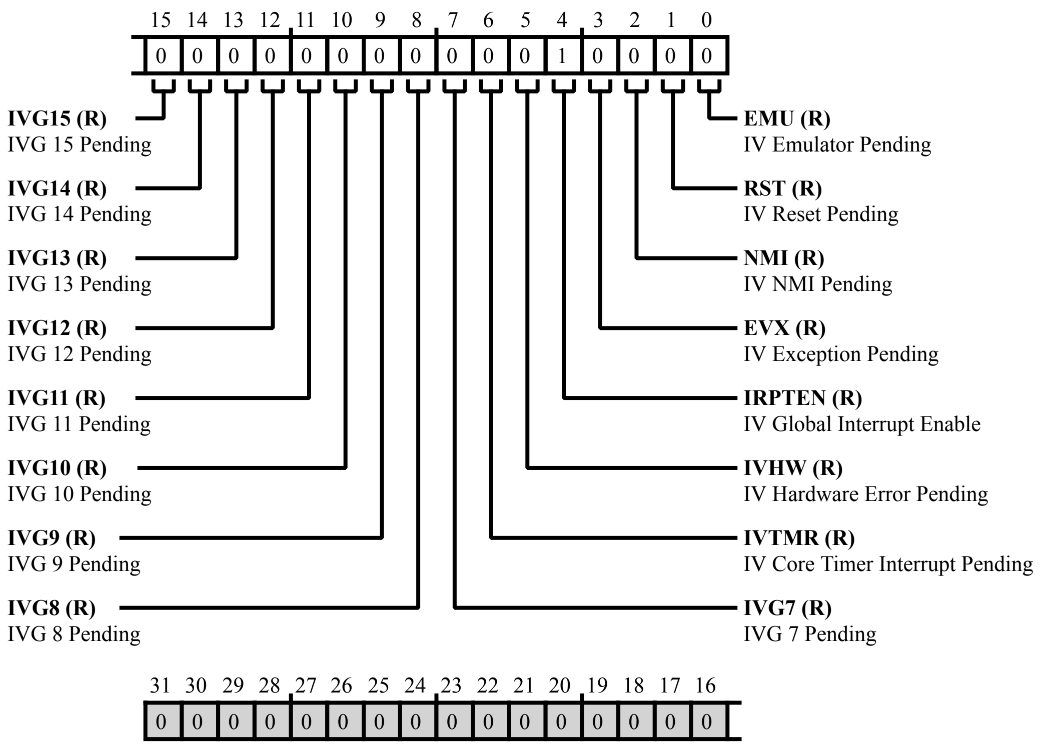


Figure 4-22: IPEND Register Diagram

Table 4-31: IPEND Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15 (R/NW)	IVG15	IVG 15 Pending. When set, the `IPEND.IVG15` bit indicates that the IVG15 interrupt is currently active or nested at some level.
14 (R/NW)	IVG14	IVG 14 Pending. When set, the `IPEND.IVG14` bit indicates that the IVG14 interrupt is currently active or nested at some level.
13 (R/NW)	IVG13	IVG 13 Pending. When set, the `IPEND.IVG13` bit indicates that the IVG13 interrupt is currently active or nested at some level.
12 (R/NW)	IVG12	IVG 12 Pending. When set, the `IPEND.IVG12` bit indicates that the IVG12 interrupt is currently active or nested at some level.
11 (R/NW)	IVG11	IVG 11 Pending. When set, the `IPEND.IVG11` bit indicates that the IVG11 interrupt is currently active or nested at some level.
10 (R/NW)	IVG10	IVG 10 Pending. When set, the `IPEND.IVG10` bit indicates that the IVG10 interrupt is currently active or nested at some level.
9 (R/NW)	IVG9	IVG 9 Pending. When set, the `IPEND.IVG9` bit indicates that the IVG9 interrupt is currently active or nested at some level.
8 (R/NW)	IVG8	IVG 8 Pending. When set, the `IPEND.IVG8` bit indicates that the IVG8 interrupt is currently active or nested at some level.
7 (R/NW)	IVG7	IVG 7 Pending. When set, the `IPEND.IVG7` bit indicates that the IVG7 interrupt is currently active or nested at some level.
6 (R/NW)	IVTMR	IV Core Timer Interrupt Pending. When set, the `IPEND.IVTMR` bit indicates that the Core Timer interrupt is currently active or nested at some level.
5 (R/NW)	IVHW	IV Hardware Error Pending. When set, the `IPEND.IVHW` bit indicates that the Hardware Error interrupt is currently active or nested at some level.
4 (R/NW)	IRPTEN	IV Global Interrupt Enable.

Table 4-31: IPEND Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
		The IPEND.IRPTEN bit is managed by the core event controller to ensure that the RETI register is properly saved before a higher-priority event can be recognized by the core.
3 (R/NW)	EVX	IV Exception Pending. When set, the IPEND.EVX bit indicates that the Exception event is currently active or nested at some level.
2 (R/NW)	NMI	IV NMI Pending. When set, the IPEND.NMI bit indicates that the NMI event is currently active or nested at some level.
1 (R/NW)	RST	IV Reset Pending. When set, the IPEND.RST bit indicates that the Reset event is currently active or nested at some level.
0 (R/NW)	EMU	IV Emulator Pending. When set, the IPEND.EMU bit indicates that the Emulator event is currently active.

Blackfin+ BP Register Descriptions

Branch Predictor (BP) contains the following registers.

Table 4-32: Blackfin+ BP Register List

Name	Description
BP_CFG	BP Configuration Register
BP_STAT	BP Status Register

BP Configuration Register

The BP_CFG register configures branch predictor features such as enabling dynamic branch prediction for various types of branch instructions, controlling updates to the branch prediction table, permitting access to entries in the prediction table and prediction table memory, and clearing the branch prediction table.

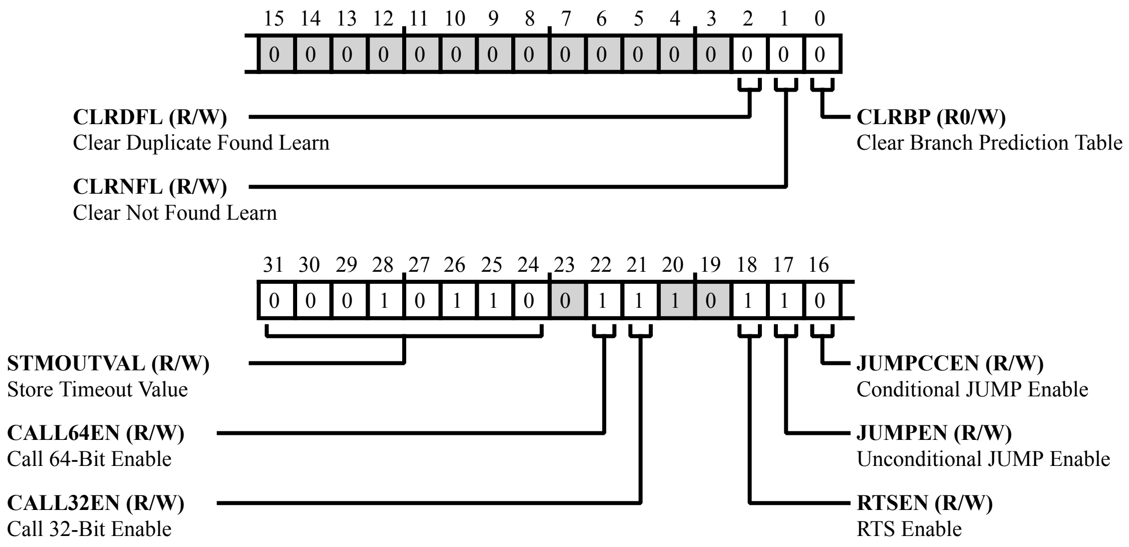


Figure 4-23: BP_CFG Register Diagram

Table 4-33: BP_CFG Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/W)	STMOUTVAL	Store Timeout Value. The BP_CFG.STMOUTVAL bits select a timeout value (in CCLK cycles) for the wait status state for access requests to the store buffer.	
22 (R/W)	CALL64EN	Call 64-Bit Enable. The BP_CFG.CALL64EN bit enables branch prediction for CALL instructions encoded as 64-bit opcodes.	
		0	Disable prediction
		1	Enable prediction
21 (R/W)	CALL32EN	Call 32-Bit Enable. The BP_CFG.CALL32EN bit enables branch prediction for CALL instructions encoded as 32-bit opcodes.	
		0	Disable prediction
		1	Enable prediction

Table 4-33: BP_CFG Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
18 (R/W)	RTSEN	RTS Enable. The BP_CFG.RTSEN bit enables branch prediction for RTS instructions.	
		0	Disable prediction
		1	Enable prediction
17 (R/W)	JUMPEN	Unconditional JUMP Enable. The BP_CFG.JUMPEN bit enables branch prediction for unconditional JUMP instructions.	
		0	Disable prediction
		1	Enable prediction
16 (R/W)	JUMPCCEN	Conditional JUMP Enable. The BP_CFG.JUMPCCEN bit enables branch prediction for conditional JUMP instructions.	
		0	Disable prediction
		1	Enable prediction
2 (R/W)	CLRDFL	Clear Duplicate Found Learn. The BP_CFG.CLRDFL bit clears the BP_STAT.DFL bit and keeps it cleared. When enabled, BP_CFG.CLRDFL prevents the normal behavior of the branch predictor reporting branch entry duplicate found errors from occurring.	
		0	Disable
		1	Enable
1 (R/W)	CLRNFL	Clear Not Found Learn. The BP_CFG.CLRNFL bit clears the BP_STAT.NFL bit and keeps it cleared. When enabled, BP_CFG.CLRNFL prevents the normal behavior of the branch predictor reporting branch entry not found errors from occurring.	
		0	Disable
		1	Enable
0 (R0/W)	CLRBP	Clear Branch Prediction Table. The BP_CFG.CLRBP bit clears (W1A) the tag valid field for all the entries in the branch prediction table.	

BP Status Register

The BP_STAT register indicates the status of the branch predictor state machine, store buffer, and current operation.

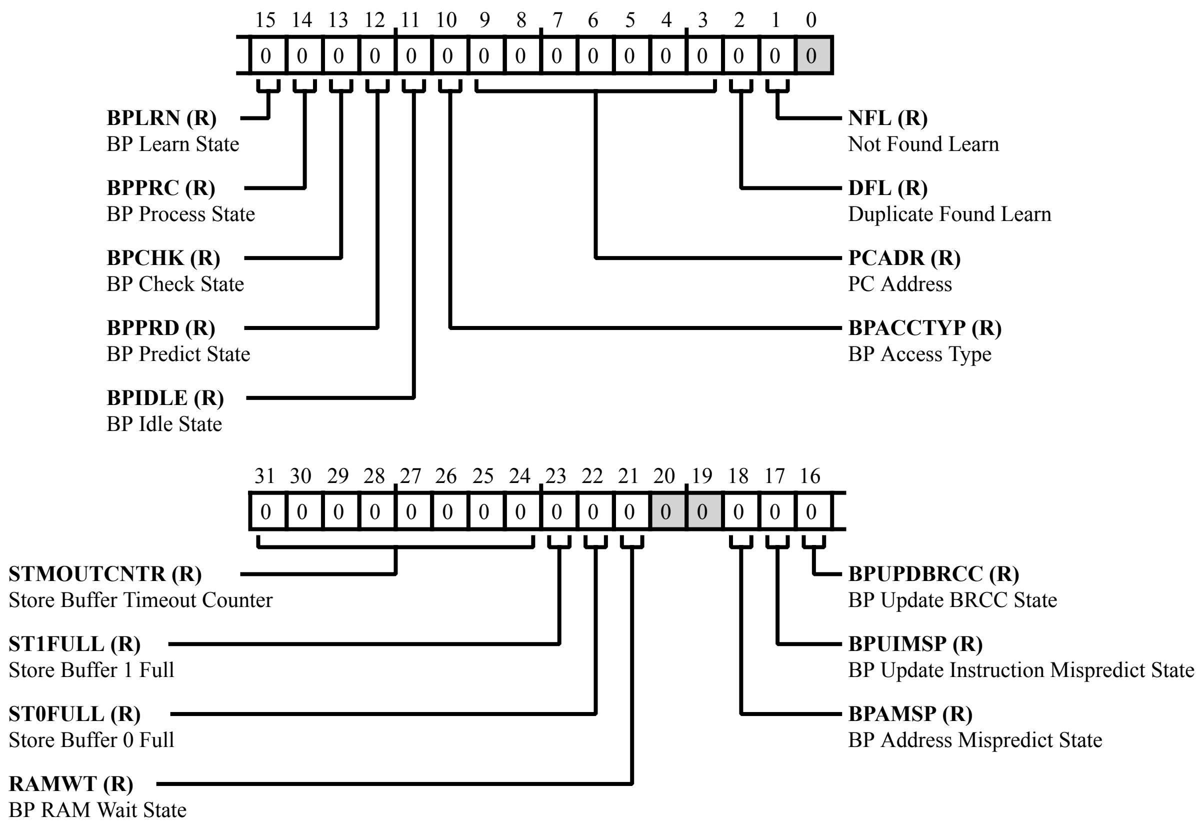


Figure 4-24: BP_STAT Register Diagram

Table 4-34: BP_STAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/NW)	STMOUTCNTR	Store Buffer Timeout Counter. The BP_STAT.STMOUTCNTR bits hold the value of the count remaining for the store buffer timeout. This field is automatically loaded when the BP_CFG.STMOUTVAL field is loaded and resets to that value when store buffer access occurs before the count expires. The count decrements every core clock (CCLK) cycle while the branch predictor is in the wait state (BP_STAT.RAMWT = 1).	
23 (R/NW)	ST1FULL	Store Buffer 1 Full. The BP_STAT.ST1FULL bit indicates whether or not the branch predictor store buffer 1 is full.	
		0	Store buffer 1 not full
		1	Store buffer 1 full

Table 4-34: BP_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
22 (R/NW)	ST0FULL	Store Buffer 0 Full. The `BP_STAT.ST0FULL` bit indicates whether or not the branch predictor store buffer 0 is full.	
		0	Store buffer 0 not full
		1	Store buffer 0 full
21 (R/NW)	RAMWT	BP RAM Wait State. The `BP_STAT.RAMWT` bit indicates whether or not the branch predictor is waiting for access to the branch predictor memory.	
		0	Not waiting
		1	Waiting
18 (R/NW)	BPAMSP	BP Address Mispredict State. The `BP_STAT.BPAMSP` bit indicates whether or not the branch predictor is in the Address Mispredict state.	
		0	Not in Address Mispredict state
		1	In Address Mispredict state
17 (R/NW)	BPUIMSP	BP Update Instruction Mispredict State. The `BP_STAT.BPUIMSP` bit indicates whether or not the branch predictor is in the Instruction Mispredict state.	
		0	Not in Instruction Mispredict state
		1	In Instruction Mispredict state
16 (R/NW)	BPUPDBRCC	BP Update BRCC State. The `BP_STAT.BPUPDBRCC` bit indicates whether or not the branch predictor is in the Update BRCC state.	
		0	Not in Update BRCC state
		1	In Update BRCC state
15 (R/NW)	BPLRN	BP Learn State. The `BP_STAT.BPLRN` bit indicates whether or not the branch predictor is in the Learn state.	
		0	Not in Learn state
		1	In Learn state

Table 4-34: BP_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
14 (R/NW)	BPPRC	BP Process State. The `BP_STAT.BPPRC` bit indicates whether or not the branch predictor is in the Process state.	
		0	Not in Process state
		1	In Process state
13 (R/NW)	BPCHK	BP Check State. The `BP_STAT.BPCHK` bit indicates whether or not the branch predictor is in the Check state.	
		0	Not in Check state
		1	In Check state
12 (R/NW)	BPPRD	BP Predict State. The `BP_STAT.BPPRD` bit indicates whether or not the branch predictor is in the Predict state.	
		0	Not in Predict state
		1	In Predict state
11 (R/NW)	BPIDLE	BP Idle State. The `BP_STAT.BPIDLE` bit indicates whether or not the branch predictor is in the Idle state.	
		0	Not in Idle state
		1	In Idle state
10 (R/NW)	BPACCTYP	BP Access Type. The `BP_STAT.BPACCTYP` bit indicates whether the most recent branch prediction table access was made for predicting or learning the branch.	
		0	Learning the branch
		1	Predicting the branch
9:3 (R/NW)	PCADR	PC Address. The `BP_STAT.PCADR` bits provide the branch predictor memory address within the branch prediction table corresponding to the most recent PC entry accessed.	
2 (R/NW)	DFL	Duplicate Found Learn. The `BP_STAT.DFL` bit indicates whether or not a duplicate branch prediction table entry was found during a BP Learn state.	
		0	No status
		1	Duplicate entry found

Table 4-34: BP_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
1 (R/NW)	NFL	Not Found Learn. The `BP_STAT.NFL` bit indicates whether or not a branch prediction table entry was found during a BP Learn state.	
		0	No status
		1	Entry not found

5 Core Timer (TMR)

Each Blackfin+ core features a dedicated timer. Unlike other peripherals, the core timer resides inside the Blackfin+ core and runs at the core clock (CCLK) rate. The core timer is typically used as a system tick clock for generating periodic operating system interrupts.

TMR Features

The core timer is a programmable 32-bit interval timer which can generate periodic interrupts. Core timer features include:

- 32-bit timer with 8-bit prescaler
- Operates at core clock (*CCLK*) rate
- Dedicated high-priority interrupt channel
- Single-shot or continuous operation

TMR Functional Description

The TMR (core timer) is a programmable 32-bit interval timer in each processor core. The following sections describe the TMR features:

- TMR Block Diagram

Blackfin+ TMR Register List

Table 5-1: Blackfin+ TMR Register List

Name	Description
`TCNTL`	Core Timer Control Register (TCNTL)
`TCOUNT`	Core Timer Count Register (TCOUNT)
`TPERIOD`	Core Timer Period Register (TPERIOD)
`TSCALE`	Core Timer Scale Register (TSCALE)

TMR Block Diagram

The *Core Timer Block Diagram* shows the core timer block diagram.

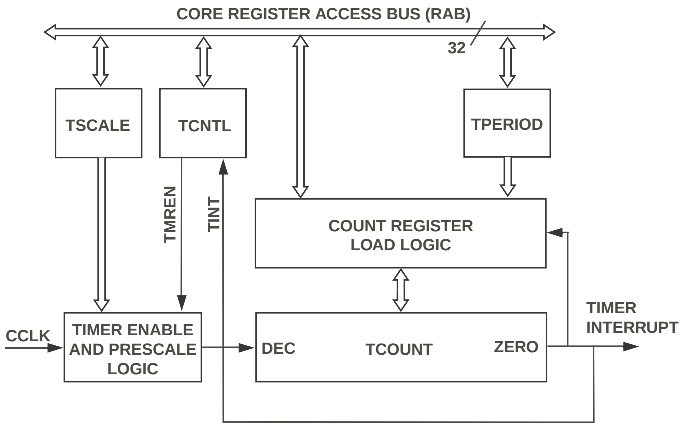

CORE REGISTER ACCESS BUS (RAB)
32
TSCALE
TCNTL
TPERIOD
TMREN
TINT
COUNT REGISTER LOAD LOGIC
CCLK
TIMER ENABLE AND PRESCALE LOGIC
DEC
TCOUNT
ZERO
TIMER INTERRUPT

Figure 5-1: Core Timer Block Diagram

External Interfaces

The core timer does not directly interact with any external pins on the device.

Internal Interfaces

The core timer is accessed through the 32-bit register access bus (RAB). The core clock (*CCLK*) is the source clock for the module. The dedicated interrupt request of the timer is a higher priority than requests from all other peripherals.

TMR Operation

The software initializes the timer count (`TCOUNT`) register before the timer is enabled. The `TCOUNT` register can be written directly, but writes to the timer period (`TPERIOD`) register also pass through to `TCOUNT`.

When the timer is enabled by setting the `TCNTL.EN` bit, the `TCOUNT` register is decremented once every `TSCALE`+ 1 *CCLK* cycles. When the value of the `TCOUNT` register reaches 0, the core timer generates an interrupt and the `TCNTL.INT` bit is set.

If the `TCNTL.AUTORLD` bit is set, then hardware automatically reloads the `TCOUNT` register with the contents of the `TPERIOD` register, and the count begins again. If the `TCNTL.AUTORLD` bit is not set, the timer stops operation.

Clear the `TCNTL.PWR` bit to put the core timer into low-power mode, which disables clocks to the core timer to reduce power consumption. Before using the timer, set the `TCNTL.PWR` bit to restore clocks to the timer unit before setting the `TCNTL.EN` bit to enable the core timer.

NOTE: Hardware behavior is undefined if `TCNTL.EN` is set when `TCNTL.PWR`= 0.

Interrupt Processing

The core timer's dedicated interrupt request is a higher priority than interrupt requests from all other peripherals. The request goes directly to the core event controller (CEC), thus bypassing the system event controller (SEC) entirely. As such, interrupt processing is completely in the *CCLK* domain.

NOTE: The core timer interrupt request is edge-sensitive. Hardware clears it automatically as soon as the interrupt is serviced.

The `TCNTL.INT` bit indicates that the core timer has generated an interrupt. Programs must write a 0 (not W1C) to clear it, though this write is optional. The core timer module does not provide any further interrupt enable bit. When the timer is enabled, interrupts can be masked in the CEC controller.

Blackfin+ TMR Register Descriptions

Timer (TMR) contains the following registers.

Table 5-2: Blackfin+ TMR Register List

Name	Description
TCNTL	Core Timer Control Register (TCNTL)
TCOUNT	Core Timer Count Register (TCOUNT)
TPERIOD	Core Timer Period Register (TPERIOD)
TSCALE	Core Timer Scale Register (TSCALE)

Core Timer Control Register (TCNTL)

The TCNTL register is used for timer configuration and status.

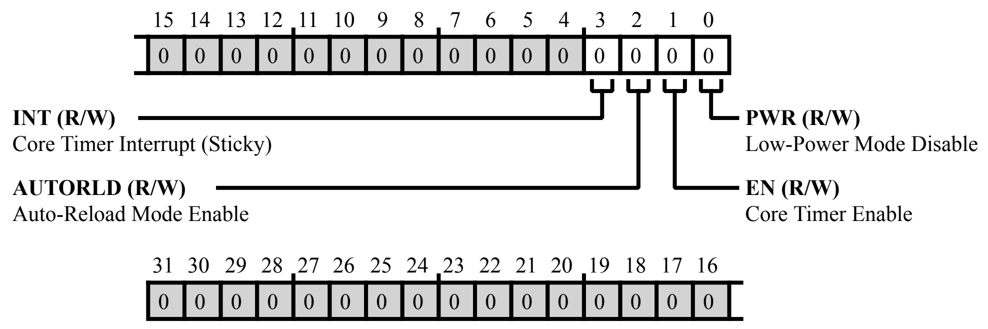


Figure 5-2: TCNTL Register Diagram

Table 5-3: TCNTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
3 (R/W)	INT	Core Timer Interrupt (Sticky). The TCNTL bit indicates whether or not the Core Timer has generated an interrupt.	
		0	Core timer has not generated an interrupt
		1	Core timer has generated an interrupt
2 (R/W)	AUTORLD	Auto-Reload Mode Enable. By default, when TCOUNT reaches 0, the timer generates an interrupt and halts. When the TCNTL.AUTORLD bit is set, Auto-reload mode is enabled and TCOUNT will automatically be reloaded from TPERIOD when the timer generates the interrupt, and the timer continues to count.	
		0	Disable Auto-reload mode
		1	Enable Auto-reload mode
1 (R/W)	EN	Core Timer Enable. The TCNTL.EN denotes whether or not the core timer is enabled. It is only meaningful when the TCNTL.PWR bit is also set.	
		0	Disable core timer
		1	Enable core timer
0 (R/W)	PWR	Low-Power Mode Disable. The TCNTL.PWR bit determines whether or not the core timer is in the Low-power mode, where the CCLK is gated from clocking the core timer unit.	
		0	Timer is in Low-power mode
		1	Timer is in active state

Core Timer Count Register (TCOUNT)

The TCOUNT register decrements once every TSCALE + 1 core clock cycles. When the value of TCOUNT reaches 0, an interrupt is generated, and the TCNTL.INT bit is set.

Values written to the TPERIOD register are automatically copied to the TCOUNT register as well. Nevertheless, the TCOUNT register can be written directly. In Auto-reload mode, the value written to TCOUNT may differ from the TPERIOD setting to let the initial period be shorter or longer than the rest that follow. To accomplish this, write to TPERIOD first and then subsequently overwrite TCOUNT.

Writes to TCOUNT are ignored once the timer is running.

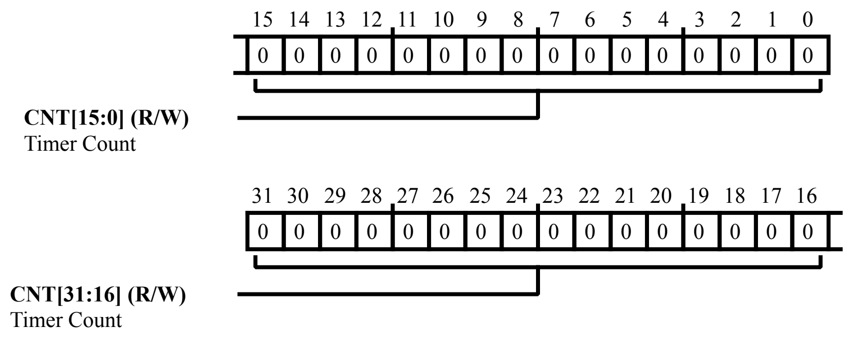


Figure 5-3: TCOUNT Register Diagram

Table 5-4: TCOUNT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	Timer Count.

Core Timer Period Register (TPERIOD)

The TPERIOD register is used to configure the periodicity of the core timer interrupt. Writes to the TPERIOD register automatically propagate to the TCOUNT register to set the time-out for the core timer to generate the interrupt.

When Auto-reload is enabled by the TCNTL.AUTORLD bit, the TCOUNT register is reloaded with the contents of TPERIOD whenever it reaches 0. Writes to TPERIOD are ignored when the timer is running.

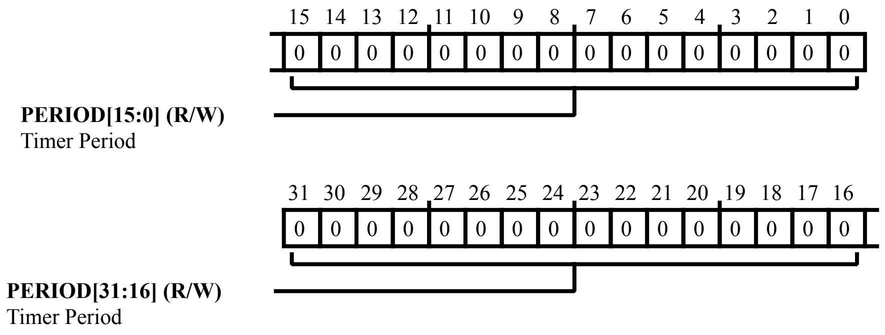


Figure 5-4: TPERIOD Register Diagram

Table 5-5: TPERIOD Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	PERIOD	Timer Period.

Core Timer Scale Register (TSCALE)

The TSCALE register contains the scaling value that is one less than the number of core clock cycles between decrements of the TCOUNT register. For example, if the value in the TSCALE register is 0, the TCOUNT register will decrement every CCLK cycle. If TSCALE is 1, TCOUNT decrements once every two CCLK cycles.

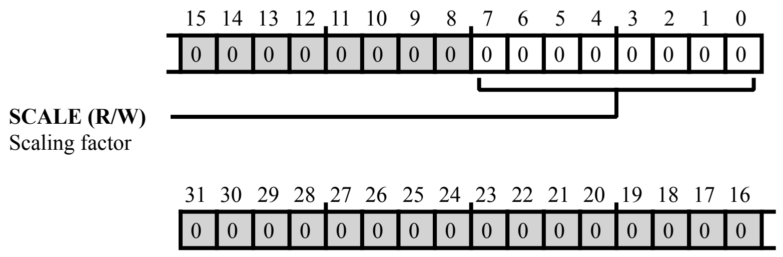


Figure 5-5: TSCALE Register Diagram

Table 5-6: TSCALE Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7:0 (R/W)	SCALE	Scaling factor.

6 Address Arithmetic Unit

Like most digital signal processor (DSP) and reduced instruction set computer (RISC) platforms, the Blackfin+ processors have a load/store architecture. Computation operands and results are always represented by core registers. Prior to computation, data is loaded from memory into core registers, and results are stored back by explicit move operations. The Address Arithmetic Unit (AAU) provides all the required support to keep data transport between memory and core registers efficient and seamless. Having a separate arithmetic unit for address calculations prevents the data computation block from being burdened by address operations. Not only can the load and store operations occur in parallel to data computations, but memory addresses can also be calculated at the same time.

The AAU uses Data Address Generators (DAGs) to generate addresses for data moves to and from memory. By generating addresses, the DAGs let programs refer to addresses indirectly, using a DAG register instead of an absolute address. The figure shows the AAU block diagram.

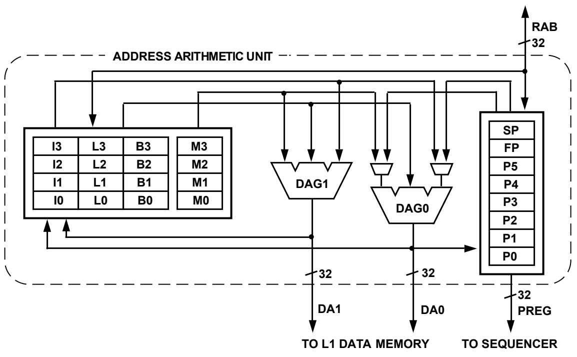


Figure 6-1: AAU Block Diagram

The AAU architecture supports several functions that minimize overhead in data access routines. These functions include:

- Supply address - provides an address during a data access
- Supply address and post-modify - provides an address during a data move and auto-increments/decrements the stored address for the next move
- Supply address with offset - provides an address from a base with an offset without incrementing the original address pointer
- Modify address - increments or decrements the stored address without performing a data move

- Bit-reversed carry address - provides a bit-reversed carry address during a data move without reversing the stored address

The AAU comprises two DAGs, nine pointer registers, four index registers and four complete sets of related modify, base, and length registers. These registers (shown in the AAU figure) hold the values that the DAGs use to generate addresses. The types of registers are:

- Index (`I[3:0]`) registers. Unsigned 32-bit index registers hold an address pointer to memory. For example, the `R3 = [I0];` instruction loads the data value found at the memory location pointed to by the `I0` register. Index registers can be used for 16- and 32-bit memory accesses.
- Modify (`M[3:0]`) registers. Signed 32-bit modify registers provide the increment, or step size, by which an index register is modified after a register move. For example, the `R0 = [I0 ++ M1];` instruction directs the DAG to output the address in register `I0`, load the contents of the memory address pointed to by the`I0` register into the `R0` register, and then modify the value of the`I0` register by the value in the `M1` register
- Base (`B[3:0]`) and length (`L[3:0]`) registers. Unsigned 32-bit base and length registers set up the the starting address and length of a buffer, respectively. Each `B`/`L` pair is always grouped with a corresponding index register. For example, `I3`, `B3`, and `L3` are used collectively to handle a single buffer, but any modify register can be used to update the dedicated index register. If the length register is set to 0, the buffer is unbound and linear. If the length register is non-zero, the buffer is bound and circular, meaning that index modification beyond the end of the buffer will wrap back to the base address. For more information on circular buffers, see Addressing Circular Buffers.
- Pointer registers. The core has six general-purpose (`P[5:0]`) pointer registers, a Frame Pointer (`FP`) register, a User (Mode) Stack Pointer (`USP`) register, and a Stack Pointer (`SP`) register. Each is a 32-bit pointer register holding the value of an address in memory and can be manipulated and used in various instructions. For example, the `R3 = [P0];` instruction loads the `R3` register with the data found at the memory location pointed to by the `P0` register. The pointer registers have no effect on circular buffer addressing and can be used for 8-, 16-, and 32-bit memory accesses. For added mode protection, the `SP` register is only accessible in Supervisor mode, and the `USP` register is an alias that is either implicitly accessed via the stack pointer from User mode or explicitly accessed from Supervisor mode.

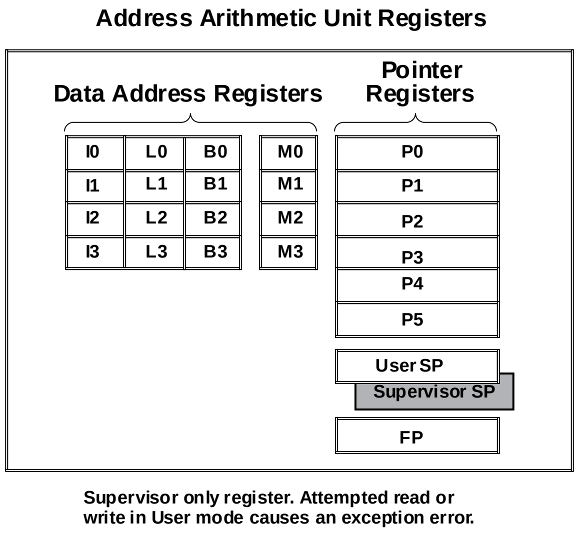

Address Arithmetic Unit Registers
Data Address Registers
Pointer Registers
I0
I1
I2
I3
L0
L1
L2
L3
B0
B1
B2
B3
M0
M1
M2
M3
P0
P1
P2
P3
P4
P5
User SP
Supervisor SP
FP
Supervisor only register. Attempted read or write in User mode causes an exception error.

Figure 6-2: Address Arithmetic Unit

Addressing with the AAU

The DAGs can generate an address that is incremented by an immediate value or by a value in another register. In post-modify addressing, the DAG outputs the index register value unchanged, then adds the content of a modify register or an immediate value to the value in the index register.

In indexed addressing, the DAG can add a small value in the pointer register without updating the pointer register with the new value, thus providing an offset for that particular memory access.

In direct addressing, the entire address is specified in the instruction and does not depend upon the value in any register.

The processor is byte-addressed. Depending on the type of data used, increments and decrements to the address registers can be by 1, 2, or 4 bytes to align with the respective 8-, 16-, or 32-bit accesses.

For example, consider the following instruction:

```
R0 = [ P3++ ];
```

This instruction fetches the 32-bit word pointed to by the address in the `P3` register and places it in the 32-bit `R0` register. It then post-increments `P3` by four to point to the data after the 32-bit word that was just fetched.

Now consider this instruction:

```
R0.L = W [ I3++ ];
```

The W modifier in this instruction indicates that this is a 16-bit access to the address pointed to by the `I3` register. As such, the destination register must be a 16-bit entity, in this case the low half of one of the data registers (`R0.L`), and the address in the `I3` register is post-incremented by two after the access is made.

Finally, there is the byte access:

```
R0 = B [ P3++ ] (Z) ;
```

This instruction fetches a byte that is pointed to by the address in the `P3` register, places it in the destination register, `R0`, and then post-increments the address in `P3` by one. Unlike the previous 16-bit move instruction that chose a destination register of the same width as the access, the 32-bit destination register is satisfied by this instruction's inclusion of the zero extension (Z), which fills the upper 24 bits with zeros. Sign-extension (X) of bit 7 through the upper 24 bits of the register is also supported.

Instructions using index registers can use either a modify register or a small immediate value (+/- 2 or 4) as the modifier. Instructions using pointer registers use a small immediate value or another pointer register as the modifier. For more details, see AAU Instruction Summary.

There are no restrictions on data alignment. A 32-bit word can be fetched from any address, and the four contiguous bytes starting from the specified address are fetched. Similarly, a 16-bit access may be to any two adjacent addresses in memory. The byte order of the memory is little-endian, so the lower addressed byte always contains the least significant bits of the stored value.

Pointer Register File

The general-purpose pointer registers (or Preg), are organized as:

- a 6-entry pointer register file (`P[5:0]`, see Pointer Register)
- a Frame Pointer (`FP`), used to point to the current procedure's activation record (see Frame Pointer Register)
- a Stack Pointer (`SP`), used to point to the last used location on the run-time stack (see Stack Pointer Register)

Pointer registers are 32 bits wide. Although pointer registers are primarily used for address calculations, they may also be used for general integer arithmetic with a limited set of arithmetic operations. However, unlike data registers, pointer register arithmetic does not affect the bits in the Arithmetic Status register (`ASTAT`).

Frame and Stack Pointers

In many respects, the frame and stack pointer registers perform like the other pointer registers, `P[5:0]`. They can act as general pointers in any of the load/store instructions (e.g., `R1 = B[SP] (Z);`). However, `FP` and `SP` have additional functionality. For more information, see the following:

- Frame Pointer Register
- Stack Pointer Register
- User Stack Pointer Register .

Stack pointer registers include:

- a User Stack Pointer (`USP` in Supervisor mode, `SP` in User mode)
- a Supervisor Stack Pointer (`SP` in Supervisor mode)

The User Stack Pointer register and the Supervisor Stack Pointer register are accessed using the register alias `SP`. Depending on the current processor operating mode, only one of these registers is active and accessible as `SP`:

- In User mode, any reference to `SP` (e.g., the `R0 = [ SP++ ];` stack pop instruction) implicitly uses the `USP` as the effective address to access.
- In Supervisor mode, the same reference to `SP` uses the Supervisor Stack Pointer as the effective address to access. To manipulate the User Stack Pointer from code running in Supervisor mode, explicitly use the register alias `USP`. When the processor is in Supervisor mode, a move from the `USP` register (e.g., `R0 = USP;`) moves the current User Stack Pointer into `R0`. The `USP` register alias can only be used in Supervisor mode.

The following load/store instructions use `FP` and `SP`:

- `FP`-indexed load/store, which extends the addressing range for 16-bit encoded load/stores
- Stack push/pop instructions, including those for pushing and popping multiple registers
- Link/unlink instructions implicitly use both, as they control stack frame space and manage the frame pointer register (`FP`) for that space

DAG Register Set

Embedded processor instructions primarily use the 32-bit Data Address Generator (DAG) register set for addressing, which is comprised of:

- `I[3:0]` - index addresses (see Index (Circular Buffer) Register)
- `M[3:0]` - modify values (see Modify (Circular Buffer) Register)
- `B[3:0]` - base addresses (see Base (Circular Buffer) Register)
- `L[3:0]` - buffer length values (see Length (Circular Buffer) Register)

The I (index) and B (base) registers always contain addresses of individual bytes in memory, with the index registers containing an effective address. The M (modify) registers contain an offset value that is added to or subtracted from one of the index registers.

The B and L (length) registers define buffers. The B register contains the starting address of a buffer, and the L register contains the length (in bytes) for any buffer defined to be circular in nature. If the L register is 0, the buffer is unbounded, where indexing will never wrap. If the L register is non-zero, indexing through the buffer will wrap from the "base plus length" address back to the base address. Each B/L register pair is associated with the corresponding I register. For example, the buffer defined by the `L0` and `B0` register pair is always indexed using the `I0` register. However, any M register may be associated with any I register (i.e., `I0` may be modified by `M3`).

Indexed Addressing with Index and Pointer Registers

Indexed addressing instructions use the value in the index or pointer register as an effective address. This type of instruction can load or store 16- or 32-bit values, and the default is a 32-bit transfer. If a 16-bit transfer is required, then the `W` (16-bit word) designator is used to preface the load or store.

For example:

```
R0 = [ I2 ];
```

loads a 32-bit value from the address pointed to by `I2` and stores it in the 32-bit destination register `R0`.

```
R0.H = W [ I2 ];
```

loads a 16-bit value from the address pointed to by `I2` and stores it in the 16-bit destination register `R0.H`.

```
[ P1 ] = R0 ;
```

is an example of a 32-bit store operation.

Pointer registers can also be used for 8-bit loads and stores. For example:

```
B [ P1 ] = R0 ;
```

stores the 8-bit value from the least signficant byte of the `R0` register to the address pointed to by the `P1` register.

Loads with Zero- or Sign-Extension

When a 32-bit register is loaded by an 8- or 16-bit memory read, the value can be extended to the full register width. A trailing (Z) on the instruction is used to zero-extend the loaded value, whereas an (X) forces sign-extension. The following examples assume that `P1` points to a memory location that contains a value of 0x8080.

```
R0 = W[P1] (Z) ; /* R0 = 0x0000 8080 */

R1 = W[P1] (X) ; /* R1 = 0xFFFF 8080 */

R2 = B[P1] (Z) ; /* R2 = 0x0000 0080 */

R3 = B[P1] (X) ; /* R3 = 0xFFFF FF80 */
```

Indexed Addressing with Immediate Offset

Indexed addressing allows programs to obtain values from data tables with references to the base of that table. The pointer register is modified by the immediate field and that value is then used as the effective address to access. The value of the pointer register, however, is not updated.

For example, if `P1 = 0x13`, then `R0 = [P1 + 0x11]` would effectively be equal to `R0 = [0x24]`, but the value of the pointer register would be unchanged when this instruction executes.

Auto-increment and Auto-decrement Addressing

Auto-increment addressing updates the pointer and index registers after the access. The amount the address is incremented by depends on the size of the access. An 32-bit access results in an update of the pointer by four. A 16-bit access updates the pointer by 2, and an 8-bit access updates the pointer by 1. Both 8- and 16-bit read operations may specify to either sign- or zero-extend the read contents into the upper bits of the destination register. Pointer registers may be used for 8-, 16-, and 32-bit accesses, while index registers may only be used for 16- and 32-bit accesses. For example:

```
R0 = W [ P1++ ] (Z);
```

loads a 16-bit word into a 32-bit destination register from an address pointed to by the `P1` pointer register. The pointer is then incremented by two, and the word is zero-extended to fill the 32-bit destination register.

Auto-decrement works the same way by decrementing the address after the access. For example:

```
R0 = [ I2-- ] ;
```

loads a 32-bit value into the destination register and decrements the index register by four.

Pre-modify Stack Pointer Addressing

The only pre-modify instruction in the processor uses the stack pointer register, `SP`. The address in `SP` is first decremented by four and then used as the effective address for the store. The `[ --SP ] = R0;` instruction is used for stack push operations and can support only 32-bit word transfers.

Post-modify Addressing

Post-modify addressing uses the value in the index or pointer registers as the effective address and then modifies it by the contents of another register. Pointer registers are modified by other pointer registers, whereas index registers are modified by modify registers. Post-modify addressing does not support the pointer registers as destination registers, nor does it support byte-addressing. For example:

```
R5 = [ P1++P2 ] ;
```

loads a 32-bit value into the `R5` register from the memory location pointed to by the `P1` register. The value in the `P2` register is then added to the value in the `P1` register.

```
R2 = W [ P4++P5 ] (Z) ;
```

loads a 16-bit word from the memory location pointed to by the P4 register into the low half of the destination register `R2`, zero-filling it to 32 bits. The value in the `P5` register is then added to the value in the `P4` register.

```
R2 = [ I2++M1 ];
```

loads a 32-bit word from the address pointed to by I2 into the destination register `R2`, and then the value in the `I2` index register is then modified by the value in the `M1` modify register.

Direct Addressing

Direct addressing uses the immediate value field in the instruction as the effective address. The location addressed does not depend upon the contents of any register. The source or destination may be a pointer register, a data register, or a data register half. Both 8- and 16-bit read operations may specify to either sign- or zero-extend the value into the upper bits of the destination register. For example:

```
[ 0x100 ] = SP ;
```

stores the stack pointer register in the 32-bit word at address 0x100.

Sometimes, the address can be specified as a symbolic value defined at the assembler level:

```
R0 = B [ myvar ] (Z) ;
```

This instruction loads a byte from an address identified by the symbolic value `myvar`, which might be defined with a `.VAR` directive in either a native or compiler-produced assembly source file.

Direct address instructions are convenient for accessing memory-mapped registers, which are always at defined addresses. They also enable context saving without the need to modify any core registers. For example, a CPLB miss handler could be written to switch to a private stack in a known safe region of memory to protect against the case where the stack pointer itself caused the CPLB miss to begin with:

```
.SECTION scratchpad;
.ALIGN 4;
.VAR save_sp, safe_stack[BIG_ENOUGH];
...
[ save_sp ] = SP;    // save stack pointer
SP = safe_stack+BIG_ENOUGH; // make it point to the safe stack
// now registers can be saved into the safe area
[ --SP ] = ( R7:5, P5:P4 );
[ --SP ] = ASTAT;
...
// restore registers
ASTAT = [SP++];
( R7:5, P5:P4 ) = [ SP++ ];
SP = [ save_sp ];    // restore stack pointer
RTX;
```

Addressing Circular Buffers

The DAGs support addressing of circular buffers. Circular buffers are a range of addresses containing data that the DAG steps through repeatedly, wrapping around to the beginning of the buffer again and repeating stepping through the same range of addresses in a circular fashion. The DAGs use four types of data address registers for addressing circular buffers:

- The index (I) register contains the value that the DAG outputs on the address bus. For more information, see Index (Circular Buffer) Register .
- The modify (M) register contains the post-modify value (positive or negative) that the DAG applies to the index register at the end of each memory access. Any modify register can be used with any index register. The modify value can also be an immediate value rather than a value contained in a modify register. The size of the modify value must be less than or equal to the length register of the circular buffer. For more information, see Modify (Circular Buffer) Register .
- The length (L) register sets the size of the circular buffer and the address range through which the DAG circulates the index register. L is positive and cannot have a value greater than 2^{32} - 1. If a length register's value is zero, its circular buffer operation is disabled. For more information, see Length (Circular Buffer) Register .
- The base (B) register plus the length register is the value with which the DAG compares the modified index register value after each access to check for the wrap condition. When the condition is met, indexing continues to the base register address. For more information, see Base (Circular Buffer) Register .

To address a circular buffer, the DAG steps the index register through the buffer values, post-modifying and updating the index on each access with a positive or negative modify value from the modify register. If the resulting index pointer falls outside the buffer's range, the DAG automatically adjusts the value to wrap the index pointer to a location that is in the buffer.

The starting address that the DAG wraps around to is called the buffer's base address (base register). There are no restrictions on the value of the base address for circular buffers that contains 8-bit data. Circular buffers that contain 16- or 32-bit data must be 16-bit-aligned or 32-bit-aligned, respectively. Circular buffering uses post-modify addressing.

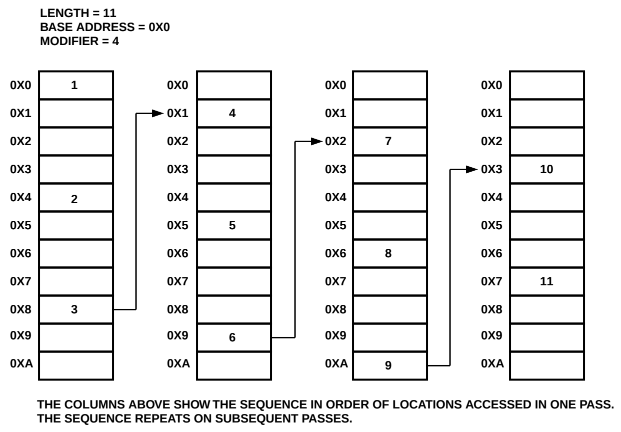

LENGTH = 11
BASE ADDRESS = 0X0
MODIFIER = 4
0X0
0X1
0X2
0X3
0X4
0X5
0X6
0X7
0X8
0X9
0XA
1
2
3
4
5
6
7
8
9
10
11
THE COLUMNS ABOVE SHOW THE SEQUENCE IN ORDER OF LOCATIONS ACCESSED IN ONE PASS.
THE SEQUENCE REPEATS ON SUBSEQUENT PASSES.

Figure 6-3: Circular Data Buffers

As seen in the ***Circular Data Buffers*** figure, on the first post-modify access to the buffer, the DAG outputs the index register value on the address bus, then modifies the address by adding the modify value.

- If the updated index value is within the buffer length, the DAG writes the value to the index register.
- If the updated index value exceeds the buffer length, the DAG subtracts (for a positive modify value) or adds (for a negative modify value) the length register value before writing the updated index value to the index register.

In equation form, these post-modify and wrap-around operations work as follows, shown for "I+M" operations.

- If M is positive:
  - $I_{new} = I_{old} + M$, if $I_{old} + M <$ buffer base + length (end of buffer)
  - $I_{new} = I_{old} + M - L$, if $I_{old} + M >$ buffer base + length (end of buffer)
- If M is negative:
  - $I_{new} = I_{old} + M$, if $I_{old} + M >$ buffer base (start of buffer)
  - $I_{new} = I_{old} + M + L$, if $I_{old} + M <$ buffer base (start of buffer)

Addressing with Bit-reversed Addresses

To obtain results in sequential order, programs need bit-reversed carry addressing for some algorithms, particularly Fast Fourier Transform (FFT) calculations. To satisfy the requirements of these algorithms, the DAG's bit-reversed addressing feature permits repeatedly subdividing data sequences and storing this data in bit-reversed order. For detailed information about bit-reversed addressing, see the description of the modify/increment instruction.

Modifying Index and Pointer Registers

The DAGs support operations that modify an address value in an index register without outputting an address. The operation, address-modify, is useful for maintaining pointers.

The address-modify operation modifies addresses in any index and pointer register (`I[3:0]`, `P[5:0]`, `FP`, `SP`) without accessing memory. If the index register's corresponding base and length registers are set up for circular buffering, the address-modify operation performs the specified buffer wrap-around (if needed).

The syntax is similar to post-modify addressing (`index += modifier`). For index registers, a modify register is used as the modifier. For pointer registers, another pointer register is used as the modifier.

Consider the example, `I1 += M2 ;` . This instruction adds `M2` to `I1` and updates `I1` with the new value.

Addressing Mode Summary

The ***Addressing Modes, Transfers, and Sizes*** table summarizes the types of transfers and transfer sizes supported by the addressing modes.

Table 6-1: Addressing Modes, Transfers, and Sizes

Addressing Mode	Types of Transfers Supported	Transfer Sizes
Auto-increment Auto-decrement Indirect Indexed Direct	To and from data registers	LOADS: 32-bit word 16-bit, zero-extended half word 16-bit, sign-extended half word 8-bit, zero-extended byte 8-bit, sign-extended byte STORES: 32-bit word 16-bit half word 8-bit byte
	To and from pointer registers	LOAD: 32-bit word STORE: 32-bit word
Post-increment	To and from data registers	LOADS: 32-bit word 16-bit half word to data register high half

Table 6-1: Addressing Modes, Transfers, and Sizes (Continued)

Addressing Mode	Types of Transfers Supported	Transfer Sizes
		16-bit half word to data register low half 16-bit, zero-extended half word 16-bit, sign-extended half word STORES: 32-bit word 16-bit half word from data register high half 16-bit half word from data register low half

The *Addressing Modes* table summarizes the addressing modes. In the table, an asterisk (*) indicates the processor supports the addressing mode.

Table 6-2: Addressing Modes

	32-bit Word	16-bit Half-Word	8-bit Byte	Sign/zero Extend	Data Register	Pointer Register	Data Register Half
P Auto-inc [P0++]	*	*	*	*	*	*	
P Auto-dec [P0--]	*	*	*	*	*	*	
P Indirect [P0]	*	*	*	*	*	*	*
P Indexed [P0+im]	*	*	*	*	*	*	
FP indexed [FP+im]	*				*	*	
P Post-inc [P0++P1]	*	*		*	*		*
I Auto-inc [I0++]	*	*			*		*
I Auto-dec [I0--]	*	*			*		*
I Indirect [I0]	*	*			*		*
I Post-inc	*				*		

Table 6-2: Addressing Modes (Continued)

	32-bit Word	16-bit Half-Word	8-bit Byte	Sign/zero Extend	Data Register	Pointer Register	Data Register Half
[I0++M0]							
Direct [im]	*	*	*	*	*	*	*

AAU Instruction Summary

The *AAU Instructions* table lists the AAU instructions. In the table, note the meaning of these symbols:

- Dreg denotes any data register file register.
- Dreg_lo denotes the lower 16 bits of any data register file register.
- Dreg_hi denotes the upper 16 bits of any data register file register.
- Preg denotes any pointer register, `FP`, or `SP` register.
- Ireg denotes any index register.
- Mreg denotes any modify register.
- W denotes a 16-bit wide value.
- B denotes an 8-bit wide value.
- immA denotes a signed, A-bit wide, immediate value.
- uimmAmB denotes an unsigned, A-bit wide, immediate value that is an even multiple of B.
- Z denotes the zero-extension qualifier.
- X denotes the sign-extension qualifier.
- BREV denotes the bit-reversal qualifier.

AAU instructions do not affect the `ASTAT` register status bits.

Table 6-3: AAU Instructions

Instruction
Preg = [Preg] ;
Preg = [Preg ++] ;
Preg = [Preg --] ;
Preg = [Preg + uimm6m4] ;
Preg = [Preg + uimm17m4] ;
Preg = [Preg - uimm17m4] ;

Table 6-3: AAU Instructions (Continued)

Instruction
Preg = [FP - uimm7m4] ;
Dreg = [Preg] ;
Dreg = [Preg ++] ;
Dreg = [Preg --] ;
Dreg = [Preg + uimm6m4] ;
Dreg = [Preg + uimm17m4] ;
Dreg = [Preg - uimm17m4] ;
Dreg = [Preg ++ Preg] ;
Dreg = [FP - uimm7m4] ;
Dreg = [uimm32] ;
Dreg = [Ireg] ;
Dreg = [Ireg ++] ;
Dreg = [Ireg --] ;
Dreg = [Ireg ++ Mreg] ;
Dreg =W [Preg] (Z) ;
Dreg =W [Preg ++] (Z) ;
Dreg =W [Preg --] (Z) ;
Dreg =W [Preg + uimm5m2] (Z) ;
Dreg =W [Preg + uimm16m2] (Z) ;
Dreg =W [Preg - uimm16m2] (Z) ;
Dreg =W [Preg ++ Preg] (Z) ;
Dreg =W [uimm32] (Z) ;
Dreg = W [Preg] (X) ;
Dreg = W [Preg ++] (X) ;
Dreg = W [Preg --] (X) ;
Dreg =W [Preg + uimm5m2] (X) ;
Dreg =W [Preg + uimm16m2] (X) ;
Dreg =W [Preg - uimm16m2] (X) ;
Dreg =W [Preg ++ Preg] (X) ;
Dreg =W [uimm32] (X) ;
Dreg_hi = W [Ireg] ;

Table 6-3: AAU Instructions (Continued)

Instruction
Dreg_hi = W [Ireg ++] ;
Dreg_hi = W [Ireg --] ;
Dreg_hi = W [Preg] ;
Dreg_hi = W [Preg ++ Preg] ;
Dreg_hi = W [uimm32] ;
Dreg_lo = W [Ireg] ;
Dreg_lo = W [Ireg ++] ;
Dreg_lo = W [Ireg --] ;
Dreg_lo = W [Preg] ;
Dreg_lo = W [Preg ++ Preg] ;
Dreg_lo = W [uimm32] ;
Dreg = B [Preg] (Z) ;
Dreg = B [Preg ++] (Z) ;
Dreg = B [Preg --] (Z) ;
Dreg = B [Preg + uimm15] (Z) ;
Dreg = B [Preg - uimm15] (Z) ;
Dreg = B [uimm32] (Z) ;
Dreg = B [Preg] (X) ;
Dreg = B [Preg ++] (X) ;
Dreg = B [Preg --] (X) ;
Dreg = B [Preg + uimm15] (X) ;
Dreg = B [Preg - uimm15] (X) ;
Dreg = B [uimm32] (X) ;
[Preg] = Preg ;
[Preg ++] = Preg ;
[Preg --] = Preg ;
[Preg + uimm6m4] = Preg ;
[Preg + uimm17m4] = Preg ;
[Preg - uimm17m4] = Preg ;
[FP - uimm7m4] = Preg ;
[uimm32] = Preg ;

Table 6-3: AAU Instructions (Continued)

Instruction
[Preg] = Dreg ;
[Preg ++] = Dreg ;
[Preg --] = Dreg ;
[Preg + uimm6m4] = Dreg ;
[Preg + uimm17m4] = Dreg ;
[Preg - uimm17m4] = Dreg ;
[Preg ++ Preg] = Dreg ;
[FP - uimm7m4] = Dreg ;
[uimm32] = Dreg ;
[Ireg] = Dreg ;
[Ireg ++] = Dreg ;
[Ireg --] = Dreg ;
[Ireg ++ Mreg] = Dreg ;
W [Ireg] = Dreg_hi ;
W [Ireg ++] = Dreg_hi ;
W [Ireg --] = Dreg_hi ;
W [Preg] = Dreg_hi ;
W [Preg ++ Preg] = Dreg_hi ;
W [uimm32] = Dreg_hi ;
W [Ireg] = Dreg_lo ;
W [Ireg ++] = Dreg_lo ;
W [Ireg --] = Dreg_lo ;
W [Preg] = Dreg_lo ;
W [Preg ++ Preg] = Dreg_lo ;
W [uimm32] = Dreg_lo ;
W [Preg] = Dreg ;
W [Preg ++] = Dreg ;
W [Preg --] = Dreg ;
W [Preg + uimm5m2] = Dreg ;
W [Preg + uimm16m2] = Dreg ;
W [Preg - uimm16m2] = Dreg ;

Table 6-3: AAU Instructions (Continued)

Instruction
W [uimm32] = Dreg ;
B [Preg] = Dreg ;
B [Preg ++] = Dreg ;
B [Preg --] = Dreg ;
B [Preg + uimm15] = Dreg ;
B [Preg - uimm15] = Dreg ;
B [uimm32] = Dreg ;
Preg = imm7 (X) ;
Preg = imm16 (X) ;
Preg = uimm32;
Preg += Preg (BREV) ;
Ireg += Mreg (BREV) ;
Preg = Preg << 2 ;
Preg = Preg >> 2 ;
Preg = Preg >> 1 ;
Preg = Preg + Preg << 1 ;
Preg = Preg + Preg << 2 ;
Preg -= Preg ;
Ireg -= Mreg ;

ADSP-BF70x Address Arithmetic Unit Register Descriptions

The AAU Register File contains the following registers.

Table 6-4: ADSP-BF70x AAU Register List

Name	Description
`FP`	Frame Pointer Register
`SP`	Stack Pointer Register
`USP`	User Stack Pointer Register
`P[n]`	Pointer Register (n = 0 - 5)
`I[n]`	Index (Circular Buffer) Register (n = 0 - 3)
`M[n]`	Modify (Circular Buffer) Register (n = 0 - 3)
`B[n]`	Base (Circular Buffer) Register (n = 0 - 3)

Table 6-4: ADSP-BF70x AAU Register List (Continued)

Name	Description
`L[n]`	Length (Circular Buffer) Register (n = 0 - 3)

Pointer Register

There are six 32-bit general-purpose pointer registers `P[n]` that are primarily used for load/store operations. Although pointer registers are primarily used for address calculations, these registers may also be used for general integer arithmetic with a limited set of arithmetic operations; however, unlike computations involving data registers (`R[n]`), pointer register arithmetic does not affect the `ASTAT` status bits.

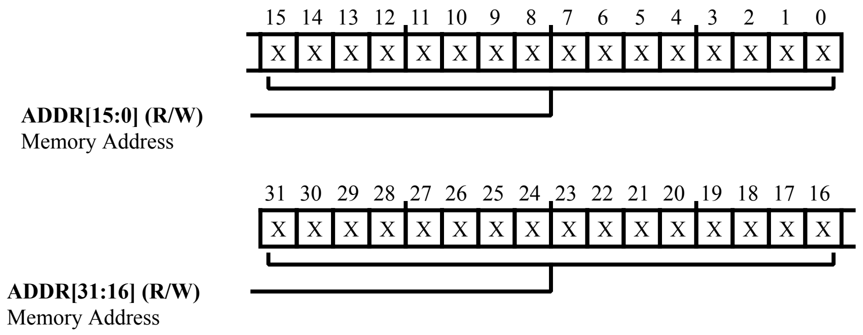


Figure 6-4: P[n] Register Diagram

Table 6-5: P[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Memory Address. The `P[n].ADDR` bit field (bits 31:0) holds either an address (for address calculations or load/store operations) or data for arithmetic operations.

Frame Pointer Register

The FP register contains the address of the current frame on the system stack. Frames are required to control program flow in the context of sub-routines and consist of the previous FP value, the function return information (RETS), and the sub-routine's local stack.

The FP register performs like the general-purpose P[n] pointer registers, acting as a general pointer in any load/ store instruction.

The LINK and UNLINK instructions, which control stack frame space, implicitly use and modify the FP register.

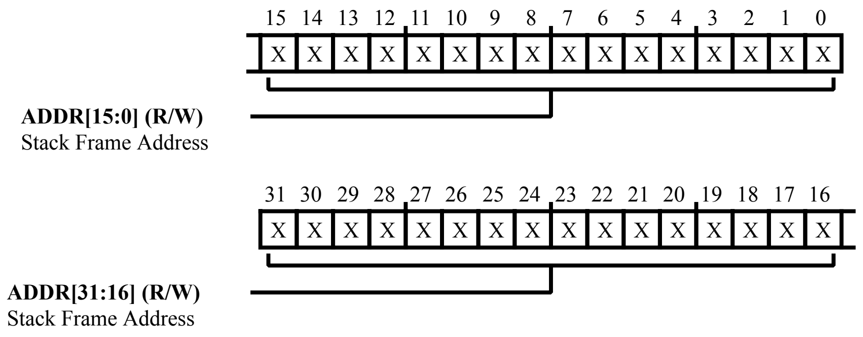


Figure 6-5: FP Register Diagram

Table 6-6: FP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Stack Frame Address. The FP.ADDR bit field (bits 31:0) hold either an address for address calculations or the address of the current frame on the system stack.

Stack Pointer Register

The SP register monitors the current index into the run-time stack, which contains critical run-time information such as stored context and local variables/arguments. In many respects, the SP register is like the general-purpose P[n] pointer registers and can be used in any load/store instruction.

To speed up context switching, there are two stack pointer registers, a User stack pointer (USP) and a Supervisor stack pointer (SP). In assembly code, only the SP syntax is used, as the correct stack pointer register will be used based on whether the processor is in Supervisor or User mode.

The LINK and UNLINK instructions, which control stack frame space, implicitly use and modify the SP register.

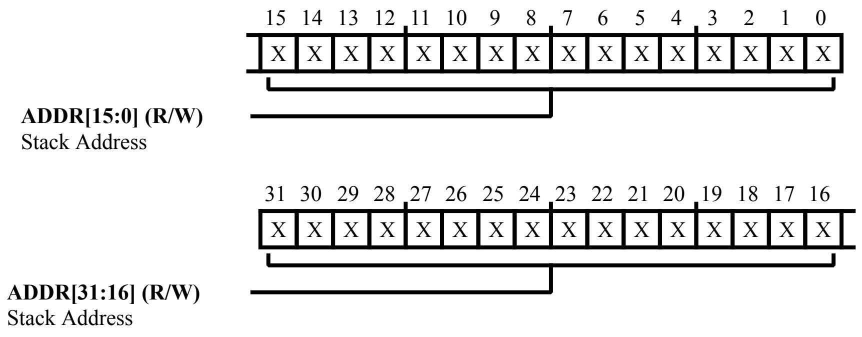


Figure 6-6: SP Register Diagram

Table 6-7: SP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Stack Address. The SP.ADDR bit field (bits 31:0) holds an address for address calculations or stack operations in Supervisor mode.

User Stack Pointer Register

The User (mode) Stack Pointer (USP) is a virtual register that is used by the processor for SP accesses when it is in User mode. For context switching while in Supervisor mode, the USP register can be explicitly referenced as USP; however, if an explicit USP access is attempted while executing in User mode, an exception is generated.

The USP register can be used like the general-purpose P[n] pointer registers, acting as a general pointer in any of the load/store instructions.

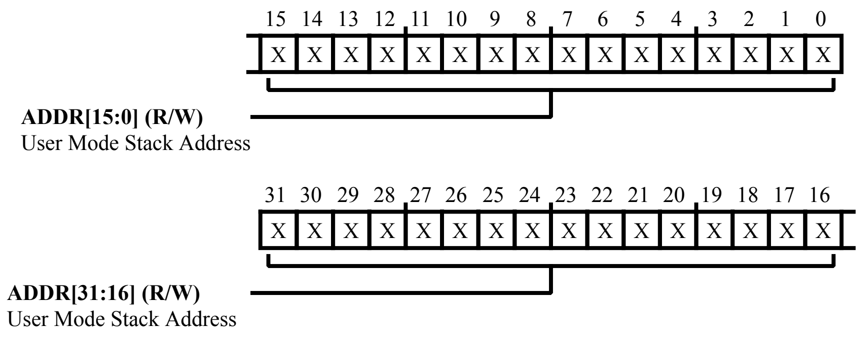


Figure 6-7: USP Register Diagram

Table 6-8: USP Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	User Mode Stack Address. The USP.ADDR bit field (bits 31:0) holds an address for address calculations or stack operations in User mode.

Index (Circular Buffer) Register

Instructions primarily use the 32-bit data address generator (DAG) register set for addressing. The `I[n]` Index registers (I3 through I0) typically contain indexed addresses within a buffer, but they can be used to access individual data elements in memory as well.

The `I[n]` registers always contain byte addresses in memory and are used in conjunction with their associated base (`B[n]`) and length (`L[n]`) registers to manage buffers (i.e., the I0/B0/L0 register set define buffer 0). When a `L[n]` register contains a non-zero value, the buffer pointed to by the corresponding `B[n]` register is defined to be circular, meaning that indexing through the buffer using the corresponding `I[n]` register will wrap back to the `B[n]` address when the length defined in the corresponding `L[n]` register is exceeded. While the `I[n]`/`B[n]`/`L[n]` register set are grouped, any `M[n]` modify register can be used to post-modify an `I[n]` register after a load/store operation (e.g., I0 may be modified by M3).

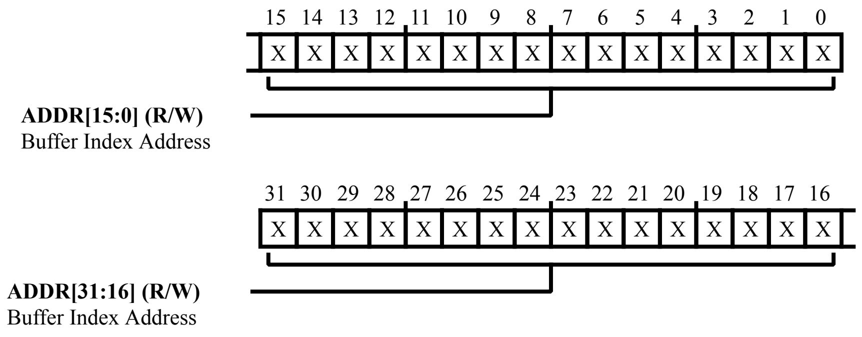


Figure 6-8: I[n] Register Diagram

Table 6-9: I[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Buffer Index Address. The `I[n].ADDR` bits contain the indexed address within a buffer in memory.

Modify (Circular Buffer) Register

The 32-bit M[n] modify registers (M3 through M0) contain byte offsets to be applied to the designated I[n] index register in any load/store with modify instruction. While the I[n]/B[n]/L[n] register set is grouped by number, any M[n] modify register can be used to modify an I[n] register when a load/store operation is executed (e.g., I0 may be modified by M3).

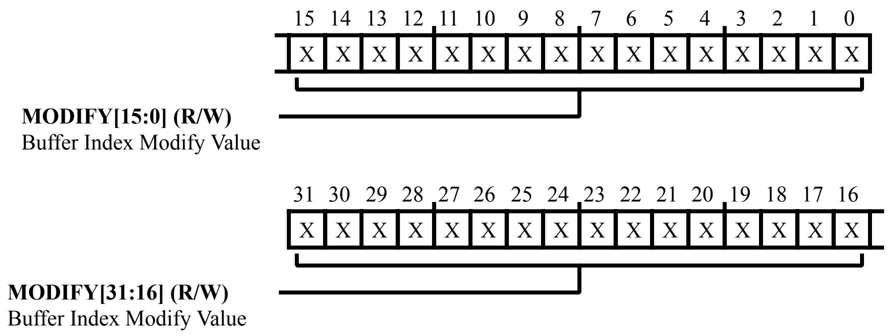


Figure 6-9: M[n] Register Diagram

Table 6-10: M[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	MODIFY	Buffer Index Modify Value. The M[n].MODIFY bits contain the modify value applied to the designated I[n] register when a load/store with modify operation is executed.

Base (Circular Buffer) Register

The 32-bit B[n] base registers (B3 through B0) contain byte addresses and are used in conjunction with their associated index (I[n]) and length (L[n]) registers to manage buffers in memory (i.e., the I0/B0/L0 register set is grouped and defines buffer 0). When a L[n] register contains a non-zero value, the buffer pointed to by the corresponding B[n] register is defined to be circular, meaning that indexing through the buffer using the corresponding I[n] register will wrap back to the B[n] address when the length defined in the corresponding L[n] register is exceeded.

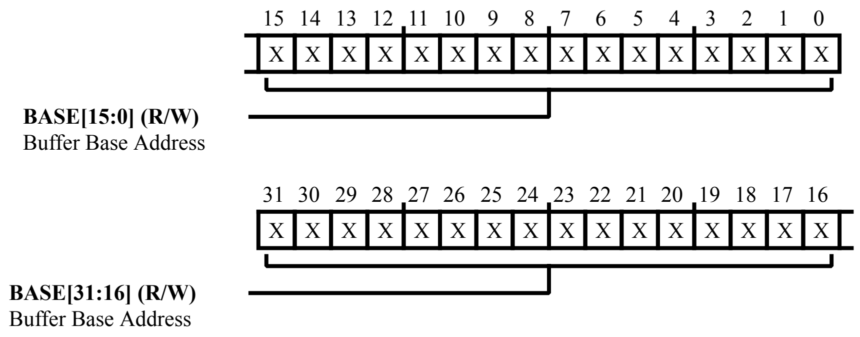


Figure 6-10: B[n] Register Diagram

Table 6-11: B[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	BASE	Buffer Base Address. The B[n].BASE bits contain the base address for a buffer in memory. When the associated L[n] register is non-zero, indexing through the buffer wraps to this address when the length is exceeded.

Length (Circular Buffer) Register

The 32-bit L[n] length registers (L3 through L0) contain the length in bytes of a buffer in memory. They are used in conjunction with their associated base (B[n]) and index (I[n]) registers to manage buffers (i.e., the I0/B0/L0 register set is grouped and defines buffer 0). When a L[n] register contains a non-zero value, the buffer pointed to by the corresponding B[n] register is defined to be circular, meaning that indexing through the buffer using the corresponding I[n] register will wrap back to the B[n] address when the length defined in the corresponding L[n] register is exceeded.

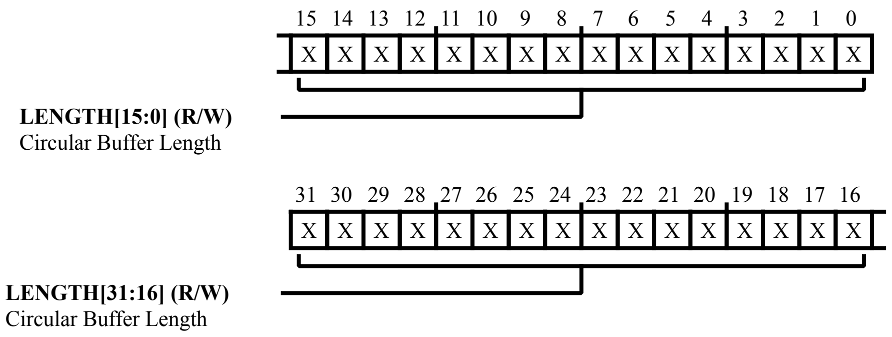


Figure 6-11: L[n] Register Diagram

Table 6-12: L[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	LENGTH	Circular Buffer Length. The L[n].LENGTH bits contain the length of the circular buffer, which determines the offset from the associated B[n] base address where indexing through the buffer will wrap back to the base. When L[n].LENGTH is 0, the buffer is defined to be unbounded (indexing will never wrap).

7 Memory

Blackfin+ processors support a hierarchical memory model with different performance and size parameters, depending on the memory location within the hierarchy. Level 1 (L1) instruction and data memories interconnect closely and efficiently with the Blackfin+ core to achieve the best performance. Separate blocks of L1 memory can be accessed simultaneously through multiple bus systems. Instruction memory is separated from data memory, but unlike classic Harvard architectures, all L1 memory blocks are accessed by a unified addressing scheme. Portions of L1 memory can be configured to function as cache memory.

The Blackfin+ processors feature on-chip Level 2 (L2) memory, which can freely store both instructions and data but take more core clock cycles to access than L1 memory, as well as external memory space (including asynchronous memory for static RAM devices and synchronous memory for dynamic RAM, such as DDR SDRAM devices).

This chapter discusses the architecture and principles of L1 memories, as well as memory protection and caching mechanisms. For memory sizes, locations, and definitions for both L2 and off-chip memory interfaces, refer to the specific *Blackfin+ Processor Hardware Reference*.

Memory Architecture

Blackfin+ processors have a unified 4 GB address range that spans a combination of on-chip and off-chip memory and memory-mapped I/O resources. Of this range, some of the address space is dedicated to internal, on-chip resources, populated as:

- Level-1 (Core) Static Random Access Memories (L1 SRAM)
- Level-2 (Core) Static Random Access Memories (L2 SRAM)
- A set of memory-mapped registers (MMRs)
- A boot Read-Only Memory (ROM)

The Processor Memory Architecture figure shows a block diagram.

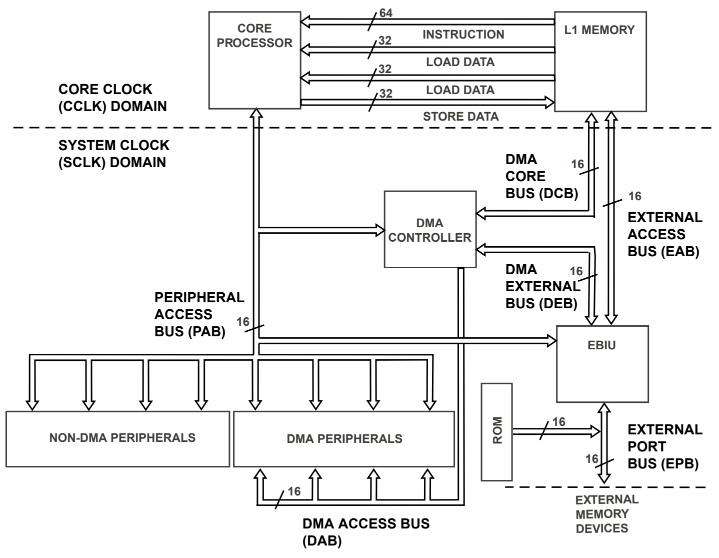


Figure 7-1: Processor Memory Architecture

Overview of On-Chip Level-1 (L1) Memory

The L1 memory system performance provides high bandwidth and low latency. Because SRAMs provide deterministic access time and very high throughput, embedded processing systems have traditionally achieved performance improvements by providing fast SRAM on the chip.

The addition of instruction and data caches (SRAMs with cache control hardware) provides both higher performance and a simple programming model. Caches eliminate the need to explicitly manage data movement into and out of L1 memory. Code can be ported to or developed for the processor quickly, without requiring performance optimization for the memory organization.

L1 memory provides:

- A modified Harvard architecture, allowing up to four core memory accesses per clock cycle (one 64-bit instruction fetch, two 32-bit data loads, and one pipelined 32-bit data store)
- Simultaneous system DMA, cache maintenance, and core accesses
- SRAM accesses at the processor's core clock rate (CCLK) for critical algorithms and fast context switching
- Instruction and data cache options for microcontroller code
- Excellent High-Level Language (HLL) support
- Ease-of-programming cache control instructions such as `PREFETCH` and `FLUSH`
- Memory protection

L1 instruction memory is a continuous region of memory which may only be used to store instructions. A portion of this memory is dedicated instruction SRAM, and the remainder may be configured as instruction cache or as dedicated instruction SRAM. L1 instruction memory cannot normally be accessed by load or store instructions by the core.

Typical L1 data memory is divided into three blocks. Two, known as block A and block B, are equal-sized and may be configured as cache or SRAM, and the third, known as block C or scratchpad, is a small region of dedicated data SRAM often used for system stacks and heaps. Concurrent accesses by the core or DMA to separate blocks proceed in parallel, whereas concurrent accesses to the same block may be stalled due to a sub-bank conflict.

The L1 memory system contains special-purpose memory spaces, such as cache tags and parity bits, to which direct access by load or store instructions is normally restricted. An Extended Data Access mode is supported, in which all restricted memory spaces including L1 instruction memory may be directly accessed by suitably privileged software.

Overview of Other On-Chip (L2) and Off-Chip (L3) Memories

Blackfin+ processors feature an on-chip Level-2 (L2) memory, so named because it forms an on-chip memory hierarchy with L1 memory. On-chip L2 memory provides more capacity than L1 memory, but the access latency is higher. It is capable of storing both instructions and data biut cannot be configured as cache. The L2 memory also contains the processor Boot ROM.

Blackfin+ processors also feature external (off-chip) memory controllers which enable access to dynamic external DRAM via standard protocols such as DDR2, as well as to static RAM and flash memory devices. The external memory is sometimes referred to as Level-3 (L3) memory.

For details of the on-chip and off-chip memory map, refer to the *Blackfin+ Processor Hardware Reference*.

L1 Instruction Memory

L1 instruction memory is a continuous region of the address space which may only be used for storing instructions. On all Blackfin+ processors, a subset of this space must be used as directly addressable SRAM, but a dedicated portion of this L1 instruction memory space can be optionally be configured as instruction cache memory. Control bits in the `L1IM_ICTL` register can be used to configure this portion of L1 instruction memory as a 4-Way, set-associative instruction cache.

The L1 instruction memory content is not accessible by core load or store operations during normal operation. This memory's content may be read and modified using either DMA or by loads and stores in Extended Data Access mode.

L1 Instruction SRAM

The processor core reads instruction memory via a 64-bit instruction fetch bus. All addresses from this bus are 64-bit aligned. Each instruction fetch can return any combination of 16-, 32- and 64-bit instructions (i.e., four 16-bit instructions, two 16-bit instructions and one 32-bit instruction, two 32-bit instructions, or one 64-bit instruction).

The pointer registers and index registers, which are described in the *Address Arithmetic Unit* chapter, may only access L1 instruction memory directly when Extended Data Access is enabled (see Extended Data Access), otherwise a direct access to an address in instruction memory SRAM space generates an exception. Write access to the L1 instruction SRAM memory may also be made through the 64-bit system DMA port.

Typically, the SRAM is implemented as a collection of single-ported sub-banks. Writes to one sub-bank may proceed in parallel with reads from another, effectively making the instruction memory dual-ported.

L1 Instruction Cache

For information about cache terminology, see Terminology.

A portion of the L1 instruction memory can be configured as a 16 KB 4-way set-associative instruction cache, featuring a cache line size of 32 bytes. To improve the average access latency for critical code sections, each line of the cache can be locked independently. When the memory is configured as cache, it can only be accessed directly by loads and stores if Extended Data Access mode is enabled.

When cache is enabled, only memory pages further specified as cacheable by the cacheability protection lookaside buffers (CPLBs) are cached. When CPLBs are enabled, any memory location that is accessed must have an associated page definition available, else a CPLB exception is generated. CPLBs are described in Memory Protection and Properties.

The figures in Cache Lines show the organization of the Blackfin+ processor instruction cache memory.

Enabling L1 Instruction Cache

The `L1IM_ICTL.CFG` bit reserves a portion of L1 instruction SRAM to serve as cache. Setting this bit also causes the cache tags and dirty bits to be initialized to the Invalid state.

To store instructions in L2 and off-chip memories in the cache, enable the ICPLBs using the `L1IM_ICTL.ENCPLB` bit and the ICPLB descriptors (`L1IM_ICPLB_DFLT` or `L1IM_ICPLB_DATAx` and `L1IM_ICPLB_ADDRx` registers). The configuration must specify the desired memory pages as cacheable. For more information, see Memory Protection and Properties.

The memory system must maintain a consistent view of the cacheability of any memory word. As such, use SSYNC instructions to ensure there is no memory traffic in progress while the cache mode is being changed. The code that enables or disables the instruction cache must also not be cacheable itself. Additionally, do not enable ICPLBs when the cache is initializing.

The recommended sequence for enabling the instruction cache is:

- with ICPLBs disabled, execute an `SSYNC;` instruction to ensure that any outstanding instruction fetches have completed
- set `L1IM_ICTL.CFG` to reserve the cache and initialize the cache tags
- program the `L1IM_ICPLB_DFLT`, `L1IM_ICPLB_DATAx` and `L1IM_ICPLB_ADDRx` registers as required
- execute a `CSYNC;` instruction to ensure tag initialization is complete
- set `L1IM_ICTL.ENCPLB` to enable ICPLBs
- execute one final `SSYNC;` instruction

NOTE: The entire code sequence above must not be in a cacheable region of memory.

Cache Lines

As shown in the Instruction Cache Organization figure, the cache consists of a collection of cache lines. Each cache line is made up of a tag component and a data component.

- The tag component incorporates a 20-bit address tag, replacement policy bits (including a Priority bit), and a Valid bit.
- The data component is made up of four 64-bit words of instruction data.

The tag and data components of cache lines are stored in the tag and data memory arrays, respectively.

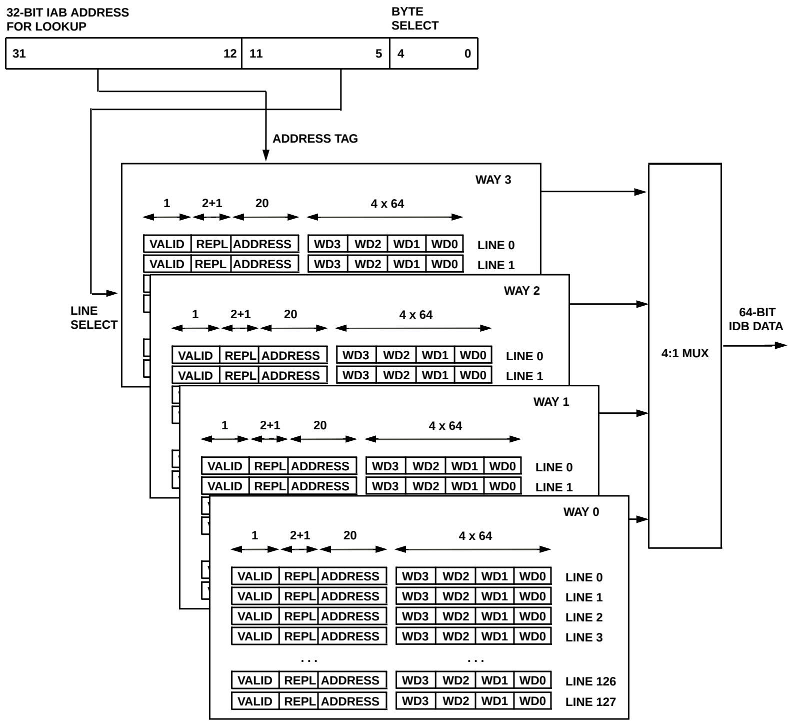


Figure 7-2: Instruction Cache Organization

The Replacement bits are part of a round-robin replacement algorithm used to determine which cache line should be replaced if a cache miss occurs.

The Valid bit indicates the state of a cache line. A cache line is always valid or invalid.

- Invalid cache lines have their Valid bit cleared, indicating the line will be ignored during an address-tag compare operation.
- Valid cache lines have their Valid bit set, indicating the line contains valid instruction data that is consistent with the source memory.

The tag and data components of a cache line are illustrated in the Cache Line - Tag and Data Portions figure.

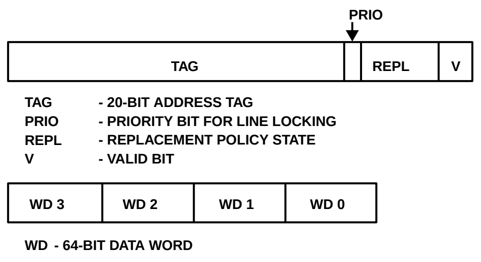


Figure 7-3: Cache Line - Tag and Data Portions

Cache Hits and Misses

A cache hit occurs when the address for an instruction fetch request from the core matches a valid entry in the cache. Specifically, a cache hit is determined by comparing the upper 20 bits of the instruction fetch address to the address tags of valid lines currently stored in a cache set. The cache set (cache line across ways) is selected using bits 11 through 5 of the instruction fetch address. If the address tag compare operation results in a match in any of the four ways and the respective cache line is valid, a cache hit occurs. If the address tag compare operation does not result in a match in any of the four ways or the respective line is not valid, a cache miss occurs.

When a cache miss occurs, the instruction memory unit generates a cache line fill access to retrieve the missing instruction from the source memory external to the core. The address for the external memory access is the address of the 32-byte block containing the target instruction word.

A location in the cache is selected to store the missing block once the cache line fill has completed. If the tag address compare operation results in a cache miss, the Valid, Replacement and Priority bits for the selected set are examined by a cache line replacement unit to determine the entry to use for the new cache line (whether that is Way0, Way1, Way2, or Way3).

The cache line replacement unit first checks for invalid entries (entries with the Valid bit cleared). If there are no invalid entries in the set, an entry with the Priority bit cleared is selected. In the last resort, the cache line replacement unit will select an entry with its Priority bit set. Valid entries are selected using a replacement algorithm designed to choose the entry that is statistically least likely to be accessed again. The Replacement bits play a role in the implementation of this algorithm.

When a cache miss occurs, the core halts until the target instruction word is returned from external memory.

Instruction Cache Management

The system DMA controller and the core DAGs cannot normally access the instruction cache directly. By a combination of instructions and the use of Extended Data Access mode, it is possible to initialize the instruction tag and data arrays and provide a mechanism for instruction cache test, initialization, and debug.

Instruction Cache Bypass Mode

The `L1IM_ICTL.CBYPASS` bit enables Instruction Cache Bypass mode. In this mode, instructions in cacheable memory are loaded directly from the system without disturbing the contents of the cache. To ensure a consistent view of cacheability in the memory, system code that enables, disables and configures cache must respect the following guidelines:

- bypass mode must not be cacheable
- execute a SSYNC; instruction before enabling and after disabling cache bypass mode

Instruction Cache Locking by Line

The CPRIO bits in the L1IM_ICPLB_DATAx registers (see Memory Protection and Properties) are used to enhance control over which code remains resident in the instruction cache. When a cache line is filled, the state of this bit is stored along with the line's tag. It is then used with the replacement policy to determine which way is victimized if all the cache ways are occupied when a new cacheable line is fetched. This bit indicates that a line is of either "low" or "high" importance. In a modified replacement policy, a high can replace a low, but a low cannot replace a high. If all four ways are occupied by highs, a cacheable low may replace a high. If all previously cached highs ever become less important, they may be simultaneously transformed into lows by setting the L1IM_ICTL.CPRIORST bit.

When the L1IM_ICTL.CPRIORST bit is set to 1, the cached states of all CPRIO bits are cleared. This simultaneously forces all cached lines to be of equal (low) importance. L1IM_ICTL.CPRIORST must be 0 to allow the state of the CPRIO bits to be stored when new lines are cached.

Instruction Cache Invalidation

The instruction cache can be invalidated as a whole, by address, or by cache line.

The simplest method for invalidating the entire instruction cache is to disable and then re-enable it. Setting L1IM_ICTL.CFG to 0, and then, in a second write, setting L1IM_ICTL.CFG to 1 causes the cache to be re-initialized with all Valid bits set to the Invalid state. As with any code that changes the instruction cache mode, these writes must be preceded and followed by an SSYNC instruction, and the instruction sequence must not be in cacheable memory.

The IFLUSH instruction can explicitly invalidate cache lines based on their line addresses. The target address of the instruction is generated from the pointer registers. Because the instruction cache should not contain modified (dirty) data, the cache line is simply invalidated and is not "flushed".

In the following example, the P2 register contains the address of a valid memory location. If this address has been brought into cache, the corresponding cache line is invalidated after the execution of this IFLUSH instruction:

```
IFLUSH [ P2 ] ;   /* Invalidate cache line containing address that P2 points to */
```

Because the IFLUSH instruction is used to invalidate a specific address in the memory map and its corresponding cache line, it is most useful when the buffer being invalidated is less than the cache size.

Finally, larger portions of the cache can be invalidated by writing directly to the Valid bits while Extended Data Access is enabled (see Extended Data Access).

L1 Data Memory

L1 data memory is organized as a number of distinct blocks, each of which constitutes a separate contiguous region of the address space. Accesses to different blocks are guaranteed not to collide, whereas accesses within a single block will not collide if they are to different sub-banks. When there are no collisions, the following L1 data traffic could occur in a single core clock cycle:

- Two 32-bit data accesses (two loads or one load and one store)
- One DMA I/O, up to 32 bits
- One 64-bit cache fill or victim access

There are three blocks of L1 data SRAM, blocks A, B and C. Parts of the two larger blocks, A and B, may be configured as data cache, as controlled by bits in the `L1DM_DCTL` register.

The processor cannot fetch instructions directly from L1 data memory.

L1 Data SRAM

L1 data SRAM is directly addressable by the processor core and by the DMA controller. Core accesses are performed with no stalls, so long as access collisions are avoided; therefore, configuring the whole of L1 data memory as SRAM is potentially the most efficient way to use it. However, this is at the cost of additional software complexity when data structures are larger than available L1 memory. In a common use case, DMA engines transfer data between L1 and the system while the core processes data previously loaded to L1.

To optimize memory performance, the programmer must ensure that stalls due to collisions are avoided. The Blackfin+ architecture guarantees that accesses to different blocks do not collide. For example, a tight loop which loads two operands per cycle will not incur stalls due to the loads if the operands are placed in separate blocks.

Depending on the memory microarchitecture, two accesses to the same block may not collide. However, a collision due to two loads in a parallel issue instruction will not take more cycles than the additional cycles resulting from executing one of the loads in a separate instruction.

Collisions between the core and DMA are less common because DMA runs at system clock speeds, which are some fraction of the core clock. DMA accesses are also usually delayed behind core accesses, but after some delay a fairness algorithm ensures that the DMA gets access to L1. If possible, DMA accesses should be to a different block than concurrent core accesses.

L1 Data Cache

For definitions of cache terminology, see Terminology.

Each of the A and B blocks of L1 data memory can be configured to serve as a 16 KB 2-way set associative data cache (up to 32 KB total), featuring a cache line length of 32 bytes. When the memory is configured as cache, it can only be accessed directly by loads and stores if Extended Data Access mode is enabled.

If cache is enabled in the L1DM_CTL.CFG[1:0] bits, data CPLBs must also be enabled by setting the L1DM_CTL.ENDCPLB bit. Only memory pages specified as cacheable by data CPLBs will be cached. The default behavior when data CPLBs are disabled is for nothing to be cached.

Access to core MMR space is not controlled by the data CPLBs, so this region cannot be configured as cacheable.

Enabling L1 Data Cache

The L1DM_DCTL.CFG[1:0] bits can reserve a portion of L1 data SRAM to serve as cache. At reset, all L1 data memory serves as SRAM by default, but these bits allow for either 16 KB (from block A only) or 32 KB (16 KB from each of blocks A and B) to be enabled as cache. Configuring either one or two blocks as cache also causes the cache tags to be initialized with the Valid bit set to the Invalid state.

To store data from L2 and off-chip memories in the cache, enable the DCPLBs using the L1DM_DCTL.ENCPLB bit. The DCPLB descriptors (L1DM_DCPLB_DFLT or LDIM_DCPLB_DATAx and L1DM_DCPLB_ADDRx registers) must specify the desired memory pages as cacheable. For more information, see Memory Protection and Properties.

The memory system must maintain a consistent view of the cacheability of any memory word. Use SSYNC instructions to ensure that there is no memory traffic in progress while the cache is being enabled or disabled. Also, avoid enabling the cache while the cache tags are initializing. The recommended sequence for enabling the data cache is:

- Execute a SSYNC instruction with DCPLBs disabled to ensure that any outstanding memory traffic has completed.
- Set the L1DM_DCTL.CFG[1:0] bits, as appropriate, to reserve the cache and initialize the cache tags.
- Program the L1DM_DCPLB_DFLT, L1DM_DCPLB_DATAx and L1DM_DCPLB_ADDRx registers, as required.
- Execute a CSYNC instruction to ensure tag initialization is complete.
- Set L1DM_CTL.ENCPLB to enable DCPLBs.
- Execute one final SSYNC instruction to ensure strong ordering against potential subsequent read instructions.

Data Cache Access

The cache controller tests the address from the DAGs against the tag bits. If the logical address is present in L1 cache, a cache hit occurs, and the data is accessed in L1. If the logical address is not present, a cache miss occurs, and the memory transaction is passed to the next level of memory via the system interface.

The set index and replacement policy for the cache controller determines the cache tag and data space that are allocated for the data coming back from external memory.

A data cache line is in one of three states: Invalid, Exclusive (valid and clean), or Modified (valid and dirty). If valid data already occupies the allocated line and the cache is configured for write-back storage, the controller checks the state of the cache line and treats it accordingly:

- If the state of the line is exclusive (clean), the new tag and data overwrite the old line.

- If the state of the line is modified (dirty), then the cache contains the only valid copy of the data. This is copied back to external memory before the new tag and data is written to the cache.

Cache Write Method

Cache write memory operations can be implemented by using either a write-through method or a write-back method:

- For each store operation, write-through caches initiate a write to external memory immediately upon the write to cache. If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.
- A write-back cache does not write to external memory until the line is replaced by a load operation that needs the line. For most applications, a write-back cache is more efficient than a write-through cache, as the external memory accesses are less frequent.

The cache-write method is selected by the data CPLB descriptors (see Memory Protection and Properties). Since the cache mode is independently selectable between memory pages, these cache modes can be used simultaneously.

Data Cache Block Select

When both data blocks A and B have memory serving as cache, the `L1DM_DCTL.DCBS` bit may be used to control whether the sets in each block are to act as a single cache or as two caches which may be accessed in parallel. For the best general-purpose operation, a single cache (DCBS=0) should be selected.

When both blocks A and B are configured as cache, they operate as two independent 16 KB 2-way set-associative caches that can be independently mapped into the Blackfin+ processor address space. The `L1DM_DCTL.DCBS` bit designates address bit `A[14]` or `A[23]` as the cache selector, which selects the cache implemented by data block A or the cache implemented by data block B:

- If `DCBS = 0`, then `A[14]` is part of the address index. All addresses in which `A[14] = 0` use data block B, and all addresses in which `A[14] = 1` use data block A. In this case, `A[23]` is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.
- If `DCBS = 1`, then `A[23]` is part of the address index. All addresses in which `A[23] = 0` use data block B, and all addresses in which `A[23] = 1` use data block A. In this case, `A[14]` is treated as merely another bit in the address that is stored with the tag in the cache and compared for hit/miss processing by the cache.

The result of choosing `DCBS = 0` or `DCBS = 1` is:

- If `DCBS = 0`, `A[14]` selects data block A instead of data block B. Alternating 16 KB regions of the non-L1 memory map will target the two respective 16 KB caches implemented by the two data blocks A and B. As a result, the cache operates as if it were a single, contiguous, 2-way set-associative 32 KB cache. Each way is 16 KB in length, and all of the data elements with the same first 14 address bits will index to a unique set, in which up to two elements can be stored (one in each way).

  - For a given large region of memory, data in the first 16 KB of that memory (offset 0x0000 - 0x3FFF) will be cached only in data block B. Data in the next 16 KB address range (offset 0x4000 - 0x7FFF) will be cached only in data block A, and so on.
- If `DCBS = 1`, `A[23]` selects data block A instead of data block B. With `DCBS = 1`, the system functions more like two independent 16 KB caches, each being 2-way set-associative and serving alternating blocks of 8 MB source memory regions. Data block B caches all data accesses for the first 8 MB of the memory address range, with each access vying for the two line entries. Likewise, data block A caches data located above 8 MB and below 16 MB, and so on.

For example, if `DCBS = 1` and the application utilizes a 1 MB data buffer located entirely in the first 8 MB of memory, it is effectively served by only half the cache, as the 2-Way set associative 16 KB cache associated with data block B is the only cache memory it can target. In this instance, the application never derives any benefit from data block A.

However, if the application is working from two data sets located at least 8 MB apart in memory, closer control over how the cache maps to the data is possible. For example, if the program is doing a series of dual-MAC operations in which both DAGs are accessing data on every cycle, the DAG0 data set can be mapped to one 8 MB region of memory while the DAG1 data is mapped to another, thus ensuring that:

- DAG0 gets its data from data block A for all of its accesses, and
- DAG1 gets its data from data block B.

This arrangement causes the core to use both data buses for cache line transfers and achieves the maximum data bandwidth between the cache and the core.

The Data Cache Mapping figure shows an example of how mapping is performed when `DCBS = 1`.

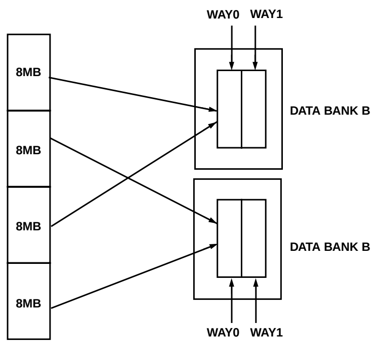


Figure 7-4: Data Cache Mapping When DCBS = 1

Data Cache Bypass Mode

The `L1DM_DCTL.CBYPASS` bit enables Data Cache Bypass mode, where data in cacheable memory is loaded directly from the system without disturbing the contents of the cache. To ensure a consistent view of cacheability in the memory, system code that enables, disables and operates in data cache bypass mode should be preceded by and followed by an `SSYNC` instruction.

Data Cache Control Instructions

The processor defines three data cache control instructions that are accessible in User and Supervisor modes. The instructions are `PREFETCH`, `FLUSH`, and `FLUSHINV`.

- `PREFETCH` (Data Cache Prefetch) attempts to allocate a line into the L1 cache. If the prefetch hits in the cache, generates an exception, or addresses a cache-inhibited region, `PREFETCH` functions like a `NOP`. To improve performance, it can be used to begin a data fetch prior to when the processor needs the data.
- `FLUSH` (Data Cache Flush) causes the data cache to synchronize the specified cache line with external memory. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or does not exist, `FLUSH` functions like a `NOP`.
- `FLUSHINV` (Data Cache Line Flush and Invalidate) causes the data cache to perform the same function as the `FLUSH` instruction and then invalidate the specified line in the cache. If the line is in the cache and dirty, the cache line is written out to external memory, and the Valid bit in the cache line is cleared. If the line is not in the cache, `FLUSHINV` functions like a `NOP`.

If software requires synchronization with system hardware, place an `SSYNC` instruction after the `FLUSH` instruction to ensure that the flush operation has completed. If strong ordering is desired to ensure that previous stores have been pushed through all the logic, place an `SSYNC` instruction before the `FLUSH`.

Data Cache Invalidation

Besides the `FLUSHINV` instruction, explained in the previous section, two additional methods are available to invalidate the data cache when flushing is not required.

The simplest method for invalidating the entire data cache is to disable and then re-enable it by first setting `L1DM_DCTL.CFG` to 0. Then, in a second write, set `L1DM_DCTL.CFG` to 1 to cause the cache to be re-initialized with all Valid bits set to the Invalid state. As with any code that changes the data cache mode, these write operations must be preceded by and followed by an `SSYNC` instruction.

The second technique directly invalidates cache lines by writing directly to the Valid bits while Extended Data Access mode is enabled (see Extended Data Access).

Extended Data Access

The L1 memory system contains special-purpose memory spaces to which direct access by load or store instructions is normally restricted. When extended data accesses are enabled, all restricted memory spaces including L1 instruction memory may be directly accessed by suitably-privileged software. The extended data accesses are typically not as fast as regular L1 data SRAM accesses.

Setting the Extended Data Access Enable bit (`L1DM_DCTL.ENX`) enables data access to restricted L1 memory spaces. When enabled, regular load and store instructions may be used to read or write these memory spaces:

- L1 data SRAM currently being used as cache
- L1 instruction SRAM, whether being used as cache or not

Extended data access is also possible to these additional memory spaces which are mapped into otherwise unused parts of the core's L1 address space:

- cache tags
- cache Dirty bits
- Parity bits

When extended data accesses are enabled, loads or stores to all the restricted regions (including L1 instruction SRAM) are controlled by the data CPLBs if they are enabled. When extended data accesses are disabled, a load or store to any of these restricted memory regions will cause a Data Access CPLB Protection Violation exception, irrespective of the CPLB setting.

Loads and stores to restricted memory regions can only be through DAG0. An attempt to access them through DAG1 will cause a Data Access CPLB Protection Violation exception.

Loads and stores to cache tags and dirty bits should be 32-bit accesses and 32-bit aligned, else incorrect data may be returned with no exception raised.

Writes to L1 instruction SRAM to initialize instruction memory should be followed by a `CSYNC` instruction to ensure the processor flushes its pipeline and fetches the next instruction from the modified SRAM.

Extended data access mode does not affect DMA access to these memory regions. DMA access is never permitted to Parity bits, cache tags, Dirty bits, or SRAM configured as cache.

Memory Protection and Properties

This section describes the Memory Management Unit (MMU), memory pages, Cacheability Protection Lookaside Buffer (CPLB) management, MMU management, and CPLB registers.

Memory Management Unit (MMU)

The Blackfin+ processor contains a page-based Memory Management Unit (MMU). This mechanism provides control over the cacheability of memory ranges, as well as management of protection attributes at a page level. The MMU provides great flexibility in allocating memory and I/O resources between tasks, with complete control over access rights and cache behavior.

The MMU is implemented as two 16-entry Content Addressable Memory (CAM) blocks and two default descriptors. Each entry is referred to as a Cacheability Protection Lookaside Buffer (CPLB) descriptor. When enabled, every valid entry in the MMU is examined on any fetch, load, or store operation to determine whether or not there is a match between the address being requested and the page described by the CPLB entry. If a match occurs, the cacheability and protection attributes contained in the descriptor are used for the memory transaction with no additional cycles added to the execution of the instruction. If no valid CPLB entry matches, the cacheability and protection attributes are provided by the default descriptor.

Because the L1 memories are separated into instruction and data memories, the CPLB entries are also divided between instruction and data CPLBs. Sixteen CPLB entries and one default descriptor are used for instruction fetch requests (*ICPLBs*). Another sixteen CPLB entries are used for data transactions (*DCPLBs*). The ICPLBs and DCPLBs are enabled by setting the appropriate bits in the L1 Instruction Memory Control (`L1IM_ICTL`) and L1 Data Memory Control (`L1DM_DCTL`) registers, respectively.

Data accesses to system and core MMR space are never controlled by CPLBs. If the data CPLBs are enabled, data accesses to all memory spaces (including extended data accesses, when enabled) are controlled by the data CPLBs.

Instruction CPLB

The Instruction CPLB (ICPLB) governs instruction fetches. Each of the 16 ICPLB page descriptors consists of a pair of 32-bit values:

- `L1IM_ICPLB_ADDR[n]` defines the start address of the page described by the ICPLB descriptor. For more information, see Instruction Memory CPLB Address Registers .
- `L1IM_ICPLB_DATA[n]` defines the properties of the page described by the ICPLB descriptor. For more information, see Instruction Memory CPLB Data Registers .

The `L1IM_ICPLB_DFLT` register provides default properties should no valid page descriptor match. For more information, see Instruction Memory CPLB Default Settings Register .

NOTE: To ensure proper behavior and future compatibility, all reserved bits in the `L1IM_ICPLB_DATAx` and `L1IM_ICPLB_DFLT` registers must be set to 0 whenever these registers are written.

Data CPLB

The Data CPLB (DCPLB) governs data accesses by load and store instructions. Each of the 16 DCPLB page descriptors consists of a pair of 32-bit values:

- The `L1DM_DCPLB_ADDR[m]` defines the start address of the page described by the DCPLB descriptor. For more information, see the Data Memory CPLB Address Registers .
- `L1DM_DCPLB_DATA[m]` defines the properties of the page described by the DCPLB descriptor. For more information, see the Data Memory CPLB Data Registers .

The `L1DM_DCPLB_DFLT` register provides default properties should no valid page descriptor match. For more information, see the Data Memory CPLB Default Settings Register .

CAUTION: OTP memory does not support burst transfers, which is required to support cache line fills. As such, OTP memory should not be covered by a cache-enabled DCPLB. If it is, the OTP controller will return an error when a read access is attempted.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in the `L1DM_DCPLB_DATAx` and `L1DM_DCPLB_DFLT` register must be set to 0 whenever these registers are written.

CPLB Page Descriptors

A CPLB page descriptor is a two-word descriptor consisting of an address descriptor word (`L1IM_ICPLB_ADDR[n]` or `L1DM_DCPLB_ADDR[n]`) and a properties descriptor word (`L1IM_ICPLB_DATA[n]` or `L1DM_DCPLB_DATA[n]`). Each valid CPLB entry describes a memory page.

The 4 GB address space of the processor can be divided into smaller ranges of memory or I/O, referred to as memory pages. Every address within a page shares the attributes defined for that page. The architecture supports 11 different page sizes:

- 1 KB
- 4 KB
- 16 KB
- 64 KB
- 256 KB
- 1 MB
- 4 MB
- 16 MB
- 64 MB
- 256 MB
- 1 GB

Different page sizes provide a flexible mechanism for matching the mapping of attributes to different kinds of memory and I/O.

The CPLB address descriptor word provides the base address of the page in memory. Each page must be aligned on a boundary that is an integer multiple of its size (e.g., a 4 MB page must start on an address divisible by 4 MB, whereas a 1 KB page can start on any 1 KB boundary).

Memory Page Properties

The second word of a CPLB page descriptor, `L1IM_ICPLB_DATA[n]` or `L1DM_DCPLB_DATA[n]`, specifies the other properties or attributes of the page. These properties include:

- Page size - any power of 4 between 1 KB and 1 GB.
- Cacheability properties:
  - Cacheable/non-cacheable: accesses to this page use the cache or bypass the cache.

    Cacheability may be specified separately for L1 and L2 cache.

    - If cacheable: write-through or write-back determines whether data writes propagate directly to memory or are deferred until the cache line is reallocated.
    - If non-cacheable: I/O device space or regular memory. Data reads from I/O device space are non-speculative. This is suitable for use with memory-mapped devices with read side-effects, but it is considerably less efficient than a regular memory access and should not be used indiscriminately.
- Protection properties:
    - Supervisor write access permission: enables or disables writes to this page when in Supervisor mode (applies to data pages only).
    - User write access permission: enables or disables writes to this page when in User mode (applies to data pages only).
    - User read access permission: enables or disables reads from this page when in User mode.
- CPLB entry status:
    - Valid: the processor ignores the CPLB page descriptor unless this bit is set.
    - Dirty: the data in this page in memory has changed since the CPLB was last loaded. Writes to a page without this bit set cause a CPLB protection exception. Software is responsible for setting the bit to enable writes to the page and for propagating any subsequent modifications to the page further down the memory hierarchy. Ensure this bit is always set in data CPLBs if it is not required to track modifications to individual pages.
    - Lock: keep this entry in the MMU, and do not participate in CPLB replacement policy. This bit is ignored by the hardware and reserved for use by software implementing the CPLB replacement policy.

Default Memory Properties

Cacheability and protection properties may be set for memory spaces not described by the CPLB page descriptors. Once the processor has found no valid CPLB page descriptor matching an address, it looks up the default properties in the `L1IM_ICPLB_DFLT` register for instruction fetches and in the `L1DM_DCPLB_DFLT` register for data accesses.

The registers specify properties for L1 and non-L1 (system) memory separately.

- EOM (No Exception on Miss): when this bit is cleared, an access to the memory space (L1 or system) which does not match a valid CPLB page descriptor causes a CPLB miss exception. When this bit is set, the properties from this register are used when no valid CPLB page descriptor matches.
- Protection properties:
    - Supervisor write access permission (applies to data CPLB only).
    - User write access permission (applies to data CPLB only).
    - User read access permission.
- Cacheability properties (system memory only):

- Cacheable or non-cacheable in L1 and/or L2 cache
- If cacheable: write-through or write-back mode (applies to data CPLB only).
- If non-cacheable: I/O device space or regular memory (applies to data CPLB only).

The default memory properties are only used when CPLBs are enabled and the EOM bit is set.

CPLB Status Registers

Bits in the DCPLB Status (L1DM_DSTAT) and ICPLB Status (L1IM_ISTAT) registers provide information about the cause of CPLB-related exceptions and help identify the CPLB entry that has triggered the exception. The exception service routine may also inspect the SEQSTAT.EXCAUSE field and the CPLB entries to infer the cause of the fault. For more information, see the Data Memory CPLB Status Register and the Instruction Memory CPLB Status Register .

NOTE: The L1DM_DSTAT and L1IM_ISTAT registers are valid only while in the faulting exception service routine.

DCPLB and ICPLB Fault Address Registers

The DCPLB Fault Address (L1DM_DCPLB_FAULT_ADDR) and ICPLB Fault Address (L1IM_ICPLB_FAULT_ADDR) registers hold the address that has caused a fault in L1 Data Memory and L1 Instruction Memory, respectively. For more information, see the Data Memory CPLB Fault Address Register and the Instruction Memory CPLB Fault Address Register .

CPLB Management

Use of CPLBs is optional. If all L1 memory is configured as SRAM and no memory protection is required by the application, then CPLBs need not be enabled. However, CPLBs must be used if cache or memory protection is required.

NOTE: Before caches are enabled, the MMU and its supporting data structures must be set up and enabled.

Upon reset, CPLBs are disabled, and the Memory Management Unit (MMU) is not used. CPLBs are enabled separately for instruction fetches (by setting the L1IM_DCTL.ENCPLB bit) and data accesses (by setting the L1DM_ICTL.ENCPLB bit).

Once the CPLBs are enabled, an exception occurs when the Blackfin+ processor issues a memory operation for which no valid CPLB page descriptor exists and the default CPLB register indicates EOM (exception on miss). This exception places the processor into Supervisor mode and vectors to the MMU exception handler. The handler is typically part of the operating system (OS) kernel that implements the CPLB replacement policy.

The MMR storage locations for CPLB entries are limited to 16 page descriptors for instruction fetches and 16 page descriptors for data load and store operations.

For small and/or simple memory models, it may be possible to define a set of CPLB page descriptors combined with defaults that fit into these 32 entries, cover the entire addressable space, and never need to be replaced. This type of definition is referred to as a static memory management model.

However, operating environments commonly define more CPLB descriptors (to cover the addressable memory and I/O spaces) than will fit into the available on-chip CPLB MMRs. When this happens, a Page Descriptor Table is used, which stores all the potentially required CPLB descriptors. The specific format for the Page Descriptor Table is not defined as part of the Blackfin+ processor architecture. Different operating systems, which have different memory management models, can implement Page Descriptor Table structures that are consistent with the OS requirements. This allows adjustments to be made between the level of protection afforded versus the performance attributes of the memory-management support routines.

NOTE: Before CPLBs are enabled, valid CPLB page descriptors (defaults) must be in place for both the Page Descriptor Table and the MMU exception handler. The `LOCK` bits of these CPLB page descriptors are commonly set so that they are not inadvertently replaced in software.

The MMU exception handler uses the faulting address to index into the Page Descriptor Table structure to find the correct CPLB descriptor data to load into one of the on-chip CPLB page descriptor register pairs. If all on-chip registers contain valid CPLB entries, the handler selects one of the descriptors to be replaced, and the new descriptor information is loaded. Before loading new descriptor data into any CPLBs, the corresponding group of sixteen CPLBs must be disabled by clearing the `ENCPLB` bit in either `L1DM_DCTL` or `L1IM_ICTL`.

After the new CPLB page descriptor is loaded, the CPLB is re-enabled, the exception handler returns, and the faulting memory operation is restarted. This operation should now find a valid CPLB descriptor for the requested address, and it should proceed normally.

A single instruction may generate an instruction fetch as well as one or two data accesses. It is possible that more than one of these memory operations references data for which there is no valid or default CPLB page descriptor. In this case, the exceptions are prioritized and serviced in this order:

- Instruction page miss
- Data page miss using DAG0
- Data page miss using DAG1

CrossCore® Embedded Studio provides an MMU exception handler and automatic generation of the Page Descriptor Table structure. Please refer to the *Cache and CPLBs* section of the *System Run-Time Documentation*.

CPLB Exception Cause

An exception service routine can inspect the `SEQSTAT.EXCAUSE` field to determine the cause of a fault. If the cause is a CPLB- or L1 memory-related exception, further information can be obtained from the CPLB status and fault address registers.

The DCPLB Status (`L1DM_DSTAT`) and ICPLB Status (`L1IM_ISTAT`) registers provide information about the cause of CPLB-related exceptions and help identify the CPLB entry that has triggered the exception. The exception service routine can also inspect the CPLB entries to infer the cause of the fault. For more information, see the Data Memory CPLB Status Register and the Instruction Memory CPLB Status Register .

The DCPLB Fault Address (`L1DM_DCPLB_FAULT_ADDR`) and ICPLB Fault Address (`L1IM_ICPLB_FAULT_ADDR`) registers hold the address that caused a fault in L1 data memory and L1 instruction memory, respectively. For more information, see the Data Memory CPLB Fault Address Register and the Instruction Memory CPLB Fault Address Register .

NOTE: The `L1DM_DSTAT`, `L1IM_ISTAT`, `L1DM_DCPLB_FAULT_ADDR`, and `L1IM_ICPLB_FAULT_ADDR` registers are valid only while in the faulting exception service routine.

L1 Parity Protection

The following sections provide details of L1 parity error support on Blackfin+ processors.

Parity Protection Coverage

Data and instruction L1 SRAM are parity-protected using one bit per byte. Data and instruction L1 cache tag arrays and dirty bits are also parity-protected using at least one bit per byte.

The parity bits are calculated and stored on every write to L1 SRAM or cache, either by the processor or by system traffic such as DMA and cache fill requests. Parity error checking is disabled by default. On power-up and upon reset, L1 SRAM and the parity bits are in an undefined state. Software must write to all locations of L1 to initialize the memory and parity bits before enabling parity read checking. This is performed automatically by the processor Boot ROM code upon power-up.

Parity Error Detection and Notification

Parity error checking may be enabled separately for L1 instruction memory and for L1 data memory by setting the Read Parity Checking Enable (`RDCHK`) bit in the `L1IM_ICTL` and `L1DM_DCTL` registers, respectively.

Parity errors are checked for whenever L1 memory is read. Parity checking is distributed so that all simultaneous L1 read traffic (DAG reads, instruction reads, DMA reads, and victim reads) can be simultaneously examined.

If a parity error is detected during any L1 read, an NMI is raised. The error is immediately signaled to the processor, even if the read is speculative in nature. L1 reads by the processor are intercepted before they lead to further immediate consequence. The core will receive an NMI, be immediately stalled, and will remain stalled until it vectors to the handler in response to the NMI. This guarantees that core state is not modified based on a corrupted L1 memory state.

If the read was not initiated by the local processor, an NMI is raised but the transfer is unstoppable. To guarantee that the system is not corrupted by an L1 parity error, external reads of L1 by DMA or another processor must not be used, and write-back cache must also not be used. Note that write-through cache is a safe alternative to write-back cache, since it does not generate victim traffic.Typically, it is better to accept the risk of some system corruption, which can be isolated by the parity error handler, to benefit from the increased performance of write-back cache or DMA.

The core will signal a double fault error to the SEC if a parity error is detected while servicing a Reset or Emulation event. An NMI is not raised if a Parity Error is detected while servicing an NMI, but the Parity Error Status registers are updated.

Parity Error Recovery

When a parity error is detected on a L1 read, the L1 memory has already been corrupted and must be restored to a known good state. Read-only data in SRAM might be restored from a known good copy in ROM or ECC-protected L2 or L3 memory. Volatile data may need to be recomputed by rerunning a computation from a checkpointed good state. Caches can be recovered by completely invalidating the cache (by disabling and then immediately re-enabling it) so that all lines would be reacquired from non-L1 memory.

The location of a parity error can be determined by inspecting the `SEQSTAT` register in combination with the Instruction Memory Parity Error Status (`L1IM_IPERR_STAT`) or Data Memory Parity Error Status (`L1DM_DPERR_STAT`) registers. For more information, see the Instruction Parity Error Status Register and the Data Memory Parity Error Status Register .

Four bits in the `SEQSTAT` register indicate whether the parity error occurred in instruction or data memory and whether it was detected on a read by the processor or a read by the system:

- `PEIC` indicates a parity error on a L1 instruction memory read by the processor.
- `PEDC` indicates a parity error on a L1 data memory read by the processor.
- `PEIX` indicates a parity error on a L1 instruction memory read by the system.
- `PEDX` indicates a parity error on a L1 data memory read by the system.

Once the general location of the error has been identified, the precise location can be discovered by reading `L1IM_IPERR_STAT` (for an error in L1 instruction memory) or `L1DM_DPERR_STAT` (for an error in L1 data memory).

If the fault is in SRAM, the `LOCATION` field is set to zero, the `ADDRESS` field is set to bits 21:3 of the aligned address of the eight bytes containing the faulting location, and a bit is set in the `BYTELOC` for each byte of the eight bytes that contains an error (e.g., `BYTELOC[0]` is set if byte 0 has an error, and so on).

When a misaligned memory read crosses an 8-byte boundary and errors occur on both sides of the boundary, then only the errors on the lower side of the boundary are reported.

If the fault is in a cache tag array, the `LOCATION` field is set to a non-zero value. In this case, knowledge of the memory microarchitecture is required to interpret the `LOCATION`, `ADDRESS` and `BYTELOC` fields. See Extended Data Access to L1 Caches.

The `LOCATION` and `BYTELOC` fields of the `L1IM_IPERR_STAT` and `L1DM_DPERR_STAT` registers are sticky. They are set when hardware encounters a parity error and are cleared only on reset or when the processor explicitly writes to the MMR. As these bits are write-1-to-clear (W1C), they can be cleared by writing back the value read from the registers.

If a parity error is detected while servicing an NMI or a higher-priority event, the Parity Error Status registers are set. When the processor is at thread level or servicing a lower-priority event and bits in the `LOCATION` or `BYTELOC` fields are set, then an NMI is raised. As such, it is recommended that the NMI handler read and write back the Parity Error Status registers to clear these bits for all NMI events associated with parity errors. If any further parity errors are detected while still in the NMI handler, the status registers will update again, thus causing the processor to vector again to the NMI handler immediately upon exiting it to handle the new error(s).

Parity Errors Simultaneous with Exceptions and Interrupts

The Concurrent Parity Error and Interrupt bit (`SEQSTAT.CPARINT`) indicates a parity error was detected simultaneously with an exception or an interrupt.

If an instruction causes both a parity error and an exception, `CPARINT` is set, `IPEND[3]` and `IPEND[2]` are set, `RETN` is set to the address of the exception handler, and `RETX` is set to the address of the instruction that caused the parity error. It may not be possible to recover from this situation, as the exception may actually be a side-effect of the parity error (if that is what caused the instruction to be corrupted).

If an interrupt is latched at the same time that a parity error is detected, `CPARINT` is set, `IPEND[3]` and the `IPEND` bit for the interrupt are set, `RETN` is set to the address of the interrupt handler, and `RETX` is set to the address of the instruction that caused the parity error. This enables the interrupt handler to be executed upon returning from the NMI handler, but this also means that `RETN` should not be inspected to determine the location of the parity error on an instruction read; rather, `L1IM_IPERR_STAT` should be used.

Direct Access To Parity Bits for L1 SRAM

In support of both parity error interrupt servicing and hardware test, it is possible to inspect the state of a parity bit without generating a parity error and write the state of a parity bit to an intentionally incorrect state by reading from and writing to extended L1 memory locations. When extended data access to L1 memory is enabled (by setting the `L1DM_DCTL.ENX` bit), a number of extended accesses are allowed, as listed in the ***Permitted Extended Accesses*** table.

Table 7-1: Permitted Extended Accesses

Access Type	Address	Parity
Read	L1 SRAM address + `0x80000`	Read data at address without parity checking
Read	L1 SRAM address + `0xC0000`	Read parity bits for 32-bit word at address
Write	L1 SRAM address + `0xC0000`	Write 32-bit value to word at address and invert parity bits to create an intentional parity error for test purposes

Parity bits are read into the low-order bit of each byte. The value of the remaining bits may not be zero and cannot be relied upon.

Direct access to parity bits for L1 cache tags and dirty bits is possible only when extended data access is enabled. Knowledge of the memory microarchitecture is required to do this (see Extended Data Access to L1 Caches).

L1 Initialization Requirements

If parity checking is to be used, software must write all locations of L1 after each processor power-up. This includes initial device power-up, as well as subsequent exits from the Hibernate state. Normally, this process takes place in the processor's boot code. These writes are necessary to initialize the otherwise random states of parity bits to legitimate states. All of L1 must be initialized, rather than just those locations expected to be read, otherwise speculative accesses have the potential to trigger unintended parity errors.

To allow the processor to be able to initialize L1 without the risk of triggering these unwanted parity errors, L1 parity error-checking on reads is disabled by default. Writes are always performed with parity bits calculated and stored. This mode is controlled by the Read Parity Checking Enable (`RDCHK`) bit in the `L1IM_ICTL` and `L1DM_DCTL` registers. This bit is deasserted by hardware reset and must be set by software (after L1 initialization) to enable read parity checking. Initialization of L1 may be achieved through any combination of DMA and processor L1 accesses.

When the cache is enabled, all the tags are written to. As such, enabling the cache also initializes the parity bits protecting the tags. However, enabling the cache does not initialize the parity bits for the SRAM holding the cache data arrays. These must be initialized prior to enabling the cache as part of the L1 SRAM initialization process described above.

Additional Notes on Parity Errors

Although hardware provides a means for determining the locations of corrupted L1, this may not provide sufficient clues to identify the locations outside the core that receive corrupted data from L1. The system can receive corrupted L1 data through either direct access by another core, DMA, or via cache victimization. In the case of direct system access, an L1 source address may not alone implicate the system target address. In the case of cache victimization, the victim address stored in the Tag array is typically overwritten with a Fill address prior to all victim data being extracted from L1 memory. Therefore, cache will not natively be able to provide the system address of a corrupt victim.

A cache bypass mode is provided to allow direct access to system memory. Cache bypass is enabled by setting the appropriate bypass bit (`L1DM_ICTL.CBYPASS` for data cache and `L1IM_DCTL.CBYPASS` for instruction cache). This can be useful during servicing of L1 parity errors. While cache is bypassed, CPLB cache settings are ignored, and cache hits are blocked. This preserves the cache in the precise state it was in prior to being bypassed, ignoring any already active fill, flush and victimization servicing, which continues until completed. While cache is being bypassed, extended data accesses may still be used to access cache content.

Example Parity Handler

```
.section NONCACHED_ECC_PROTECTED_DATA;
.var saved_sp;
.var nmi_handler_stack[BIG_ENOUGH];

.section NONCACHED_ECC_PROTECTED_CODE;
.extern nmi_handler;
nmi_handler:
/* switch to ECC-protected stack */
```

```
[saved_sp] = SP;
SP = nmi_handler_stack + BIG_ENOUGH;
/* save other registers on ECC-protected stack */

/* If program used system MMRs or memory with read side-effects, check for non-speculative
access abort. */
R7 = SEQSTAT;
CC = BITTST(R7, BITP_SEQSTAT_NSPECABT);
IF CC JUMP unrecoverable_error;

/* If there are other sources of NMI, check for external NMI which vectors to the same
handler as parity. */
CC = BITTST(R7, BITP_SEQSTAT_SYSNMI);
IF CC JUMP nmi_handler;

/* CPARINT indicates parity error simultaneous with exception or interrupt, not necessarily
recoverable. */
CC = BITTST(R7, BITP_SEQSTAT_CPARINT);
IF CC JUMP unrecoverable_error;

/* If DMA may have read L1 or Write-back cache is enabled, check for parity error on system
read */
CC = BITTST(R7, BITP_SEQSTAT_PEIX);
IF CC JUMP unrecoverable_error;
CC = BITTST(R7, BITP_SEQSTAT_PEDX);
IF CC JUMP unrecoverable_error;

/* We have a recoverable parity error. */
CC = BITTST(R7, BITP_SEQSTAT_PEIC);
IF CC JUMP parity_in_instruction_L1;

/* Parity error in data L1. */
R7 = [REG_L1DM_DPERR_STAT];
/* clear the error by writing BYTELOC and LOCATION */
[REG_L1DM_DPERR_STAT] = R7;
/* test for error in cache TAG or MOD */
R5 = R7 << 29;
CC = R5 == 0;
IF !CC JUMP parity_error_in_data_cache;

/* compute the error address */
R6 = [REG_L1DM_SRAM_BASE_ADDR];
R5 = R7 << 8;
R5 = R5 >> 8;
R6 = R6 + R5;
/* if address is in non-cache SRAM goto reload */

/* otherwise reset data cache */
parity_in_data_cache:
R7 = [REG_L1DM_DCTL];
```

```
R6 = R7;
BITCLR(R6, BITP_L1DM_DCTL_CFG+1);
BITCLR(R6, BITP_L1DM_DCTL_CFG);
[REG_L1DM_DCTL] = R6; /* disable cache */
CSYNC;
[REG_L1DM_DCTL] = R7; /* re-enable cache */
CSYNC; /* wait for cache to reinitialize */
JUMP return_from_parity_error;

parity_in_instruction_L1:
R7 = [REG_L1IM_IPERR_STAT];
/* clear the error by writing BYTELOC and LOCATION */
[REG_L1IM_IPERR_STAT] = R7;
/* test for error in cache TAG or MOD */
R5 = ~BITM_L1DM_DPERR_STAT_LOCATION;
R5 = R7 & R5;
CC = R5 == 0;
IF !CC JUMP parity_error_in_instruction_cache;
/* compute the error address */
R6 = [REG_L1DM_SRAM_BASE_ADDR];
R5 = BITM_L1DM_DPERR_STAT_ADDRESS;
R5 = R7 & R5;
R6 = R6 + R5;
/* if address is in non-cache SRAM goto reload */

/* otherwise reset instruction cache */
parity_in_instruction_cache:
R7 = [REG_L1IM_ICTL];
BITCLR(R7, BITP_L1IM_ICTL_CFG);
[REG_L1IM_ICTL] = R7; /* disable cache */
CSYNC;
BITSET(R7, BITP_L1IM_ICTL_CFG);
[REG_L1IM_ICTL] = R7; /* re-enable cache */
CSYNC;
BITSET(R7, 1);
[REG_L1IM_ICTL] = R7; /* re-enable cache */
CSYNC;
BITSET(R7, BITP_L1IM_ICTL_CFG);
[REG_L1IM_ICTL] = R7; /* re-enable cache */
CSYNC; /* wait for cache to reinitialize */

return_from_parity_error:
/* restore registers */
SP = [saved_sp];  /* restore stack */
RTN;
```

Memory Transaction Model

Both internal and external memory locations are accessed in little-endian byte order. The Data Stored in Little Endian Order figure shows a data word stored in register `R0` and in memory at address location *addr*. B0 refers to the least significant byte of the 32-bit word.

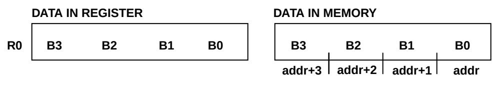


Figure 7-5: Data Stored in Little Endian Order

The Instructions Stored in Little Endian Order figure shows 16- and 32-bit instructions stored in memory. The diagram on the left shows 16-bit instructions stored in memory, with the most significant byte of the instruction stored in the high address (byte B1 in *addr+1*) and the least significant byte in the low address (byte B0 in *addr*).

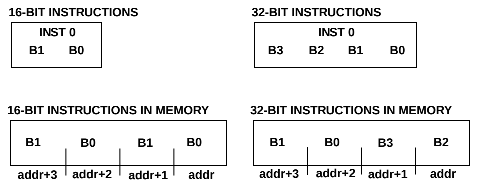


Figure 7-6: Instructions Stored in Little Endian Order

The diagram on the right shows 32-bit instructions stored in memory. Note that the most significant 16-bit half-word of the instruction (bytes B3 and B2) is stored in the low addresses (*addr+1* and *addr*), and the least significant half-word (bytes B1 and B0) is stored in the high addresses (*addr+3* and *addr+2*).

Load/Store Operation

The Blackfin+ processor architecture supports the RISC concept of a Load/Store machine. This machine is the characteristic in RISC architectures whereby memory operations (loads and stores) are intentionally separated from the arithmetic functions that use the targets of the memory operations. The separation is made because memory operations, particularly instructions that access off-chip memory or I/O devices, often take multiple cycles to complete and would normally halt the processor, preventing an instruction execution rate of one instruction per cycle.

In write operations, the store instruction is considered complete as soon as it executes, even though many cycles may elapse before the data is actually written to an external memory or I/O location. This arrangement allows the processor to execute one instruction per clock cycle, and it implies that the synchronization between when writes complete and when subsequent instructions execute is not guaranteed. Moreover, this synchronization is considered unimportant in the context of most memory operations.

Interlocked Pipeline

In the execution of instructions, the Blackfin+ processor architecture implements an interlocked pipeline. When a load instruction executes, the target register of the read operation is marked as busy until the value is returned from the memory system. If a subsequent instruction tries to access this register before the new value is present, the pipeline will stall until the memory operation completes. This stall guarantees that instructions that require the use of data resulting from the load do not use the previous or invalid data in the register, even though instructions are allowed to start executing before the memory read completes.

This mechanism allows the execution of independent instructions between the load and the instructions that use the read target without requiring the programmer or compiler to know how many cycles are actually needed for the memory read operation to complete. If the instruction immediately following the load uses the same register, it simply stalls until the value is returned. Consequently, it operates as the programmer expects. However, if four other instructions are placed after the load but before the instruction that uses the same register, all of them execute, and the overall throughput of the processor is improved.

Alignment

Non-aligned memory operations are supported. Loads and stores with addresses which are not a multiple of the data size access the bytes at sequential addresses starting with the address passed to the instruction, as expected. This may generate multiple memory read or write operations, but generally the instruction will not take more cycles than the equivalent two aligned loads and stores.

Aligned addresses are required in special circumstances, such as access to MMRs, I/O device space, exclusive loads and stores, and extended data access. In these cases, an address which is not a multiple of the data size causes a Misaligned Address exception.

For backward compatibility, some instructions in the quad 8-bit group and those used with the `DISALGNEXCPT` instruction do not cause alignment exceptions, but ignore the low order bits of a non-aligned address to access aligned data.

Ordering of Loads and Stores

The relaxation of synchronization between memory access instructions and their surrounding instructions is referred to as weak ordering of loads and stores. Weak ordering implies that the timing of the actual completion of the memory operations - even the order in which these events occur - may not align with how they appear in the sequence of the program source code. All that is guaranteed is:

- Load operations will complete before the returned data is used by a subsequent instruction.
- Load operations using data previously written will use the updated values.
- Store operations will eventually propagate to their ultimate destination.

Because of weak ordering, the memory system is allowed to prioritize reads over writes. In this case, a write that is queued anywhere in the pipeline, but not completed, may be deferred by a subsequent read operation, and the read is allowed to be completed before the write. Reads are prioritized over writes because the read operation has a dependent operation waiting on its completion, whereas the processor considers the write operation complete, and the write does not stall the pipeline if it takes more cycles to propagate the value out to memory. This behavior could cause a read that occurs in the program source code after a write in the program flow to actually return its value before the write has been completed. This ordering provides significant performance advantages in the operation of most memory instructions.

Speculative Load Execution

Load operations from memory do not change the state of the memory value. Consequently, issuing a speculative memory read operation for a subsequent load instruction usually has no undesirable side-effect. In some code sequences, such as a conditional branch instruction followed by a load, performance may be improved by speculatively issuing the read request to the memory system before the conditional branch is resolved. For example:

```
  IF CC JUMP away_from_here;
  RO = [P2];
  . . .
away_from_here:
```

If the branch is taken, then the load is flushed from the pipeline, and any results that are in the process of being returned can be ignored. Conversely, if the branch is not taken, the memory will have returned the correct value earlier than if the operation were stalled until the branch condition was resolved.

Store operations never access memory speculatively because this could cause modification of a memory value before it is determined whether or not the instruction should have executed.

Interruptible Load Behavior

Because it is interruptible, a load instruction may generate more than one memory read operation. If an interrupt of sufficient priority occurs between the load instruction entering the pipeline and the completion of the load instruction, the sequencer cancels the instruction. After execution of the interrupt, the interrupted load is executed again. This approach minimizes interrupt latency. However, it is possible that a memory read operation was initiated before the load was canceled, and this would be followed by a second read operation after the load is executed again. For most memory accesses, multiple reads of the same memory address have no side-effects.

There is no corresponding issue with store instructions, as the memory write operation only happens after a store instruction has committed. As such, the store can only be interrupted before the memory write has been initiated, and it is canceled with no visible side-effect. The store instruction is re-executed on return from the interrupt and ultimately initiates the memory write.

For alternative load behavior, see Non-Speculative, Non-Interruptible Loads.

Hazards of the High-Performance Memory Architecture

The Blackfin+ memory model, with weak ordering of reads and writes and redundant read operations, enables a long pipeline while avoiding stalls and maintaining fast interrupt response. However, it can cause side-effects that the programmer must be aware of to avoid improper system operation.

When sharing data with another core or device via memory accessible to both, the order of how read and write operations complete is often significant, as the writing core must be sure the data is visible to the reading device before signalling that the data is available. Similarly, when writing to or reading from non-memory locations such as off-chip I/O device registers, the order of how read and write operations complete is also significant. For example, a read of a status register may depend on a write to a control register. If the address is the same, the read would return a value from the store buffer rather than from the actual I/O device register, and the order of the read and write at the register may be reversed. Both these phenomena could cause undesirable side-effects in the intended operation of the program and peripheral.

Redundant memory reads can also be an issue. Interruptible load behavior can cause multiple memory read operations where only one was intended. For most memory accesses, multiple reads of the same memory address have no side-effects; however, for some off-chip memory-mapped devices such as peripheral data FIFOs, reads are destructive (i.e., each time the device is read, the FIFO advances and the data cannot be recovered and re-read). The redundant memory reads due to speculation will also cause problems if the load from a peripheral with destructive read behavior, such as a FIFO, is subsequently aborted.

Speculation can also be a problem where a load from an illegal address is aborted. A redundant memory read operation from a non-L1 address which does not map to any memory or device in the system will cause an External Memory Addressing error. Note that the load might be aborted because it is in the shadow of a conditional jump that tests whether or not the address is valid.

In summary, the following hazards exist:

- Reordering of an externally visible write with another externally visible action.
- Reordering of an externally visible read and write to the same location.
- Reordering of an externally visible read and write to different locations.
- Caches and write buffers can prevent memory operations becoming externally visible at all.
- Destructive reads that are not generated because they are serviced from the write buffer or cache.
- Repeated destructive reads due to interruptible loads.
- Unintended destructive reads due to speculative loads.
- Unintended access to illegal addresses causing spurious error interrupts.

The Blackfin+ architecture provides a number of solutions to these problems. Synchronization instructions (`CSYNC` or `SSYNC`) may be used to impose a precise ordering at the points in the code where it is required while generally retaining the benefits of weak ordering.

Cachebility properties may be specified in the CPLBs, which control the external visibility of memory operations and specify some regions as I/O device space. Loads from MMRs and I/O device space are never executed speculatively and are non-interruptable. All reads and writes to MMRs are strongly ordered.

CPLBs may be used to avoid spurious illegal address exceptions, as the memory read operation will not be initiated for a load that causes a page miss exception, but the exception will be suppressed in the case of a speculative load.

Synchronizing Instructions

When strong ordering of loads and stores is required, as may be the case for sequential accesses to shared memory, use the core or system synchronization instructions, CSYNC or SSYNC, respectively.

The CSYNC instruction ensures that all pending core operations have completed and that the store buffer between the processor core and the L1 memories has been flushed before proceeding to the next instruction. Pending core operations include any pending interrupts, speculative states (such as branch predictions), and exceptions. Consider the following example code sequence:

```
  IF CC JUMP away_from_here;
  CSYNC;
  R0 = [P0];
away_from_here:
```

The CSYNC instruction ensures:

- The conditional branch (IF CC JUMP away_from_here) is resolved, forcing stalls into the execution pipeline until the condition is resolved and any entries in the processor store buffer have been flushed.
- All pending interrupts or exceptions have been processed before CSYNC completes.
- The load is not speculatively fetched from memory.

The SSYNC instruction ensures that all side-effects of previous operations are propagated out through the interface between the L1 memories and the rest of the chip. In addition to performing the core synchronization functions of CSYNC, the SSYNC instruction flushes any write buffers between the L1 memory and the system domain and generates a sync request to the system that requires acknowledgement before SSYNC completes.

Where the external visibility of memory operations or interaction with system MMRs is a concern, SSYNC must be used. CSYNC is sufficient to control interaction with core MMRs and the L1 memory system.

Cache Coherency

For shared data, software must provide cache coherency support, as required. To accomplish this, use the FLUSH instruction (see the FLUSH description in Data Cache Control Instructions) and/or explicit line invalidation (see Data Cache Invalidation).

Whenever the external visibility of reads and writes is a concern, the cachebility properties specified in the CPLBs must be considered. A store to write-back L1 cache is only guaranteed to become visible externally after a FLUSH instruction is executed, whereas stores to write-through L1 is visible after the memory write completes.

If the memory region is written by another core or device, there is no automatic coherence mechanism to ensure earlier values in the cache are invalidated. As such, a load operation might return stale data from the cache unless explicit line invalidation is used to ensure coherency.

Commonly shared data which is updated by more than one writer should be maintained in non-cacheable regions.

I/O Device Space

The I/O device space property may be specified in the CPLBs. Load instructions from memory with this property behave in a manner more suitable for access to devices with destructive reads, such as FIFOs.

I/O device space is not cached. Reads from I/O device space are executed in a non-interruptible manner and are never executed speculatively. However, writes to I/O device space may be buffered. Reads will not be serviced from the write buffer, so a write followed by a read to the same location will always become externally visible and execute in the correct order. Reads and writes to different locations, however, may get reordered unless they are separated by a `SSYNC` instruction.

Loads and stores to I/O device space may not be executed in parallel with another memory operation, and addresses must be aligned to the data size.

Memory-Mapped Registers

A portion of the address space is reserved for Memory-Mapped Registers (MMRs), which is split into a region for system MMRs and a region for core MMRs. System MMRs are located in the memory space from `0x20000000-0x2FFFFFFF`, and core MMRs are mapped to `0x1FC00000-0x1FFFFFFF`. Refer to the Blackfin Processor Hardware Reference for more information.

All MMRs are only accessible in Supervisor mode. Accesses to MMRs in User mode generate an Illegal Use of Supervisor Resource exception. The same exception is also raised if a load or store to an MMR is issued in parallel with another load or store. Loads from MMRs are non-speculative and non-interruptible. All loads and stores to each MMR space are strongly ordered.

The core MMR space is located in the same memory region on every Blackfin+ core in a system. Core MMRs may only be accessed by load and store instructions executed by the local core and are not accessible via DMA. Like non-memory-mapped registers, the core MMRs connect to the 32-bit Register Access Bus (RAB) and are accessed at the CCLK rate.

All core MMRs must be read and written with 32-bit-aligned accesses; however, some MMRs have fewer than 32 bits defined. In this case, the unused bits are reserved and must be written as zero when writing the register. System MMRs connect through the system crossbars (SCBs) and must be accessed aligned to the data size. Accesses to non-existent MMRs generate an Illegal Access exception, and writes to read-only MMRs are ignored.

Each chapter in this manual describing a portion of the processor architecture includes a description of any related core MMRs. System MMRs are described in each chapter of the processor's hardware reference manual.

Non-Speculative, Non-Interruptible Loads

Loads from MMRs and I/O device space are non-speculative and non-interruptible. When a load from one of these spaces is encountered, a non-speculative request is sent to the sequencer. The sequencer will then disable all interrupts that can be disabled and stall the pipeline below the load instruction, at which point the non-speculative read is issued. Interrupts are subsequently re-enabled after the read data is returned. Even though most interrupts are disabled, there are four interrupt levels (IVG0-IVG3) that are effectively non-maskable and therefore might interrupt a non-speculative read:

- Emulator hardware interrupt (IVG0): emulator hardware interrupts can interrupt I/O accesses. However, this would only be during debugging sessions.
- Reset (IVG1): a reset event can interrupt non-speculative accesses. Since the core is being reset, this is expected behavior.
- NMI (IVG2): Non-Maskable Interrupts (NMIs) come from three possible sources:
  - External events on the NMI pin
  - `RAISE 2;` instruction (which is committed before I/O starts, so it cannot interrupt an non-speculative access)
  - Parity errors (which can interrupt a non-speculative access if caused by DMA of cache victim traffic)
- Exception (IVG3): always related to an instruction executing. The non-speculative read mechanism is designed to allow all exceptions in the pipeline before the non-speculative read to be taken before the non-speculative read is placed on the bus. As such, there will never be an exception during an non-speculative read.

If a non-speculative read is interrupted, whether to an I/O device page or a MMR, the Non-Speculative Access Was Aborted bit (`SEQSTAT.NSPECABRT`) is set. As the effect of the interrupted non-speculative read might be that a read side-effect occurred but the read data was lost, this is a non-recoverable error condition.

Exclusive Load, Store, and Sync (Spin Lock Example)

The load from memory exclusive, store to memory exclusive, and synchronize exclusive state instructions enable the implementation of software semaphores to control the interaction between tasks on separate processor cores or to separate tasks running on a single processor core.

A load exclusive instruction reads data from memory in the same manner as a regular load instruction. The load exclusive instruction also establishes exclusive access to that memory location. A store exclusive instruction only modifies memory if the task still has exclusive access to the location. An intervening load exclusive instruction from another task causes the exclusive access to be lost. The state of exclusive operations is tracked in the `XMONITOR`, `XWACTIVE` and `XWAVAIL` bits in the `SEQSTAT` register. The `SYNCEXCL` instruction synchronizes this state with the processor state, ensuring the `CC` bit in `ASTAT` has been updated to indicate whether a previous store exclusive instruction was performed successfully. The address passed to an exclusive load or store instruction must be aligned to the size of the data.

The following code sequence implements a spin lock:

```
P0 = lock;   /* address of lock */
R1 = 1;      /* lock value */
spin:
R0 = B[P0] (Z,EXCL);
CC = R0==0;                  /* is semaphore unlocked? */
if !CC JUMP spin;            /* no - try again */
CC = (B[P0] = R1) (EXCL);    /* try to lock */
SYNCEXCL;   /* wait for write and copy to CC */
IF !CC JUMP spin;   /* failed - try again */
```

```
/* critical section */

R1 = 0;       /* unlocked value */
B[P0] = R1;   /* unlock */
```

Semaphores controlling interaction between tasks on separate cores should be placed in non-cacheable, non-L1 memory that is accessible by both cores. In this case, the load and store exclusive instructions generate exclusive transactions on the system bus, and the memory controller participates in a protocol that ensures that a store exclusive instruction will fail if the memory location has been modified by another core since the corresponding load exclusive instruction.

Semaphores controlling interaction between tasks on the same core may be placed in cacheable memory or L1 SRAM. In this case, the load and store exclusive instructions do not generate special memory transactions. The `SYNCEXCL` instruction must be called in context switch code to clear any pending exclusive transactions and to preserve the result of any store exclusive instructions in the `CC` bit of the preserved `ASTAT` register.

```
/* context switch */
SYNCEXCL;
[--SP] = ASTAT; /* saves store excl result if one was pending */
```

Interrupt handlers that are known not to use exclusive operations may leave the exclusive state unmodified. Any pending exclusive write operations will complete and update the state in `SEQSTAT` which will be read by a `SYNCEXCL` instruction upon returning from the interrupt.

`SYNCEXCL` should also be used in the exception handler to reset exclusive state on exceptions caused by load or store exclusive instructions.

Load exclusive and store exclusive instructions must be aligned and may not be used with MMRs, I/O device space, or extended data accesses.

Execution results for exclusive load instructions and exclusive store instructions vary, depending on whether the memory addressed is shareable or non-shareable. The shareability of memory spaces is determined from the memory space and the CPLB settings, as shown in the ***Memory Kinds*** table.

An exclusive load or exclusive store to an *illegal* memory location causes an exception. An exclusive load or exclusive store to a *non-shareable* memory location succeeds, but the operation is not exclusive with respect to other cores. The operation is exclusive with respect to other threads running on the core executing the instruction. An exclusive load or exclusive store to a *shareable* memory location ensures exclusivity with respect to other cores by using exclusive transactions on the memory bus. Exclusive transactions require hardware support in the memory device. If that support is not available, an uncached exclusive load from that memory will cause an exception.

Table 7-2: Memory Kinds (Example for ADSP-BF70x Processors)

Memory	CPROPS	Meaning	"0" Shareability
MMR	any	Core or system MMR	Illegal
L1	any	L1 sram	Non-shareable
non-L1	CPLBEN=0	CPLB Disabled	Shareable

Table 7-2: Memory Kinds (Example for ADSP-BF70x Processors) (Continued)

Memory	CPROPS	Meaning	"0" Shareability
non-L1	CPLBBYPASS=1	Cache temporarily disabled	Shareable
non-L1	000	Page is non-cacheable memory	Shareable
non-L1	001	Write-Back Cacheable in L1	Non-shareable
non-L1	100	I/O Device Space	Illegal
non-L1	101	Write-Through Cacheable in L1	Non-shareable

Atomic TESTSET Instruction (Spin Lock Example)

The processor provides an atomic TESTSET instruction. This is primarily provided for backward compatability and is recomended to use with Exclusive Load and Store instructions, which make more efficient use of system resources. The TESTSET instruction reads an indirectly-addressed memory byte, tests whether it is zero, and then writes the byte back to memory with the most significant bit (MSB) set, all as one indivisible operation. If the byte is originally zero, the instruction sets the CC bit. If the byte is originally non-zero, the instruction clears the CC bit.

The TESTSET instruction is used with a regular store instruction to implement a spin lock, as follows:

```
P0 = lock;
spin:
TESTSET (P0);
IF !CC JUMP spin;

/* critical section */

R1 = 0;     /* unlock value */
B[P0] = R1; /* unlock */
```

NOTE: For more information, see the TESTSET instruction's reference page.

L1 Memory Microarchitecture

This section provides an overview of the L1 memory system in the Blackfin+ processors.

L1 Memory Access

Processor access to the L1 memory space is intended to occur in a single cycle. The L1 memory has four virtual ports: DAG0 read, DAG1 read, Store, and DMA read/write. The memories used to implement L1 are physically single-ported, so access conflicts are possible. To reduce the likelihood of such memory conflicts, L1 memory is divided into individually accessible 4 KB sub-banks, each having its own port multiplexor and a dedicated data bus. A memory conflict will only occur when multiple ports request at least one byte from the same sub-bank in the same cycle.

The incoming addresses for the four L1 ports are centrally decoded into sub-bank selects, which are then compared for collisions. The collisions are prioritized, and the winner is allowed to access the memory. The losers receive a stall and try again in the next cycle.

Load instructions present the read address to the memory system in pipeline stage F (DF1). The memory read occurs in stage G (DF2). The result is returned to the processor's history buffer in stage H (EX1). Store instructions are described in L1 Data Stores.

The DCPLB page descriptors generate exceptions in pipe stage G (DF2). The exception information is pipelined along with the memory operation and deposited into the history buffer in pipe stage H (EX1).

DMA accesses use the same timing as DAG reads and writes, which is three cycles for reads and two cycles for writes.

Memory Logical Sub-Bank Arrangement

The L1 memory sub-banks are logically arranged into rows and columns, as shown in the ***ADSP-BF70x Data Block A Address Mapping to Sub-Banks*** figure. The datapath width consists of the number of memory columns times the width of a sub-bank. All sub-banks are identical. The sub-banks are four bytes wide, and there are two columns; thus, the datapath width is eight bytes. The number of rows is determined by the desired amount of L1 memory.

The address provided by the DAGs is a byte address. The lower three bits of the address index bytes across the eight byte datapath width. The next set of bits address memory words within a sub-bank. In data blocks A and B, each sub-bank is also logically split into an upper and lower half, with bit 14 selecting which half. As such, address bits 31:20 select L1 memory and the block within L1 memory. Address bits 19:15, 13, 12, and 2 select the sub-bank. Address bit 14 and 11:3 select the row within the sub-bank. Finally, address bits 1:0 select the byte within the row. Data bank C only contains two sub-banks, where bit 2 selects the sub-bank and bits 13:3 select the row. As described, the addresses are contiguous.

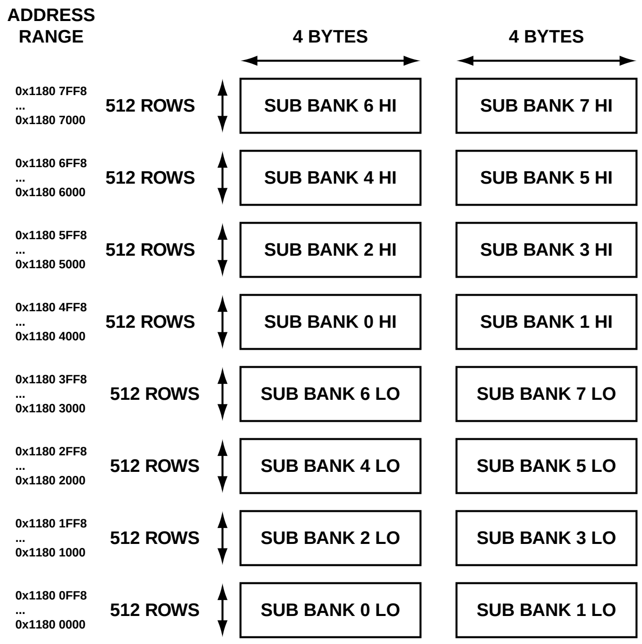


Figure 7-7: ADSP-BF70x Data Block A Address Mapping to Sub-Banks

Misaligned Data Access to L1

Misaligned memory accesses are supported. If all the bytes of an access lie within the address offset from 0-7, then the misaligned access can be serviced in a single cycle (assuming no sub-bank conflicts). If this requirement is not satisfied, then the access is considered a "crossed access" and is broken into two pieces within the MMU. The two halves of a crossed access are handled sequentially and are reassembled in the history buffer and returned to the core as a single entity. Thus, the core is isolated from the effects of the misaligned access, aside from the extra cycle required to retrieve all the data.

L1 Data Stores

Processor stores to data memory are more complicated than loads because the store address arrives in pipe stage F (DF1), but the data does not arrive until pipe stage J (WB). As such, the Memory Controller must create a placeholder for the address. When the data arrives, it is matched with the address. The address/data pair is then delivered to L1 memory on the Store port. The block that records the write address is called the Read/Write Buffer.

The name Read/Write Buffer is derived from the fact that it handles uncached loads from the system, as well as all stores. Each Read/Write buffer entry contains an address which describes a particular set of eight aligned memory bytes, and data storage for those bytes. The number of bytes stored in a given buffer is determined by the datapath width. When a new store enters pipe stage G (DF2), its address is compared to the addresses in all currently valid Read/Write entries. If there is a match, then the existing buffer entry is used, otherwise a new buffer entry is allocated. The ID of the buffer entry is placed in a queue called the Store Queue. The Store Queue performs the function of matching incoming store data with the proper Read/Write entry. When valid write data is present in the Read/Write buffer, it gets scheduled for "draining" to L1 memory using the L1 Store port.

Write Gathering is supported. Consider a sequence of byte writes to L1 memory, starting with an aligned address. The first byte will allocate a Read/Write entry, and the address of the second byte will match the Read/Write entry of the first. These two bytes will use the same Read/Write entry and is called write gathering. The Read/Write buffer entry tracks which of the eight bytes has been loaded with valid data. Only these bytes are written to memory.

Write Data Forwarding is supported. Consider a byte write, followed by a read of the same byte. The read operation must wait for the write data to become available in the memory. Once the write data arrives at the memory system, it often takes several cycles to get the data into the L1 memory sub-bank. Write forwarding short circuits this process by delivering the write data to the history buffer as soon as it arrives at the memory controller.

System Slave Interface

The Blackfin+ core interface is an SCB slave interface used to access a core's L1 memory. The width of the read and write data buses are 32 bits. This interface is used for DMA access to the L1 memory spaces.

SCB transaction burst lengths of one to sixteen are supported. All burst types (Fixed address, Incrementing address, and Wrap mode) and sizes less than or equal to the bus width are supported, as are arbitrary write strobes. However, locked/exclusive accesses and SCB cache information signaling are not supported by the interface.

The slave interface does not reorder read or write transactions; however, no ordering is enforced between the read and write transaction streams. The slave interface will respond with a slave error for illegal accesses. During read responses, an error will be reported for every access within the read burst that attempted an illegal access. For write responses, an error will be reported if any access of the burst attempted an illegal access. Illegal accesses do not result in transactions being initiated to the L1 memory.

The following accesses result in a slave error:

- Access to a non-populated L1 region
- Access to an L1 region configured as cache

Core MMR Access

Reads of core MMR registers in the MMU are non-speculative in nature and use the same timing as L1 memory reads. Core MMR reads must be 32-bit-aligned accesses performed in Supervisor mode, else an MMR exception is generated. Core MMR reads are stalled in pipe stage G (DF2) until all prior MMR writes have committed. This behavior is necessary because of the functional side effects of MMR writes. MMR writes take effect in the cycle immediately after write commit in pipe stage J (WB). There is no MMR store queue which could create timing uncertainty. CPLBs do not control the MMR address space.

System Memory Access

Unlike L1 loads, system loads allocate a Read/Write buffer entry. This is because, like stores, the data arrives late and an address placeholder and a way to associate the address and data is needed. Unlike stores, each system load is allocated a new Read/Write entry. There is no "read gathering" allowed. The ID of the Read/Write entry allocated for the system read is passed to the System Read Queue and the history buffer. System reads gather in the System Read Queue waiting to be dispatched to the System. When the System gasket accepts the read request, the request is plucked from the queue. When the read data returns, still tagged with the Read/Write ID, it is forwarded directly to the history buffer.

System stores behave in a similar fashion to L1 stores. Multiple system stores into the same eight-byte aligned space will gather into the same Read/Write buffer entry. Write drains to the system gasket are scheduled the same way as L1 writes. When a system store causes a Read/Write entry to be allocated, subsequent system loads matching that Read/Write address will use that entry's ID. This is necessary to ensure that the write data is properly forwarded to the history buffer along with the bytes read from system memory.

System accesses may also be misaligned and result in a crossed access. In this case, two Read/Write entries will be allocated. The two components of the crossed access are treated as separate transactions by the Read/Write buffer and system gasket. The history buffer understands that crossed system reads are split and rejoins them before delivery to the core.

Reads of MMRs, I/O device space, and the read part of the `TESTSET` instruction are non-speculative and non-interruptible. These reads are not added to the System Read Queue. Instead, when the read operation is passed to the history buffer in pipe stage H (EX1), a non-speculative request is sent to the sequencer. The sequencer will then disable most interrupts and stall the pipeline below the non-speculative read while holding the non-speculative read in stage H (EX1). Once this is done, the system read request is sent to the System Memory Interface. The non-speculative read is then issued, and interrupts are re-enabled when the read data is returned.

System Memory Interface

The Blackfin+ core memory interface to the system is used to access memory regions outside of L1 and MMR space. The data widths of the read and write data buses are 64 bits.

Cache and non-cache transactions are signaled across the memory interface. Cache transactions are 256 bits and result in a burst of four 64-bit words. The burst type signaled for cache transactions is `WRAP` mode. For cache reads, a core expects the critical word to be the first one returned, as signified by the address sent during the read transaction. Non-cache transactions, whether for data or instructions, use a burst type of `INCR` with a transaction length of one.

The `TESTSET` instruction is supported via a bus-locked transaction. When a `TESTSET` instruction is committed, all pending memory transactions are completed before the read portion of the `TESTSET` is initiated. After the read data is returned, the write portion of the `TESTSET` instruction will be initiated to the same memory location. After a write response is received by the interface, other memory transactions will once again be allowed. If the `TESTSET` instruction is killed in the processor pipe before it is commited, a dummy write will be performed on the system crossbar with all byte enables set to inactive to clear the lock.

Load and Store Exclusive instructions are supported by exclusive bus transactions.

As access to the System Memory space crosses from the core clock domain to the system clock domain, synchronization circuitry is required to force the timing of the accesses to align properly, which incurs additional read latencies. As this processor features a single Clock Generation Unit (CGU) to generate all the on-chip clocks, synchronization among the clock domains is guaranteed for integer CCLK::SYSCLK ratios (M::1). When this is true, the synchronizers can optionally be bypassed to reduce read latency by setting the System Memory Sync Bypass bit (`SYSCFG.MEMSBYP`).

NOTE: Do NOT set the SYSCFG.MEMSBYP bit when the system is programmed with a fractional CCLK::SYSCLK ratio (M::N).

System MMR Interface

The Blackfin+ system MMR interface is a crossbar master interface used to access the system MMR memory region. The data width of the read and write data buses are 32 bits. Only a single system MMR transaction can be active at any time; therefore, the read and write address buses are shared in the interface. The length of all transactions is one, and the supported transaction sizes are 8-, 16-, and 32-bit.

All reads via the system MMR interface are non-speculative.

As access to the System MMR space crosses from the core clock domain to the system clock domain, synchronization circuitry is required to force the timing of the accesses to align properly, which incurs additional read latencies. As this processor features a single Clock Generation Unit (CGU) to generate all the on-chip clocks, synchronization among the clock domains is guaranteed for integer CCLK::SYSCLK ratios (M::1). When this is true, the synchronizers can optionally be bypassed to reduce read latency by setting the System MMR Sync Bypass bit (`SYSCFG.MMRSBYP`).

NOTE: Do NOT set the SYSCFG.MMRSBYP bit when the system is programmed with a fractional CCLK::SYSCLK ratio (M::N).

L1 Cache Details

The upper 16 KB of L1 instruction SRAM, L1 data block A and L1 data block B, may be individually configured as cache. When so configured, these 16 KB are reserved for cache lines fetched from non-L1 memory. The cache tags occupy this memory space only when cache is enabled.

Both data block A and data block B can be configured as data cache. When both are enabled as cache, Address bits 14 and 23 of a cacheable load/store operation select which data cache is searched for a particular cache line (see Data Cache Block Select).

When accessing a cache line present in L1 cache, the access timing is identical to a standard L1 memory access. On a cache read, all lines in the set are read in parallel with the cache tag access, and the correct data is selected. On an instruction fetch, all eight subbanks are read, and a data load may access up to four subbanks. These accesses will collide with accesses to the other half of these subbanks, which remain in use as L1 SRAM.

Extended Data Access to L1 Caches

When extended data access is enabled by setting the `L1DM_DCTL.ENX` bit, cache tags, dirty bits and associated parity bits may be directly accessed with load and store instructions. The data cache tag memory and dirty array reside in the upper portion of the 256 KB region of L1 data block A. All accesses to these regions must be 32 bits wide, have an 8-byte-aligned address, issued from DAG0, and not be in parallel with another load or store.

Two bits of the extended data access address encode `PARCTL` and `PARSEL`. Setting `PARCTL` disables parity checking for the access. When accessing L1 SRAM, setting `PARSEL` with `PARCTL` enables parity bits to be toggled to cause parity errors (for test purposes) when writing to the address and enables SRAM parity bits to be read when reading from the address. When accessing cache tag and dirty arrays, `PARSEL` has no meaning, as the parity bits can be directly read or written.

The following tables provide details of the data cache tags. Separate copies of each tag are maintained for accesses by DAG0 and for accesses by DAG1.

Table 7-3: Data Cache Tags Extended Data Access Address

Address Bit	Meaning
`A[31:20]`	L1 Data Block A Address
`A[19]`	`PARCTL`
`A[18]`	`PARSEL`
`A[17:14]`	`1111`
`A[13]`	`1`=Tag for DAG0, `0`=Tag for DAG1
`A[12:4]`	Set Index
`A[3]`	Way
`A[2:0]`	`000`

Table 7-4: Data Cache Tags Extended Data Access Value

Data Bit	Value
`D[31:29]`	`0`
`D[28]`	Priority Parity
`D[27]`	Fill Pending Parity
`D[26]`	Next Victim Parity
`D[25]`	Valid Parity
`D[24]`	Tag Parity
`D[23]`	Priority
`D[22]`	Fill Pending
`D[21]`	Next Victim
`D[20]`	Valid
`D[19:0]`	Tag

Each access to the data cache dirty bits reads or writes the bits for both ways of two sets. Only even-addressed sets can be addressed. As such, the following table always shows Set Index bit 0 and Way as zero.

Table 7-5: Data Cache Dirty Bits Extended Data Access Address

Address Bit	Meaning
`A[31:20]`	L1 Data Block A Address

Table 7-5: Data Cache Dirty Bits Extended Data Access Address (Continued)

Address Bit	Meaning
`A[19]`	`PARCTL`
`A[18]`	`PARSEL`
`A[17:11]`	`1110100`
`A[10:5]`	Set Index Bits 7 to 1
`A[4:0]`	`00000`

Table 7-6: Data Cache Dirty Bits Extended Data Access Value

Data Bit	Value
`D[31:8]`	`0`
`D[7]`	Odd Set Way 1 Dirty Parity
`D[6]`	Odd Set Way 0 Dirty Parity
`D[5]`	Odd Set Way 1 Dirty
`D[4]`	Odd Set Way 0 Dirty
`D[3]`	Even Set Way 1 Dirty Parity
`D[2]`	Even Set Way 0 Dirty Parity
`D[1]`	Even Set Way 1 Dirty
`D[0]`	Even Set Way 0 Dirty

Cached data is stored in SRAM banks which may be accessed at their native address in `ENX` mode.

Table 7-7: Data Cache Data Extended Data Access Address and Mapping to L1 SRAM Subbank

L1 SRAM Address Bits	Cache Location
`A[31:21]`	L1 Data SRAM Address
`A[20]`	Set Index Bit 8 Selects Block A or B
`A[19]`	`PARCTL`
`A[18]`	`PARSEL`
`A[17:15]`	Usually 0 (Depends on Size of Data Block)
`A[14]`	`1`
`A[13]`	Set Index Bit 7
`A[12]`	Way
`A[11:5]`	Set Index Bits 6:0
`A[4:0]`	Byte Offset in Cache Line

The 20-bit Tag, 9-bit Set Index, and 5-bit offset within the cache line can be combined to form the home address of data in the cache.

Table 7-8: Data Address from Tag, Set Index, and Offset

L1DM_DCTL.CFG[1]	L1DM_DCTL.DCBS	Address
0	0	Tag[19:12], Tag[11], Tag[10:3], Tag[2], Tag[1], Set[7:0], Offset[4:0]
0	1	Tag[19:12], Tag[2], Tag[10:3], Tag[11], Tag[1], Set[7:0], Offset[4:0]
1	0	Tag[19:12], Tag[11], Tag[10:3], Set[8], Tag[1], Set[7:0], Offset[4:0]
1	1	Tag[19:12], Set[8], Tag[10:3], Tag[11], Tag[1], Set[7:0], Offset[4:0]

The instruction cache has a slightly different format because it has four ways and 128 sets, unlike the data cache (which has two ways and 256 sets).

Table 7-9: Instruction Cache Tags Extended Data Access Address

Address Bit	Meaning
`A[31:20]`	L1 Instruction SRAM Address
`A[19]`	`PARCTL`
`A[18]`	`PARSEL`
`A[17:13]`	`11111`
`A[12:5]`	Set Index
`A[4:3]`	Way
`A[2:0]`	`000`

The Instruction Cache Tags extended data access value is the same as for data cache.

Table 7-10: Instruction Cache Data Extended Data Access Address and Mapping to L1 SRAM Subbank

L1 SRAM Address Bits	Cache Location
`A[31:20]`	L1 Instruction SRAM Address
`A[19]`	`PARCTL`
`A[18]`	`PARSEL`
`A[17:15]`	Usually b#001 (Depends on Size of L1 Instruction SRAM)
`A[14]`	`1`
`A[13:12]`	Way
`A[11:5]`	Set Index

Table 7-10: Instruction Cache Data Extended Data Access Address and Mapping to L1 SRAM Subbank (Continued)

L1 SRAM Address Bits	Cache Location
`A[4:0]`	Byte Offset in Cache Line

Cache Fills and Victims

Cacheable accesses do not use the Read/Write buffer, but instead use a similar structure called the Fill/Victim buffer. These buffers are used on a cache read miss and when a cache read miss generates a victim cache line that needs to be written back to memory. A Fill/Victim buffer entry consists of an address, 32 bytes of data, and cache line status information.

The CPLBs describe the cache mode to be used for various memory regions, either non-cacheable, Write-Back cacheable, or Write-Through cacheable. Traffic for non-cacheable memory is described in System Memory Access.

- In write-through mode, a write to external memory is initiated immediately upon the write to cache. If the cache line is present in cache, it is also updated there, but the cache line is not fetched on a write miss. If the cache line is replaced or explicitly flushed by software, the contents of the cache line are invalidated rather than written back to external memory.
- In write-back mode, the cache does not write to external memory until the line is replaced by a load operation that needs the line. For most applications, a write-back cache is more efficient than a write-through cache, as the external memory accesses are less frequent. On a write miss, the cache line is filled from the system and then modified in the cache by the write data.

When a read miss occurs, a cache line fill is always initiated.

A cacheable write or a read which misses the cache will allocate a Fill/Victim buffer entry. If a write also causes a cache fill, the write data and fill data will be merged into the Fill buffer before being drained to L1 memory.

When a modified cache line is evicted from the cache, it must be written back to memory. In the reverse process of a fill, a Fill buffer is allocated to receive the cache line from L1 memory before it is then forwarded to system memory. The process of reading the cache victim from L1 requires the DAG interface to be stalled, as the DAG0 read port is used to read the victimized cache line from L1 memory. Cache victims are created by cache flush operations or when a cache fill displaces a cache line in the same set because the cache set is full.

The Fill/Victim buffer performs data forwarding in the same manner as the Read/Write buffer. If valid data exists in the Fill/Victim buffer, it can be forwarded to the history buffer with few limitations. Write gathering is also done in the Fill/Victim buffer.

The Fill/Victim buffer can stall the L1 memory pipeline if buffers are not available or if it is in a state where it cannot respond to a request.

Terminology

The following terminology is used to describe memory.

cache block

The smallest unit of memory that is transferred to/from the next level of memory from/to a cache memory as a result of a cache miss.

cache hit

A memory access that is satisfied by a present entry in the cache with its Valid bit set.

cache line

Same as cache block.

cache miss

A memory access that does not match any valid entry in the cache.

direct-mapped

Cache architecture in which each line has only one place in which it can appear in the cache. Also described as 1-Way associative.

dirty/modified

A state bit, stored along with the tag, indicating whether the data in the data cache line has been changed since it was copied from the source memory and, therefore, needs to be updated in that source memory.

exclusive, clean

The state of a data cache line, indicating that the line is valid and that the data contained in the line matches that in the source memory. The data in a clean cache line does not need to be written to source memory before it is replaced.

fully associative

Cache architecture in which each line can be placed anywhere in the cache.

index

Address portion that is used to select an array element (for example, a line index).

invalid

Describes the state of a cache line. When a cache line is invalid, a cache line match cannot occur.

least recently used (LRU) algorithm

Replacement algorithm, used by some caches, that first replaces lines that have been unused for the longest time (this is the legacy Blackfin implementation).

Level 1 (L1) memory

Memory that is directly accessed by the core with no intervening memory subsystems between it and the core.

little endian

The native data store format of the Blackfin processor. Words and half words are stored in memory (and registers) with the least significant byte at the lowest byte address and the most significant byte in the highest byte address of the data storage location.

replacement policy

The function used by the processor to determine which line to replace on a cache miss. In Blackfin+ processors, a round-robin algorithm is employed (ways are iteratively cycled through as replacement is required).

set

A group of *N*-line storage locations in the Ways of an *N*-Way cache, selected by the `INDEX` field of the address (see the Cache Lines figures).

set associative

Cache architecture that limits line placement to a number of sets (or Ways).

tag

Upper address bits, stored along with the cached data line, to identify the specific address source in memory that the cached line represents.

valid

A state bit, stored with the tag, indicating that the corresponding tag and data are current/correct and can be used to satisfy memory access requests.

victim

A dirty cache line that must be written to memory before it can be replaced to free space for a cache line allocation.

Way

An array of line storage elements in an *N*-Way cache (see the Cache Lines figures).

write back

A cache write policy, also known as *copyback*

Blackfin+ L1IM Register Descriptions

L1 Instruction Memory Unit (L1IM) contains the following registers.

Table 7-11: Blackfin+ L1IM Register List

Name	Description
L1IM_ICPLB_ADDR[n]	Instruction Memory CPLB Address Registers
L1IM_ICPLB_DATA[n]	Instruction Memory CPLB Data Registers
L1IM_ICPLB_DFLT	Instruction Memory CPLB Default Settings Register
L1IM_ICPLB_FAULT_ADDR	Instruction Memory CPLB Fault Address Register
L1IM_ICTL	Instruction Memory Control Register
L1IM_IPERR_STAT	Instruction Parity Error Status Register
L1IM_ISTAT	Instruction Memory CPLB Status Register

Instruction Memory CPLB Address Registers

Each of the L1IM_ICPLB_ADDR[n] registers, along with its corresponding L1IM_ICPLB_DATA[n] register, comprise a single valid Instruction Cacheability Protection Lookaside Buffer (ICPLB) table entry descriptor pair required by the Memory Management Unit to configure and enable cache and/or memory protection.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared whenever this register is written.

For instruction fetch operations, L1IM_ICPLB_ADDR[n] defines the start address of the page described by the ICPLB descriptor, and the associated L1IM_ICPLB_DATA[n] register defines the properties of the page described by the ICPLB descriptor.

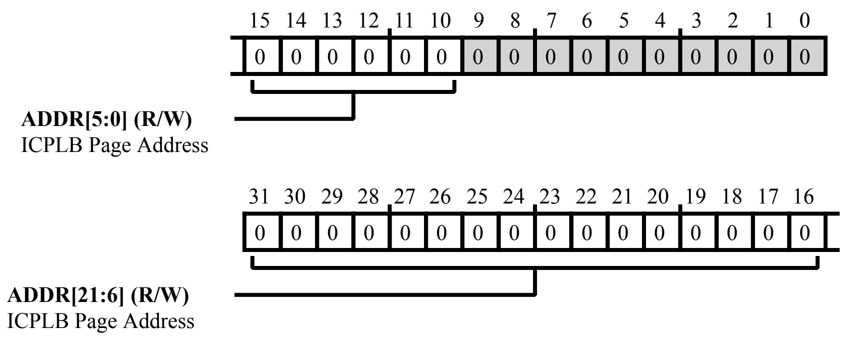


Figure 7-8: L1IM_ICPLB_ADDR[n] Register Diagram

Table 7-12: L1IM_ICPLB_ADDR[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:10 (R/W)	ADDR	ICPLB Page Address. The L1IM_ICPLB_ADDR[n].ADDR bits contain the start address for the page defined by the associated L1IM_ICPLB_DATA[n] register for address match operations by the Memory Management Unit.

Instruction Memory CPLB Data Registers

Each of the L1IM_ICPLB_DATA[n] registers, along with its corresponding L1IM_ICPLB_ADDR[n] register, comprise a single valid Instruction Cacheability Protection Lookaside Buffer (ICPLB) table entry descriptor pair required by the Memory Management Unit to configure and enable cache and/or memory protection.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared whenever this register is written.

For instruction fetch operations, L1IM_ICPLB_ADDR[n] defines the start address of the page described by the ICPLB descriptor, and the associated L1IM_ICPLB_DATA[n] register defines the properties of the page described by the ICPLB descriptor.

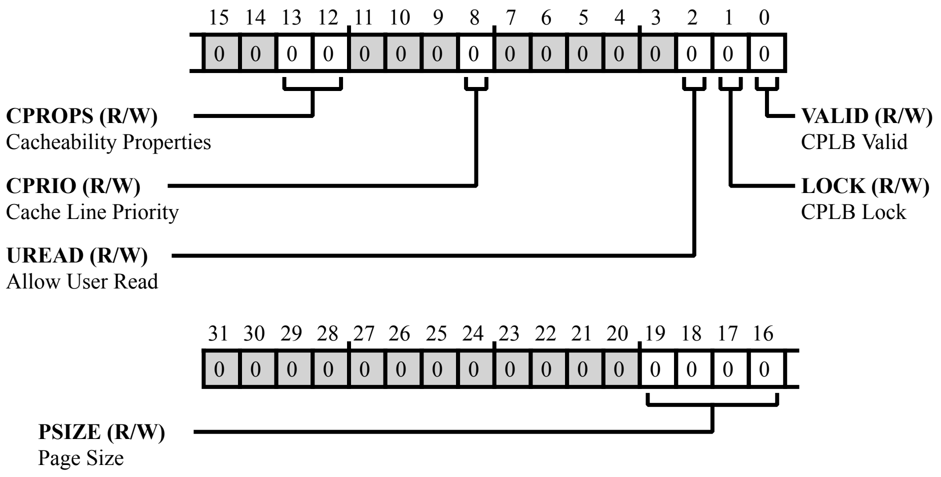


Figure 7-9: L1IM_ICPLB_DATA[n] Register Diagram

Table 7-13: L1IM_ICPLB_DATA[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19:16 (R/W)	PSIZE	Page Size. The L1IM_ICPLB_DATA[n].PSIZE bits select the page size according to the formula: page size = $4^{(\text{L1IM\_ICPLB\_DATA[n].PSIZE}+5)}$.	
		0	1 KB
		1	4 KB
		2	16 KB
		3	64 KB
		4	256 KB
		5	1 MB

Table 7-13: L1IM_ICPLB_DATA[n] Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		6	4 MB
		7	16 MB
		8	64 MB
		9	256 MB
		10	1 GB
		11-15	RESERVED
13:12 (R/W)	CPROPS	Cacheability Properties. The `L1IM_ICPLB_DATA[n].CPROPS` field determines whether or not the defined memory region can be placed in L1 instruction cache memory.	
		0	Non-cacheable
		1	Cacheable in L1
		2	Non-cacheable
		3	Cacheable in L1
8 (R/W)	CPRIO	Cache Line Priority. The `L1IM_ICPLB_DATA[n].CPRIO` bit indicates whether the cache line priority is marked as low or high importance when an entry from this page enters the cache.	
		0	Low importance
		1	High importance
2 (R/W)	UREAD	Allow User Read. The `L1IM_ICPLB_DATA[n].UREAD` bit indicates whether or not reads to this memory region are permitted when the processor in in User mode.	
		0	Restricted
		1	Permitted
1 (R/W)	LOCK	CPLB Lock. The `L1IM_ICPLB_DATA[n].LOCK` bit indicates whether or not the CPLB is locked (cannot be replaced) in the CPLB table.	
		0	Not locked
		1	Locked
0 (R/W)	VALID	CPLB Valid. The `L1IM_ICPLB_DATA[n].VALID` bit indicates whether or not the CPLB entry is valid in the CPLB table.	
		0	Invalid
		1	Valid

Instruction Memory CPLB Default Settings Register

The L1IM_ICPLB_DFLT register selects the default CPLB settings for new instruction memory CPLB entries. These default settings may be changed by writing the CPLB descriptor in the L1IM_ICPLB_DATA[n] register after the new entry is added.

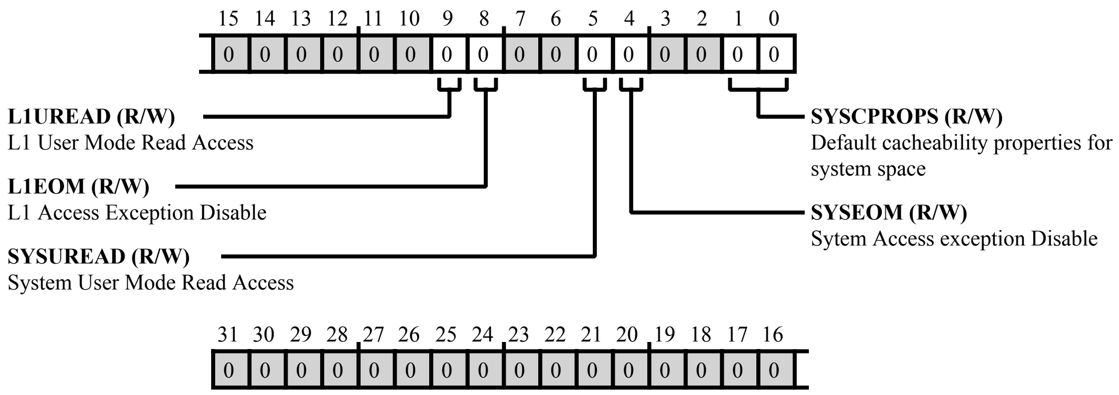


Figure 7-10: L1IM_ICPLB_DFLT Register Diagram

Table 7-14: L1IM_ICPLB_DFLT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
9 (R/W)	L1UREAD	L1 User Mode Read Access. The L1IM_ICPLB_DFLT.L1UREAD bit configures whether the default setting is to permit or restrict L1 read accesses from User mode.	
		0	Restricted
		1	Permitted
8 (R/W)	L1EOM	L1 Access Exception Disable. The L1IM_ICPLB_DFLT.L1EOM bit configures whether or not an ICPLB miss to L1 memory space generates an exception.	
		0	Generate exception
		1	Disable exception generation
5 (R/W)	SYSUREAD	System User Mode Read Access. The L1IM_ICPLB_DFLT.SYSUREAD bit configures whether the default setting is to permit or restrict system memory read accesses from User mode.	
		0	Restricted
		1	Permitted

Table 7-14: L1IM_ICPLB_DFLT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
4 (R/W)	SYSEOM	Sytem Access exception Disable. The `L1IM_ICPLB_DFLT.SYSEOM` bit configures whether or not an ICPLB miss to system memory space generates an exception.	
		0	Generate exception
		1	Disable exception generation
1:0 (R/W)	SYSCPROPS	Default cacheability properties for system space. The `L1IM_ICPLB_DFLT.SYSCPROPS` field determines the default behavior as to whether or not the defined memory region can be placed in L1 instruction cache memory.	
		0	Non-cacheable
		1	Cacheable in L1
		2	Non-cacheable
		3	Cacheable in L1

Instruction Memory CPLB Fault Address Register

The L1IM_ICPLB_FAULT_ADDR register holds the address of the memory location that caused a fault.

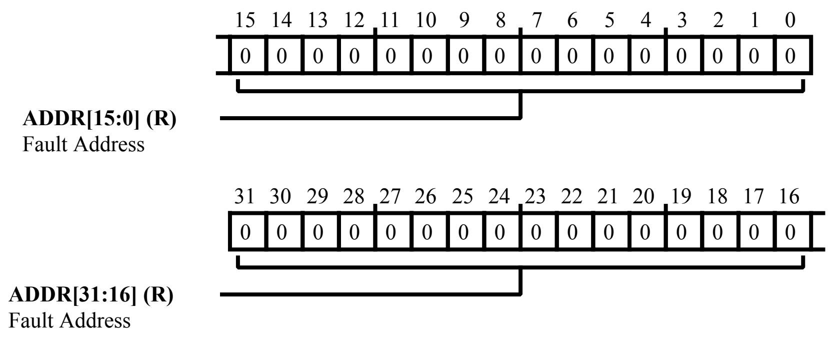


Figure 7-11: L1IM_ICPLB_FAULT_ADDR Register Diagram

Table 7-15: L1IM_ICPLB_FAULT_ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/NW)	ADDR	Fault Address.

Instruction Memory Control Register

The `L1IM_ICTL` register controls memory management unit operation of ICPLBs. This register enables CPLB operation, configures memory block usage, and selects the configuration of other ICPLB controls.

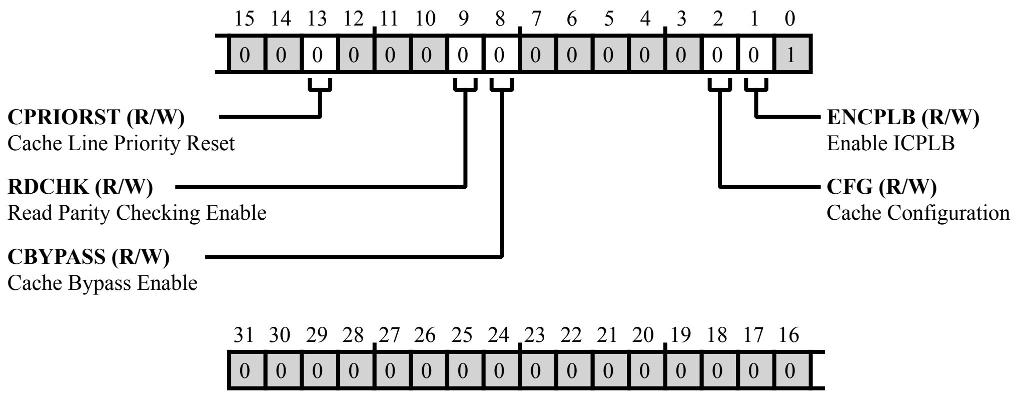


Figure 7-12: L1IM_ICTL Register Diagram

Table 7-16: L1IM_ICTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
13 (R/W)	CPRIORST	Cache Line Priority Reset. The `L1IM_ICTL.CPRIORST` bit determines whether or not the CPRIO bits are stored when cache lines are stored.	
		0	Store CPRIO bits
		1	Do not store CPRIO bits
9 (R/W)	RDCHK	Read Parity Checking Enable. The `L1IM_ICTL.RDCHK` bit determines whether or not read parity checking is enabled for this region of memory.	
		0	Disabled
		1	Enabled
8 (R/W)	CBYPASS	Cache Bypass Enable. The `L1IM_ICTL.CBYPASS` bit determines whether or not the cache is bypassed when accesses are made to this region of memory.	
		0	Do not bypass cache
		1	Bypass cache

Table 7-16: L1IM_ICTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
2 (R/W)	CFG	Cache Configuration. The `L1IM_ICTL.CFG` bit determines whether or not the configurable block of L1 Instruction SRAM is enabled as cache.	
		0	SRAM
		1	Cache
1 (R/W)	ENCPLB	Enable ICPLB. The `L1IM_ICTL.ENCPLB` bit determines whether or not ICPLBs are enabled to provide memory protection and/or cache support.	
		0	Disable ICPLBs
		1	Enable ICPLBs

Instruction Parity Error Status Register

The L1IM_IPERR_STAT register contains status information for identifying the location and properties of a parity error occurring during a read access of an address in L1 instruction memory.

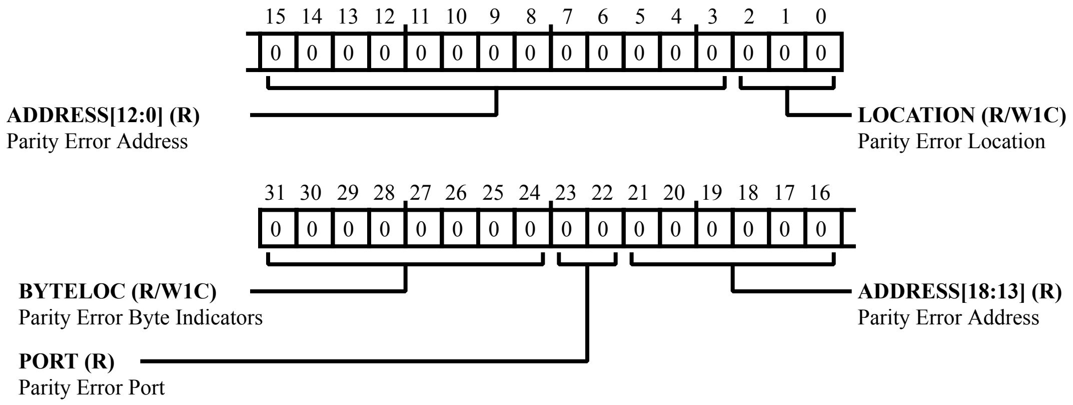


Figure 7-13: L1IM_IPERR_STAT Register Diagram

Table 7-17: L1IM_IPERR_STAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/W1C)	BYTELOC	Parity Error Byte Indicators. The L1IM_IPERR_STAT.BYTELOC bits indicate the byte locations within the 8-byte chunk associated with the L1IM_IPERR_STAT.ADDRESS field that produced read parity error faults. The error indication for bits in this field is quad-word aligned (i.e, if a byte access to address 0x11A00005 generated a parity error, the L1IM_IPERR_STAT.BYTELOC[5] bit is set, not the L1IM_IPERR_STAT.BYTELOC[0] bit). These bits are sticky, thus a W1C action is required to clear them.	
23:22 (R/NW)	PORT	Parity Error Port. The L1IM_IPERR_STAT.PORT bits contain the encoding for the read port on which the parity error occurred.	
		0	Port 0
		1	Port 1
		2	DMA
		3	Cache victim
21:3 (R/NW)	ADDRESS	Parity Error Address. The L1IM_IPERR_STAT.ADDRESS bits provide the byte address of the memory location in the most recent read that produced a parity error. This address is the quad-word aligned base address for the faulting location. Use the L1IM_IPERR_STAT.BYTELOC field to determine the location of the faulting byte(s).	

Table 7-17: L1IM_IPERR_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
2:0 (R/W1C)	LOCATION	Parity Error Location. The `L1IM_IPERR_STAT.LOCATION` bits contain the encoding for the memory type for the location where the parity error occurred. These bits are sticky, thus a W1C action is required to clear them.	
		0	L1 SRAM
		1	Tag 0 memory
		2-7	Reserved

Instruction Memory CPLB Status Register

The L1IM_ISTAT register holds information regarding the ICPLB fault that occurred during an access to instruction memory. These bits indicate the processor mode during the access, whether or not the access was to an illegal address, and which ICPLB entry is associated with the fault. This status information is only valid while in the context of the ICPLB exception service routine.

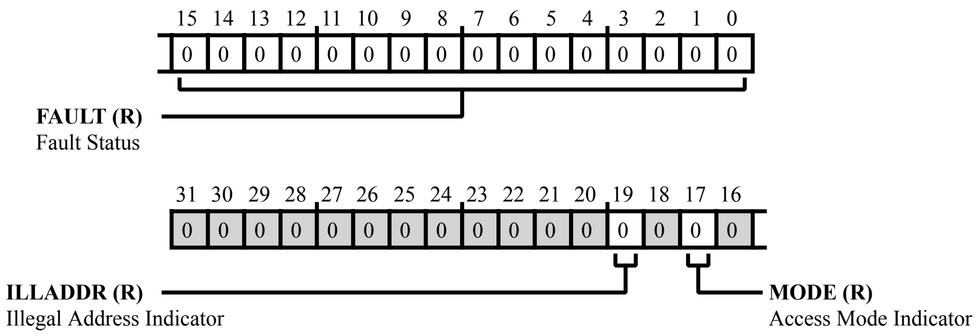


Figure 7-14: L1IM_ISTAT Register Diagram

Table 7-18: L1IM_ISTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19 (R/NW)	ILLADDR	Illegal Address Indicator. The L1IM_ISTAT.ILLADDR bit indicates whether or not the attempted access was to an illegal memory location.	
		0	No status
		1	Illegal address fault
17 (R/NW)	MODE	Access Mode Indicator. The L1IM_ISTAT.MODE bit indicates which mode the processor was in when the fault address was accessed.	
		0	User mode
		1	Supervisor mode
15:0 (R/NW)	FAULT	Fault Status. Each bit in the L1IM_ISTAT.FAULT field is associated with an ICPLB entry in the ICPLB table. A set bit indicates that the fault occurred on that ICPLB entry (e.g., if the fault occurred on the page defined by the L1IM_ICPLB_DATA5 register, the L1IM_ISTAT.FAULT[5] bit is set).	

Blackfin+ L1DM Register Descriptions

L1 Data Memory Unit (L1DM) contains the following registers.

Table 7-19: Blackfin+ L1DM Register List

Name	Description
L1DM_DCPLB_ADDR[n]	Data Memory CPLB Address Registers
L1DM_DCPLB_DATA[n]	Data Memory CPLB Data Registers
L1DM_DCPLB_DFLT	Data Memory CPLB Default Settings Register
L1DM_DCPLB_FAULT_ADDR	Data Memory CPLB Fault Address Register
L1DM_DCTL	Data Memory Control Register
L1DM_DPERR_STAT	Data Memory Parity Error Status Register
L1DM_DSTAT	Data Memory CPLB Status Register
L1DM_SRAM_BASE_ADDR	SRAM Base Address Register

Data Memory CPLB Address Registers

Each of the `L1DM_DCPLB_ADDR[n]` registers, along with its corresponding `L1DM_DCPLB_DATA[n]` register, comprise a single valid Data Cacheability Protection Lookaside Buffer (DCPLB) table entry descriptor pair required by the Memory Management Unit to configure and enable cache and/or memory protection.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared whenever this register is written.

For data fetch operations, `L1DM_DCPLB_ADDR[n]` defines the start address of the page described by the DCPLB descriptor, and the associated `L1DM_DCPLB_DATA[n]` register defines the properties of the page described by the DCPLB descriptor.

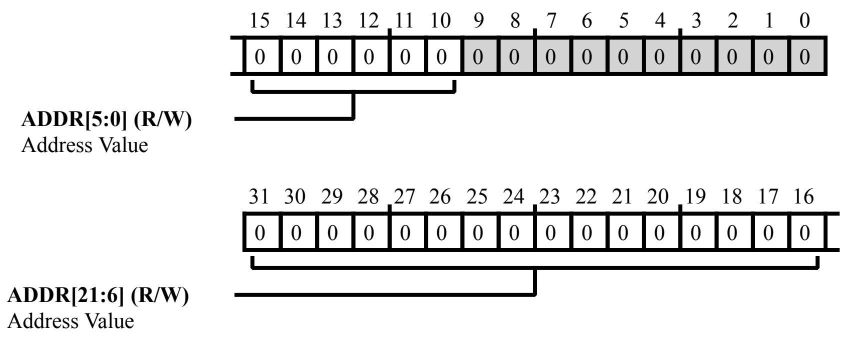


Figure 7-15: L1DM_DCPLB_ADDR[n] Register Diagram

Table 7-20: L1DM_DCPLB_ADDR[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:10 (R/W)	ADDR	Address Value. The `L1DM_DCPLB_ADDR[n].ADDR` bits contain the start address for the page defined by the associated `L1DM_DCPLB_DATA[n]` register for address match operations by the Memory Management Unit.

Data Memory CPLB Data Registers

Each of the L1DM_DCPLB_DATA[n] registers, along with its corresponding L1DM_DCPLB_ADDR[n] register, comprise a single valid Data Cacheability Protection Lookaside Buffer (DCPLB) table entry descriptor pair required by the Memory Management Unit to configure and enable cache and/or memory protection.

NOTE: To ensure proper behavior and future compatibility, all reserved bits in this register must be cleared whenever this register is written.

For data fetch operations, L1DM_DCPLB_ADDR[n] defines the start address of the page described by the DCPLB descriptor, and the associated L1DM_DCPLB_DATA[n] register defines the properties of the page described by the DCPLB descriptor.

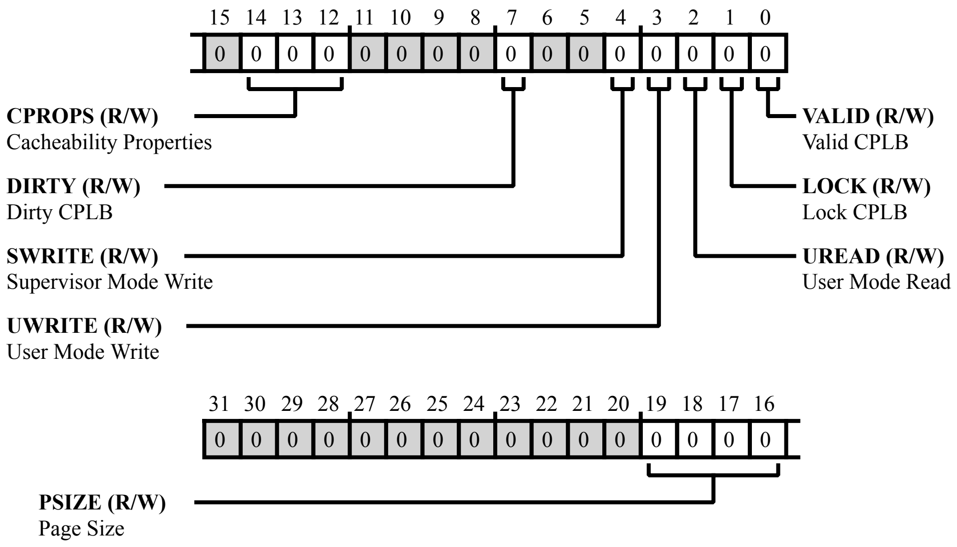


Figure 7-16: L1DM_DCPLB_DATA[n] Register Diagram

Table 7-21: L1DM_DCPLB_DATA[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19:16 (R/W)	PSIZE	Page Size. The L1DM_DCPLB_DATA[n].PSIZE bits select the page size according to the formula: page size = $4^{(\text{L1DM\_DCPLB\_DATA[n].PSIZE}+5)}$.	
		0	1 KB
		1	4 KB
		2	16 KB
		3	64 KB

Table 7-21: L1DM_DCPLB_DATA[n] Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		4	256 KB
		5	1 MB
		6	4 MB
		7	16 MB
		8	64 MB
		9	256 MB
		10	1 GB
		11-15	RESERVED
14:12 (R/W)	CPROPS	Cacheability Properties. The `L1DM_DCPLB_DATA[n].CPROPS` bits select the cacheability properties for the page, determining whether or not the page is cacheable (write-back or write-through) in L1 memory.	
		0	Not cacheable
		1	Write-back cacheable in L1
		2	Not cacheable
		3	Write-back cacheable in L1
		4	I/O device space
		5	Write-through cacheable in L1
		6	Not cacheable
		7	Write-through cacheable in L1
7 (R/W)	DIRTY	Dirty CPLB. The `L1DM_DCPLB_DATA[n].DIRTY` bit is used by software to indicate that a write has been made to the page since the CPLB entry was installed. On the first write to the page after its installation, the MMU exception handler raises a CPLB dirty exception and sets this bit. To avoid this exception, software may set this bit when the CPLB entry is installed. Software may use the `L1DM_DCPLB_DATA[n].DIRTY` bit to detect whether a write as been made to a page so that the write may be propagated to further levels of the memory hierarchy.	
		0	Clean - CPLB dirty exception raised on page write A CPLB dirty exception is raised if the page is written to.
		1	Dirty - No CPLB dirty exception raised on page write A CPLB dirty exception is not raised when the page is written to.

Table 7-21: L1DM_DCPLB_DATA[n] Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
4 (R/W)	SWRITE	Supervisor Mode Write. The `L1DM_DCPLB_DATA[n].SWRITE` bit selects whether to permit or restrict Supervisor mode write access. If access is attempted when restricted, the access generates a protection violation exception.	
		0	Restricted
		1	Permitted
3 (R/W)	UWRITE	User Mode Write. The `L1DM_DCPLB_DATA[n].UWRITE` bit selects whether to permit or restrict User mode write access. If access is attempted when restricted, the access generates a protection violation exception.	
		0	Restricted
		1	Permitted
2 (R/W)	UREAD	User Mode Read. The `L1DM_DCPLB_DATA[n].UREAD` bit selects whether to permit or restrict User mode read access. If access is attempted when restricted, the access generates a protection violation exception.	
		0	Restricted
		1	Permitted
1 (R/W)	LOCK	Lock CPLB. The `L1DM_DCPLB_DATA[n].LOCK` bit locks or unlocks the CPLB entry. When locked, the MMU is directed to keep this entry in the MMRs rather than participate in the CPLB replacement policy algorithm.	
		0	Unlocked
		1	Locked
0 (R/W)	VALID	Valid CPLB. The `L1DM_DCPLB_DATA[n].VALID` bit indicates whether or not the CPLB entry contains valid data. Software uses this bit to identify valid CPLB entries in the MMU exception handler when executing the CPLB replacement policy.	
		0	Invalid
		1	Valid

Data Memory CPLB Default Settings Register

The L1DM_DCPLB_DFLT register selects the default DCPLB settings for new data memory CPLB entries. These default settings may be changed by writing the DCPLB descriptor in the L1DM_DCPLB_DATA[n] register after the new entry is added.

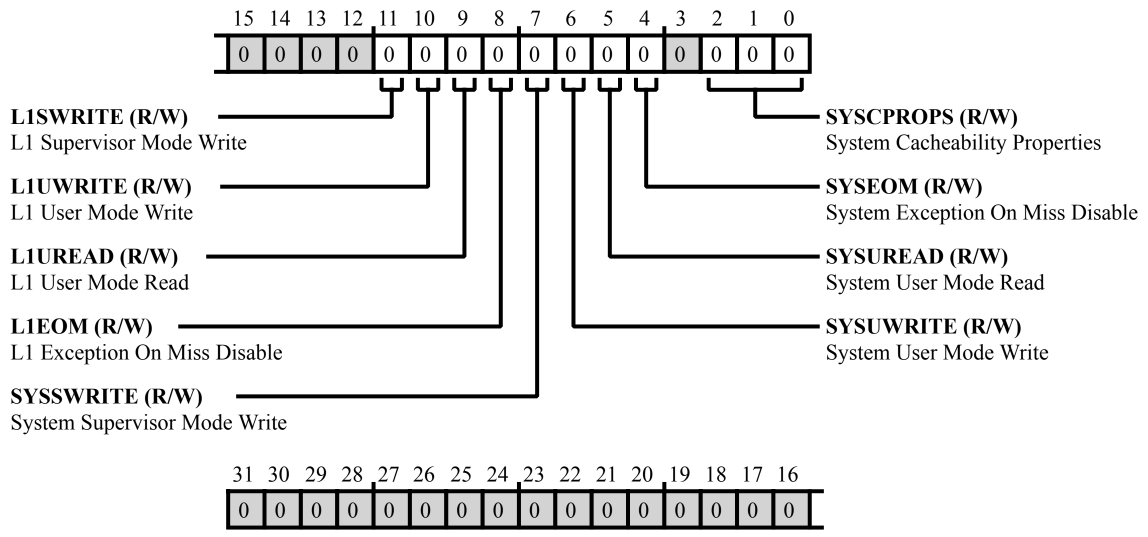


Figure 7-17: L1DM_DCPLB_DFLT Register Diagram

Table 7-22: L1DM_DCPLB_DFLT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
11 (R/W)	L1SWRITE	L1 Supervisor Mode Write. The L1DM_DCPLB_DFLT.L1SWRITE bit selects the default for Supervisor mode write access to L1 memory, which determines whether to permit or restrict Supervisor mode write access. If a write is attempted while restricted, the access generates a protection violation exception. This default setting is overridden by the L1DM_DCPLB_DATA[n].SWRITE bit in a valid enabled DCPLB entry.	
		0	Restricted
		1	Permitted
10 (R/W)	L1UWRITE	L1 User Mode Write. The L1DM_DCPLB_DFLT.L1UWRITE bit selects the default for User mode write access to L1 memory, which determines whether to permit or restrict User mode write access. If a write is attempted while restricted, the access generates a protection violation exception. This default setting is overridden by the L1DM_DCPLB_DATA[n].UWRITE bit in a valid enabled DCPLB.	
		0	Restricted
		1	Permitted

Table 7-22: L1DM_DCPLB_DFLT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
9 (R/W)	L1UREAD	L1 User Mode Read. The L1DM_DCPLB_DFLT.L1UREAD bit selects the default for User mode read access to L1 memory, which determines whether to permit or restrict User mode read access. If a read is attempted while restricted, the access generates a protection violation exception. This default setting is overridden by the L1DM_DCPLB_DATA[n].UREAD bit in a valid enabled DCPLB entry.	
		0	Restricted
		1	Permitted
8 (R/W)	L1EOM	L1 Exception On Miss Disable. The L1DM_DCPLB_DFLT.L1EOM bit disables access exception generation on a DAG DCPLB miss to L1 memory space. Default access protection for L1 memory space is controlled by the L1DM_DCPLB_DFLT.L1UREAD, L1DM_DCPLB_DFLT.L1UWRITE, and L1DM_DCPLB_DFLT.L1SWRITE bits.	
		0	Generate exception
		1	Disable exception generation
7 (R/W)	SYSSWRITE	System Supervisor Mode Write. The L1DM_DCPLB_DFLT.SYSSWRITE bit selects the default for Supervisor mode write access to system memory space, which determines whether to permit or restrict Supervisor mode write access. If a write is attempted while restricted, the access generates a protection violation exception. This default setting is overridden by the L1DM_DCPLB_DATA[n].SWRITE bit in a valid enabled DCPLB.	
		0	Restricted
		1	Permitted
6 (R/W)	SYSUWRITE	System User Mode Write. The L1DM_DCPLB_DFLT.SYSUWRITE bit selects the default for User mode write access to system memory space, which determines whether to permit or restrict User mode write access. If a write is attempted while restricted, the access generates a protection violation exception. This default setting is overridden by the L1DM_DCPLB_DATA[n].UWRITE bit in a valid enabled DCPLB.	
		0	Restricted
		1	Permitted

Table 7-22: L1DM_DCPLB_DFLT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
5 (R/W)	SYSUREAD	System User Mode Read. The L1DM_DCPLB_DFLT.SYSUREAD bit selects the default for User mode read access to system memory space, which determines whether to permit or restrict User mode read access. If a read is attempted while restricted, the access generates a protection violation exception. This default setting is overridden by the L1DM_DCPLB_DATA[n].UREAD bit in a valid enabled DCPLB.	
		0	Restricted
		1	Permitted
4 (R/W)	SYSEOM	System Exception On Miss Disable. The L1DM_DCPLB_DFLT.SYSEOM bit disables access exception generation on a DAG DCPLB miss to system memory space. Default access protection for system memory space is controlled by the L1DM_DCPLB_DFLT.SYSCPROPS, L1DM_DCPLB_DFLT.SYSUREAD, L1DM_DCPLB_DFLT.SYSUWRITE, and L1DM_DCPLB_DFLT.SYSSWRITE bits.	
		0	Generate exception
		1	Disable exception generation
2:0 (R/W)	SYSCPROPS	System Cacheability Properties. The L1DM_DCPLB_DFLT.SYSCPROPS bits select the default system memory cacheability properties, which determine whether or not the region cacheable in L1 memory. This default setting is overridden by the L1DM_DCPLB_DATA[n].CPROPS bits in a valid enabled DCPLB.	
		0	Non-cacheable
		1	Write-back cacheable in L1
		2	Non-cacheable
		3	Write-back cacheable
		4	I/O device space
		5	Write-through cacheable in L1
		6	Non-cacheable
		7	Write-through cacheable in L1

Data Memory CPLB Fault Address Register

The L1DM_DCPLB_FAULT_ADDR register holds the address of the memory location that caused a fault in data memory.

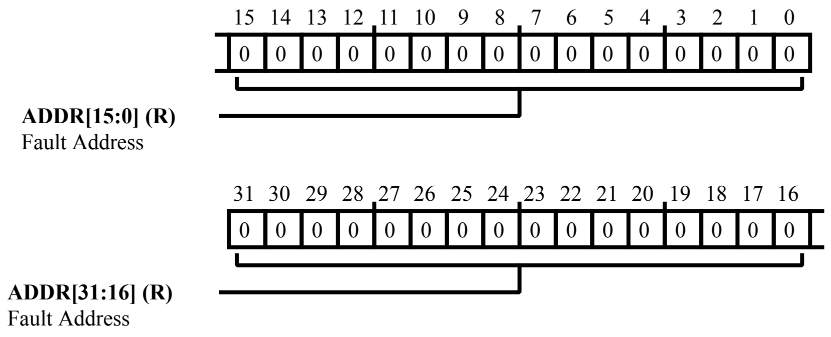


Figure 7-18: L1DM_DCPLB_FAULT_ADDR Register Diagram

Table 7-23: L1DM_DCPLB_FAULT_ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/NW)	ADDR	Fault Address.

Data Memory Control Register

The L1DM_DCTL register controls memory management unit operation for DCPLBs. This register enables DCPLB operation, configures memory block usage, and selects the configuration of other DCPLB controls.

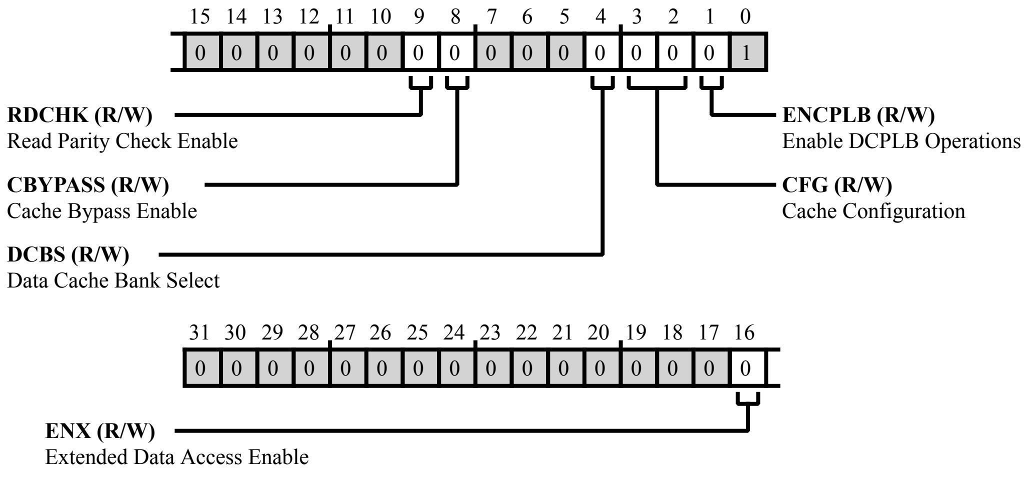


Figure 7-19: L1DM_DCTL Register Diagram

Table 7-24: L1DM_DCTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
16 (R/W)	ENX	Extended Data Access Enable. The L1DM_DCTL.ENX bit enables Extended Data Access mode, where all restricted memory spaces (including L1 instruction memory) may be directly accessed by suitably-privileged software. For more information, see the Extended Data Access section.	
		0	Disable (default)
		1	Enable
9 (R/W)	RDCHK	Read Parity Check Enable. The L1DM_DCTL.RDCHK bit enables parity checking for read accesses.	
		0	Disable (default)
		1	Enable
8 (R/W)	CBYPASS	Cache Bypass Enable. The L1DM_DCTL.CBYPASS bit enables cache bypass, thus disabling processor use of the data cache.	
		0	Do not bypass (default)
		1	Bypass

Table 7-24: L1DM_DCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
4 (R/W)	DCBS	Data Cache Bank Select. The `L1DM_DCTL.DCBS` bit selects whether the memory management unit uses address bit 14 or 23 to choose between L1 data memory banks A and B. See the Data Cache Block Select section for details.	
		0	Bit 14
		1	Bit 23
3:2 (R/W)	CFG	Cache Configuration. The `L1DM_DCTL.CFG` bits set the usage of the configurable regions in L1 memory data blocks A and B as either cache or SRAM. When this field is cleared, all cache lines for regions previously configured as cache are invalidated.	
		0	No cache
		1	Data block A cache enable
		3	Data blocks A and B cache enable
1 (R/W)	ENCPLB	Enable DCPLB Operations. The `L1DM_DCTL.ENCPLB` bit enables data memory CPLB operation. When disabled, the memory management unit only performs minimal address checking.	
		0	Disable DCPLBs
		1	Enable DCPLBs

Data Memory Parity Error Status Register

The L1DM_DPERR_STAT register contains status information for identifying the location and properties of a parity error occurring during a read access of an address in L1 data memory.

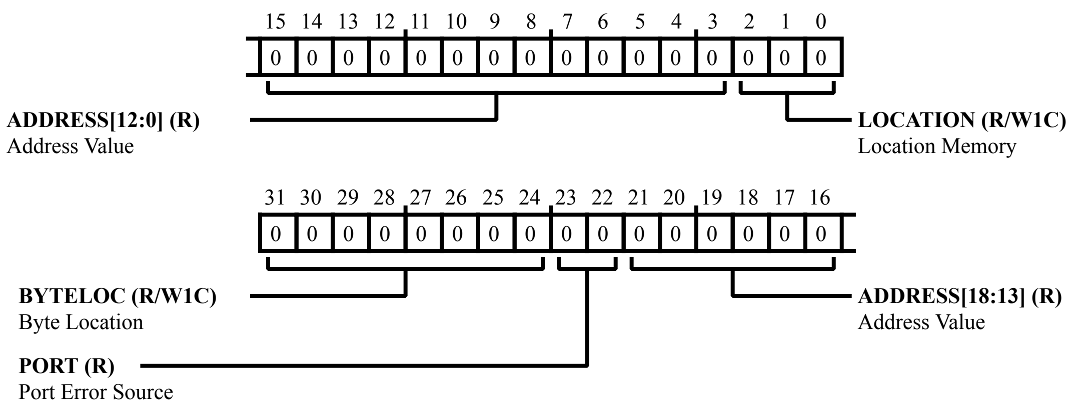


Figure 7-20: L1DM_DPERR_STAT Register Diagram

Table 7-25: L1DM_DPERR_STAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/W1C)	BYTELOC	Byte Location. The L1DM_DPERR_STAT.BYTELOC bits indicate the byte locations within the 8-byte chunk associated with the L1DM_DPERR_STAT.ADDRESS field that produced read parity error faults. The error indication for bits in this field is quad-word aligned (i.e, if a byte access to address 0x11800005 generated a parity error, the L1DM_DPERR_STAT.BYTELOC[5] bit is set, not the L1DM_DPERR_STAT.BYTELOC[0] bit). These bits are sticky, thus a W1C action is required to clear them.	
23:22 (R/NW)	PORT	Port Error Source. The L1DM_DPERR_STAT.PORT bits contain the encoding for the read port on which the parity error occurred.	
		0	Port 0
		1	Port 1
		2	DMA
		3	Cache victim
21:3 (R/NW)	ADDRESS	Address Value. The L1DM_DPERR_STAT.ADDRESS bits provide the byte address of the memory location in the most recent read that produced a parity error. This address is the quad-word aligned base address for the faulting location. Use the L1DM_DPERR_STAT.BYTELOC field to determine the location of the faulting byte(s).	

Table 7-25: L1DM_DPERR_STAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
2:0 (R/W1C)	LOCATION	Location Memory. The `L1DM_DPERR_STAT.LOCATION` bits contain the encoding for the memory type for the location where the parity error occurred. These bits are sticky, thus a W1C action is required to clear them.	
		0	L1 SRAM
		1	Tag 0
		2	Tag 1
		4	Dirty L1 cache memory
		5-7	Reserved

Data Memory CPLB Status Register

The `L1DM_DSTAT` register identifies status information for a CPLB fault occurring during an access to data memory. These bits indicate the processor mode during the access, whether the access was a read or a write, whether or not it was a DAG access, whether or not the access was to an illegal address, and which DCPLB entry is associated with the fault. The status information in the `L1DM_DSTAT` register is only valid while in the context of a fault exception service routine.

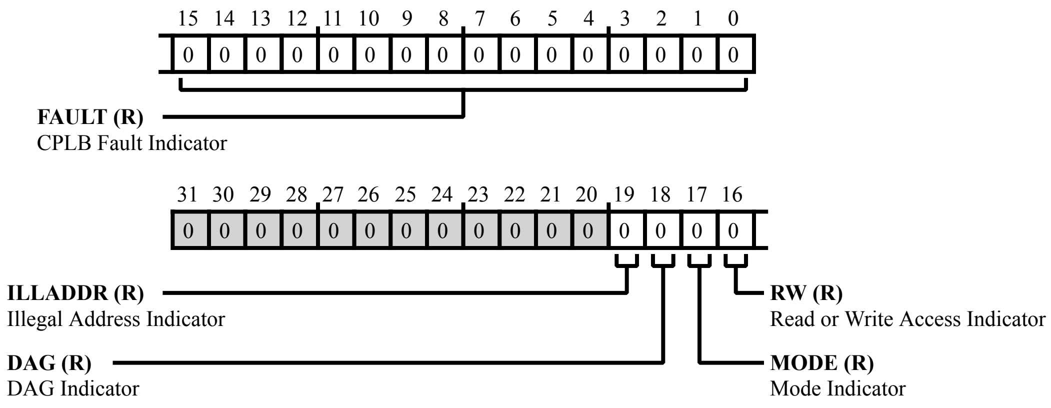


Figure 7-21: L1DM_DSTAT Register Diagram

Table 7-26: L1DM_DSTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
19 (R/NW)	ILLADDR	Illegal Address Indicator. The `L1DM_DSTAT.ILLADDR` bit indicates whether or not the fault occurred as a result of an attempt to access non-existent memory.	
		0	Not illegal access
		1	Illegal address
18 (R/NW)	DAG	DAG Indicator. The `L1DM_DSTAT.DAG` bit indicates whether the fault access was made by DAG0 or DAG1.	
		0	DAG 0
		1	DAG 1
17 (R/NW)	MODE	Mode Indicator. The `L1DM_DSTAT.MODE` bit indicates whether the processor mode was User or Supervisor during the faulting access.	
		0	User mode
		1	Supervisor mode

Table 7-26: L1DM_DSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
16 (R/NW)	RW	Read or Write Access Indicator. The L1DM_DSTAT.RW bit indicates whether the faulting access was a read or a write.	
		0	Read
		1	Write
15:0 (R/NW)	FAULT	CPLB Fault Indicator. Each bit in the L1DM_DSTAT.FAULT field is associated with a DCPLB entry in the DCPLB table. A set bit indicates that the fault occurred on that DCPLB entry (e.g., if the fault occurred on the page defined by the L1DM_DCPLB_DATA5 register, the L1DM_DSTAT.FAULT[5] bit is set).	

SRAM Base Address Register

When the data or instruction memories are configured as SRAM (see the `L1DM_DCTL` and register descriptions), the base address is determined from the `L1DM_SRAM_BASE_ADDR` register. The SRAM base address inputs to the core are latched into this register at reset. The SRAM base address is aligned to a 4 MB boundary and cannot overlap the uppermost 4 MB region because this region is reserved for the processor core and chip-level memory-mapped registers. The `L1DM_SRAM_BASE_ADDR` register is only accessible in Supervisor mode.

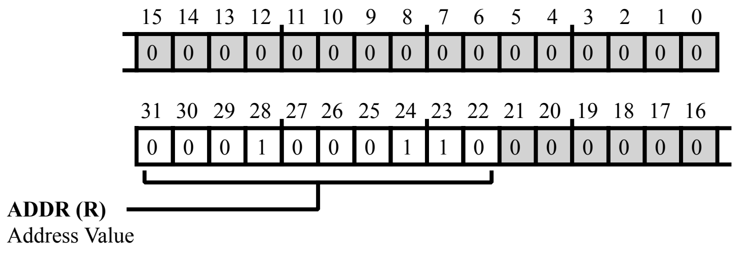


Figure 7-22: L1DM_SRAM_BASE_ADDR Register Diagram

Table 7-27: L1DM_SRAM_BASE_ADDR Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:22 (R/NW)	ADDR	Address Value. The `L1DM_SRAM_BASE_ADDR.ADDR` bits hold the SRAM base address. This address is aligned to any 4 MB boundary except for the uppermost 4 MB region (reserved for memory-mapped registers).

8 Instruction Reference Pages

The instruction reference pages provide detailed information about the syntax and operation of each instruction in the processor's instruction set. The reference groups the instructions by type and by operation. This grouping stems from the portion of the processor core (see the ***Blackfin+ Core Block Diagram*** figure), on which each instruction executes. Because each instruction uses specific resources (portions of the processor architecture), understanding the relationship between the instructions and the architecture can greatly influence how to write efficient code and achieve optimum code density (applying instruction parallelism).

- Arithmetic Instructions -- execute within the data arithmetic unit
- Sequencer Instructions -- execute within the control unit
- Memory or Pointer Instructions -- execute within the address arithmetic unit
- Specialized Compute Instructions -- execute within the data arithmetic unit

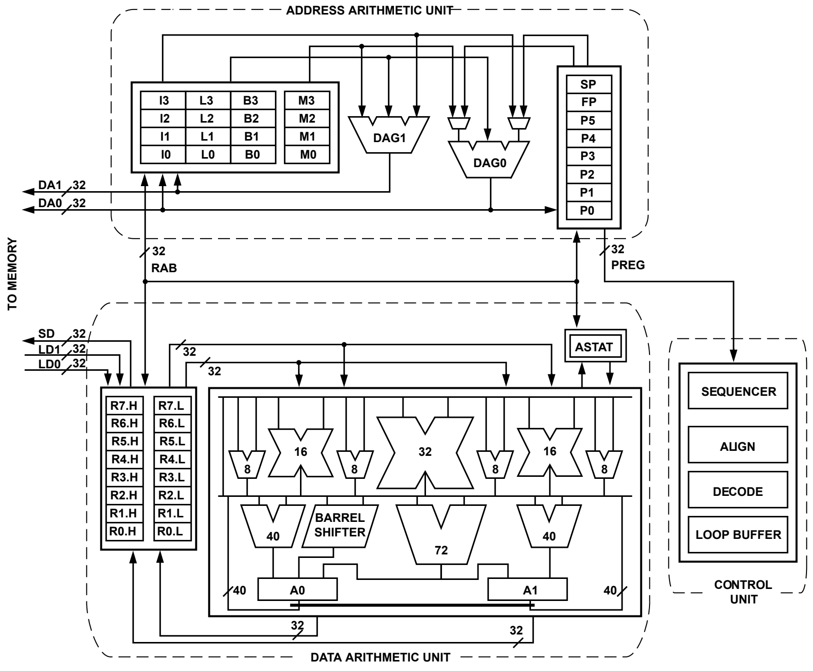


Figure 8-1: Blackfin+ Core Block Diagram

NOTE: Arithmetic instructions generate status, indicating information about the result of the operation. For more information, see Arithmetic Status Register . To optimize program execution, many 16- and 32-bit instructions may be issued in parallel. For more information, see Issuing Parallel Instructions.

For more information about ADSP-BF70x processor family core architecture or memory infrastructure, see the corresponding chapters of this text. For information about ADSP-BF70x processor peripherals, see the hardware reference manual.

Each instruction reference page provides the following information:

- ***Syntax*** -- each section of a syntax table identifies the underlying instruction encoding (e.g., ALU Operations (Dsp32Alu)). Each line of a syntax table defines the permitted processor resource classes (e.g., a register type) that is allowed in that syntax position. To see the list of resources in a particular class, follow the link for that resource class from the syntax line (e.g., DDST0_HL).
- ***Data Flow*** -- for instructions with sophisticated data placement options, a data flow diagram is provided (not provided for all instructions).
- ***Abstract*** -- brief (1-2 sentence) description of the instruction.
- ***Description*** -- provides a full description of the instruction, including execution options, instruction encoding size, instruction parallelism (if applicable), any special applications, and affect on status flags (if applicable).
- ***ASTAT Flags*** -- a table (where applicable) detailing the status flags affected by the instruction's execution. For instructions that do not affect status, this section is omitted.
- ***Example*** -- provide a code snippet that demonstrates the instruction and its options.

Arithmetic Instructions

The arithmetic instructions provide operations which execute on the ***data arithmetic unit*** in the processor core. Users can take advantage of these instructions to add, subtract, divide, and multiply, as well as to calculate and store absolute values, detect exponents, round, saturate, and return the number of sign bits.

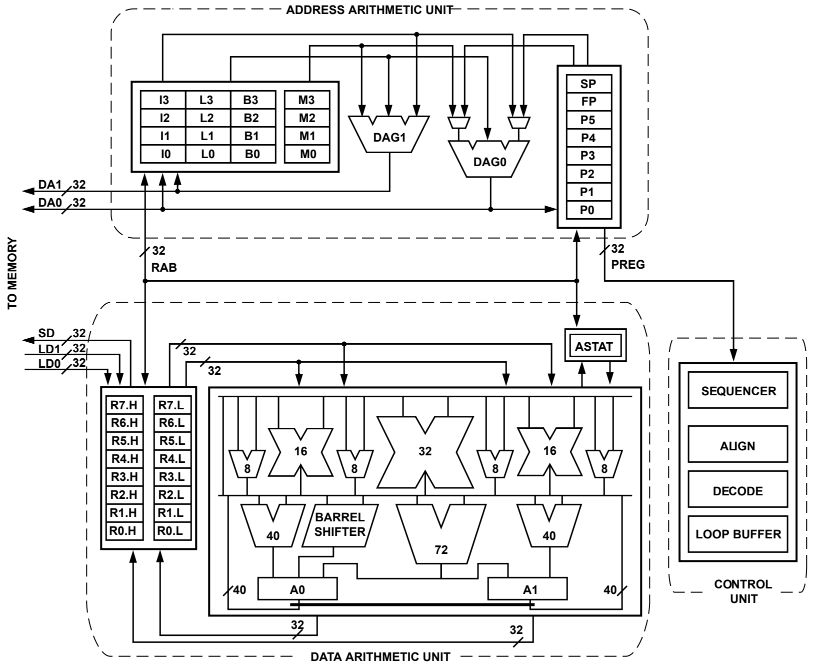


Figure 8-2: Blackfin+ Core Block Diagram

The operation types of arithmetic instructions include:

- Add and Subtract Operations
- Bit Operations
- Comparison Operations
- Conversion Operations
- Logic Operations
- Multiplication Operations
- Rotate Operations
- Shift Operations

Add and Subtract Operations

These operations provide addition and/or subtraction operations on register and immediate value operands:

- 16-Bit Add or Subtract (AddSub16)
- Vectored 16-Bit Add or Subtract (AddSubVec16)
- 32-bit Add or Subtract (AddSub32)
- 32-bit Add and Subtract (AddSub32Dual)
- 32-Bit Add or Subtract with Carry (AddSubAC0)

- 32-bit Add Constant (AddImm)
- Accumulator Add or Subtract (AddSubAcc)
- Accumulator Add and Extract (AddAccExt)
- Dual Accumulator Add and Subtract to Registers (AddSubAccExt)
- 32-bit Add then Shift (AddSubShift)

16-Bit Add or Subtract (AddSub16)

General Form

ALU Operations (Dsp32Alu)
DDST0_HL = DREG_L Register Type + DREG_L Register Type SAT2
DDST0_HL = DREG_L Register Type + DREG_H Register Type SAT2
DDST0_HL = DREG_H Register Type + DREG_L Register Type SAT2
DDST0_HL = DREG_H Register Type + DREG_H Register Type SAT2
DDST0_HL = DREG_L Register Type - DREG_L Register Type SAT2
DDST0_HL = DREG_L Register Type - DREG_H Register Type SAT2
DDST0_HL = DREG_H Register Type - DREG_L Register Type SAT2
DDST0_HL = DREG_H Register Type - DREG_H Register Type SAT2

Abstract

This instruction adds or subtracts two signed register halves.

See Also (Vectored 16-Bit Add or Subtract (AddSubVec16))

AddSub16 Description

The AddSub16 instruction adds or subtracts two source values and places the result in a destination register with or without result saturation.

AddSub16 accepts any combination of upper and lower half-register operands, and places the results in the upper or lower half of the destination register at the user's discretion.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

In the syntax, where `SAT2` appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see the Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSub16 Example

```
/* If r0.l = 0x7000 and r7.l = 0x2000, then . . . */
r4.l = r0.l + r7.l (ns) ;   /* produces r4.l = 0x9000 (no saturation is enforced) */
r4.l = r0.l + r7.h (s) ;    /* produces r4.l = 0x7FFF (saturated to maximum positive value)
*/

r0.l = r2.h + r4.l (ns) ;
r1.l = r3.h + r7.h (ns) ;
r4.h = r0.l + r7.l (ns) ;
r4.h = r0.l + r7.h (ns) ;
r0.h = r2.h + r4.l (s) ; /* saturate the result */
r1.h = r3.h + r7.h (ns) ;

r4.l = r0.l - r7.l (ns) ;
r4.l = r0.l - r7.h (s) ; /* saturate the result */
r0.l = r2.h - r4.l (ns) ;
r1.l = r3.h - r7.h (ns) ;
r4.h = r0.l - r7.l (ns) ;
r4.h = r0.l - r7.h (ns) ;
r0.h = r2.h - r4.l (s) ; /* saturate the result */
r1.h = r3.h - r7.h (ns) ;
```

Vectored 16-Bit Add or Subtract (AddSubVec16)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = DREG Register Type AOPL DREG Register Type SX
DREG Register Type = DREG Register Type +|+ DREG Register Type, DREG Register Type = DREG Register Type -|- DREG Register Type SXA
DREG Register Type = DREG Register Type +|- DREG Register Type, DREG Register Type = DREG Register Type -|+ DREG Register Type SXA

Abstract

This instruction adds or subtracts two set s of two signed 16-bit vectors, and it deposits them into two destination registers. Optionally, the result of the additions can be saturated. Also, the y inputs can be "crossed" so that instead of adding `Rx.H + Ry.H`, the crossed inputs allow for `Rx.H + Ry.L`(and so on). The output halves can also be crossed on compute unit 0.

See Also (16-Bit Add or Subtract (AddSub16))

AddSubVec16 Description

The Vector Add / Subtract instruction simultaneously adds and/or subtracts two pairs of registered numbers. It then stores the results of each operation into a separate 32-bit data register or 16-bit half register, according to the syntax used. The destination register for each of the quad or dual versions must be unique.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

The AddSubVec16 instruction supports dual and quad 16-Bit operations.

In the syntax, where *`SX`* appears (for dual 16-bit operations), substitute a saturation and/or cross output option (s, co, or sco) . In the syntax, where *`SXA`* appears (for quad 16-bit operations), substitute one of the *`SX`* values, substitute an arithmetic shift right or left option (asr or asl) ASR (arithmetic shift right). The options shown for quad 16-bit operations are scaling options. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

NOTE: A ***special application*** of the AddSubVec16 instruction is the FFT butterfly routines in which each of the registers is considered a single complex number often use the Vector Add / Subtract instruction.

```
/* If r1 = 0x0003 0004 and r2 = 0x0001 0002, then . . . */
r0 = r2 +|- r1(co) ; /* . . . produces r0 = 0xFFFE 0004 */
```

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see the Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	*AC1*	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubVec16 Example

```
r5 = r3 +|+ r4 ;
   /* dual 16-bit operations, add|add */
```

```
r6 = r0 -|+ r1(s) ;
   /* same as above, subtract|add with saturation */
r0 = r2 +|- r1(co) ;
   /* add|subtract with half-word results crossed over in the destination register */
r7 = r3 -|- r6(sco) ;
   /* subtract|subtract with saturation and half-word results crossed over in the
      destination register */
r5 = r3 +|+ r4, r7 = r3-|-r4 ;
   /* quad 16-bit operations, add|add, subtract|subtract */
r5 = r3 +|- r4, r7 = r3 -|+ r4 ;
   /* quad 16-bit operations, add|subtract, subtract|add */
r5 = r3 +|- r4, r7 = r3 -|+ r4(asr) ;
   /* quad 16-bit operations, add|subtract, subtract|add, with all results divided
      by 2 (right shifted 1 place) before storing into destination register */
r5 = r3 +|- r4, r7 = r3 -|+ r4(asl) ;
   /* quad 16-bit operations, add|subtract, subtract|add, with all results
      multiplied by 2 (left shifted 1 place) before storing into destination register dual
*/
```

32-bit Add Constant (AddImm)

General Form

Destructive Binary Operations, dreg with 7bit immediate (CompI2opD)
DREG Register Type += imm7 Register Type

Abstract

This instruction allows the user to add a constant to a register. This instruction does not saturate on overflow

See Also (32-bit Add or Subtract (AddSub32), 32-bit Add and Subtract (AddSub32Dual), 32-Bit Add or Subtract with Carry (AddSubAC0))

AddImm Description

The Add Immediate instruction adds a constant value to a register without saturation.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddImm Example

```
r0 += 40 ; /* increment r0 value by 40 and store in r0 */
```

32-bit Add or Subtract (AddSub32)

General Form

Compute with 3 operands (Comp3op)
DREG Register Type = DREG Register Type + DREG Register Type
DREG Register Type = DREG Register Type - DREG Register Type
ALU Operations (Dsp32Alu)
DREG Register Type = DREG Register Type + DREG Register Type NSAT
DREG Register Type = DREG Register Type - DREG Register Type NSAT

Abstract

This instruction adds or subtracts two signed registers. The ALU does not saturate the result by default.

See Also (32-bit Add and Subtract (AddSub32Dual), 32-Bit Add or Subtract with Carry (AddSubAC0), 32-bit Add Constant (AddImm))

AddSub32 Description

The AddSub32 instruction adds or subtracts two source values and places the result in a destination register with or without result saturation.

AddSub32 accepts any combination of register operands, and places the results in the destination register at the user's discretion.

This instruction is encoded as a ***16-bit instruction*** if the `NSAT` option is omitted. The 16-bit encoded instruction 16-bit instruction takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

When the `NSAT` option is included, the instruction is encoded as a ***32-bit instruction***. The 32-bit encoded instruction can sometimes save execution time (over a 16-bit encoded instruction), because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

In the syntax, where `NSAT` appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSub32 Example

```
r5 = r2 + r1 ;    /* 16-bit instruction length add, no saturation */
r5 = r2 + r1 (ns) ;    /* same result as above, but 32-bit instruction length */
r5 = r2 + r1 (s) ;    /* saturate the result */

r5 = r2 - r1 ;        /* 16-bit instruction length subtract, no saturation */
r5 = r2 - r1 (ns) ;    /* same result as above, but 32-bit instruction length */
r5 = r2 - r1 (s) ;     /* saturate the result */
```

32-bit Add and Subtract (AddSub32Dual)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = DREG Register Type + DREG Register Type, DREG Register Type = DREG Register Type - DREG Register Type SAT

Abstract

This instruction adds and subtracts two signed registers. The ALU does not saturate the result by default.

See Also (32-bit Add or Subtract (AddSub32), 32-Bit Add or Subtract with Carry (AddSubAC0), 32-bit Add Constant (AddImm))

AddSub32Dual Description

The AddSub32Dual instruction simultaneously adds and/or subtracts two pairs of registered numbers. Then, the instruction stores the results of each operation into a separate 32-bit data register with or without result saturation. Each destination register must be unique.

AddSub32Dual accepts any combination of register operands, and places the results in the destination registers at the user's discretion.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

In the syntax, where *`SAT`* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSub32Dual Example

```
r2=r0+r1, r3=r0-r1 ;   /* dual 32-bit operations */
r2=r0+r1, r3=r0-r1 (s) ;   /* dual 32-bit operations with saturation */
```

32-Bit Add or Subtract with Carry (AddSubAC0)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = DREG Register Type + DREG Register Type + ac0 SAT
DREG Register Type = DREG Register Type - DREG Register Type + ac0 - 1 SAT

Abstract

This instruction adds or subtracts two 32-Bit numbers plus a carry bit. This operation is used to to multi-precision addition. Optionally, the user can saturate the result.

See Also (32-bit Add or Subtract (AddSub32), 32-bit Add and Subtract (AddSub32Dual), 32-bit Add Constant (AddImm))

AddSubAC0 Description

The AddSubAC0 instruction adds or subtracts two source values plus a carry bit and places the result in a destination register with or without result saturation.

AddSub32 accepts any combination of register operands, and places the results in the destination register at the user's discretion.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

When the *SAT* option is included, the instruction is encoded as a ***32-bit instruction***. The 32-bit encoded instruction can sometimes save execution time (over a 16-bit encoded instruction), because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

In the syntax, where *SAT* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubAC0 Example

```
r5 = r2 + r1 + ac0;          /* add with carry, no saturation implied */
r5 = r2 + r1 + ac0 (ns) ;    /* same result as above */
r5 = r2 + r1 + ac0 (s) ;     /* saturate the result */

r5 = r2 - r1 + ac0 - 1 ;     /* sub with carry, no saturation implied */
r5 = r2 - r1 + ac0 -1 (ns) ; /* same result as above */
r5 = r2 - r1 + ac0 -1 (s) ;  /* saturate the result */
```

Accumulator Add and Extract (AddAccExt)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = (a0 += a1)
DDST0_HL = (a0 += a1)

Abstract

This instruction adds the two signed accumulators together, then extracts the result to a register.

See Also (Accumulator Add or Subtract (AddSubAcc), Dual Accumulator Add and Subtract to Registers (AddSubAccExt))

AddAccExt Description

The AddAccExt instruction increments the 40-bit A0 accumulator register by A1 with saturation at 40 bits, then extract the result into a 32-bit register with saturation at 32 bits.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	*AV0S*	*AV0*
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddAccExt Example

```
r5 = (a0 += a1) ;
r2.l = (a0 += a1) ;
r5.h = (a0 += a1) ;
```

Accumulator Add or Subtract (AddSubAcc)

General Form

ALU Operations (Dsp32Alu)
a0 += a1
a0 += a1 (w32)
a0 -= a1
a0 -= a1 (w32)

Abstract

This instruction adds or subtracts two signed accumulators. The ALU saturates the result on overflow.

See Also (Accumulator Add and Extract (AddAccExt), Dual Accumulator Add and Subtract to Registers (AddSubAccExt))

AddSubAcc Description

The AddSubAcc instruction adds or subtracts two source values in the accumulator registers and places the result in a destination acculator register with or without result saturation.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

The syntax of this instruction provides optional saturation/sign-extension of the result.

- (W32) - signed saturate the result at 32 bits, sign extended

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	***AV0S***	***AV0***
...	...	AC1	***AC0***	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	***AN***	***AZ***
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubAcc Example

```
a0 += a1 ; /* no saturation */
a0 += a1 (w32) ; /* signed saturate at 32 bits, sign extended */

a0 -= a1 ; /* no saturation */
a0 -= a1 (w32) ; /* signed saturate at 32 bits, sign extended */
```

Dual Accumulator Add and Subtract to Registers (AddSubAccExt)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = a1 + a0, DREG Register Type = a1 - a0 SAT
DREG Register Type = a0 + a1, DREG Register Type = a0 - a1 SAT

Abstract

This instruction adds and subtracts the two accumulators together, then extracts the results.

See Also (Accumulator Add or Subtract (AddSubAcc), Accumulator Add and Extract (AddAccExt))

AddSubAccExt Description

The AddSubAccExt instruction simultaneously adds and subtracts the two 40-bit accumulator registers. Then, the instruction stores the results of each operation into a separate 32-bit data register with or without result saturation. Each destination register must be unique.

AddSubAccExt accepts any combination of register operands, and places the results in the destination registers at the user's discretion.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

In the syntax, where *`SAT`* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	*AC1*	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubAccExt Example

```
r4=a1+a0, r6=a1-a0 ;
   /* dual 40-bit accumulator operations with no saturation, A0 added/subtracted from A1 */
r4=a0+a1, r6=a0-a1(s) ;
   /* dual 40-bit accumulator operations with saturation, A1 subtracted from A0 */
```

32-bit Add then Shift (AddSubShift)

General Form

ALU Binary Operations (ALU2op)
DREG Register Type = (DREG Register Type + DREG Register Type) << 1
DREG Register Type = (DREG Register Type + DREG Register Type) << 2

Abstract

This instruction adds then shifts left one or two places. This instruction always saturates on overflow.

AddSubShift Description

The AddSubShift instruction combines an addition operation with a one- or two-place logical shift left. The left shift accomplishes a x2 (for shift 1) or x4 (for shift 2) multiplication on sign-extended numbers. Saturation is not supported.

This instruction does not intrinsically modify values that are strictly input. However, the destination register (*`DDST`*) serves as both an input operand and the result destination, so the *`DDST`* is intrinsically modified.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubShift Example

```
r3 = (r3+r2)<<1 ;    /* r3 = (r3 + r2) * 2 */
r3 = (r3+r2)<<2 ;    /* r3 = (r3 + r2) * 4 */
```

Bit Operations

These operations provide bitwise shift operations on registers operands:

- Ones Count (Shift_Ones)
- Redundant Sign Bits (Shift_SignBits32)
- Redundant Sign Bits (Shift_SignBitsAcc)
- Bit Mux (BitMux)
- Bit Modify (Shift_BitMod)
- Bit Test (Shift_BitTst)
- Deposit Bits (Shift_Deposit)
- Extract Bits (Shift_Extract)

Ones Count (Shift_Ones)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = ones DREG Register Type

Abstract

This instruction counts the number of 1's in a XOP register.

See Also (Redundant Sign Bits (Shift_SignBits32), Redundant Sign Bits (Shift_SignBitsAcc))

Shift_Ones Description

The Shift_Ones instruction (one's-population count) loads the number of 1's contained in the souce register (*DSRC1*) into the lower half of the destination register (*DDST_L*).

The range of possible values loaded into *DDST_L* is 0 through 32.

The *DDST_L* and *DSRC1* can be the same D-register. Otherwise, the One's-Population Count instruction does not modify the contents of *DSRC1*.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Shift_Ones Example

```
r3.l = ones r7 ;
```

If R7 contains 0xA5A5A5A5, R3.L contains the value 16, or 0x0010.

If R7 contains 0x00000081, R3.L contains the value 2, or 0x0002.

Redundant Sign Bits (Shift_SignBits32)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = signbits DREG Register Type
DREG_L Register Type = signbits DREG_L Register Type
DREG_L Register Type = signbits DREG_H Register Type

Abstract

This instruction returns the number of redundant sign bits. For example, if there are five sign bits, this instruction returns 4. The result can then be used with ASHIFT to normalize the data.

See Also (Ones Count (Shift_Ones), Redundant Sign Bits (Shift_SignBitsAcc))

Shift_SignBits32 Description

The SignBits32 instruction returns the number of sign bits in a number, and can be used in conjunction with a shift to normalize numbers. This instruction can operate on 16-bit or 32-bit input numbers.

- For a 16-bit input, Sign Bit returns the number of leading sign bits minus one, which is in the range 0 through 15. There are no special cases. An input of all zeros returns +15 (all sign bits), and an input of all ones also returns +15.
- For a 32-bit input, Sign Bit returns the number of leading sign bits minus one, which is in the range 0 through 31. An input of all zeros or all ones returns +31 (all sign bits).

The result of the SignBits32 instruction can be used directly as the argument to an arithmetic shift instruction (AShift) to normalize the number. Resultant numbers will be in the following formats (S == signbit, M == magnitude bit).

16-bit:	S.MMM MMMM MMMM MMMM
32-bit:	S.MMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM

In addition, the SignBits32 instruction result can be subtracted directly to form the new exponent.

The SignBits32 instruction does not implicitly modify the input value. For 32-bit and 16-bit input, the destination register (*DDST_L*) and source sample register (*DSRC1*) can be the same D-register. Doing this explicitly modifies the *DSRC1*.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Shift_SignBits32 Example

```
r2.l = signbits r7 ;
r1.l = signbits r5.l ;
r0.l = signbits r4.h ;
```

Redundant Sign Bits (Shift_SignBitsAcc)

General Form

Shift (Dsp32Shf)

DREG_L Register Type = signbits a0
DREG_L Register Type = signbits a1

Abstract

This instruction returns the number of redundant sign bits. For example, if there are five sign bits, this instruction returns 4. The result can then be used with ASHIFT to normalize the data.

See Also (Ones Count (Shift_Ones), Redundant Sign Bits (Shift_SignBits32))

Shift_SignBitsAcc Description

The SignBitsAcc instruction returns the number of sign bits in a number, and can be used in conjunction with a shift to normalize numbers. This instruction can operate on 40-bit input numbers.

- For a 40-bit Accumulator input, Sign Bit returns the number of leading sign bits minus 9, which is in the range -8 through +31. A negative number is returned when the result in the Accumulator has expanded into the extension bits; the corresponding normalization will shift the result down to a 32-bit quantity (losing precision). An input of all zeros or all ones returns +31.

The result of the SignBitsAcc instruction can be used directly as the argument to an arithmetic shift instruction (AShift) to normalize the number. Resultant numbers will be in the following formats (S == signbit, M == magnitude bit).

40-bit:	SSSS SSSS S.MMM MMMM MMMM MMMM MMMM MMMM MMMM MMMM

In addition, the SignBitsAcc instruction result can be subtracted directly to form the new exponent.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Shift_SignBitsAcc Example

```
r6.l = signbits a0 ;
r5.l = signbits a1 ;
```

Bit Mux (BitMux)

General Form

Shift (Dsp32Shf)
bitmux (DREG Register Type, DREG Register Type, a0) (asr)
bitmux (DREG Register Type, DREG Register Type, a0) (asl)

Abstract

This instruction merges two bit streams into the accumulator. Each time you call this instruction, it takes a single bit from the two source registers, muxes them together, and deposits them into the accumulator. The streams can be taken from the MSBs of the register pair and shifted into the LSBs of the accumulator (ASL) or taken from the LSBs of the registers and deposited into the MSBs of the accumulator (ASR).

See Also (Bit Modify (Shift_BitMod), Bit Test (Shift_BitTst))

BitMux Description

The BitMux instruction merges bit streams.

The instruction has two versions, shift right and shift left. This instruction overwrites the contents of source 1 (`DSRC1`) and source 0 (`DSRC0`). See the ***Contents Before Shift*** table, ***A Shift Right Instruction*** table, and ***A Shift Left Instruction*** table.

In the Shift Right version, the processor performs the following sequence.

1. Right shift Accumulator `A0` by one bit. Right shift the LSB of `DSRC1` into the MSB of the Accumulator.
2. Right shift Accumulator `A0` by one bit. Right shift the LSB of `DSRC0` into the MSB of the Accumulator.

In the Shift Left version, the processor performs the following sequence.

1. Left shift Accumulator `A0` by one bit. Left shift the MSB of `DSRC0` into the LSB of the Accumulator.
2. Left shift Accumulator `A0` by one bit. Left shift the MSB of `DSRC1` into the LSB of the Accumulator.

`DSRC1` and `DSRC0` must not be the same D-register.

Table 8-1: Contents Before Shift

IF	39...........32	31...........24	23...........16	15.............8	7...............0
source_1:		`xxxx xxxx`	`xxxx xxxx`	`xxxx xxxx`	`xxxx xxxx`
source_0:		`yyyy yyyy`	`yyyy yyyy`	`yyyy yyyy`	`yyyy yyyy`
Accumulator A0:	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzzz`

Table 8-2: A Shift Right Instruction

IF	39...........32	31...........24	23...........16	15.............8	7...............0
source_1:*1		`0xxx xxxx`	`xxxx xxxx`	`xxxx xxxx`	`xxxx xxxx`
source_0:*2		`0yyy yyyy`	`yyyy yyyy`	`yyyy yyyy`	`yyyy yyyy`
Accumulator A0:*3	`yxzz zzzz`	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzzz`

*1 source_1 is shifted right 1 place

*2 source_0 is shifted right 1 place

*3 Accumulator A0 is shifted right 2 places

Table 8-3: A Shift Left Instruction

IF	39...........32	31...........24	23...........16	15.............8	7................0
source_1:[*1]		`xxxx xxxx`	`xxxx xxxx`	`xxxx xxxx`	`xxxx xxx0`
source_0:[*2]		`yyyy yyyy`	`yyyy yyyy`	`yyyy yyyy`	`yyyy yyy0`
Accumulator A0:[*3]	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzzz`	`zzzz zzyx`

*1 source_1 is shifted left 1 place

*2 source_0 is shifted left 1 place

*3 Accumulator A0 is shifted left 2 places

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

BitMux Example

```
bitmux (r2, r3, a0) (asr) ;   /* right shift*/
```

- If
  - `R2=0b1010 0101 1010 0101 1100 0011 1010 1010`
  - `R3=0b1100 0011 1010 1010 1010 0101 1010 0101`
  - `A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111`

  then the Shift Right instruction produces:
  - `R2=0b0101 0010 1101 0010 1110 0001 1101 0101`
  - `R3=0b0110 0001 1101 0101 0101 0010 1101 0010`
  - `A0=0b1000 0000 0000 0000 0000 0000 0000 0000 0000 0001`

```
bitmux (r3, r2, a0) (asl) ;   /* left shift*/
```

- If
  - `R3=0b1010 0101 1010 0101 1100 0011 1010 1010`
  - `R2=0b1100 0011 1010 1010 1010 0101 1010 0101`
  - `A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0000 0111`

  then the Shift Left instruction produces:
  - `R2=0b1000 0111 0101 0101 0100 1011 0100 1010`
  - `R3=0b0100 1011 0100 1011 1000 0111 0101 0100`

- `A0=0b0000 0000 0000 0000 0000 0000 0000 0000 0001 1111`

Bit Modify (Shift_BitMod)

General Form

Logic Binary Operations (Logi2Op)
bitset (DREG Register Type, uimm5 Register Type)
bittgl (DREG Register Type, uimm5 Register Type)
bitclr (DREG Register Type, uimm5 Register Type)

Abstract

This instruction takes the data register specified and clears, sets, or toggles a bit.

See Also (Bit Mux (BitMux), Bit Test (Shift_BitTst))

Shift_BitMod Description

The BitMod instruction includes BitSet, BitTgl, and BitTst forms:

- The BitSet (bit set) instruction sets the bit designated by the bit position (source immediate value, *`SRCI`*) in the specified D-register destination (*`DDST`*). It does not affect other bits in the D-register.

  The *`SRCI`* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

- The BitTgl (bit toggle) instruction inverts the bit designated by *`SRCI`* in the specified D-register. The instruction does not affect other bits in the D-register.

  The *`SRCI`* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

- The BitClr (bit clear) instruction clears the bit designated by *`SRCI`* in the specified D-register. It does not affect other bits in that register.

  The *`SRCI`* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

This ***16-bit instruction*** instruction takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode.***

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_BitMod Example

BitSet Example

```
bitset (r2, 7) ;   /* set bit 7 (the eighth bit from LSB) in R2 */
```

For example, if `R2` contains 0x00000000 before this instruction, it contains 0x00000080 after the instruction.

BitTgl Example

```
bittgl (r2, 24) ;   /* toggle bit 24 (the 25th bit from LSB in R2 */
```

For example, if `R2` contains 0xF1FFFFFF before this instruction, it contains 0xF0FFFFFF after the instruction. Executing the instruction a second time causes the register to contain 0xF1FFFFFF.

BitClr Example

```
bitclr (r2, 3) ;   /* clear bit 3 (the fourth bit from LSB) in R2 */
```

For example, if `R2` contains 0xFFFFFFFF before this instruction, it contains 0xFFFFFFF7 after the instruction.

Bit Test (Shift_BitTst)

General Form

Logic Binary Operations (Logi2Op)
cc = !bittst (DREG Register Type, uimm5 Register Type)
cc = bittst (DREG Register Type, uimm5 Register Type)

Abstract

This instruction sets CC bits if the specified condition is true. In the bittst case, the CC bit is set if the specified bit is a 1. For the !bittst case, it is set if the bit is a zero.

See Also (Bit Mux (BitMux), Bit Modify (Shift_BitMod))

Shift_BitTst Description

The Bit Test instruction sets or clears the `CC` bit, based on the bit designated by the bit position (source immediate value, *`SRCI`*) in the specified D-register destination (*`DDST`*).

One version tests whether the specified bit is set; the other tests whether the bit is clear. The instruction does not affect other bits in the D-register.

The *SRCI* range of values is 0 through 31, where 0 indicates the LSB, and 31 indicates the MSB of the 32-bit D-register.

This ***16-bit instruction*** instruction takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_BitTst Example

```
cc = bittst (r7, 15) ;   /* test bit 15 TRUE in R7 */
```

For example, if `R7` contains 0xFFFFFFFF before this instruction, `CC` is set to 1, and `R7` still contains 0xFFFFFFFF after the instruction.

```
cc = ! bittst (r3, 0) ;   /* test bit 0 FALSE in R3 */
```

If `R3` contains 0xFFFFFFFF, this instruction clears `CC` to 0.

Deposit Bits (Shift_Deposit)

General Form

Shift (Dsp32Shf)
DREG Register Type = deposit (DREG Register Type, DREG Register Type)
DREG Register Type = deposit (DREG Register Type, DREG Register Type) (x)

Abstract

The bit field deposit instruction merges the background data in SRC1 with a foreground bit field in SRC0.h and saves the result into DEST.

See Also (Extract Bits (Shift_Extract))

Shift_Deposit Description

The Bit Field Deposit instruction merges the background data in SRC1 with a foreground bit field in SRC0.h and saves the result into DEST. The length of the bit field is stored in SRC0.b0 and the position of the field is stored in SRC0.b1. This takes the lower SRC0.b0 bits from SRC0.h and deposits them into SRC1 at bit SRC0.b1. The (X) syntax sign-extends the field. If you are not sign extending the bits above the inserted field are unchanged.

```
Field deposit
       31               16 15     8 7      0
      +----------------+--------+--------+
src0: |xxxxxxxxxxxxxSNN |    P   |    L   |   L--length, P--position
      +----------------+--------+--------+
      +----------------+-----------------+
src1  |bbbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbbb|
      +----------------+-----------------+

      +----------------+-----------------+
dst0: |bbbbbbbbbbbbbbSNN bbbbbbbbbbbbbbbbb|  SN--inserted field in src0
      +----------------+-----------------+   b--previous contents of src1

with sign extension
      +----------------+-----------------+
dst0: |SSSSSSSSSSSSSSSNN bbbbbbbbbbbbbbbbb|
      +----------------+-----------------+
```

The Bit Field Deposit instruction merges the background bit field in the background register (`DSRC1`) with the foreground bit field in the upper half of the foreground register (`DSRC0`) and saves the result into the destination register (`DDST`). The user determines the length of the foreground bit field and its position in the background field.

The input register bit field definitions appear in the ***Input Register Bit Field Definitions*** table.

Table 8-4: Input Register Bit Field Definitions

	31...............24	23...............16	15.................8	7...................0
DSRC1[*1]	bbbb bbbb	bbbb bbbb	bbbb bbbb	bbbb bbbb
DSRC0[*2]	nnnn nnnn	nnnn nnnn	xxxp pppp	xxxL LLLL

*1 where b = background bit field (32 bits)

*2 where:

- - n = foreground bit field (16 bits); the L field determines the actual number of foreground bits used.
- - p = intended position of foreground bit field LSB in dest_reg (valid range 0 through 31)
- - L = length of foreground bit field (valid range 0 through 16)

The operation writes the foreground bit field of length *L* over the background bit field with the foreground LSB located at bit *p* of the background.

There are a number of *boundary cases* related to Shift_Deposit instruction operation that should be considered.

- Unsigned syntax, L = 0: The architecture copies `DSRC1` contents without modification into `DDST`. By definition, a foreground of zero length is transparent.
- Sign-extended, L = 0 and p = 0: This case loads 0x0000 0000 into `DDST`. The sign of a zero length, zero position foreground is zero; therefore, sign-extended is all zeros.
- Sign-extended, L = 0 and p = 0: The architecture copies the lower order bits of `DSRC1` below position *p* into `DDST`, then sign-extends that number. The foreground value has no effect. For instance, if:
  - `DSRC1` = 0x0000 8123,
  - L = 0, and
  - p = 16,
  - then:
  - `DDST` = 0xFFFF 8123.

  In this example, the architecture copies bits 15-0 from `DSRC1` into `DDST`, then sign-extends that number.
- Sign-extended, (L + p) > 32: Any foreground bits that fall outside the range 31-0 are truncated.

The Bit Field Deposit instruction does not modify the contents of the two source registers. One of the source registers can also serve as `DDST`.

The `(X)` option syntax sign-extends the deposited bit field. If you specify the sign-extended syntax, the operation does not affect the `DDST` bits that are less significant than the deposited bit field.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_Deposit Example

Bit Field Deposit Unsigned

```
r7 = deposit (r4, r3) ;
```

- If
    - `R4=0b1111 1111 1111 1111 1111 1111 1111 1111` where this is the background bit field
    - `R3=0b0000 0000 0000 0 000 0000 0111 0000 0011` where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:

    - `R7=0b1111 1111 1111 1111 1111 11 00 0 111 1111`
- If
    - `R4=0b1111 1111 1111 1111 1111 1111 1111 1111` where this is the background bit field
    - `R3=0b0000 000 0 1111 1010 0000 1101 0000 1001` where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:

    - `R7=0b1111 1111 11 01 1111 010 1 1111 1111 1111`

Bit Field Deposit Sign-Extended

```
r7 = deposit (r4, r3) (x) ;   /* sign-extended*/
```

- If
    - `R4=0b1111 1111 1111 1111 1111 1111 1111 1111` where this is the background bit field
    - `R3=0b0101 1010 0101 1 010 0000 0111 0000 0011` where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:

    - `R7=0b 0000 0000 0000 0000 0000 0001 0 111 1111`
- If
    - `R4=0b1111 1111 1111 1111 1111 1111 1111 1111` where this is the background bit field
    - `R3=0b0000 100 1 1010 1100 0000 1101 0000 1001` where bits 31-16 are the foreground bit field, bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Deposit (unsigned) instruction produces:

    - `R7=0b 1111 1111 1111 0101 100 1 1111 1111 1111`

Extract Bits (Shift_Extract)

General Form

Shift (Dsp32Shf)
DREG Register Type = extract (DREG Register Type, DREG_L Register Type) (z)
DREG Register Type = extract (DREG Register Type, DREG_L Register Type) (x)

Abstract

This instruction extracts specified bits from the SRC1 register and writes them to the low order bits of the destination register.

See Also (Deposit Bits (Shift_Deposit))

Shift_Extract Description

Extracts specified bits from the SRC1 register and writes them to the low order bits of the destination register. The bit position is stored in SRC0.b1 and the length is stored in SRC0.b0. The field is either sign extended or zero extended to fill the 32-bit output register. ((Z) zero fills, (X) sign extends))

```
Field extraction
      31     16 15 8 7   0
      +--------+----+-----+
src0: |xxxxxxxx|  P |  L |   L--length, P--position
      +--------+----+-----+
      +--------+---------+
src1  |bbbbbbbbbSNNbbbbb|
      +--------+---------+

      +--------+---------+
dst0: |000000000000000SNN|  SN--inserted field in hi half of src0
      +--------+---------+   b--previous contents of src1
x--unused
with sign extension          0--zero
      +--------+---------+
dst0: |SSSSSSSSSSSSSSSSNN|
      +--------+---------+
```

The Bit Field Extraction instruction moves only specific bits from the scene register (`DSRC1`) into the low-order bits of the destination register (`DDST`). The user determines the length of the pattern bit field and its position in the scene field using the pattern register (`DSRC0_L`).

The input register bit field definitions appear in the ***Input Register Bit Field Definitions*** table.

Table 8-5: Input Register Bit Field Definitions

	31................24	23................16	15..................8	7....................0
DSRC1:[*1]	ssss ssss	ssss ssss	ssss ssss	ssss ssss

Table 8-5: Input Register Bit Field Definitions (Continued)

`DSRC0_L:`[*2]			`xxxp pppp`	`xxxL LLLL`

*1 The *s* characters indicate the scene bit field (32 bits).

*2 The *p* characters indicate the position of pattern bit field LSB in scene_reg (valid range 0 through 31). The *L* characters indicate the length of pattern bit field (valid range 0 through 31).

The operation reads the pattern bit field of length *L* from the scene bit field, with the pattern LSB located at bit *p* of the scene. See "Example", below, for more.

There are a number of *boundary cases* related to Shift_Extract instruction operation that should be considered.

If (p + L) > 32: In the zero-extended and sign-extended versions of the instruction, the architecture assumes that all bits to the left of the `DSRC1` are zero. In such a case, the user is trying to access more bits than the register actually contains. Consequently, the architecture fills any undefined bits beyond the MSB of the `DSRC1` with zeros.

The Bit Field Extraction instruction does not modify the contents of the two source registers. One of the source registers can also serve as `DDST`.

The user has the choice of using the `(X)` option syntax to perform sign-extend extraction or the `(Z)` option syntax to perform zero-extend extraction.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_Extract Example

Bit Field Extraction Unsigned

```
r7 = extract (r4, r3.l) (z) ;   /* zero-extended*/
```

- If

- `R4=0b1010 0101 1010 0101 1100 0 011 1 010 1010` where this is the scene bit field
- `R3=0bxxxx xxxx xxxx xxxx 0000 0111 0000 0100` where bits 15-8 are the position, and bits 7-0 are the length

then the Bit Field Extraction (unsigned) instruction produces:

- `R7=0b0000 0000 0000 0000 0000 0000 0000 0111`

- If
  - `R4=0b1010 0101 10 10 0101 110 0 0011 1010 1010` where this is the scene bit field
  - `R3=0bxxxx xxxx xxxx xxxx 0000 1101 0000 1001` where bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Extraction (unsigned) instruction produces:

  - `R7=0b0000 0000 0000 0000 0000 000 1 0010 1110`

Bit Field Extraction Sign-Extended

```
r7 = extract (r4, r3.l) (x) ;   /* sign-extended*/
```

- If
  - `R4=0b1010 0101 1010 0101 1100 0 011 1 010 1010` where this is the scene bit field
  - `R3=0bxxxx xxxx xxxx xxxx 0000 0111 0000 0100` where bits 15-8 are the position, and bits 7-0 are the length

  then the Bit Field Extraction (sign-extended) instruction produces:

  - `R7=0b0000 0000 0000 0000 0000 0000 0000 0111`
- If
  - `R4=0b1010 0101 10 10 0101 110 0 0011 1010 1010` where this is the scene bit field
  - `R3=0bxxxx xxxx xxxx xxxx 0000 1101 0000 1001` where bits 15-8 are the position, and bits 7-0 are the length

  Then the Bit Field Extraction (sign-extended) instruction produces:

  - `R7=0b1111 1111 1111 1111 1111 111 1 0010 1110`

Comparison Operations

These operations provide 16- and 32-bit maximum/minimum comparison and array search operations on register operands:

- Vectored 16-Bit Maximum (Max16Vec)

- Vectored 16-Bit Minimum (Min16Vec)
- 32-bit Maximum (Max32)
- 32-Bit Minimum (Min32)
- Vectored 16-Bit Search (Search)

Vectored 16-Bit Maximum (Max16Vec)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = max(DREG Register Type, DREG Register Type) (v)

Abstract

This instruction calculates the maximum of one or two pairs of signed 16-Bit words.

See Also (Vectored 16-Bit Minimum (Min16Vec))

Max16Vec Description

The vector maximum instruction returns the maximum value (meaning the largest positive value, nearest to 0x7FFF) of the 16-bit half-word source registers to the `dest_reg`.

The instruction compares the upper half-words of `src_reg_0` and `src_reg_1` and returns that maximum to the upper half-word of dest_reg. It also compares the lower half-words of `src_reg_0` and `src_reg_1` and returns that maximum to the lower half-word of `dest_reg`. The result is a concatenation of the two 16-bit maximum values.

The vector maximum instruction does not implicitly modify input values. The `dest_reg` can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Max16Vec Example

```
r7 = max (r1, r0) (v) ;
```

- Assume `R1` = 0x0007 0000 and `R0` = 0x0000 000F, then `R7` = 0x0007 000F.
- Assume `R1` = 0xFFF7 8000 and `R0` = 0x000A 7FFF, then `R7` = 0x000A 7FFF.
- Assume `R1` = 0x1234 5678 and `R0` = 0x0000 000F, then `R7` = 0x1234 5678.

Vectored 16-Bit Minimum (Min16Vec)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = min(DREG Register Type, DREG Register Type) (v)

Abstract

This instruction calculates the minimum of two pairs of signed word vectors.

See Also (Vectored 16-Bit Maximum (Max16Vec))

Min16Vec Description

The Vector Minimum instruction returns the minimum value (the most negative value or the value closest to 0x8000) of the 16-bit half-word source registers to the *`dest_reg`*.

This instruction compares the upper half-words of *`src_reg_0`* and *`src_reg_1`* and returns that minimum to the upper half-word of *`dest_reg`*. It also compares the lower half-words of *`src_reg_0`* and *`src_reg_1`* and returns that minimum to the lower half-word of *`dest_reg`*. The result is a concatenation of the two 16-bit minimum values.

The input values are not implicitly modified by this instruction. The *`dest_reg`* can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0

...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Min16Vec Example

```
r7 = min (r1, r0) (v) ;
```

- Assume `R1` = 0x0007 0000 and `R0` = 0x0000 000F, then `R7` = 0x0000 0000.
- Assume `R1` = 0xFFF7 8000 and `R0` = 0x000A 7FFF, then `R7` = 0xFFF7 8000.
- Assume `R1` = 0x1234 5678 and `R0` = 0x0000 000F, then `R7` = 0x0000 000F.

32-bit Maximum (Max32)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = max(DREG Register Type, DREG Register Type)

Abstract

This instruction calculates the maximum of two signed 32-bit values.

See Also (32-Bit Minimum (Min32))

Max32 Description

The maximum instruction returns the maximum, or most positive, value of the source registers. The operation subtracts *src_reg_1* from *src_reg_0* and selects the output based on the signs of the input values and the arithmetic status bits.

The maximum instruction does not implicitly modify input values. The *dest_reg* can be the same D-register as one of the source registers. Doing this explicitly modifies the source register.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode.***

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0

...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Max32 Example

```
r5 = max (r2, r3) ;
```

- Assume `R2` = 0x00000000 and `R3` = 0x0000000F, then `R5` = 0x0000000F.
- Assume `R2` = 0x80000000 and `R3` = 0x0000000F, then `R5` = 0x0000000F.
- Assume `R2` = 0xFFFFFFFF and `R3` = 0x0000000F, then `R5` = 0x0000000F.

32-Bit Minimum (Min32)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = min(DREG Register Type, DREG Register Type)

Abstract

This instruction calculates the minimum of two signed 32-bit values.

See Also (32-bit Maximum (Max32))

Min32 Description

The minimum instruction returns the minimum value of the source registers to the dest_reg. (The minimum value of the source registers is the value closest to –∞.) The operation subtracts `src_reg_1` from `src_reg_0` and selects the output based on the signs of the input values and the arithmetic status bits.

The minimum instruction does not implicitly modify input values. The `dest_reg` can be the same D-register as one of the source registers. Doing this explicitly modifies the source register.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode.***

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0

...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Min32 Example

```
r5 = min (r2, r3) ;
```

- Assume `R2` = 0x00000000 and `R3` = 0x0000000F, then `R5` = 0x00000000.
- Assume `R2` = 0x80000000 and `R3` = 0x0000000F, then `R5` = 0x80000000.
- Assume `R2` = 0xFFFFFFFF and `R3` = 0x0000000F, then `R5` = 0xFFFFFFFF.

Vectored 16-Bit Search (Search)

General Form

ALU Operations (Dsp32Alu)
(DREG Register Type, DREG Register Type) = search DREG Register Type (gt)
(DREG Register Type, DREG Register Type) = search DREG Register Type (ge)
(DREG Register Type, DREG Register Type) = search DREG Register Type (lt)
(DREG Register Type, DREG Register Type) = search DREG Register Type (le)

Abstract

This instruction is used in a loop to locate a minimum or maximum in an array. For each compute unit, a value is compared against the current signed max or min in the accumulator. Two values are tested at a time, the current winner will be stored in the accumulator and the current value of P0 will be written to the result register if the comparison is true.

Search Description

This instruction is used in a loop to locate a maximum or minimum element in an array of 16-bit packed data. Two values are tested at a time. The vector search instruction compares two 16-bit, signed half-words to values stored in the Accumulators. Then, it conditionally updates each accumulator and destination pointer based on the comparison. Pointer register P0 is always the implied array pointer for the elements being searched.

More specifically, the signed high half-word of `src_reg` is compared in magnitude with the 16 low-order bits in A1. If `src_reg_hi` meets the comparison criterion, then A1 is updated with `src_reg_hi`, and the value in pointer register P0 is stored in `dest_pointer_hi`. The same operation is performed for `src_reg_low` and A0.

Based on the search mode specified in the syntax, the instruction tests for maximum or minimum signed values.

Values are sign extended when copied into the accumulator(s). See the examples for one way to implement the search loop. After the vector search loop concludes, A1 and A0 hold the two surviving elements, and `dest_pointer_hi` and `dest_pointer_lo` contain their respective addresses. The next step is to select the final value from these two surviving elements.

Modes

The four supported compare modes are specified by the mandatory searchmode flag.

Table 8-6: Compare Modes

Mode	Description
`(GT)`	Greater than. Find the location of the first maximum number in an array.
`(GE)`	Greater than or equal. Find the location of the last maximum number in an array.
`(LT)`	Less than. Find the location of the first minimum number in an array.
`(LE)`	Less than or equal. Find the location of the last minimum number in an array.

Summary (assumed Pointer P0)

`src_reg_hi`

Compared to least significant 16 bits of A1. If compare condition is met, overwrites lower 16 bits of A1 and copies P0 into `dest_pointer_hi`.

`src_reg_lo`

Compared to least significant 16 bits of A0. If compare condition is met, overwrites lower 16 bits of A0 and copies P0 into `dest_pointer_lo`.

This ***32-bit instruction*** can be issued in parallel with the combination of one 16-bit length load instruction to the P0 register and one 16-bit NOP. No other instructions can be issued in parallel with the vector search instruction. Note the following legal and illegal forms.

```
(r1, r0) = search r2 (LT) || r2 = [p0++p3]; /* ILLEGAL */
(r1, r0) = search r2 (LT) || r2 = [p0++]; /* LEGAL */
(r1, r0) = search r2 (LT) || r2 = [p0++]; /* LEGAL */
```

Search Example

```
/* Initialize Accumulators with appropriate value for the type of search. */
  r0.l=0x7fff ;
  r0.h=0 ;
  a0=r0 ; /* max positive 16-bit value */
  a1=r0 ; /* max positive 16-bit value */
/* Initialize R2. */
  r2=[p0++] ;
/* Assume P1 is initialized to the size of the vector length. */
  LSETUP (loop_, loop_) LC0=P1>>1 ; /* set up the loop */
  loop_: (r1,r0) = SEARCH R2 (LE) || R2=[P0++];
    /* search for the last minimum in all but the
    last element of the array */
    (r1,r0) = SEARCH R2 (LE);
    /* finally, search the last element */
/* The lower 16 bits of A1 and A0 contain the last minimums of the
array. R1 contains the value of P0 corresponding to the value in
A1. R0 contains the value of P0 corresponding to the value in A0.
Next, compare A1 and A0 together and R1 and R0 together to find
the single, last minimum in the array.
Note: In this example, the resulting pointers are past the actual
surviving array element due to the post-increment operation. */
  cc = a0 <= a1 ;
  r0 += -4 ;
  r1 += -2 ;
  if !cc r0 = r1 ;
/* the pointer to the survivor is in r0 */
```

Conversion Operations

These operations provide absolute value, negate, pass, and saturate operations on register operands:

- Vectored 16-Bit Absolute Value (Abs2x16)
- 32-bit Absolute Value (Abs32)
- Accumulator0 Absolute Value (AbsAcc0)
- Accumulator Absolute Value (AbsAcc1)
- Accumulator Absolute Value (AbsAccDual)
- Vectored 16-bit Negate (Neg16Vec)
- 32-Bit Negate (Neg32)
- Accumulator0 Negate (NegAcc0)
- Accumulator1 Negate (NegAcc1)
- Dual Accumulator Negate (NegAccDual)
- Fractional 32-bit to 16-Bit Conversion (Pass32Rnd16)
- Accumulator0 32-Bit Saturate (ALU_SatAcc0)
- Accumulator1 32-Bit Saturate (ALU_SatAcc1)
- Dual Accumulator 32-Bit Saturate (ALU_SatAccDual)

Vectored 16-Bit Absolute Value (Abs2x16)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = abs DREG Register Type (v)

Abstract

This instruction calculates the absolute value of the signed 16-bit input vector. Saturation only applies when the input is 0x8000.

Abs2x16 Description

The vector absolute value instruction calculates the individual absolute values of the upper and lower halves of a single 32-bit data register. The results are placed into a 32-bit `dest_reg`, using the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination.

This instruction saturates the result.

For example, as shown in the figure, if the source register contains the data shown, the destination register receives the data shown.

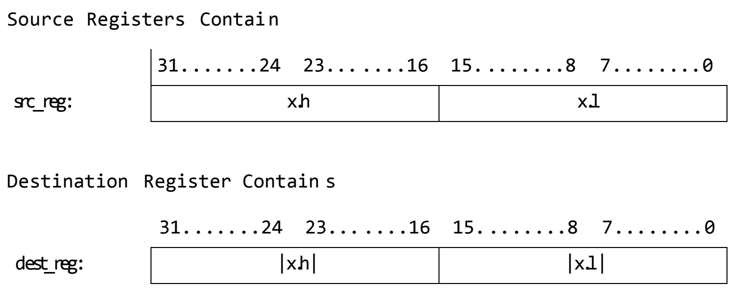


Figure 8-3: Source/Destination Value Placement

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Abs2x16 Example

```
/* If r1 = 0xFFFF 7FFF, then . . . */
r3 = abs r1 (v) ;
/* . . . produces 0x0001 7FFF */
```

32-bit Absolute Value (Abs32)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = abs DREG Register Type

Abstract

This instruction calculates the absolute value of the 32-bit input. Saturation only applies when the input is 0x80000000.

Abs32 Description

This instruction calculates the absolute value of a 32-bit register and stores it into a 32-bit `dest_reg`. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Abs32 Example

```
r3 = abs r1 ;
```

Accumulator0 Absolute Value (AbsAcc0)

General Form

ALU Operations (Dsp32Alu)
a0 = abs a0
a0 = abs a1

Abstract

This instruction calculates the absolute value of the A0 (accumulator 0) register.

See Also (Accumulator Absolute Value (AbsAcc1), Accumulator Absolute Value (AbsAccDual))

AbsAcc0 Description

This instruction takes the absolute value of a 40-bit input value in a register and produces a 40-bit result. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AbsAcc0 Example

```
a0 = abs a0 ;
```

```
a0 = abs a1 ;
```

Accumulator Absolute Value (AbsAcc1)

General Form

ALU Operations (Dsp32Alu)
a1 = abs a0
a1 = abs a1

Abstract

This instruction calculates the absolute value of the A1 (accumulator 1) register.

See Also (Accumulator0 Absolute Value (AbsAcc0), Accumulator Absolute Value (AbsAccDual))

AbsAcc1 Description

This instruction takes the absolute value of a 40-bit input value in a register and produces a 40-bit result. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AbsAcc1 Example

```
a1 = abs a0 ;
a1 = abs a1 ;
```

Accumulator Absolute Value (AbsAccDual)

General Form

ALU Operations (Dsp32Alu)
a1 = abs a1, a0 = abs a0

Abstract

This instruction calculates the absolute value of the A0 (accumulator 0) register and A1 (accumulator 1) register.

See Also (Accumulator0 Absolute Value (AbsAcc0), Accumulator Absolute Value (AbsAcc1))

AbsAccDual Description

This instruction performs the ABS operation on both accumulators by a single instruction, taking the absolute value of 40-bit input values in two registers and producing two 40-bit results. Calculation is done according to the following rules.

- If the input value is positive or zero, copy it unmodified to the destination.
- If the input value is negative, subtract it from zero and store the result in the destination. Saturation is automatically performed with the instruction, so taking the absolute value of the largest- magnitude negative number returns the largest-magnitude positive number.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AbsAccDual Example

```
a1 = abs a1, a0=abs a0 ;
```

Vectored 16-bit Negate (Neg16Vec)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = -DREG Register Type (v)

Abstract

This instruction negates the input operands. The maximum negative inputs (0x8000) saturates to maximum positive.

Neg16Vec Description

The vector negate instruction returns the same magnitude with the opposite arithmetic sign, saturated for each 16-bit half-word in the source. The instruction calculates by subtracting the source from zero.

For more information, see the Saturation section in the Introduction chapter.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Neg16Vec Example

```
r5 =-r3 (v) ;
/* R5.H becomes the negative of R3.H and R5.L becomes the negative of R3.L.  */
/* If r3 = 0x0004 7FFF the result is r5 = 0xFFFC 8001 */
```

32-Bit Negate (Neg32)

General Form

ALU Binary Operations (ALU2op)

DREG Register Type = -DREG Register Type
ALU Operations (Dsp32Alu)
DREG Register Type = -DREG Register Type NSAT

Abstract

This instruction negates the input operands. If saturation is specified, the special case of negate (`MAX_NEG_32`) returns `MAX_POS_32`. If not, it returns `MAX_NEG_32`.

Neg32 Description

The negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign. The instruction calculates by subtracting from zero.

The Dreg version of the negate (two's-complement) instruction is offered with or without saturation. The only case where the nonsaturating negate would overflow is when the input value is 0x8000 0000. The saturating version returns 0x7FFF FFFF; the nonsaturating version returns 0x8000 0000.

In the syntax, where *`NSAT`* appears, substitute a saturation option (s or ns). See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Neg32 Example

```
r5 =-r0 ; /* default is no saturation */
r5 =-r0 (s) ; /* saturation */
r5 =-r0 (ns) ; /* no saturation */
```

Accumulator0 Negate (NegAcc0)

General Form

ALU Operations (Dsp32Alu)
a0 = -a0
a0 = -a1

Abstract

This instruction negates the input operands.

See Also (Accumulator1 Negate (NegAcc1), Dual Accumulator Negate (NegAccDual))

NegAcc0 Description

The negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign. The accumulator versions saturate the result at 40 bits. The instruction calculates by subtracting from zero.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	***AV0S***	***AV0***
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	***AN***	***AZ***
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NegAcc0 Example

```
a0 =-a0 ;
a0 =-a1 ;
```

Accumulator1 Negate (NegAcc1)

General Form

ALU Operations (Dsp32Alu)
a1 = -a0

a1 = -a1

Abstract

This instruction negates the input operands.

See Also (Accumulator0 Negate (NegAcc0), Dual Accumulator Negate (NegAccDual))

NegAcc1 Description

The negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign. The accumulator versions saturate the result at 40 bits. The instruction calculates by subtracting from zero.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NegAcc1 Example

```
a1 =-a0 ;
a1 =-a1 ;
```

Dual Accumulator Negate (NegAccDual)

General Form

ALU Operations (Dsp32Alu)
a1 = -a1, a0 = -a0

Abstract

This instruction negates the input operands.

See Also (Accumulator0 Negate (NegAcc0), Accumulator1 Negate (NegAcc1))

NegAccDual Description

The dual negate (two's-complement) instruction returns the same magnitude with the opposite arithmetic sign for each accumulator. The accumulator versions saturate the result at 40 bits. The instruction calculates by subtracting from zero.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NegAccDual Example

```
a1 =-a1, a0=-a0 ;
```

Fractional 32-bit to 16-Bit Conversion (Pass32Rnd16)

General Form

ALU Operations (Dsp32Alu)
DDST0_HL = DREG Register Type (rnd)

Abstract

This instruction converts a 32-bit, normalized-fraction number into a 16-Bit normalized-fraction number by adding a round bit at bit 15, then saturating and extracting bits 31-16, then discarding bits 15-0. The instruction supports only biased rounding, which adds a half LSB (bit 15) before truncating bits 15-0. The `RND_MOD` bit in the `ASTAT` register has no bearing on the rounding behavior of this instruction.

Pass32Rnd16 Description

The round to half-word instruction rounds a 32-bit, normalized-fraction number into a 16-bit, normalized-fraction number by extracting and saturating bits 31–16, then discarding bits 15–0. The instruction supports only biased rounding, which adds a half LSB (in this case, bit 15) before truncating bits 15–0. The ALU performs the rounding. The `RND_MOD` bit in the `ASTAT` register has no bearing on the rounding behavior of this instruction.

Fractional data types such as the operands used in this instruction are always signed.

For more information, see the Saturation section and the Rounding and Truncation section in the Introduction chapter.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode.***

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Pass32Rnd16 Example

```
   /* If r6 = 0xFFFC FFFF, then rounding to 16-bits with . . . */
r1.l = r6 (rnd) ;
   /* . . . produces r1.l = 0xFFFD */
   /* If r7 = 0x0001 8000, then rounding . . . */
r1.h = r7 (rnd) ;
   /* . . . produces r1.h = 0x0002 */
```

Accumulator0 32-Bit Saturate (ALU_SatAcc0)

General Form

ALU Operations (Dsp32Alu)
a0 = a0 (s)

Abstract

This instruciton saturates the accumulator at 32-bits (a0.w). The resulting saturated value is sign extended into the accumulator extension bits (a0.x).

See Also (Accumulator1 32-Bit Saturate (ALU_SatAcc1), Dual Accumulator 32-Bit Saturate (ALU_SatAccDual))

ALU_SatAcc0 Description

The saturate instruction saturates the 40-bit Accumulators at 32 bits. The resulting saturated value is sign extended into the Accumulator extension bits.

For more information, see the Saturation section in the Introduction chapter.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either *User or Supervisor mode*.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	***AV0S***	***AV0***
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	***AN***	***AZ***
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ALU_SatAcc0 Example

```
a0 = a0 (s) ;
```

Accumulator1 32-Bit Saturate (ALU_SatAcc1)

General Form

ALU Operations (Dsp32Alu)
a1 = a1 (s)

Abstract

This instruction saturates the accumulator at 32-bits (a1.w). The resulting saturated value is sign extended into the accumulator extension bits (a1.x).

See Also (Accumulator0 32-Bit Saturate (ALU_SatAcc0), Dual Accumulator 32-Bit Saturate (ALU_SatAccDual))

ALU_SatAcc1 Description

The saturate instruction saturates the 40-bit Accumulators at 32 bits. The resulting saturated value is sign extended into the Accumulator extension bits.

For more information, see the Saturation section in the Introduction chapter.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either *User or Supervisor mode*.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ALU_SatAcc1 Example

```
a1 = a1 (s) ;
```

Dual Accumulator 32-Bit Saturate (ALU_SatAccDual)

General Form

ALU Operations (Dsp32Alu)
a1 = a1 (s), a0 = a0 (s)

Abstract

This instruction saturates the accumulator at 32-bits (a1.w). The resulting saturated value is sign extended into the accumulator extension bits (a1.x).

See Also (Accumulator0 32-Bit Saturate (ALU_SatAcc0), Accumulator1 32-Bit Saturate (ALU_SatAcc1))

ALU_SatAccDual Description

The dual saturate instruction saturates the 40-bit Accumulators at 32 bits. The resulting saturated values are sign extended into the Accumulators extension bits.

For more information, see the Saturation section in the Introduction chapter.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode.***

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*

...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ALU_SatAccDual Example

```
a1 = a1 (s), a0 = a0 (s) ;
```

Logic Operations

These operations provide one's complement and other logic operations on register operands:

- 32-Bit One's Complement (Not32)
- 32-Bit Logic Operations (Logic32)

32-Bit Logic Operations (Logic32)

General Form

Compute with 3 operands (Comp3op)
DREG Register Type = DREG Register Type & DREG Register Type
DREG Register Type = DREG Register Type | DREG Register Type
DREG Register Type = DREG Register Type ^ DREG Register Type

Abstract

This instruction performs logic operations on two 32-bit values. It does either an AND, OR, or XOR.

See Also (32-Bit One's Complement (Not32))

Logic32 Description

The AND instruction performs a 32-bit, bit-wise logical AND operation on the two source registers and stores the results into the `dest_reg`. The instruction does not implicitly modify the source registers. The `dest_reg` and one `src_reg` can be the same D-register; this operation explicitly modifies the `src_reg`.

The OR instruction performs a 32-bit, bit-wise logical OR operation on the two source registers and stores the results into the `dest_reg`. The instruction does not implicitly modify the source registers. The `dest_reg` and one `src_reg` can be the same D-register; this operation explicitly modifies the `src_reg`.

The Exclusive-OR (XOR) instruction performs a 32-bit, bit-wise logical exclusive OR operation on the two source registers and loads the results into the `dest_reg`.

The XOR instruction does not implicitly modify source registers. The `dest_reg` and one `src_reg` can be the same D-register; this operation explicitly modifies the `src_reg`.

This ***16-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Logic32 Example

```
r4 = r4 & r3 ; /* AND */
r4 = r4 | r3 ; /* OR */
r4 = r4 ^ r3 ; /* XOR */
```

32-Bit One's Complement (Not32)

General Form

ALU Binary Operations (ALU2op)
DREG Register Type = ~DREG Register Type

Abstract

This instruction (NOT one's complement) toggles every bit in the 32-bit register.

See Also (32-Bit Logic Operations (Logic32))

Not32 Description

The NOT one's-complement instruction toggles every bit in the 32-bit register. The instruction does not implicitly modify the src_reg. The `dest_reg` and `src_reg` can be the same D-register. Using the same D-register as the `dest_reg` and `src_reg` would explicitly modify the `src_reg`.

This ***16-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Not32 Example

```
r3 = ~ r4 ; /* NOT */
```

Move Operations

These operations provide register move operations on register, half register, and accumulator register operands:

- Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE)
- Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL)
- Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH)
- Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO)
- Move Register to Accumulator0 (MvAxToAx)
- Move Accumulator to Register (MvAxToDreg)
- Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL)
- Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg)
- Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL)
- Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX)
- Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg)
- Move Register to Accumulator1 (MvDregToAx)
- Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual)
- Move Register to Register (MvRegToReg)
- Conditional Move Register to Register (MvRegToRegCond)
- Dual Move Accumulators to Half Registers (ParaMvA1ToDregHwithMvA0ToDregL)
- Dual Move Accumulators to Register (ParaMvA1ToDregOwithMvA0ToDregE)

Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE)

General Form

Multiply Accumulate (Dsp32Mac)

DREG_E Register Type = a0 MMODE

Abstract

This instruction moves an 32-bit section of an accumulator to an even register.

See Also (Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO), Move Accumulator to Register (MvAxToDreg), Move Register to Accumulator0 (MvAxToAx))

MvA0ToDregE Description

The move accumulator register to even data register instruction copies the contents of the source accumulator register into the destination even data register. The operation does not affect the source register contents.

In the syntax, where `MMODE` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(iss2)`, `(iu)`, or `(s2rnd)`. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA0ToDregE Example

```
r2 = a0 ; /* 32-bit move with saturation */
r0 = a0 (iss2) ; /* 32-bit move with scaling, truncation and saturation */
```

Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL)

General Form

Multiply Accumulate (Dsp32Mac)

DREG_L Register Type = a0 MMOD1

Abstract

This instruction moves an 16-bit section of an accumulator to a low half register (16-bit section of a data register).

See Also (Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO), Move Accumulator to Register (MvAxToDreg), Move Register to Accumulator0 (MvAxToAx))

MvA0ToDregL Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where `MMOD1` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(ih)`, `(is)`, `(iss2)`, `(iu)`, `(s2rnd)`, `(t)`, or `(tfu)`. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA0ToDregL Example

```
r3.l = a0 ;
r7.l = a0 (fu) ; /* fractional unsigned format */
r2.l = a0 (s2rnd) ; /* signed fraction, scaled */
```

Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_H Register Type = a1 MMLMMOD1

Abstract

This instruction moves an 16-bit section of an accumulator to a high half register (16-bit section of a data register).

See Also (Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO), Move Accumulator to Register (MvAxToDreg), Move Register to Accumulator0 (MvAxToAx))

MvA1ToDregH Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where `MMLMMOD1` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(ih)`, `(is)`, `(iss2)`, `(iu)`, `(m)`, `(m,fu)`, `(m,ih)`, `(m,is)`, `(m,iss2)`, `(m,iu)`, `(m,s2rnd)`, `(m,t)`, `(m,tfu)`, `(s2rnd)`, `(t)`, or `(tfu)` . See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode.***

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA1ToDregH Example

```
r3.h = a1 ;
r7.h = a1 (fu) ; /* fractional unsigned format */
r2.h = a1 (s2rnd) ; /* signed fraction, scaled */
```

Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_O Register Type = a1 MMLMMODE

Abstract

This instruction moves an 32-bit section of an accumulator to an odd register.

See Also (Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move Accumulator to Register (MvAxToDreg), Move Register to Accumulator0 (MvAxToAx))

MvA1ToDregO Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where `MMLMMODE` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(iss2)`, `(iu)`, `(m)`, `(m,fu)`, `(m,is)`, `(m,iss2)`, `(m,iu)`, `(m,s2rnd)`, `(s2rnd)` . See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvA1ToDregO Example

```
r3 = a1 ;
r7 = a1 (fu);
r1 = a1 (s2rnd);
```

Move Register to Accumulator0 (MvAxToAx)

General Form

ALU Operations (Dsp32Alu)
a0 = a1
a1 = a0

Abstract

This instruction moves the contents of one accumulator register to the other accumulator register.

See Also (Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO), Move Accumulator to Register (MvAxToDreg))

MvAxToAx Description

The move accumulator register to accumulator register instruction copies the contents of the source accumulator register into the destination accumulator register. The operation does not affect the source register contents.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MvAxToAx Example

```
a0 = a1 ;
a1 = a0 ;
```

Move Accumulator to Register (MvAxToDreg)

General Form

Multiply with 3 operands (Dsp32Mult)
DREG_O Register Type = MUL1 MML, DREG_E Register Type = MUL0 MMODE
DREG Register Type = a1:0 M32MMOD2
DREG_PAIR Register Type = a1:0 M32MMOD

Abstract

This instruction moves the value in the accumulator register to the selected data register.

See Also (Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO), Move Register to Accumulator0 (MvAxToAx))

MvAxToDreg Description

The move accumulator register to low half data register instruction copies the contents of the source accumulator register into the destination low half data register. The operation does not affect the source register contents.

In the syntax, where `M32MMOD` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, `(m,is,ns)`, `(m,t)`, `(t)`, or `(tfu)` . See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvAxToDreg Example

```
r3 = a0 (iu,ns); /* integer unsigned no saturate */
r4 = a1 (fu); /* fractional unsigned */
r2 = a0 (m); /* mixed mode */
r5 = a1 (tfu); /* fractional unsigned truncated */
```

Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL)

General Form

ALU Operations (Dsp32Alu)
DREG_L Register Type = a0.x
DREG_L Register Type = a1.x

Abstract

This instruction moves an 8-bit section of an accumulator to a low half register (16-bit section of a data register).

See Also (Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE), Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL), Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH), Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO), Move Accumulator to Register (MvAxToDreg), Move Register to Accumulator0 (MvAxToAx))

MvAxXToDregL Description

The move accumulator register extension copies 8 bits from an accumulator extension source register into a low half data register. The instruction does not affect the unspecified half of the destination register. It supports only data registers and the accumulator.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MvAxXToDregL Example

```
r7.l = a0.x ;
r0.l = a1.x ;
```

Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg)

General Form

ALU Binary Operations (ALU2op)
DREG Register Type = DREG_B Register Type (x)
DREG Register Type = DREG_B Register Type (z)

Abstract

This instruction copies the least significant 8-bits from the source register into the least significant 8-bits of the destination and either sign or zero extends it the upper bits.

See Also (Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX), Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL), Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg), Move Register to Accumulator1 (MvDregToAx), Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual))

MvDregBToDreg Description

The move data register byte to data register instruction converts a signed byte to a signed word (32 bits). It copies the least significant 8 bits from a source register into the least significant 8 bits of a 32-bit register. The instruction sign-extends or zero-extends the upper bits of the destination register. This instruction supports only data registers.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), and this instruction may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvDregBToDreg Example

```
r7 = r2.b (x) ; /* sign extended */
r7 = r2.b (z) ; /* zero extended */
```

Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL)

General Form

ALU Operations (Dsp32Alu)
A0_HL = DSRC0_HL
A1_HL = DSRC0_HL

Abstract

This instruction moves a 16-bit section of a data register to a single 16-bit section of an accumulator.

See Also (Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX), Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg), Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg), Move Register to Accumulator1 (MvDregToAx), Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual))

MvDregHLToAxHL Description

The move high/low half register to high/low half accumulator instruction copies 16 bits from a source register into half of an accumulator register. The instruction does not affect the unspecified half of the destination register. It supports only data registers and the accumulator.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MvDregHLToAxHL Example

```
a0.l = r0.l /* least significant 16 bits of Dreg into least significant 16 bits of A0.W */
a0.h = r1.l
a1.l = r2.l /* least significant 16 bits of Dreg into least significant 16 bits of A1.W */
a1.h = r3.l
a0.l = r4.h
a0.h = r5.h /* most significant 16 bits of Dreg into most significant 16 bits of A0.W */
a1.l = r6.h
a1.h = r7.h /* most significant 16 bits of Dreg into most significant 16 bits of A1.W */
```

Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX)

General Form

ALU Operations (Dsp32Alu)
a0.x = DREG_L Register Type
a1.x = DREG_L Register Type

Abstract

This instruction moves the 8 LSBs from a low half register (16-bit section of a data register) to an 8-bit section of an accumulator.

See Also (Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL), Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg), Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg), Move Register to Accumulator1 (MvDregToAx), Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual))

MvDregLToAxX Description

The move low half data register to accumulator register extension instruction copies 8 bits from a low half data register source into an accumulator extension register. The instruction does not affect the unspecified portion of the destination register. It supports only data registers and the accumulator.

The accumulator extension registers `A0.X` and `A1.X` are defined only for the 8 low-order bits 7 through 0 of `A0.X` and `A1.X`. This instruction truncates the upper byte of *`DDST0_L`* before moving the value into the accumulator extension register (`A0.X` or `A1.X`).

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MvDregLToAxX Example

```
a0.x = r1.l ;
a1.x = r4.l ;
```

Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg)

General Form

ALU Binary Operations (ALU2op)
DREG Register Type = DREG_L Register Type (x)
DREG Register Type = DREG_L Register Type (z)

Abstract

This instruction zero extends or sign extends a 16-Bit register half and deposits it into a 32-bit destination register. The X option signifies sign extension signifies sign extension while the Z signifies zero extension.

See Also (Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX), Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL), Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg), Move Register to Accumulator1 (MvDregToAx), Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual))

MvDregLToDreg Description

The move low half register to data register instruction converts an unsigned half word (16 bits) to an unsigned word (32 bits). The instruction copies the least significant 16 bits from a source register into the lower half of a 32-bit register and sign- or zero-extends the upper half of the destination register. The operation supports only data registers. Zero extension is appropriate for unsigned values. If used with signed values, a small negative 16-bit value will become a large positive value.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), and this instruction may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0

...	...	AC1	*AC0*	...	...	...	RND_MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvDregLToDreg Example

```
/* If r0.l = 0xFFFF, before move with zero extend ... */
r4 = r0.l (z) ; /* zero-extends; equivalent operation to r4.l = r0.l and r4.h = 0 */
/* . . . then r4 = 0x0000FFFF, after move with zero extend */
r4 = r0.l (x) ; /* sign-extends */
```

Move Register to Accumulator1 (MvDregToAx)

General Form

ALU Operations (Dsp32Alu)
a0 = DREG Register Type XMODE
a1 = DREG Register Type XMODE

Abstract

This instruction moves the contents of a data register to an accumulator register.

See Also (Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX), Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL), Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg), Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg), Move Register to Accumulator0 & Accumulator1 (MvDreg-ToAxDual))

MvDregToAx Description

The move data register to accumulator instruction copies 32 bits from a source register into Ax.W section of an accumulator register with zero- or sign-extension. The instruction does not affect the unspecified portion of the destination register. It supports only data registers and the accumulator.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MvDregToAx Example

```
a0 = r7 (z) ; /* move R7 to 32-bit A0.W, zero extended */
a1 = r3 (x) ; /* move R3 to 32-bit A1.W, sign-exteneded */
```

Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual)

General Form

ALU Operations (Dsp32Alu)
a1 = DREG Register Type SMODE, a0 = DREG Register Type XMODE

Abstract

This instruction moves the contents of two data registers to the accumulator registers (A0, A1).

See Also (Move Register Half (LSBs) to 8-Bit Accumulator Section (MvDregLToAxX), Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL), Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg), Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg), Move Register to Accumulator1 (MvDregToAx))

MvDregToAxDual Description

The dual move data register to accumulator instruction copies 32 bits from a source register into Ax.W section of an accumulator register with zero- or sign-extension, and perform a second move in parallel as indicated. The instruction does not affect the unspecified portion of the destination register. It supports only data registers and the accumulator.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MvDregToAxDual Example

```
a1 = r3 (z), a0 = r7 (z) ; /* move R3 to 32-bit A1.W (zero extended), move R7 to 32-bit
A0.W (zero extended) */
a1 = r3 (z), a0 = r7 (x) ; /* move R3 to 32-bit A1.W (zero extended), move R7 to 32-bit
A0.W (sign-exteneded) */
a1 = r3 (x), a0 = r7 (z) ; /* move R3 to 32-bit A1.W (sign-exteneded), move R7 to 32-bit
A0.W (zero extended) */
a1 = r3 (x), a0 = r7 (x) ; /* move R3 to 32-bit A1.W (sign-exteneded), move R7 to 32-bit
A0.W (sign-exteneded) */
```

Move Register to Register (MvRegToReg)

General Form

Register to register transfer operation (RegMv)
GDST = GSRC

Abstract

This instruction moves data from any register in the data arithmetic unit, address arithmetic unit, or control unit to any other register in those units.

See Also (Conditional Move Register to Register (MvRegToRegCond))

MvRegToReg Description

The move any register to any register instruction copies from a source register into a destination register with zero- or sign-extension. The instruction does not affect the unspecified portion of the destination register. It supports all processor core registers. All moves from smaller to larger registers are sign extended.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), and this instruction may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either ***User or Supervisor mode***, except for cases where register access restrictions only permit ***Supervisor mode*** .

MvRegToReg Example

```
r3 = r0 ;
r7 = p2 ;
r2 = a0 ;
a0.w = r7 ; /* move R7 to 32-bit A0.W */
r3 = a1.x ; /* move 8-bit A1.X to R3 with sign extension*/
retn = p0 ; /* must be in Supervisor mode */
r7 = a0 ; /* move A0 to odd data register */
r2 = a1 ; /* move A1 to even data register */
```

Conditional Move Register to Register (MvRegToRegCond)

General Form

Conditional Move (CCMV)
if cc GDST = GSRC
if !cc GDST = GSRC

Abstract

This instruction conditionally moves registers.

See Also (Move Register to Register (MvRegToReg))

MvRegToRegCond Description

The Move Conditional instruction moves source register contents into a destination register, depending on the value of CC.

- `IF CC DPreg = DPreg`, the move occurs only if CC = 1.
- `IF ! CC DPreg = DPreg`, the move occurs only if CC = 0.

The source and destination registers are any data register or pointer register.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MvRegToRegCond Example

```
if cc r3 = r0 ; /* move if CC=1 */
if cc r2 = p4 ;
if cc p0 = r7 ;
if cc p2 = p5 ;
if ! cc r3 = r0 ; /* move if CC=0 */
if ! cc r2 = p4 ;
if ! cc p0 = r7 ;
if ! cc p2 = p5 ;
```

Dual Move Accumulators to Half Registers (ParaMvA1ToDregHwithMvA0To-DregL)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_H Register Type = a1 MML, DREG_L Register Type = a0 MMOD1

Abstract

This dual move instruction moves the contents of the accumulator registers to half data registers.

See Also (Dual Move Accumulators to Register (ParaMvA1ToDregOwithMvA0ToDregE))

ParaMvA1ToDregHwithMvA0ToDregL Description

The dual move accumulator 1 to high half register with move accumulator 0 to low half register instruction provide a the combination of operations in the Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL) and Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH).

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvA1ToDregHwithMvA0ToDregL Example

```
r3.h = a1 , r3.l = a0 ;
r2.h = a1 (s2rnd) , r7.l = a0 (fu) ; /* signed fraction, scaled, fractional unsigned format
*/
r7.h = a1 (fu) , r2.l = a0 (s2rnd) ; /* fractional unsigned format, signed fraction, scaled
*/
```

Dual Move Accumulators to Register (ParaMvA1ToDregOwithMvA0ToDregE)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_O Register Type = a1 MML, DREG_E Register Type = a0 MMODE

Abstract

This dual move instruction moves the contents of the accumulator registers to data registers.

See Also (Dual Move Accumulators to Half Registers (ParaMvA1ToDregHwithMvA0ToDregL))

ParaMvA1ToDregOwithMvA0ToDregE Description

The dual move accumulator 1 to high half register with move accumulator 0 to low half register instruction provide a the combination of operations in the Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO) with Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE).

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

31															
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvA1ToDregOwithMvA0ToDregE Example

```
r3 = a1 , r0 = a0 (iss2) ;
r7 = a1 (fu) , r2 = a0 ;
r1 = a1 (s2rnd) , r2 = a0 ;
```

Multiplication Operations

These operations provide multiply and multiply-accumulate operations on register and immediate value operands:

- 16 x 16-Bit MAC (Mac16)
- 16 x 16-Bit MAC with Move to Register (Mac16WithMv)
- 32 x 32-Bit MAC (Mac32)
- 32 x 32-Bit MAC with Move to Register (Mac32WithMv)
- Complex Multiply to Accumulator (Mac32Cmplx)
- Complex Multiply to Register (Mac32CmplxWithMv)
- Complex Multiply to Register with Narrowing (Mac32CmplxWithMvN)
- 16 x 16-Bit Multiply (Mult16)
- 32 x 32-bit Multiply (Mult32)
- 32 x 32-Bit Multiply, Integer (MultInt)
- Dual 16 x 16-Bit MAC (ParaMac16AndMac16)
- Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)
- Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16)
- Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)
- Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)
- Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMv)
- Dual 16 x 16-Bit Multiply (ParaMult16AndMult16)
- Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)
- Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)

16 x 16-Bit MAC (Mac16)

General Form

Multiply Accumulate (Dsp32Mac)
MAC0 MMOD0
MAC0 MMOD0
MAC1 MMLMMOD0
MAC1 MMLMMOD0

Abstract

This multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required.

See Also (16 x 16-Bit MAC with Move to Register (Mac16WithMv))

Mac16 Description

The Multiply and Multiply-Accumulate to Accumulator instruction multiplies two 16-bit half-word operands. It stores, adds or subtracts the product into a designated Accumulator with saturation.

The Multiply-and-Accumulate Unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0. MAC1 performs A1 operations.

By default, the instruction treats both operands of both MACs as signed fractions with left-shift correction as required.

In the syntax, where `MMOD0` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, or `(w32)` .

In the syntax, where `MMLMMOD0` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(m)`, `(W32)`, `(m,fu)`, `(m,is)`, or `(m,w32)` .

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac16 Example

```
a0 = r3.h * r2.h ; /* MAC0, only. Both operands are signed fractions. Load the product into
A0. */
a0 += r6.h * r4.l (fu) ; /* MAC0, only. Both operands are unsigned fractions. Accumulate
into A0 */
a0 -= r3.h * r2.h ; /* MAC0, only. Both operands are signed fractions.  Accumulate into A0.
*/
a1 = r6.h * r4.l (fu) ; /* MAC1, only. Both operands are unsigned fractions. Load the
product into A1 */
a1 += r3.h * r2.h ; /* MAC1, only. Both operands are signed fractions. Accumulate into A1.
*/
a1 -= r6.h * r4.l (fu) ; /* MAC1, only. Both operands are unsigned fractions. Accumulate
into A1 */
```

16 x 16-Bit MAC with Move to Register (Mac16WithMv)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_L Register Type = (MAC0) MMOD1
DREG_L Register Type = (MAC0) MMOD1
DREG_H Register Type = (MAC1) MMLMMOD1
DREG_H Register Type = (MAC1) MMLMMOD1

Abstract

This multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required.

See Also (16 x 16-Bit MAC (Mac16))

Mac16WithMv Description

The multiply and multiply-accumulate to half register (with move) instruction multiplies two 16-bit half-word operands. The instruction stores, adds or subtracts the product into a designated accumulator. Then, it copies 16 bits (saturated at 16 bits) of the accumulator into a high or low half data register.

In the syntax, where `MMOD1` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(ih)`, `(is)`, `(iss2)`, `(iu)`, `(s2rnd)`, `(t)`, `(tfu)` .

In the syntax, where `MMLMMOD1` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(ih)`, `(is)`, `(iss2)`, `(iu)`, `(m)`, `(m,fu)`, `(m,ih)`, `(m,is)`, `(m,iss2)`, `(m,iu)`, `(m,s2rnd)`, `(m,t)`, `(m,tfu)`, `(s2rnd)`, `(t)`, `(tfu)` .

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

The fraction versions of this instruction (the default and `(fu)` options) transfer the Accumulator result to the destination register according to the ***Result to Destination Register ((IS) and (IU) Options)*** diagram.

The integer versions of this instruction (the `(is)` and `(iu)` options) transfer the Accumulator result to the destination register according to the ***Result to Destination Register ((IS) and (IU) Options)*** diagram.

The multiply-and-accumulate unit 0 (MAC0) portion of the architecture performs operations that involve accumulator A0 and loads the results into the lower half of the destination data register. MAC1 performs A1 operations and loads the results into the upper half of the destination data register.

All versions of this instruction that support rounding are affected by the `RND_MOD` bit in the ASTAT register when they copy the results into the destination register. `RND_MOD` determines whether biased or unbiased rounding is used.

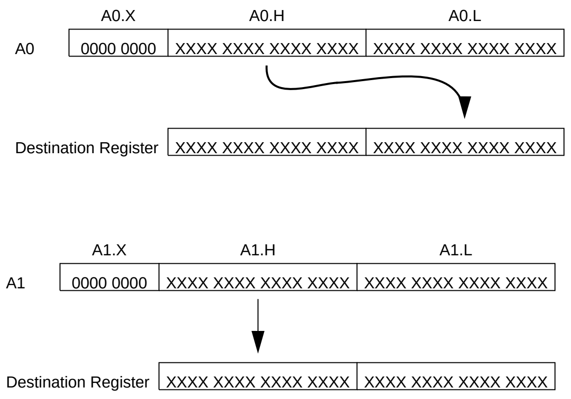


Figure 8-4: Result to Destination Register (Default and (FU) Options)

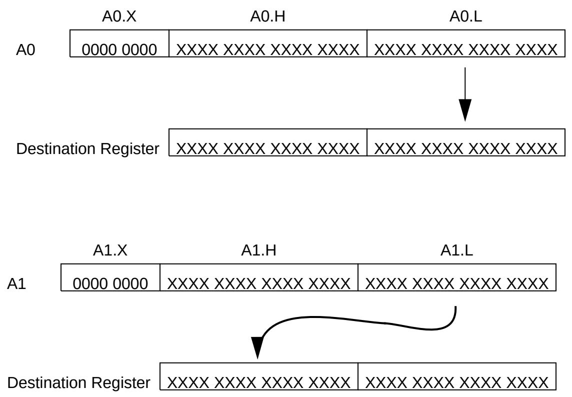


Figure 8-5: Result to Destination Register ((IS) and (IU) Options)

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

This instruction has ***special applications***. DSP filter applications often use the multiply and multiply-accumulate to half-register instruction to calculate the dot product between two signal vectors.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac16WithMvExample

```
r3.l = ( a0 = r3.h * r2.h ) ;
/* MAC0, only. Both operands are signed fractions. Load the product into A0, then copy to
r3.l. */
r3.h = ( a1 += r6.h * r4.l ) (fu) ;
/* MAC1, only. Both operands are unsigned fractions. Add the product into A1, then copy to
r3.h */
```

32 x 32-Bit MAC (Mac32)

General Form

Multiply with 3 operands (Dsp32Mult)

a1:0 = DREG Register Type * DREG Register Type M32MMOD
a1:0 += DREG Register Type * DREG Register Type M32MMOD
a1:0 -= DREG Register Type * DREG Register Type M32MMOD

Abstract

This instruction executes a multiply accumulate operation on 32-bit registers.

See Also (32 x 32-Bit MAC with Move to Register (Mac32WithMv))

Mac32 Description

The multiply-accumulate to accumulator instruction multiplies two 32-bit half-word operands. It stores, adds or subtracts the product into a designated accumulator with saturation.

The multiply-and-accumulate unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0. MAC1 performs A1 operations.

By default, the instruction treats both operands of both MACs as signed fractions with left-shift correction as required.

In the syntax, where `M32MMOD0` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, or `(m,is,ns)` .

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	***AV0S***	***AV0***
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32 Example

```
a0 = r0 * r7 ; /* (default) fractional signed, place product in a0 */
a0 += r1 * r6 (fu) ; /* fractional unsigned, accumulate in a0 */
a0 -= r2 * r5 (is) ; /* integer signed, accumulate in a0 */
a1 = r3 * r4 (is,ns) ; /* integer signed (no saturation), place product in a1 */
```

```
a1 += r4 * r3 (iu) ; /* integer unsigned, accumulate in a1 */
a1 -= r5 * r2 (iu,ns) ; /* integer unsigned (no saturation), accumulate in a1 */
a0 = r6 * r1 (m) ; /* mixed mode and fractional signed, place product in a0 */
a0 += r7 * r0 (m,is) ; /* mixed mode and integer signed, accumulate in a0 */
a0 -= r0 * r7 (m,is,ns) ; /* mixed mode and integer signed (no saturation), accumulate in
a0 */
a1 = r1 * r6 ; /* (default) fractional signed, place product in a1 */
a1 += r2 * r5 (fu) ; /* fractional unsigned, accumulate in a1 */
a1 -= r3 * r4 (is) ; /* integer signed, accumulate in a1 */
```

32 x 32-Bit MAC with Move to Register (Mac32WithMv)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_E Register Type = (MAC0) MMODE
DREG_E Register Type = (MAC0) MMODE
DREG_O Register Type = (MAC1) MMLMMODE
DREG_O Register Type = (MAC1) MMLMMODE
Multiply with 3 operands (Dsp32Mult)
DREG Register Type = (a1:0 = DREG Register Type * DREG Register Type) M32MMOD1
DREG Register Type = (a1:0 += DREG Register Type * DREG Register Type) M32MMOD1
DREG Register Type = (a1:0 -= DREG Register Type * DREG Register Type) M32MMOD1
DREG_PAIR Register Type = (a1:0 = DREG Register Type * DREG Register Type) M32MMOD
DREG_PAIR Register Type = (a1:0 += DREG Register Type * DREG Register Type) M32MMOD
DREG_PAIR Register Type = (a1:0 -= DREG Register Type * DREG Register Type) M32MMOD

Abstract

This multiply-accumulate instruction multiplies two 32-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required.

See Also (32 x 32-Bit MAC (Mac32))

Mac32WithMv Description

The multiply-accumulate to accumulator instruction multiplies two 32-bit half-word operands. It stores, adds or subtracts the product into a designated accumulator with saturation. Then, the instruction moves the result to the selected register or register pair.

The multiply-and-accumulate unit 0 (MAC0) portion of the architecture performs operations that involve Accumulator A0. MAC1 performs A1 operations.

By default, the instruction treats both operands of both MACs as signed fractions with left-shift correction as required.

In the syntax, where `MMODE` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(iss2)`, `(iu)`, or `(s2rnd)`.

In the syntax, where `MMLMMODE` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(iss2)`, `(iu)`, `(m)`, `(m,fu)`, `(m,is)`, `(m,iss2)`, `(m,iu)`, `(m,s2rnd)`, or `(s2rnd)`.

In the syntax, where `M32MMOD1` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m,is)`, `(m,is,ns)`, `(m,t)`, `(t)`, or `(tfu)`.

In the syntax, where `M32MMOD` appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, or `(m,is,ns)`.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32WithMv Example

```
r0 = (a0 = r0 * r3) (fu) ; /* MMODE options, place product in a0 and move it to EVEN data
register */
r2 = (a0 += r1 * r2) (is) ; /* MMODE options, accumulate in a0 and move it to EVEN data
register */
r4 = (a0 -= r2 * r1) (iss2) ; /* MMODE options, accumulate in a0 and move it to EVEN data
register */
r1 = (a1 = r3 * r0) (m,fu) ; /* MMLMMODE options, place product in a1 and move it to ODD
data register */
r3 = (a1 += r4 * r7) (m,is) ; /* MMLMMODE options, accumulate in a1 and move it to ODD data
register */
r5 = (a1 -= r5 * r6) (m,s2rnd) ; /* MMLMMODE options, accumulate in a1 and move it to ODD
data register */
r1 = (a0 = r6 * r5) (t) ; /* M32MMOD1 options, place product in a0 and move it to any data
register */
```

```
r2 = (a0 += r7 * r4) (tfu) ; /* M32MMOD1 options, accumulate in a0  and move it to any data
register*/
r3 = (a0 -= r0 * r3) (m,is,ns) ; /* M32MMOD1 options, accumulate in a0  and move it to any
data register*/
r4 = (a1 = r1 * r2) (iu,ns) ; /* M32MMOD1 options, place product in a1  and move it to any
data register*/
r5 = (a1 += r2 * r1) (fu) ; /* M32MMOD1 options, accumulate in a1  and move it to any data
register*/
r6 = (a1 -= r3 * r0) (m,is) ; /* M32MMOD1 options, accumulate in a1  and move it to any
data register*/
r1:0 = (a0 = r4 * r7) (fu) ; /* M32MMOD options, place product in a0 and move it to
register pair */
r3:2 = (a0 += r5 * r6) ; /* M32MMOD options, accumulate in a0 and move result to register
pair */
r5:4 = (a0 -= r6 * r5) (m) ; /* M32MMOD options, accumulate in a0 and move result to
register pair */
r7:6 = (a1 = r7 * r4) (is,ns) ; /* M32MMOD options, place product in a1 and move it to
register pair */
r5:4 = (a1 += r0 * r3) (iu,ns) ; /* M32MMOD options, accumulate in a1 and move result to
register pair */
r3:2 = (a1 -= r1 * r2) (is) ; /* M32MMOD options, accumulate in a1 and move result to
register pair */
```

Complex Multiply to Accumulator (Mac32Cmplx)

General Form

Multiply Accumulate (Dsp32Mac)
a1:0 = CMPLXOP CMODE
a1:0 += CMPLXOP CMODE
a1:0 -= CMPLXOP CMODE

Abstract

This instruction executes a complex multiply-accumulate operation, placing the results in an accumulator register.

See Also (Complex Multiply to Register (Mac32CmplxWithMv), Complex Multiply to Register with Narrowing (Mac32CmplxWithMvN))

Mac32Cmplx Description

The multiply-accumulate complex values instruction performs a number of parallell multiply-accumulate operations to produce complex results. To understand the operations, it is important to understand the placement of the imaginary part and real part of the data. Let operand A = (A_r + j *B_i), operand B = (B_r + j *B_i) and result C = (C_r + j *C_i), where A_i (the imaginary part) is stored in the most significant 16 bits of a 32 bit register, and A_r (the real part) is stored in the least significant 16 bits. Other notations (such as B_i, C_i, and others) are similarly defined regarding data placement of imaginary and real parts. Complex multiplication and complex multiplication of conjugates is defined as follows:

Table 8-7: Complex Multiplication and Complex Conjugates

Complex Multiplication	Imaginary Conjugate	Real Conjugate
C = cmul(A, B)	$C_i = A_r*B_i + A_i*B_r$	$C_r = A_r*B_r - A_i*B_i$
C = cmul(A, B*)	$C_i = A_i*B_r - A_r*B_i$	$C_r = A_r*B_r + A_i*B_i$
C = cmul(A*, B*)	$C_i = -(A_r*B_i + A_i*B_r)$	$C_r = A_r*B_r - A_i*B_i$

This complex multiply syntax for placing the product in the accumulator registers corresponds to the commented operations:

```
a1:0 = cmul(r1,r0); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = (r1.l * r0.h) + (r1.h * r0.l), a0 = (r1.l * r0.l) - (r1.h * r0.h) */

a1:0 = cmul(r1,r0*); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = (r1.h * r0.l) - (r1.l * r0.h), a0 = (r1.l * r0.l) + (r1.h * r0.h) */

a1:0 = cmul(r1*,r0*); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = - [ (r1.l * r0.h) + (r1.h * r0.l) ], a0 = (r1.l * r0.l) - (r1.h * r0.h) */
```

In the syntax, where `CMODE` appears, substitute a complex multiply mode for the accumulator copy format option: default (none) or `(is)`.

Default operation is ***signed fraction multiplication***. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. This operation uses ***signed fraction rounding***.

If the `(is)` option is used, the operation is ***signed integer multiplication***. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum 2^{31}-1.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

31															
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32Cmplx Example

```
a1:0 = (r1,r0) ; /* fractional signed complex multiply; place complex product in a1
(imaginary) and a0 (real) */
a1:0 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in a1
(imaginary) and a0 (real) */
a1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in a1
(imaginary) and a0 (real) */
a1:0 += (r3*,r1*) (is) ;  /* integer signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 += (r5,r2*) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 += (r6,r1) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 -= (r2,1) (is) ;  /* integer signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 -= (r3*,r7*) /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
a1:0 -= (r4,r5*) (is) ; /* integer signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real) */
```

Complex Multiply to Register (Mac32CmplxWithMv)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_PAIR Register Type = CMPLXOP CMODE
DREG_PAIR Register Type = (a1:0 = CMPLXOP) CMODE
DREG_PAIR Register Type = (a1:0 += CMPLXOP) CMODE
DREG_PAIR Register Type = (a1:0 -= CMPLXOP) CMODE

Abstract

This instruction executes a complex multiply-accumulate operation, placing the results in a register or register pair.

See Also (Complex Multiply to Accumulator (Mac32Cmplx), Complex Multiply to Register with Narrowing (Mac32CmplxWithMvN))

Mac32CmplxWithMv Description

The multiply-accumulate complex values instruction performs a number of parallell multiply-accumulate operations to produce complex results with a move. The product of the multiplication is placed in a pair of data registers. Alternately, the instruction may accumulate the result in the accumulator registers, then move the result to a pair of data registers. To understand the operations, it is important to understand the placement of the imaginary part and real part of the data. Let operand A = (A_r + j *B_i), operand B = (B_r + j *B_i) and result C = (C_r + j *C_i), where A_i (the imaginary part) is stored in the most significant 16 bits of a 32 bit register, and A_r (the real part) is stored in the least significant 16 bits. Other notations (such as B_i, C_i, and others) are similarly defined regarding data placement of imaginary and real parts. Complex multiplication and complex multiplication of conjugates is defined as follows:

Table 8-8: Complex Multiplication and Complex Conjugates

Complex Multiplication	Imaginary Conjugate	Real Conjugate
C = cmul(A, B)	$C_i = A_r*B_i + A_i*B_r$	$C_r = A_r*B_r - A_i*B_i$
C = cmul(A, B*)	$C_i = A_i*B_r - A_r*B_i$	$C_r = A_r*B_r + A_i*B_i$
C = cmul(A*, B*)	$C_i = -(A_r*B_i + A_i*B_r)$	$C_r = A_r*B_r - A_i*B_i$

This complex multiply syntax for placing the product in the accumulator registers corresponds to the commented operations:

```
a1:0 = cmul(r1,r0); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = (r1.l * r0.h) + (r1.h * r0.l), a0 = (r1.l * r0.l) - (r1.h * r0.h) */

a1:0 = cmul(r1,r0*); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = (r1.h * r0.l) - (r1.l * r0.h), a0 = (r1.l * r0.l) + (r1.h * r0.h) */

a1:0 = cmul(r1*,r0*); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = - [ (r1.l * r0.h) + (r1.h * r0.l) ], a0 = (r1.l * r0.l) - (r1.h * r0.h) */
```

In the syntax, where `CMODE` appears, substitute a complex multiply mode for the accumulator copy format option: default (none) or `(is)`.

Default operation is ***signed fraction multiplication***. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF.

If the `(is)` option is used, the operation is ***signed integer multiplication***. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum 2^{31}-1.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	***VS***	***V***	...	...	...	...	***AV1S***	***AV1***	***AV0S***	***AV0***
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32CmplxWithMv Example

```
r7:6 = (r1,r0) ; /* fractional signed complex multiply; place complex product in r7
(imaginary) and r6 (real) */
r5:4 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in r5
(imaginary) and r4 (real) */
r1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in r1
(imaginary) and r0 (real) */
r7:6 = a1:0 = (r1,r0) ; /* fractional signed complex multiply; place complex product in a1
(imaginary) and a0 (real); move to r7:6 */
r5:4 = a1:0 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in
a1 (imaginary) and a0 (real); move to r5:4 */
r1:0 = a1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in
a1 (imaginary) and a0 (real); move to r1:0 */
r5:4 = a1:0 += (r3*,r1*) (is) ;  /* integer signed complex mac; accumulate complex result
in a1 (imaginary) and a0 (real); move to r5:4 */
r1:0 = a1:0 += (r5,r2*) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r1:0 */
r3:2 = a1:0 += (r6,r1) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r3:2 */
r7:6 = a1:0 -= (r2,1) (is) ;  /* integer signed complex mac; accumulate complex result in
a1 (imaginary) and a0 (real); move to r7:6 */
r1:0 = a1:0 -= (r3*,r7*) /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r1:0 */
r3:2 = a1:0 -= (r4,r5*) (is) ; /* integer signed complex mac; accumulate complex result in
a1 (imaginary) and a0 (real); move to r3:2 */
```

Complex Multiply to Register with Narrowing (Mac32CmplxWithMvN)

General Form

Multiply Accumulate (Dsp32Mac)

DREG Register Type = CMPLXOP NARROWING_CMODE
DREG Register Type = (a1:0 = CMPLXOP) NARROWING_CMODE
DREG Register Type = (a1:0 += CMPLXOP) NARROWING_CMODE
DREG Register Type = (a1:0 -= CMPLXOP) NARROWING_CMODE

Abstract

This instruction executes a complex multiply-accumulate operation, placing the results in a register or register pair with narrowing.

See Also (Complex Multiply to Accumulator (Mac32Cmplx), Complex Multiply to Register (Mac32CmplxWithMv))

Mac32CmplxWithMvN Description

The multiply-accumulate complex values instruction performs a number of parallell multiply-accumulate operations to produce complex results with a narrowing move. The product of the multiplication is placed in a data register. Alternately, the instruction may accumulate the result in the accumulator registers, then move the result to a data register. To understand the operations, it is important to understand the placement of the imaginary part and real part of the data. Let operand A = (A_r + j *B_i), operand B = (B_r + j *B_i) and result C = (C_r + j *C_i), where A_i (the imaginary part) is stored in the most significant 16 bits of a 32 bit register, and A_r (the real part) is stored in the least significant 16 bits. Other notations (such as B_i, C_i, and others) are similarly defined regarding data placement of imaginary and real parts. Complex multiplication and complex multiplication of conjugates is defined as follows:

Table 8-9: Complex Multiplication and Complex Conjugates

Complex Multiplication	Imaginary Conjugate	Real Conjugate
C = cmul(A, B)	$C_i = A_r*B_i + A_i*B_r$	$C_r = A_r*B_r - A_i*B_i$
C = cmul(A, B*)	$C_i = A_i*B_r - A_r*B_i$	$C_r = A_r*B_r + A_i*B_i$
C = cmul(A*, B*)	$C_i = -(A_r*B_i + A_i*B_r)$	$C_r = A_r*B_r - A_i*B_i$

This complex multiply syntax for placing the product in the accumulator registers corresponds to the commented operations:

```
a1:0 = cmul(r1,r0); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = (r1.l * r0.h) + (r1.h * r0.l), a0 = (r1.l * r0.l) - (r1.h * r0.h) */

a1:0 = cmul(r1,r0*); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = (r1.h * r0.l) - (r1.l * r0.h), a0 = (r1.l * r0.l) + (r1.h * r0.h) */

a1:0 = cmul(r1*,r0*); /* complex multiply of r1 and r0, place imaginary product in a1 and
real product in a0 */
/* a1 = - [ (r1.l * r0.h) + (r1.h * r0.l) ], a0 = (r1.l * r0.l) - (r1.h * r0.h) */
```

In the syntax, where `NARROWING_CMODE` appears, substitute a complex multiply mode for the accumulator copy format option: default (none), `(is)`, or `(t)`.

Default operation is ***signed fraction multiplication***. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. This operation uses ***signed fraction rounding***. Round 1.31 format value at bit 16, (RND_MOD bit in the ASTAT register controls the rounding) extract the high 16 bits to produce a 1.15 result. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).

If the `(is)` option is used, the operation is ***signed integer multiplication***. Multiply 16.0 * 16.0 to produce 32.0 results. No shift correction. Saturate integer results between minimum -2^{31} and maximum 2^{31}-1. This operation uses ***signed integer saturation***. Saturate 32-bit integer values at bit 15 and extract the low 16 bits to produce a result between minimum -2^{15} and maximum 2^{15}-1.

If the `(t)` option is used, the operation is ***signed fraction multiplication with truncation***. Multiply 1.15 * 1.15 to produce 1.31 results after left-shift correction for each of the four partial products. Add or subtract corresponding partial products for real and imaginary part of result. Saturate results between minimum -1 and maximum 1^{-2-31}. The resulting hexadecimal range is minimum 0x8000 0000 through maximum 0x7FFF FFFF. This operation uses ***signed fraction truncation***. Truncate 1.31 format values for real and imaginary parts of the result at bit 16, (Perform no rounding) extract the high 16 bits to produce a 1.15 result. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	***VS***	***V***	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	***AN***	***AZ***
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mac32CmplxWithMvN Example

```
r6 = (r1,r0) ; /* fractional signed complex multiply; place complex product in r6.h
(imaginary) and r6.l (real) */
```

```
r4 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in r4.h
(imaginary) and r4.l (real) */
r0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in r0.h
(imaginary) and r0.l (real) */
r6 = a1:0 = (r1,r0) ; /* fractional signed complex multiply; place complex product in a1
(imaginary) and a0 (real); move to r6 */
r4 = a1:0 = (r7,r3*) (is) ; /* integer signed complex multiply; place complex product in a1
(imaginary) and a0 (real); move to r4 */
r0 = a1:0 = (r2*,r4*) ; /* fractional signed complex multiply; place complex product in a1
(imaginary) and a0 (real); move to r0 */
r4 = a1:0 += (r3*,r1*) (is) ;  /* integer signed complex mac; accumulate complex result in
a1 (imaginary) and a0 (real); move to r4 */
r0 = a1:0 += (r5,r2*) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r0 */
r2 = a1:0 += (r6,r1) ; /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r2 */
r6 = a1:0 -= (r2,1) (is) ;  /* integer signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r6 */
r0 = a1:0 -= (r3*,r7*) /* fractional signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r0 */
r2 = a1:0 -= (r4,r5*) (is) ; /* integer signed complex mac; accumulate complex result in a1
(imaginary) and a0 (real); move to r2 */
```

16 x 16-Bit Multiply (Mult16)

General Form

Multiply with 3 operands (Dsp32Mult)
DREG_L Register Type = MUL0 MMOD1
DREG_H Register Type = MUL1 MMLMMOD1
DREG_E Register Type = MUL0 MMODE
DREG_O Register Type = MUL1 MMLMMODE

Abstract

This instruction multiplies two 16-bit half word operands. It stores, adds, or subtracts the product into a designated accumulator register with saturation.

See Also (32 x 32-Bit Multiply, Integer (MultInt), 32 x 32-bit Multiply (Mult32))

Mult16 Description

The multiply 16-bit operands instruction multiplies the two 16-bit operands and stores the result directly into the destination register with saturation.

NOTE: This instruction is similar to the multiply-accumulate instructions, *except that* the multiply 16-bit operands does not affect the accumulators.

Operations performed by the multiply-and-accumulate unit 0 (MAC0) portion of the architecture load their 16-bit results into the lower half of the destination data register; 32-bit results go into an even numbered data register. Operations performed by MAC1 load their results into the upper half of the destination data register or an odd numbered data register.

In ***32-bit result syntax*** (result goes to a 32-bit data register), the MAC performing the operation is determined by the destination data register. Instructions placing results in even-numbered data registers (`R6`, `R4`, `R2`, or `R0`) execute on MAC0 and may use *`MMODE`* options. Instructions placing results in odd-numbered data registers (`R7`, `R5`, `R3`, or `R1`) execute on MAC1 and may use *`MMLMMODE`* options. For example, 32-bit result operations with the `(m)` option may only be performed using odd-numbered data register destinations.

In ***16-bit result syntax*** (result goes to a 16-bit half data register), the MAC performing the operation is determined by the destination data register half. Instructions placing results in low-half data registers (`R7.L` through `R0.L`) execute on MAC0 and may use *`MMOD1`* options. Instructions placing results in high-half data registers (`R7.H` through `R0.H`) execute on MAC1 and may use *`MMLMMOD1`* options. For example, 16-bit result operations using the `(m)` option may only be performed using high-half data register destinations.

The versions of this instruction that produce 16-bit results are affected by the `RND_MOD` bit in the `ASTAT` register when they copy the results into the 16-bit destination register. `RND_MOD` determines whether biased or unbiased rounding is used. `RND_MOD` controls rounding for all versions of this instruction that produce 16-bit results except the `(is)`, `(iu)` and `(iss2)` options.

In the syntax, where *`MMOD1`* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(ih)`, `(is)`, `(iss2)`, `(iu)`, `(s2rnd)`, `(t)`, or `(tfu)`.

In the syntax, where *`MMLMMOD1`* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(ih)`, `(is)`, `(iss2)`, `(iu)`, `(m)`, `(m,fu)`, `(m,ih)`, `(m,is)`, `(m,iss2)`, `(m,iu)`, `(m,s2rnd)`, `(m,t)`, `(m,tfu)`, `(s2rnd)`, `(t)`, or `(tfu)`.

In the syntax, where *`MMODE`* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(iss2)`, or `(s2rnd)`.

In the syntax, where *`MMLMMODE`* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(iss2)`, `(m)`, `(m,fu)`, `(m,is)`, `(m,iss2)`, `(m,s2rnd)`, or `(s2rnd)`.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mult16 Example

```
r3.l = r3.h * r2.h ; /* MAC0. Both operands are signed fractions. */
r3.h = r6.h * r4.l (fu) ; /* MAC1. Both operands are unsigned fractions. */
r6 = r3.h * r4.h ; /* MAC0. Signed fraction operands, results saved as 32 bits. */
```

32 x 32-bit Multiply (Mult32)

General Form

Multiply with 3 operands (Dsp32Mult)
DREG Register Type = DREG Register Type * DREG Register Type M32MMOD2
DREG_PAIR Register Type = DREG Register Type * DREG Register Type M32MMOD

Abstract

This instruction executes multiply operations on 32-bit registers and on register pairs.

See Also (32 x 32-Bit Multiply, Integer (MultInt), 16 x 16-Bit Multiply (Mult16))

Mult32 Description

The multiply 32-bit operands instruction multiplies two 32-bit half-word operands. It stores the product into a designated data register or data register pair with saturation.

In the syntax, where *M32MMOD2* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, `(m,is,ns)`, `(m,t)`, `(t)`, or `(tfu)`.

In the syntax, where *MM32MMOD* appears, substitute a MAC mode for the accumulator copy format option: default (none), `(fu)`, `(is)`, `(is,ns)`, `(iu)`, `(iu,ns)`, `(m)`, `(m,is)`, or `(m,is,ns)`.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode.***

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Mult32 Example

```
r0 = r0 * r3 (fu) ; /* fractional unsigned, place product in r0 */
r1:0 = r4 * r7 (fu) ; /* fractional unsigned, place product in r1:0 register pair */
```

32 x 32-Bit Multiply, Integer (MultInt)

General Form

ALU Binary Operations (ALU2op)
DREG Register Type *= DREG Register Type

Abstract

This instruction does a //C style//, modulo 32-bit multiply with no saturation.

See Also (16 x 16-Bit Multiply (Mult16), 32 x 32-bit Multiply (Mult32))

MultInt Description

The multiply 32-Bit operands instruction multiplies two 32-bit data registers (dest_reg and multiplier_register) and saves the product in `dest_reg`. The instruction mimics multiplication in the C language and effectively performs `Dreg1` = (`Dreg1` * `Dreg2`) modulo 2^{32}. Since the integer multiply is modulo 2^{32}, the result always fits in a 32-bit `dest_reg`, and overflows are possible but not detected. The overflow status bit in the `ASTAT` register is never set.

Users are required to limit input numbers to ensure that the resulting product does not exceed the 32-bit `dest_reg` capacity. If overflow notification is required, users should write their own multiplication macro with that capability.

Accumulators `A0` and `A1` are unchanged by this instruction.

The multiply 32-bit operands instruction does not implicitly modify the number in multiplier_register.

This instruction might be used to implement the congruence method of random number generation according to:

$$X[n+a] = (a \cdot X[n]) mod\ 2^{32}$$

Figure 8-6: Integer Multiply Equation

where:

- X[n] is the seed value,

- a is a large integer, and
- X[n+1] is the result that can be multiplied again to further the pseudo-random sequence.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

MultInt Example

```
r3 *= r0 ; /* equivalent to r3 = r3 * r0 */
```

Dual 16 x 16-Bit MAC (ParaMac16AndMac16)

General Form

Multiply Accumulate (Dsp32Mac)
MAC1 MML, MAC0 MMOD0
MAC1 MML, MAC0 MMOD0
MAC1 MML, MAC0 MMOD0
MAC1 MML, MAC0 MMOD0

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv), Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv), Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16))

ParaMac16AndMac16 Description

The dual multiply and multiply-accumulate to accumulator instruction is a dual (two instances issued in parallel) of the 16 x 16-Bit MAC (Mac16) instruction. For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16AndMac16 Example

```
a1 += r3.h * r2.h , a0 = r3.h * r2.h ;/
a1 -= r6.h * r4.l (fu) , a0 += r6.h * r4.l (fu) ;
a1 = r6.h * r4.l (fu) , a0 -= r3.h * r2.h ;
```

Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)

General Form

Multiply Accumulate (Dsp32Mac)
MAC1 MML, DREG_L Register Type = (MAC0) MMOD1
MAC1 MML, DREG_L Register Type = (MAC0) MMOD1
MAC1 MML, DREG_L Register Type = (MAC0) MMOD1
MAC1 MML, DREG_L Register Type = (MAC0) MMOD1
MAC1 MML, DREG_E Register Type = (MAC0) MMODE
MAC1 MML, DREG_E Register Type = (MAC0) MMODE
MAC1 MML, DREG_E Register Type = (MAC0) MMODE
MAC1 MML, DREG_E Register Type = (MAC0) MMODE

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (Dual 16 x 16-Bit MAC (ParaMac16AndMac16), Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv), Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16))

ParaMac16AndMac16WithMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the the 16 x 16-Bit MAC (Mac16) instruction (using MAC1) and an instance of the 16 x 16-Bit

MAC with Move to Register (Mac16WithMv) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16AndMac16WithMv Example

```
a1 += r6.h * r4.l (fu) , r3.l = ( a0 = r3.h * r2.h ) ;
```

Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_H Register Type = (MAC1) MML, MAC0 MMOD1
DREG_H Register Type = (MAC1) MML, MAC0 MMOD1
DREG_H Register Type = (MAC1) MML, MAC0 MMOD1
DREG_H Register Type = (MAC1) MML, MAC0 MMOD1
DREG_O Register Type = (MAC1) MML, MAC0 MMODE
DREG_O Register Type = (MAC1) MML, MAC0 MMODE
DREG_O Register Type = (MAC1) MML, MAC0 MMODE
DREG_O Register Type = (MAC1) MML, MAC0 MMODE

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (Dual 16 x 16-Bit MAC (ParaMac16AndMac16), Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv), Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv))

ParaMac16WithMvAndMac16 Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction (using MAC1) and an instance of the 16 x 16-Bit MAC (Mac16) (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16WithMvAndMac16 Example

```
r3.h = (a1 += r6.h * r4.l) (fu) , a0 = r3.h * r2.h ;
```

Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1
DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1
DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1
DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1
DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE
DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE
DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE
DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE

Abstract

This dual multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second MAC operation occurs in parallel.

See Also (Dual 16 x 16-Bit MAC (ParaMac16AndMac16), Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv), Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16))

ParaMac16WithMvAndMac16WithMv Description

The dual multiply and multiply-accumulate to accumulator (with move) instruction is a dual (two instances issued in parallel) of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction. For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation input operands. The instructions must NOT use the same data registers for results.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16WithMvAndMac16WithMv Example

```
r1 = (a1 = r3 * r0) (m,fu) , r0 = (a0 = r0 * r3) (fu) ;
r3 = (a1 += r4 * r7) (m,is) , r2 = (a0 += r1 * r2) (is) ;
r1 = (a1 = r3 * r0) (m,fu) , r4 = (a0 -= r2 * r1) (iss2) ;
```

Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)

General Form

Multiply Accumulate (Dsp32Mac)
MAC1 MML, DREG_L Register Type = a0 MMOD1
MAC1 MML, DREG_L Register Type = a0 MMOD1
MAC1 MML, DREG_E Register Type = a0 MMODE
MAC1 MML, DREG_E Register Type = a0 MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMv))

ParaMac16AndMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the the 16 x 16-Bit MAC (Mac16) instruction (using MAC1) and *either* an instance of the Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL) instruction (using MAC0) *or* an instance of the Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16AndMv Example

```
a1 += r6.h * r4.l (fu) , r3.l = a0 ;
a1 += r6.h * r4.l (fu) , r2 = a0 ;
```

Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMv)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_H Register Type = (MAC1) MML, DREG_L Register Type = a0 MMOD1
DREG_H Register Type = (MAC1) MML, DREG_L Register Type = a0 MMOD1
DREG_O Register Type = (MAC1) MML, DREG_E Register Type = a0 MMODE
DREG_O Register Type = (MAC1) MML, DREG_E Register Type = a0 MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv))

ParaMac16WithMvAndMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with an instance of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction (using MAC1) and *either* an instance of the Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL) instruction (using MAC0) *or* an instance of the Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMac16WithMvAndMv Example

```
r1.h = (a1 += r6.h * r4.l) (fu) , r3.l = a0 ;
r1.h = (a1 += r6.h * r4.l) (fu) , r2 = a0 ;
r2 = (a1 += r6.h * r4.l) (fu) , r3.l = a0 ;
r0 = (a1 += r6.h * r4.l) (fu) , r2 = a0 ;
```

Dual 16 x 16-Bit Multiply (ParaMult16AndMult16)

General Form

Multiply with 3 operands (Dsp32Mult)
DREG_H Register Type = MUL1 MML, DREG_L Register Type = MUL0 MMOD1

Abstract

This instruction executes a two parallel multiply operations on 16-bit registers.

ParaMult16AndMult16 Description

The dual multiply 16-bit operands instruction is a dual (two instances issued in parallel) of the 16 x 16-Bit Multiply (Mult16) instruction. One of the parallel issue instructions executes on MAC1 with its results placed in a high half data register. The other parallel issue instruction executes on MAC0 with its results placed in a low half data register. For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMult16AndMult16 Example

```
r3.h = r6.h * r4.l (fu) , r3.l = r3.h * r2.h ;
```

Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_H Register Type = a1 MML, MAC0 MMOD1
DREG_H Register Type = a1 MML, MAC0 MMOD1
DREG_O Register Type = a1 MML, MAC0 MMODE
DREG_O Register Type = a1 MML, MAC0 MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv))

ParaMvAndMac16 Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with *either* an instance of the Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH) instruction (using MAC1) *or* an instance of the Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO) instruction (using MAC1) and an instance of the 16 x 16-Bit MAC (Mac16) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvAndMac16 Example

```
r3.l = a1 , a0 += r6.h * r4.l (fu) ;
r2 = a1 , a0 += r6.h * r4.l (fu) ;
```

Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)

General Form

Multiply Accumulate (Dsp32Mac)
DREG_H Register Type = a1 MML, DREG_L Register Type = (MAC0) MMOD1
DREG_H Register Type = a1 MML, DREG_L Register Type = (MAC0) MMOD1
DREG_O Register Type = a1 MML, DREG_E Register Type = (MAC0) MMODE
DREG_O Register Type = a1 MML, DREG_E Register Type = (MAC0) MMODE

Abstract

This dual move and multiply-accumulate instruction multiplies two 16-bit half word operands. Then, the instruction stores, adds, or subtracts the product into a designated accumulator register with saturation. By default, the instruction treats all operands as signed fractions with left-shift correction as required. A second (independent) move operation occurs in parallel with the MAC operation.

See Also (Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16))

ParaMvAndMac16WithMv Description

The dual multiply and multiply-accumulate to half register (with move) instruction is a parallel issue instruction with *either* an instance of the Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH) instruction (using MAC1) *or* an instance of the Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO) instruction (using MAC1) and an instance of the 16 x 16-Bit MAC with Move to Register (Mac16WithMv) instruction (using MAC0). For more information about instruction operation, see that instruction's reference page.

The parallel issue instructions operate independently and may use the same (or different) data registers for the computation operands.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

ParaMvAndMac16WithMv Example

```
r3.h = a1 , r1.h = (a0 += r6.h * r4.l) (fu) ;
r3 = a1 , r1.h = (a0 += r6.h * r4.l) (fu) ;
r3.h = a1 , r2 = (a0 += r6.h * r4.l) (fu) ;
r7 = a1 , r0 = (a0 += r6.h * r4.l) (fu) ;
```

Pointer Math Operations

These operations provide addition and/or subtract operations on pointer register and immediate value operands:

- 32-bit Add or Subtract (DagAdd32)
- 32-bit Add then Shift (DagAddSubShift)
- 32-bit Add or Subtract Constant (DagAddImm)
- 32-bit Add Shifted Pointer (PtrOp)
- Pointer Logical Shift (LShiftPtr)

32-bit Add or Subtract (DagAdd32)

General Form

Pointer Arithmetic Operations (Ptr2op)
PREG Register Type -= PREG Register Type
PREG Register Type += PREG Register Type (brev)
Compute with 3 operands (Comp3op)
PREG Register Type = PREG Register Type + PREG Register Type
DAG Arithmetic (DAGModIm)
IREG Register Type += MREG Register Type
IREG Register Type += MREG Register Type (brev)
IREG Register Type -= MREG Register Type

Abstract

This instruction adds or subtracts two pointer registers.

See Also (32-bit Add then Shift (DagAddSubShift), 32-bit Add or Subtract Constant (DagAddImm), 32-bit Add Shifted Pointer (PtrOp))

DagAdd32 Description

The DAG AddSub32 instruction adds or subtracts source pointer registers and places the result in a destination pointer register.

The instruction versions that explicitly modify an index register (`Ireg`) support optional circular buffering. For more information, see Addressing Circular Buffers in the Address Arithmetic Unit (AAU) chapter. Unless circular buffering is desired, disable this feature prior to issuing this instruction by clearing the length register (`Lreg`) corresponding to the `Ireg` used in this instruction. For example, if using the `i2` to increment your address pointer, first clear `l2` to disable circular buffering. Failure to explicitly clear the corresponding `Lreg` beforehand can result in unexpected `Ireg` values.

The circular address buffer registers (index, length, and base) are not initialized automatically by processor reset. The recommended operation is that user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes these registers later, if needed.

When the bit reverse carry adder (BREV) is specified in the instruction syntax, the carry bit is *propagated from left-to-right*, as shown in the ***Bit Addition Flow for the Bit Reverse (BREV) Case*** figure, instead of being *propagated from right-to-left* (default operation). When bit reversal is used on the index register version of this instruction, circular buffering is disabled to support operand addressing for FFT, DCT, and DFT algorithms. The pointer register version of this instruction does not support circular buffering.

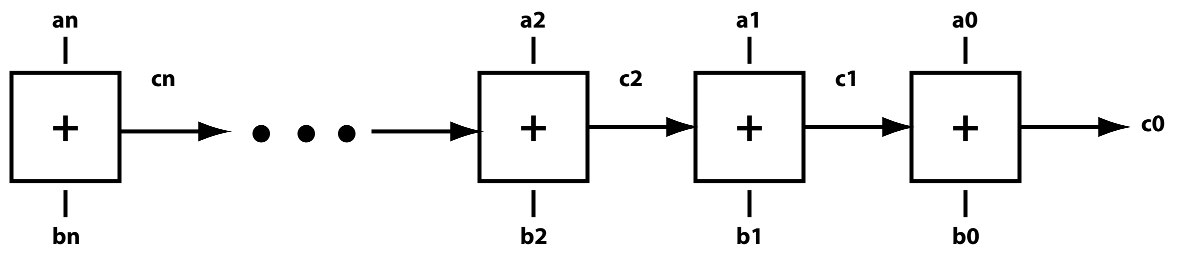


Figure 8-7: Bit Addition Flow for the Bit Reverse (BREV) Case

This instruction has a *special application*, regarding load or store operations. Typically, programs use the index register and pointer register versions of this instruction to increment or decrement indirect address pointers for load or store operations.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

DagAdd32 Example

```
p5 = p3 + p0 ; /* dest_Preg = src1_Preg + src0_Preg */
p3 -= p0 ; /* dest_Preg_new = dest_Preg_old - src_Preg */
i1 -= m2 ; /* dest_Ireg_new = dest_Ireg_old - src_Mreg */
p3 += p0 (brev) ; /* dest_Preg_new = dest_Preg_old + src_Preg (bit reversed carry, only) */
i1 += m1 ; /* dest_Ireg_new = dest_Ireg_old + src_Mreg */
i0 += m0 (brev) ; /* optional bit reverse carry, only */
```

32-bit Add or Subtract Constant (DagAddImm)

General Form

Destructive Binary Operations, preg with 7bit immediate (CompI2opP)
PREG Register Type += imm7 Register Type
DAG Arithmetic (DAGModIk)
IREG Register Type += 2
IREG Register Type -= 2
IREG Register Type += 4
IREG Register Type -= 4

Abstract

This instruction allows the user to add a constant to a register.

See Also (32-bit Add or Subtract (DagAdd32), 32-bit Add then Shift (DagAddSubShift), 32-bit Add Shifted Pointer (PtrOp))

DagAddImm Description

The DAG AddImm instruction adds or subtracts a source pointer registers and a constant value, then places the result in a destination pointer register.

The instruction versions that explicitly modify an index register (`Ireg`) support optional circular buffering. For more information, see Addressing Circular Buffers in the Address Arithmetic Unit (AAU) chapter. Unless circular buffering is desired, disable this feature prior to issuing this instruction by clearing the length register (`Lreg`) corresponding to the `Ireg` used in this instruction. For example, if using the `i2` to increment your address pointer, first clear `l2` to disable circular buffering. Failure to explicitly clear the corresponding `Lreg` beforehand can result in unexpected `Ireg` values.

The circular address buffer registers (index, length, and base) are not initialized automatically by processor reset. The recommended operation is that user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes these registers later, if needed

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

DagAddImm Example

```
p5 += -8 ; /* Preg = Preg + constant */
i0 += 2 ;  /* Ireg = Ireg + 2 */
i1 += 4 ;  /* Ireg = Ireg + 4 */
i2 -= 2 ;  /* Ireg = Ireg - 2 */
i0 -= 4 ;  /* Ireg = Ireg - 4 */
```

32-bit Add then Shift (DagAddSubShift)

General Form

Pointer Arithmetic Operations (Ptr2op)
PREG Register Type = (PREG Register Type + PREG Register Type) << 1
PREG Register Type = (PREG Register Type + PREG Register Type) << 2

Abstract

This instruction adds then shift left one or two places. Saturation is not supported.

See Also (32-bit Add or Subtract (DagAdd32), 32-bit Add or Subtract Constant (DagAddImm), 32-bit Add Shifted Pointer (PtrOp))

DagAddSubShift Description

The add with shift instruction adds two source pointer register, then applies a one- or two-bit logical shift left. The left shift accomplishes a x2 or x4 multiplication on sign-extended numbers.

The add with shift instruction does not intrinsically modify values that are strictly input. However, the `dest_reg` serves as an input as well as the result, so the `dest_reg` is intrinsically modified.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

DagAddSubShift Example

```
p3 = (p3 + p2) << 1 ;
/* dest_reg = (dest_reg + src_reg) x 2 */
/* p3 = (p3 + p2) * 2 */

p3 = (p3 + p2) << 2 ;
/* dest_reg = (dest_reg + src_reg) x 4 (a) */
/* p3 = (p3 + p2) * 4 */
```

32-bit Add Shifted Pointer (PtrOp)

General Form

Compute with 3 operands (Comp3op)
PREG Register Type = PREG Register Type + (PREG Register Type << 1)
PREG Register Type = PREG Register Type + (PREG Register Type << 2)

Abstract

This instruction adds or subtracts pointer and DAG registers.

See Also (32-bit Add or Subtract (DagAdd32), 32-bit Add then Shift (DagAddSubShift), 32-bit Add or Subtract Constant (DagAddImm))

PtrOp Description

The shift with add instruction combines a one- or two-bit logical shift left with an addition operation.

The instruction provides a shift-then-add method that supports a rudimentary multiplier sequence useful for array pointer manipulation.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

PtrOp Example

```
p3 = p0 + (p3 << 1) ;
/* p3 = (p3 * 2) + p0 */
/* adder_pntr + (src_pntr * 2) */

p3 = p0 + (p3 << 2) ;
/* p3 = (p3 * 4) + p0 */
/* adder_pntr + (src_pntr * 4) */
```

Pointer Logical Shift (LShiftPtr)

General Form

Pointer Arithmetic Operations (Ptr2op)
PREG Register Type = PREG Register Type << 2
PREG Register Type = PREG Register Type << 1
PREG Register Type = PREG Register Type >> 2
PREG Register Type = PREG Register Type >> 1

Abstract

This instruction shifts a pointer register by the specified number of bits.

LShiftPtr Description

The logical shift pointer instruction logically shifts a pointer register by a specified distance and direction.

Logical shifts discard any bits shifted out of the register and backfill vacated bits with zeros.

The logical shift pointer instruction does not implicitly modify the input *src_pntr* value. However, the dest_pntr can be the same pointer register as *src_pntr*. Doing so explicitly modifies the source register.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LShiftPtr Example

```
p3 = p2 >> 1 ; /* pointer right shift by 1 */
p3 = p3 >> 2 ; /* pointer right shift by 2 */
p4 = p5 << 1 ; /* pointer left shift by 1 */
p0 = p1 << 2 ; /* pointer left shift by 2 */
```

Rotate Operations

These operations provide bitwise rotate operations on register and immediate value operands:

- 32-Bit Rotate (Shift_Rot32)

- Accumulator Rotate (Shift_RotAcc)

32-Bit Rotate (Shift_Rot32)

General Form

Shift (Dsp32Shf)
DREG Register Type = rot DREG Register Type by DREG_L Register Type
Shift Immediate (Dsp32ShfImm)
DREG Register Type = rot DREG Register Type by imm6 Register Type

Abstract

This instruction rotates the a register through the CC bit a specified distance and direction. The CC bit is in the rotate chain.

Shift_Rot32 Description

The rotate data register instruction rotates a data register through the `CC` bit a specified distance and direction. The `CC` bit is in the rotate chain. Consequently, the first value rotated into the register is the initial value of the `CC` bit.

Rotation shifts all the bits either right or left. Each bit that rotates out of the register (the LSB for rotate right or the MSB for rotate left) is stored in the `CC` bit, and the `CC` bit is stored into the bit vacated by the rotate on the opposite end of the register.

```
If                          31                                      0
D-register:                 1010 1111 0000 0000 0000 0000 0001 1010
CC bit:                     N (“1” or “0”)

Rotate left 1 bit           31                                      0
D-register:                 0101 1110 0000 0000 0000 0000 0011 010N
CC bit:                     1

Rotate left 1 bit again     31                                      0
D-register:                 1011 1100 0000 0000 0000 0000 0110 10N1
CC bit: 0

If                          31                                      0
D-register:                 1010 1111 0000 0000 0000 0000 0001 1010
CC bit:                     N (“1” or “0”)

Rotate right 1 bit          31                                      0
D-register:                 N101 0111 1000 0000 0000 0000 0000 1101
CC bit:                     0

Rotate right 1 bit again    31                                      0
D-register:                 0N10 1011 1100 0000 0000 0000 0000 0110
CC bit:                     1
```

Figure 8-8: Left-versus-Right Bit Rotation Example

The sign of the rotate magnitude determines the direction of the rotation.

- Positive rotate magnitudes produce Left rotations.
- Negative rotate magnitudes produce Right rotations.

Valid rotate magnitudes are –32 through +31, zero included. The rotate instruction masks and ignores bits that are more significant than those allowed. The distance is determined by the lower 6 bits (sign extended) of the *shift_magnitude*.

Unlike shift operations, the rotate instruction loses no bits of the source register data. Instead, it rearranges them in a circular fashion. However, the last bit rotated out of the register remains in the CC bit, and is not returned to the register. Because rotates are performed all at once and not one bit at a time, rotating one direction or another regardless of the rotate magnitude produces no advantage. For instance, a rotate right by two bits is no more efficient than a rotate left by 30 bits. Both methods produce identical results in identical execution time.

This instruction rotates all 32 bits of the data register.

The instruction does not implicitly modify the *src_reg* values. Optionally, *dest_reg* can be the same data register as *src_reg*. Doing this explicitly modifies the source register.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift_Rot32 Example

```
r4 = rot r1 by 8 ; /* rotate left (Dreg = ROT Dreg BY imm6) */
r4 = rot r1 by -5 ; /* rotate right */
```

Accumulator Rotate (Shift_RotAcc)

General Form

Shift (Dsp32Shf)
a0 = rot a0 by DREG_L Register Type

a1 = rot a1 by DREG_L Register Type
Shift Immediate (Dsp32ShfImm)
a0 = rot a0 by imm6 Register Type
a1 = rot a1 by imm6 Register Type

Abstract

This instruction rotates the accumulator through the CC bit a specified distance and direction. The CC bit is in the rotate chain.

Shift_RotAcc Description

This instruction rotates the accumulator through the CC bit a specified distance and direction. The CC bit is in the rotate chain. Consequently, the first value rotated into the register is the initial value of the CC bit and the last bit rotated out ends up in CC. The sign of the rotate magnitude determines the direction of the rotation.

- Positive rotates left
- Negative rotates right

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Shift Operations

These operations provide arithmetic or logical shift operations on register and immediate value operands:

- 16-Bit Arithmetic Shift (AShift16)
- Vectored 16-Bit Arithmetic (AShift16Vec)
- Accumulator Arithmetic Shift (AShiftAcc)
- 32-Bit Arithmetic Shift (AShift32)
- 16-Bit Logical Shift (LShift16)
- Vectored 16-Bit Logical Shift (LShift16Vec)
- 32-Bit Logical Shift (LShift)

- Accumulator Logical Shift (LShiftA)

16-Bit Arithmetic Shift (AShift16)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = ashift DREG_L Register Type by DREG_L Register Type
DREG_L Register Type = ashift DREG_H Register Type by DREG_L Register Type
DREG_H Register Type = ashift DREG_L Register Type by DREG_L Register Type
DREG_H Register Type = ashift DREG_H Register Type by DREG_L Register Type
DREG_L Register Type = ashift DREG_L Register Type by DREG_L Register Type (s)
DREG_L Register Type = ashift DREG_H Register Type by DREG_L Register Type (s)
DREG_H Register Type = ashift DREG_L Register Type by DREG_L Register Type (s)
DREG_H Register Type = ashift DREG_H Register Type by DREG_L Register Type (s)
Shift Immediate (Dsp32ShfImm)
DREG_L Register Type = DREG_L Register Type AHSH4
DREG_L Register Type = DREG_H Register Type AHSH4
DREG_H Register Type = DREG_L Register Type AHSH4
DREG_H Register Type = DREG_H Register Type AHSH4
DREG_L Register Type = DREG_L Register Type AHSH4S
DREG_L Register Type = DREG_H Register Type AHSH4S
DREG_H Register Type = DREG_L Register Type AHSH4S
DREG_H Register Type = DREG_H Register Type AHSH4S

Abstract

This instruction shifts left or right and preserves the sign bit. For right shifts, the sign bit back-fills the left-most bit vacated by the shift. For left shifts, if the shift causes the sign bit to be lost, the result will saturate to the maximum positive or negative value depending on the lost sign bit.

See Also (Vectored 16-Bit Arithmetic (AShift16Vec))

AShift16 Description

The arithmetic shift low/high half data register destination instruction shifts a register contents a specified distance (`shift_magnitude`) and direction (based on syntax and/or `shift_magnitude` sign) while preserving the sign bit of the original number. This instruction provides arithmetic shift right, logical shift left, and arithmetic shift left (with saturation) operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on *The Science Dictionary* site are helpful.

The versions of this instruction using `ashift` syntax support the following shift operations:

- For a positive *`shift_magnitude`*, `ashift` produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative *`shift_magnitude`*, `ashift` produces a logical shift left, but does not guarantee sign bit preservation. If the negative *`shift_magnitude`* is too large, the `ashift` operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the `ashift` operation as a multiplication or division operation. Viewed this way, the *`shift_magnitude`* is the power of 2 multiplied by the *`src_reg`* number. Positive magnitudes cause multiplication ($N \times 2^n$), and negative magnitudes produce division ($N \times 2^{-n}$ or $N / 2^n$).

The versions of this instruction using >>> syntax only support arithmetic right shift operations using positive *`shift_magnitude`* values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive *`shift_magnitude`* values.

The versions of this instruction using << with `(s)` syntax only support arithmetic shift left operations (with saturation) using positive *`shift_magnitude`* values.

The ***Arithmetic Shift (16 Bit Destination Register) Operations*** table provide more detailed information about arithmetic shift operations.

Table 8-10: Arithmetic Shift (16 Bit Destination Register) Operations

Syntax	Description
">>>", << (with saturation), and `ashift`	The value in *`src_reg`* is shifted by the number of places specified in *`shift_magnitude`*, and the result is stored into *`dest_reg`*. The `ashift` versions can shift 16-bit Dreg_lo_hi registers by up to –16 through +15 places.

The *`dest_reg`* and *`src_reg`* may be a 16-bit half data register.

For 16-bit *`src_reg`*, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for half-word registers.

The half data register versions of this instruction do not implicitly modify the *`src_reg`* values. Optionally, *`dest_reg`* may be the same data register as *`src_reg`*. Doing this explicitly modifies the source register.

Where permitted (optional) or required the saturation (s) option applies saturation of the result. For shift operations *without saturation enabled*, values may be left-shifted so far that all the sign bits overflow and are lost. For shift operations *with saturation enabled*, a left shift that would otherwise shift nonsign bits off the left-hand side saturates to the maximum positive or negative value instead. The result always keeps the same sign as the pre-shifted value when saturation is enabled.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instructions*** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShift16 Example

```
/* AShift16 syntax summary */
/* Dreg_lo_hi = ashift Dreg_lo_hi BY Dreg_lo (optional_sat) ; arithmetic or logical shift
with optional saturation*/
/* Dreg_lo_hi = Dreg_lo_hi >>> uimm4 (optional_sat) ; arithmetic shift right with optional
saturation*/
/* Dreg_lo_hi = Dreg_lo_hi <<< uimm4 ; logical shift left*/
/* Dreg_lo_hi = Dreg_lo_hi <<< 0 ; logical shift left*/
/* Dreg_lo_hi = Dreg_lo_hi << uimm4 (s) ; arithmetic shift left with saturation*/
/* Dreg_lo_hi = Dreg_lo_hi << 0 (s) ; arithmetic shift left with saturation*/
/* AShift16 syntax examples */
r3.l = r0.h >>> 7 ; /* arithmetic right shift, half-word */
r3.h = r0.h >>> 5 ; /* same as above; any combination of upper and lower half-words is
supported */
r3.l = r0.h >>> 7(s) ; /* arithmetic right shift, half-word, saturated */
r3.l = r0.h << 12 (s) ; /* arithmetic left shift */
r3.l = ashift r0.h by r7.l ; /* shift, half-word */
r3.h = ashift r0.l by r7.l ;
r3.h = ashift r0.h by r7.l ;
r3.l = ashift r0.l by r7.l ;
r3.l = ashift r0.h by r7.l(s) ; /* shift, half-word, saturated */
r3.h = ashift r0.l by r7.l(s) ; /* shift, half-word, saturated */
r3.h = ashift r0.h by r7.l(s) ;
r3.l = ashift r0.l by r7.l (s) ;
/* If r0.h = -64, then performing . . . */
r3.h = r0.h >>> 4 ;
/* . . . produces r3.h = -4, preserving the sign */
```

Vectored 16-Bit Arithmetic (AShift16Vec)

General Form

Shift (Dsp32Shf)
DREG Register Type = ashift DREG Register Type by DREG_L Register Type (v)
DREG Register Type = ashift DREG Register Type by DREG_L Register Type (v,s)
Shift Immediate (Dsp32ShfImm)
DREG Register Type = DREG Register Type AHSH4 (v)
DREG Register Type = DREG Register Type AHSH4VS

Abstract

This instruction shifts a 16-bit vector left or right by the value in the XOP register. When shifting right, the sign bit will be replicated. If saturation is specified, ASHIFT lefts will saturate if any of the bits shifted off do not match the original sign bit.

See Also (16-Bit Arithmetic Shift (AShift16))

AShift16Vec Description

The arithmetic shift data register destination (vector) instruction performs two independent shifts, shifting a contents of a register's low half and high half a specified distance (`shift_magnitude`) and direction (based on syntax and/or `shift_magnitude` sign) while preserving the sign bits of the original numbers. Although the two half-word registers are shifted at the same time, the two numbers are kept separate. This instruction provides arithmetic shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on ***The Science Dictionary*** site are helpful.

The versions of this instruction using `ashift` syntax support the following shift operations:

- For a positive `shift_magnitude`, `ashift` produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative `shift_magnitude`, `ashift` produces a logical shift left, but does not guarantee sign bit preservation. If the negative `shift_magnitude` is too large, the `ashift` operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the `ashift` operation as a multiplication or division operation. Viewed this way, the `shift_magnitude` is the power of 2 multiplied by the `src_reg` number. Positive magnitudes cause multiplication ($N \times 2^n$), and negative magnitudes produce division ($N \times 2^{-n}$ or $N / 2^n$).

The versions of this instruction using >>> syntax only support arithmetic right shift operations using positive `shift_magnitude` values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive `shift_magnitude` values.

The versions of this instruction using << with `(v,s)` syntax only support logical shift left operations (with saturation) using positive `shift_magnitude` values.

The ***Arithmetic Shift (16 Bit Destination Register) Operations*** table provide more detailed information about arithmetic shift operations.

Table 8-11: Arithmetic Shift (16 Bit Destination Register) Operations

Syntax	Description
">>>", << (with saturation), and `ashift`	The value in `src_reg` is shifted by the number of places specified in `shift_magnitude`, and the result is stored into `dest_reg`. The `ashift` versions can shift the two 16-bit halves of a Dreg (independently) by up to –16 through +15 places.

The `dest_reg` and `src_reg` may be a 132-bit register.

For each 16-bit half of the `src_reg`, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for the two half-word sections of the word registers, independently.

The data register versions of this instruction always modify the `src_reg` values.

Where permitted (optional) or required the saturation (s) option applies saturation of the result. For shift operations *without saturation enabled*, values may be left-shifted so far that all the sign bits overflow and are lost. For shift operations *with saturation enabled*, a left shift that would otherwise shift nonsign bits off the left-hand side saturates to the maximum positive or negative value instead. The result always keeps the same sign as the pre-shifted value when saturation is enabled.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0

...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShift16Vec Example

```
/* AShift16Vec syntax summary */
Dreg1 = ashift Dreg1 by Dreg0_lo (v) /* arithmetic/logical shift, vector (dual) */
Dreg1 = ashift Dreg1 by Dreg0_lo (v,s) /* arithmetic/logical shift, vector (dual) */
Dreg = Dreg <<< 0 (v) /* logical shift left, vector (dual) */
Dreg = Dreg <<< UImm4 (v) /* logical shift left, vector (dual) */
Dreg = Dreg >>> UImm4N (v) /* arithmetic shift right, vector (dual) */
Dreg = Dreg << 0 (v,s) /* logical shift left with saturation, vector (dual) */
Dreg = Dreg << UImm4 (v,s) /* logical shift left with saturation, vector (dual) */
Dreg = Dreg >>> UImm4N (v,s) /* arithmetic shift right with saturation, vector (dual) */
/* AShift16Vec syntax examples */
r4=r5>>3 (v) ; /* logical right shift immediate R5.H and R5.L by 3 bits */
r4=r5<<3 (v) ; /* logical left shift immediate R5.H and R5.L by 3 bits */
r2=lshift r7 by r5.l (v) ;
/* logically shift (right or left, depending on sign of r5.l) R7.H and R7.L by magnitude of
R5.L */
```

32-Bit Arithmetic Shift (AShift32)

General Form

ALU Binary Operations (ALU2op)
DREG Register Type >>>= DREG Register Type
Logic Binary Operations (Logi2Op)
DREG Register Type >>>= uimm5 Register Type
Shift (Dsp32Shf)
DREG Register Type = ashift DREG Register Type by DREG_L Register Type
DREG Register Type = ashift DREG Register Type by DREG_L Register Type (s)
Shift Immediate (Dsp32ShfImm)
DREG Register Type = DREG Register Type ASH5
DREG Register Type = DREG Register Type ASH5S

Abstract

This instruction shifts left or right and preserves the sign bit. For right shifts, the sign bit back-fills the left-most bit vacated by the shift. For left shifts, if the shift causes the sign bit to be lost, the result will saturate to the maximum positive or negative value depending on the lost sign bit.

See Also (Accumulator Arithmetic Shift (AShiftAcc))

AShift32 Description

The arithmetic shift data register destination instruction shifts a register contents a specified distance (`shift_magnitude`) and direction (based on syntax and/or `shift_magnitude` sign) while preserving the sign bit of the original number. This instruction provides arithmetic shift right, logical shift left, and arithmetic shift left (with saturation) operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on ***The Science Dictionary*** site are helpful.

The versions of this instruction using `ashift` syntax support the following shift operations:

- For a positive `shift_magnitude`, `ashift` produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative `shift_magnitude`, `ashift` produces a logical shift left, but does not guarantee sign bit preservation. If the negative `shift_magnitude` is too large, the `ashift` operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the `ashift` operation as a multiplication or division operation. Viewed this way, the `shift_magnitude` is the power of 2 multiplied by the `src_reg` number. Positive magnitudes cause multiplication ($N \times 2^n$), and negative magnitudes produce division ($N \times 2^{-n}$ or $N / 2^n$).

The versions of this instruction using >>>= and >>> syntax only support arithmetic right shift operations using positive `shift_magnitude` values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive `shift_magnitude` values.

The versions of this instruction using << with `(s)` syntax only support arithmetic shift left operations (with saturation) using positive `shift_magnitude` values.

The ***Arithmetic Shift (32 Bit Destination Register) Operations*** table provide more detailed information about arithmetic shift operations.

Table 8-12: Arithmetic Shift (32 Bit Destination Register) Operations

Syntax	Description
>>>=	The value in `dest_reg` is right-shifted by the number of places specified by `shift_magnitude`. The data size is always 32 bits long. The entire 32 bits of the `shift_magnitude` determine the shift value. Shift magnitudes larger than 0x1F result in either 0x00000000 (when the input value is positive) or 0xFFFFFFFF (when the input value is negative).
">>>", << (with saturation), and `ashift`	The value in `src_reg` is shifted by the number of places specified in `shift_magnitude`, and the result is stored into `dest_reg`. The `ashift` versions can shift 32-bit Dreg registers by up to –32 through +31 places.

The `dest_reg` and `src_reg` may be a 32-bit register.

For 32-bit `src_reg`, valid shift magnitudes are –32 through +31, zero included.

The data register versions of this instruction shift 32 bits for word registers.

The data register versions of this instruction do not implicitly modify the `src_reg` values. Optionally, `dest_reg` can be the same data register as `src_reg`. Doing this explicitly modifies the source register.

Where permitted (optional) or required the saturation (s) option applies saturation of the result. For shift operations *without saturation enabled*, values may be left-shifted so far that all the sign bits overflow and are lost. For shift operations *with saturation enabled*, a left shift that would otherwise shift nonsign bits off the left-hand side saturates to the maximum positive or negative value instead. The result always keeps the same sign as the pre-shifted value when saturation is enabled.

See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

The versions of this instruction using >>>, <<<, <<, and `ashift` syntax are ***32-bit instructions***, which can sometimes save execution time (over a 16-bit encoded instruction) because they can be issued in parallel with certain other instructions.

The versions of this instruction using >>>= syntax are ***16-bit instructions*** (which takes up less memory space over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShift32 Example

```
/* AShift32 syntax summary */
/* Dreg >>>= Dreg ; arithmetic shift right */
/* Dreg >>>= UImm5 ; arithmetic shift right */
/* Dreg = ashift Dreg by Dreg_lo (optional_sat) ; arithmetic/logical shift with optional
saturation */
/* Dreg = Dreg <<< 0 ; logical shift left */
/* Dreg = Dreg <<< UImm5 ; logical shift left */
/* Dreg = Dreg >>> UImm5N ; arithmetic shift right */
/* Dreg = Dreg << 0 (s) ; arithmetic shift left with saturation */
/* Dreg = Dreg << UImm5 (s) ; arithmetic shift left with saturation */
/* Dreg = Dreg >>> UImm5 ; arithmetic shift right */
/* AShift32 syntax examples */
r0 >>>= 19 ; /* 16-bit instruction length arithmetic right shift */
r0 >>>= r2 ; /* 16-bit instruction length arithmetic right shift */
r5 = r2 << 24 (s) ; /* arithmetic left shift */
r4 = ashift r2 by r7.l ; /* shift, word */
r4 = ashift r2 by r7.l (s) ; /* shift, word, saturated */
```

Accumulator Arithmetic Shift (AShiftAcc)

General Form

Shift (Dsp32Shf)
a0 = ashift a0 by DREG_L Register Type
a1 = ashift a1 by DREG_L Register Type
Shift Immediate (Dsp32ShfImm)
a0 = a0 ASH5
a1 = a1 ASH5

Abstract

This instruction shifts left or right and preserves the sign bit. For right shifts, the sign bit back-fills the left-most bit vacated by the shift.

See Also (32-Bit Arithmetic Shift (AShift32))

AShiftAcc Description

The arithmetic shift accumulator register destination instruction shifts a register contents a specified distance (`shift_magnitude`) and direction (based on syntax and/or `shift_magnitude` sign) while preserving the sign bit of the original number. This instruction provides arithmetic shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on *The Science Dictionary* site are helpful.

The versions of this instruction using `ashift` syntax support the following shift operations:

- For a positive `shift_magnitude`, `ashift` produces an arithmetic shift right with sign bit preservation. The sign bit value back-fills the left-most bit positions vacated by the arithmetic shift right.
- For a negative `shift_magnitude`, `ashift` produces a logical shift left, but does not guarantee sign bit preservation. If the negative `shift_magnitude` is too large, the `ashift` operation saturates the destination register. A logical shift left that would otherwise lose non-sign bits off the left-hand side saturates to the maximum positive or negative value instead.

NOTE: One may view the `ashift` operation as a multiplication or division operation. Viewed this way, the `shift_magnitude` is the power of 2 multiplied by the `src_reg` number. Positive magnitudes cause multiplication (N x 2^n), and negative magnitudes produce division (N x 2^{-n} or N / 2^n).

The versions of this instruction using >>> syntax only support arithmetic right shift operations using positive `shift_magnitude` values.

The versions of this instruction using <<< syntax only support logical left shift operations using positive `shift_magnitude` values.

The ***Arithmetic Shift (Accumulator Destination Register) Operations*** table provide more detailed information about arithmetic shift operations.

Table 8-13: Arithmetic Shift (Accumulator Destination Register) Operations

Syntax	Description
">>>", <<<, and `ashift`	The value in `src_reg` is shifted by the number of places specified in `shift_magnitude`, and the result is stored into `dest_reg`. The `ashift` versions can shift 40-bit accumulator registers by up to –32 through +31 places.

The `dest_reg` and `src_reg` may be a 40-bit register.

For 40-bit `src_reg`, valid shift magnitudes are –32 through +31, zero included.

The accumulator versions shift all 40 bits of those registers.

The accumulator versions of this instruction always implicitly modify the `src_reg` values.

This is a ***32-bit instruction*** and can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	*AV0S*	*AV0*
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AShiftAcc Example

```
/* AShiftAcc syntax summary */
/* a0 = ashift a0 by Dreg_lo ; arithmetic/logical shift */
/* a1 = ashift a1 by Dreg_lo ; arithmetic/logical shift */
/* a0 = a0 <<< 0 ; logical shift left */
/* a0 = a0 <<< UImm5 ; logical shift left */
/* a0 = a0 >>> UImm5N ; arithmetic shift right */
/* a1 = a1 <<< 0 ; logical shift left */
/* a1 = a1 <<< UImm5 ; logical shift left */
/* a1 = a1 >>> UImm5N ; arithmetic shift right */
/ * AShiftAcc syntax examples */
a0 = a0 >>> 1 ; /* arithmetic right shift, accumulator */
a0 = ashift a0 by r7.l ; /* shift, accumulator */
a1 = ashift a1 by r7.l ; /* shift, accumulator */
```

16-Bit Logical Shift (LShift16)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = lshift DREG_L Register Type by DREG_L Register Type
DREG_L Register Type = lshift DREG_H Register Type by DREG_L Register Type
DREG_H Register Type = lshift DREG_L Register Type by DREG_L Register Type
DREG_H Register Type = lshift DREG_H Register Type by DREG_L Register Type
Shift Immediate (Dsp32ShfImm)
DREG_L Register Type = DREG_L Register Type LHSH4
DREG_L Register Type = DREG_H Register Type LHSH4
DREG_H Register Type = DREG_L Register Type LHSH4
DREG_H Register Type = DREG_H Register Type LHSH4

Abstract

This instruction shifts a register half by the specified number of bits and returns the shifted value.

See Also (Vectored 16-Bit Logical Shift (LShift16Vec))

LShift16 Description

The logical shift low/high half data register destination instruction shifts a register contents a specified distance (`shift_magnitude`) and direction (based on syntax and/or `shift_magnitude` sign), discarding any bits shifted out of the register and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on ***The Science Dictionary*** site are helpful.

The versions of this instruction using `lshift` syntax support the following shift operations:

- For a positive `shift_magnitude`, `lshift` produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative `shift_magnitude`, `lshift` produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >> syntax only support logical shift right operations using positive `shift_magnitude` values.

The versions of this instruction using << syntax only support logical shift left operations using positive `shift_magnitude` values.

The ***Logical Shift (16 Bit Destination Register) Operations*** table provide more detailed information about logical shift operations.

Table 8-14: Logical Shift (16 Bit Destination Register) Operations

Syntax	**Description**
>>, <<, and `lshift`	The value in `src_reg` is shifted by the number of places specified in `shift_magnitude`, and the result is stored into `dest_reg`. The `lshift` versions can shift 16-bit half data registers by up to –16 through +15 places.

The `dest_reg` and `src_reg` may be a 16-bit half data register.

For 16-bit `src_reg`, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for half-word registers.

The half data register versions of this instruction do not implicitly modify the `src_reg` values. Optionally, `dest_reg` may be the same data register as `src_reg`. Doing this explicitly modifies the source register.

This ***32-bit instructions*** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	***AN***	***AZ***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LShift16 Example

```
/* LShift16 syntax summary */
/* DDST_Lo_Hi = lshift DSRC1_Lo_Hi by DSRC0_Lo ; logical shift */
/* DDST_Lo_Hi = DSRC_Lo_Hi << 0 ; logical shift left */
/* DDST_Lo_Hi = DSRC_Lo_Hi << UImm4 ; logical shift lef */
/* DDST_Lo_Hi = DSRC_Lo_Hi >> UImm4N ; logical shift right */
/* LShift16 syntax examples */
r3.l = r0.l >> 4 ; /* logical shift right, half-word register */
r3.l = r0.h >> 4 ; /* logical shift right; half-word register combinations are arbitrary */
r3.h = r0.l << 12 ; /* logical shift left, half-word register */
r3.h = r0.h << 14 ; /* logical shift left; half-word register combinations are arbitrary */
r3.l = lshift r0.l by r2.l ; /* logical shift, direction controlled by sign of R2.L */
r3.h = lshift r0.l by r2.l ;
/* If r0.h = -64 (or 0xFFC0), then performing . . . */
r3.h = r0.h >> 4 ;
/* . . . produces r3.h = 0x0FFC (or 4092), losing the sign */
```

Vectored 16-Bit Logical Shift (LShift16Vec)

General Form

Shift (Dsp32Shf)
DREG Register Type = lshift DREG Register Type by DREG_L Register Type (v)
Shift Immediate (Dsp32ShfImm)
DREG Register Type = DREG Register Type LHSH4 (v)

Abstract

This instruction shifts a 16-bit vector left or right by the value in the XOP register.

See Also (16-Bit Logical Shift (LShift16))

LShift16Vec Description

The logical shift data register destination (vector) instruction performs two independent shifts, shifting a contents of a register's low half and high half a specified distance (`shift_magnitude`) and direction (based on syntax and/or `shift_magnitude` sign), discarding any bits shifted out of the two half registers and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on ***The Science Dictionary*** site are helpful.

The versions of this instruction using `lshift` syntax support the following shift operations:

- For a positive `shift_magnitude`, `lshift` produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative `shift_magnitude`, `lshift` produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >> syntax only support logical shift right operations using positive `shift_magnitude` values.

The versions of this instruction using << syntax only support logical shift left operations using positive `shift_magnitude` values.

The ***Logical Shift (16 Bit Destination Register) Operations*** table provide more detailed information about logical shift operations.

Table 8-15: Logical Shift (16 Bit Destination Register) Operations

Syntax	Description
>>, <<, and `lshift`	The value in `src_reg` is shifted by the number of places specified in `shift_magnitude`, and the result is stored into `dest_reg`. The `lshift` versions can shift 16-bit half data reg isters by up to –16 through +15 places.

The `dest_reg` and `src_reg` may be a 32-bit half data register. The shift operations are applied to the 16-bit half registers within the `src_reg`.

For 16-bit `src_reg`, valid shift magnitudes are –16 through +15, zero included.

The data register versions of this instruction shift 16 bits for half-word registers.

The half data register versions of this instruction do not implicitly modify the `src_reg` values. Optionally, `dest_reg` may be the same data register as `src_reg`. Doing this explicitly modifies the source register.

This ***32-bit instructions*** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0

...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LShift16Vec Example

```
/* LShift16Vec syntax summary */
/* DDST = lshift DSRC1 by DSRC0_L (v) ; logical shift, vector (dual) */
/* DDST = DSRC << 0 (v) ; logical shift left, vector (dual) */
/* DDST = DSRC << UImm4 (v) ; logical shift left, vector (dual) */
/* DDST = DSRC >> UImm4N (v) ; logical shift right, vector (dual) */
/* LShiftVec syntax examples */
r4=r5>>3 (v) ; /* logical right shift immediate R5.H and R5.L by 3 bits */
r4=r5<<3 (v) ; /* logical left shift immediate R5.H and R5.L by 3 bits */
r2=lshift r7 by r5.l (v) ;
/* logically shift (right or left, depending on sign of r5.l) R7.H and R7.L by magnitude of
R5.L */
```

32-Bit Logical Shift (LShift)

General Form

ALU Binary Operations (ALU2op)
DREG Register Type >>= DREG Register Type
DREG Register Type <<= DREG Register Type
Logic Binary Operations (Logi2Op)
DREG Register Type >>= uimm5 Register Type
DREG Register Type <<= uimm5 Register Type
Shift (Dsp32Shf)
DREG Register Type = lshift DREG Register Type by DREG_L Register Type
Shift Immediate (Dsp32ShfImm)
DREG Register Type = DREG Register Type LSH5

Abstract

This instruction shifts a register by the specified number of bits and returns the shifted value.

LShift Description

The logical shift data register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign), discarding any bits shifted out of the register and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on ***The Science Dictionary*** site are helpful.

The versions of this instruction using `lshift` syntax support the following shift operations:

- For a positive *`shift_magnitude`*, `lshift` produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative *`shift_magnitude`*, `lshift` produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >>= and >> syntax only support logical shift right operations using positive *`shift_magnitude`* values.

The versions of this instruction using <<= and << syntax only support logical shift left operations using positive *`shift_magnitude`* values.

The ***Logical Shift (16 Bit Destination Register) Operations*** table provide more detailed information about logical shift operations.

Table 8-16: Logical Shift (16 Bit Destination Register) Operations

Syntax	Description
>>= and <<=	The value in *`dest_reg`* is shifted by the number of places specified by *`shift_magnitude`*. The data size is always 32 bits long. The entire 32 bits of the *`shift_magnitude`* determine the shift value. Shift magnitudes larger than 0x1F produce a 0x00000000 result.
>>, <<, and `lshift`	The value in *`src_reg`* is shifted by the number of places specified in *`shift_magnitude`*, and the result is stored into *`dest_reg`*. The `lshift` versions can shift 32-bit data reg isters by up to –32 through +31 places.

The *`dest_reg`* and *`src_reg`* may be a 32-bit data register.

For 32-bit *`src_reg`*, valid shift magnitudes are –32 through +31, zero included.

The data register versions of this instruction shift 32 bits for word registers.

The data register versions of this instruction do not implicitly modify the *`src_reg`* values. Optionally, *`dest_reg`* may be the same data register as *`src_reg`*. Doing this explicitly modifies the source register.

The versions of this instruction using >>, <<, and `lshift` syntax are ***32-bit instructions***, which can sometimes save execution time (over a 16-bit encoded instruction) because they can be issued in parallel with certain other instructions.

The versions of this instruction using >>= and <<= syntax are ***16-bit instructions*** (which takes up less memory space over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LShift Example

```
/* LShift syntax summary */
/* DDST >>= DSRC ; logical shift right */
/* DDST <<= DSRC ; logical shift left */
/* DDST >>= SRCI ; logical shift right */
/* DDST <<= SRCI ; logical shift left */
/* DDST = lshift DSRC1 by DSRC0_L ; logical shift */
/* DDST = DSRC << 0 ; logical shift left */
/* DDST = DSRC << UImm5 ; logical shift left */
/* DDST = DSRC >> UImm5N ; logical shift right */
/* LShift syntax examples */
r3 >>= 17 ; /* logical shift right */
r3 <<= 17 ; /* logical shift left */
r3 = r6 >> 4 ; /* logical shift right, 32-bit word */
r3 = r6 << 4 ; /* logical shift left, 32-bit word */
r3 >>= r0 ; /* logical shift right */
r3 <<= r1 ; /* logical shift left */
```

Accumulator Logical Shift (LShiftA)

General Form

Shift (Dsp32Shf)
a0 = lshift a0 by DREG_L Register Type
a1 = lshift a1 by DREG_L Register Type
Shift Immediate (Dsp32ShfImm)
a0 = a0 LSH5
a1 = a1 LSH5

Abstract

This instruction shifts an accumulator left by the specified number of bits and returns the shifted value.

LShiftA Description

The logical shift accumulator register destination instruction shifts a register contents a specified distance (*shift_magnitude*) and direction (based on syntax and/or *shift_magnitude* sign), discarding any bits shifted out of the register and backfilling vacated bits with zeros. This instruction provides logical shift right and logical shift left operations.

NOTE: For information about the difference between arithmetic and logical shift operations, the definitions for Arithmetic Shift and Logical Shift on ***The Science Dictionary*** site are helpful.

The versions of this instruction using `lshift` syntax support the following shift operations:

- For a positive *shift_magnitude*, `lshift` produces an logical shift right, discarding any bits shifted out of the register and backfilling vacated bits with zeros.
- For a negative *shift_magnitude*, `lshift` produces a logical shift left, discarding any bits shifted out of the register and backfilling vacated bits with zeros.

NOTE: Shift magnitudes that exceed the size of the destination register produce all zeros in the result. For example, shifting a 16-bit register value by 20 bit places (a valid operation) produces 0x0000.

The versions of this instruction using >> syntax only support logical shift right operations using positive *shift_magnitude* values.

The versions of this instruction using << syntax only support logical shift left operations using positive *shift_magnitude* values.

The ***Logical Shift (Accumulator Destination Register) Operations*** table provide more detailed information about logical shift operations.

Table 8-17: Logical Shift (Accumulator Destination Register) Operations

Syntax	Description
>>, <<, and `lshift`	The value in *src_reg* is shifted by the number of places specified in *shift_magnitude*, and the result is stored into *dest_reg*. The `lshift` versions can shift 40-bit accumulator reg isters by up to –32 through +31 places.

The *dest_reg* and *src_reg* must be the same 40-bit accumulator register.

For 32-bit *src_reg*, valid shift magnitudes are –32 through +31, zero included.

The accumulator versions shift all 40 bits of those registers.

The accumulator versions of this instruction always implicitly modify the *src_reg* values.

This ***32-bit instructions*** can sometimes save execution time over a 16-bit encoded instruction, because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	***AV0***
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	***AN***	***AZ***
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

LShiftA Example

```
/* LShiftA syntax summary */
/* a0_1 = lshift a0_1 by DSRC0_L ; logical shift */
/* a0_1 = a0_1 << 0 ; logical shift left */
/* a0_1 = a0_1 << UImm5 ; logical shift left */
/* a0_1 = a0_1 >> UImm5N ; logical shift right */
/* LShiftA syntax examples */
a0 = a0 >> 7 ; /* Accumulator right shift */
a1 = a1 >> 25 ; /* Accumulator right shift */
a0 = a0 << 7 ; /* Accumulator left shift */
a1 = a1 << 14 ; /* Accumulator left shift */
a0 = lshift a0 by r7.l ;
a1 = lshift a1 by r7.l ;
```

Sequencer Instructions

The sequencer instructions provide program flow control operations, which execute on the ***control unit*** in the processor core. Users can take advantage of these instructions to force new values into the program counter and change program flow, branch conditionally, set up loops, and call and return from subroutines.

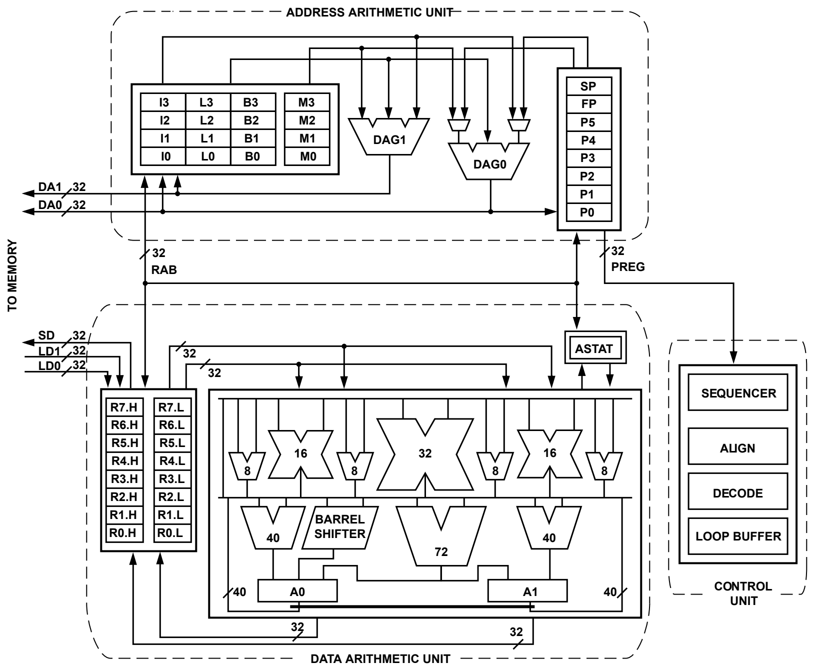


Figure 8-9: Blackfin+ Core Block Diagram

The operation types of sequencer instructions include:

- Branch Operations
- Control Code Bit Management Operations
- Event Management Operations
- Stack Operations
- Synchronization Operations

Branch Operations

These operations provide branching of program flow operations, unconditionally or with conditional operands:

- Conditional Jump Immediate (BrCC)
- Jump (Jump)
- Jump Immediate (JumpAbs)
- Call (Call)
- Return from Branch (Return)
- Hardware Loop Set Up (LoopSetup)

Conditional Jump Immediate (BrCC)

General Form

Conditional Branch PC relative on CC (BrCC)
if !cc jump imm10s2 Register Type
if !cc jump imm10s2 Register Type (bp)
if cc jump imm10s2 Register Type
if cc jump imm10s2 Register Type (bp)

Abstract

The Jump instruction forces a new value into the Program Counter (PC) to change program flow. This branches based on the value of the CC0 status bit. The BP option helps the processor improve branch instruction performance. The default is branch predicted-not-taken.

See Also (Jump (Jump), Jump Immediate (JumpAbs))

BrCC Description

The branch CC (conditional `jump`) instruction forces a new value into the Program Counter (PC) to change the program flow, based on the value of the `CC` bit.

- For `if CC`, a `CC` bit = 1 causes a branch to an address, computed by adding the signed, even offset to the current PC value.
- For `if !cc`, a `cc` bit = 0 causes a branch to an address, computed by adding the signed, even relative offset to the current PC value.

The range of valid offset values for the `jump` is –1024 through 1022.

The branch prediction option, `(bp)`, helps the processor improve branch instruction performance. The default is branch predicted-not-taken. By appending `(bp)` to the instruction, the branch becomes predicted-taken.

Typically, code analysis shows that a good default condition is to predict branch-taken for branches to a prior address (backwards branches), and to predict branch-not-taken for branches to subsequent addresses (forward branches).

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

BrCC Example

```
if cc jump 0xFFFFFE08 (bp) ;
/* offset is negative in 11 bits, so target address is a backwards branch, branch predicted
*/
```

```
if cc jump 0x0B4 ;
/* offset is positive, so target offset address is a forwards branch, branch not predicted
*/
if !cc jump 0xFFFFFC22 (bp) ;
/* negative offset in 11 bits, so target address is a backwards branch, branch predicted */
if !cc jump 0x120 ;
/* positive offset, so target address is a forwards branch, branch not predicted */
if cc jump dest_label ;
/* assembler resolved target, abstract offsets */
```

Jump (Jump)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
jump (PREG Register Type)
jump (pc+PREG Register Type)

Abstract

The Jump instruction forces a new value into the Program Counter (PC) to change program flow.

See Also (Conditional Jump Immediate (BrCC), Jump Immediate (JumpAbs))

Jump Description

The jump pointer instruction forces a new value into the Program Counter (PC) to change program flow.

The new address may be *indirect* (provided by a pointer register) or may be *indexed* (PC plus an offset provided by a pointer register). In the indirect and indexed versions of the instruction, the value in the pointer register (Preg) must be an even number (bit 0 of the register =0) to maintain 16-bit address alignment. Otherwise, an odd offset in the pointer register causes the processor to generate an address alignment exception.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Jump Example

```
jump (p5) ;
/* P5 contains the absolute address of the target */
jump (pc + p2) ;
/* P2 relative absolute address of the target and then a presentation of the absolute
values for target */
```

Jump Immediate (JumpAbs)

General Form

Unconditional Branch PC relative with 12bit offset (UJump)
jump.s imm12nxs2 Register Type
Call function with pcrel address (CallA)
jump.l imm24s2 Register Type
Jump/Call to 32-bit Immediate (Jump32)
jump.a buimm32 Register Type
jump bimm32 Register Type

Abstract

The Jump instruction forces a new value into the Program Counter (PC) to change program flow.

See Also (Conditional Jump Immediate (BrCC), Jump (Jump))

JumpAbs Description

The jump absolute instruction forces a new value into the Program Counter (PC) to change program flow.

The new address may be a `label` (a program label that provides a signed, even, PC-relative offset) or may be an *immediate value* (provides a signed, even, PC-relative offset). In the `jump label` versions of the instruction, the instruction may be mapped to the smallest of `jump.s`, `jump.l`, or `jump.a`. In the `jump immediate` versions of the instruction, the instruction should not be mapped to `jump.a` due to potential ambiguity of the offset (relative versus absolute).

This instruction encodes as a ***16-bit instruction*** or ***32-bit instruction***, depending on the size of the offset value. This instruction may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

JumpAbs Example

```
jump get_new_sample ;
/* assembler resolved target, abstract offsets */
jump 0x224 ;
/* offset is positive in 13 bits, so target address is PC + 0x224, a forward jump */
jump.s 0x224 ;
/* same as above with jump "short" syntax */
jump.l 0xFFFACE86 ;
/* offset is negative in 25 bits, so target address is PC + 0x1FA CE86, a backwards jump */
```

Call (Call)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
call (PREG Register Type)
call (pc+PREG Register Type)
Call function with pcrel address (CallA)
call imm24nxs2 Register Type
Jump/Call to 32-bit Immediate (Jump32)
call.a buimm32 Register Type
call bimm32 Register Type

Abstract

The Call instruction branches to the address specified and then updates the RETS register with the address of the instruction directly following the Call instruction.

See Also (Return from Branch (Return))

Call Description

The `CALL` instruction calls a subroutine from an address that may be *indirect* (provided by a pointer register), may be *indexed* (PC plus an offset provided by a pointer register), may be a *`label`* (a program label that provides a signed, even, PC-relative offset), or may be an *immediate value* (provides a signed, even, PC-relative offset). In the indirect and indexed versions of the instruction, the value in the pointer register (Preg) must be an even number (bit 0 of the register =0) to maintain 16-bit address alignment. Otherwise, an odd offset in the pointer register causes the processor to generate an address alignment exception.

After the `CALL` instruction executes and execution of the subroutine is completed, the program sequencer resumes program execution at the instruction address pointed to by the `RETS` register. The address write to the `RETS` register occurs when the `CALL` instruction is committed. Even when used as the last instruction of a loop, the `CALL` instruction functions correctly. If the `CALL` were placed at a loop end, the `RETS` register contains the loop top address.

This instruction encodes as a ***16-bit instruction*** or ***32-bit instruction***, depending on the size of the offset value. This instruction may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Call Example

```
call ( p5 ) ;
call ( pc + p2 ) ;
call 0x123456 ;
call get_next_sample ;
```

Return from Branch (Return)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
rts
rti
rtx
rtn
rte

Abstract

Each of these instructions branch to the address specified in their return registers. The interrupt return instructions will also clear their interrupts corresponding bit in the IPEND register.

See Also (Call (Call))

Return Description

The return instruction forces a return from a subroutine, maskable interrupt or NMI routine, exception routine, or emulation routine. The *Types of Return Instructions* table provides a description of the operations provided by each type of return. Note that the interrupt return instructions also clear their interrupt's corresponding bit in the `IPEND` register.

Table 8-18: Types of Return Instructions

Mnemonic	Description
`RTS`	Forces a return from a subroutine by loading the value of the `RETS` register into the Program Counter (PC), causing the processor to fetch the next instruction from the address contained in `RETS`. For nested subroutines, you must save the value of the `RETS` Register. Otherwise, the next subroutine `CALL` instruction overwrites it.
`RTI`	Forces a return from an interrupt routine by loading the value of the `RETI` register into the PC. When an interrupt is generated, the processor enters a non-interruptible state. Saving `RETI` to the stack re-enables interrupt detection so that subsequent, higher priority interrupts can be serviced (or nested) during the current interrupt service routine. If `RETI` is not saved to the stack, higher priority interrupts are recognized but not serviced until the current interrupt service routine concludes. Restoring `RETI` back off the stack at the conclusion of the interrupt service routine masks subsequent interrupts until the `RTI` instruction executes. In any case, `RETI` is protected against inadvertent corruption by higher priority interrupts.
`RTX`	Forces a return from an exception routine by loading the value of the `RETX` register into the PC.
`RTN`	Forces a return from a non-maskable interrupt (NMI) routine by load- ing the value of the `RETN` register into the PC.

Table 8-18: Types of Return Instructions (Continued)

Mnemonic	Description
`RTE`	Forces a return from an emulation routine and emulation mode by load- ing the value of the `RETE` register into the PC. Because only one emulation routine can run at a time, nesting is not an issue, and saving the value of the `RETE` register is unnecessary.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

The ***Required Modes for Return Instructions*** table identifies the modes required for each return instruction.

Table 8-19: Required Modes for Return Instructions

Mnemonic	Required Mode
`RTS`	User and Supervisor
`RTI`, `RTX`, and `RTN`	Supervisor only. Any attempt to execute in User mode produces a protection violation ex- ception.
`RTE`	Emulation only. Any attempt to execute in User mode or Supervi- sor mode produces an exception.

Return Example

```
rts ;
rti ;
rtx ;
rtn ;
rte ;
```

Hardware Loop Set Up (LoopSetup)

General Form

Virtually Zero Overhead Loop Mechanism (LoopSetup)
lsetup (uimm4s2o4 Register Type, uimm10s2o4 Register Type) LC *1
lsetup (uimm4s2o4 Register Type, uimm10s2o4 Register Type) LC = PREG Register Type *2
lsetup (uimm4s2o4 Register Type, uimm10s2o4 Register Type) LC = PREG Register Type >>1 *3
Virtually Zero Overhead Loop Mechanism (LoopSetup)
lsetup (uimm10s2o4 Register Type) LC = uimm10 Register Type
lsetupz (uimm10s2o4 Register Type) LC = PREG Register Type
lsetupz (uimm10s2o4 Register Type) LC = PREG Register Type >> 1
lsetuplez (uimm10s2o4 Register Type) LC = PREG Register Type
lsetuplez (uimm10s2o4 Register Type) LC = PREG Register Type >> 1

*1 Provides encoding for: `LOOP loop_name LC0 ; LOOP_BEGIN loop_name ; LOOP_END loop_name ;`

*2 Provides encoding for: `LOOP loop_name LC0 = Preg ; LOOP_BEGIN loop_name ; LOOP_END loop_name ;`

*3 Provides encoding for: `LOOP loop_name LC0 = Preg >> 1 ; LOOP_BEGIN loop_name ; LOOP_END loop_name ;`

Abstract

The zero-overhead loop set up instruction provides a flexible, count-based, hardware loop mechanism, implementing efficient, zero-overhead software loops. The term "zero-overhead" means the software does not incur a performance or code size penalty by decrementing the loop counter, evaluating a loop condition, calculating the target address, and branching to the address.

See Also (none)

LoopSetup Description

The zero-overhead loop setup instruction provides a flexible, counter- based, hardware loop mechanism that provides efficient, zero-overhead software loops. In this context, zero-overhead means that the software in the loops does not incur a performance or code size penalty by decrementing a counter, evaluating a loop condition, then calculating and branching to a new target address.

NOTE: When the *`Begin_Loop`* address is the next sequential address after the `LSETUP` instruction, the loop has zero overhead. If the *`Begin_Loop`* address is not the next sequential address after the `LSETUP` instruction, there is some overhead that is incurred on loop entry only.

The architecture includes two sets of three registers each to support two independent, nestable loops. The registers are *`Loop_Top`* (`LTx`), *`Loop_Bottom`* (`LBx`) and *`Loop_Count`* (`LCx`). The `LT0`, `LB0`, and `LC0` registers describe *`Loop0`*, and the `LT1`, `LB1`, and `LC1` registers describe *`Loop1`*.

The `LOOP` and `LSETUP` instructions permit initializing all three registers using a single instruction. The size of the `LOOP` and `LSETUP` instructions only supports a finite number of bits, so the loop range is limited. However, `LT0` and `LT1`, `LB0` and `LB1` and `LC0` and `LC1` can be initialized manually using move instructions if loop length and repetition count need to be beyond the limits supported by the `LOOP` and `LSETUP` syntax. A single loop (initialized using this method) can span the entire 4G bytes of memory space.

NOTE: When initializing `LT0` and `LT1`, `LB0` and `LB1`, and `LC0` and `LC1` manually, make sure that *`Loop_Top`* (`LTx`) and *`Loop_Bottom`* (`LBx`) are configured before setting *`Loop_Count`* (`LCx`) to the desired loop count value.

The instruction syntax supports an optional initialization value from a pointer register (Preg) or pointer register divided by 2.

NOTE: The `LOOP`, `LOOP_BEGIN`, `LOOP_END` legacy syntax from previous Blackfin processors is supported by the Blackfin+ processor assembler. The legacy syntax is encoded as `LSETUP` syntax, which contains the same information in a more compact form.

If LCx is nonzero when the fetch address equals LBx, the processor decrements LCx and places the address in LTx into the PC. The loop always executes once through because *Loop_Count* is evaluated at the end of the loop.

There are two special cases for small loop count values. A value of 0 in Loop_Count causes the hardware loop mechanism to neither decrement or loopback, causing the instructions enclosed by the loop pointers to be executed as straight-line code. A value of 1 in *Loop_Count* causes the hardware loop mechanism to decrement only (not loopback), also causing the instructions enclosed by the loop pointers to be executed as straight-line code.

In the instruction syntax, the designation of the loop counter–LC0 or LC1– determines which loop level is initialized. Consequently, to initialize *Loop0*, code LC0; to initialize *Loop1*, code LC1.

In the case of nested loops that end on the same instruction, the processor requires *Loop0* to describe the outer loop and *Loop1* to describe the inner loop. The user is responsible for meeting this requirement.

For example, if LB0=LB1, then the processor assumes loop 1 is the inner loop and loop 0 the outer loop.

Just like entries in any other register, loop register entries can be saved and restored. If nesting beyond two loop levels is required, the user can explicitly save the outermost loop register values, re-use the registers for an inner loop, and then restore the outermost loop values before terminating the inner loop. In such a case, remember that loop 0 must always be outside of loop 1. Alternately, the user can implement the outermost loop in software with the Conditional Jump structure.

Begin_Loop, the value loaded into LTx, is a 5-bit, PC-relative, even offset from the current instruction to the first instruction in the loop. The user is required to preserve half-word alignment by maintaining even values in this register. The offset is interpreted as a one's-complement, unsigned number, eliminating backwards loops.

End_Loop, the value loaded into LBx, is an 11-bit, unsigned, even, PC-relative offset from the current instruction to the last instruction of the loop. When using the LSETUP instruction, *Begin_Loop* and *End_Loop* are typically address labels. The linker replaces the labels with offset values.

A loop counter register (LC0 or LC1) counts the trips through the loop. The register contains a 32-bit unsigned value, supporting as many as 4,294,967,294 trips through the loop. The loop is disabled (subsequent executions of the loop code pass through without reiterating) when the loop counter equals 0.

If no *LoopStartLabel* is specified then the loop start is implied to be the instruction following the LSETUP instruction.

The Z suffix (LSETUPZ), means that the entire loop will be skipped if the count starts at zero. The LEZ suffix (LSETUPLEZ) means that entire loop will be skipped if the starting count is Less than or Equal to Zero. When a LSETUPZ instruction with a loop count of zero commits the LSBit of it's associated LT register will be set. This is used to mark this as an lsetupz should we interrupt the loop. The end of the loop will clear the bit.

It is important to understand the following LSETUP operations and how these affect loop operations:

- If a start address is specified in the LSETUP instruction, the address is a 5-bit, PC-relative, unsigned, even offset (4 to 30) address. If a start address is not specified in the LSETUP instruction, the address used is the address of the instruction following the LSETUP instruction. The absolute start address is computed and stored in LT0 or LT1 register on LSETUP instruction commit.

- The end address is an 11-bit, PC-relative, unsigned, even offset (4 through 2046) address. The absolute end address is computed and stored in the `LB0` or `LB1` register on `LSETUP` instruction commit.
- The values in the loop counter 0 (`LC0`) and loop counter 1 (`LC1`) registers are treated as 32-bit unsigned values, except in the `LSETUPLEZ` version of the instruction. For `LSETUPLEZ`, the loop count value is treated as a signed value. The value in the `LC0` or `LC1` register decrements each time a loop bottom instruction is executed, until the count reaches 0. When executing a loop end instruction, when the value in `LC0` or `LC1` is not 0 or 1, a loop back operation occurs.
- A loop is disabled when the its loop count (`LC0` or `LC1`) equals 0.
- The sequencer treats a ***constant*** loop count of -1 as special and loads the counter with the value 0xffffffff.

LoopSetup Example

```
/* examples for three-part loop setup ... */
/* LOOP loop_name loop_counter */
/* LOOP_BEGIN loop_name */
/* LOOP_END loop_name */

loop MyRepeatedOperations LC0 ; /* define loop 'MyRepeatedOperations' with Loop Counter 0 */
loop_begin MyRepeatedOperations ; /* place before the first instruction in the loop */
   nop;
loop_end MyRepeatedOperations ; /* place after the last instruction in the loop */

loop MyOtherRepeatedOperations LC1 ; /* define loop 'MyOtherRepeatedOperations' with Loop
Counter 1 */
loop_begin MyOtherRepeatedOperations ; /* place before the first instruction in the loop */
   nop;
loop_end MyOtherRepeatedOperations ; /* place after the last instruction in the loop */

/* a loop with a specified beginning offset */
loop loop_2 lc0 = p0;
loop_begin loop_2;
   nop;
loop_end loop_2;

/* a loop without a specified beginning offset */
loop lc0 = 45;
   nop;
loop_end;

/* note that loopz and looplez opcodes to lsetupz and lsetuplez */

/* examples for single line loop setup ... */
/* LSETUP (Begin_Loop, End_Loop) Loop_Counter */

lsetup ( 4, 4 ) lc0 ;
lsetup ( poll_bit, end_poll_bit ) lc0 ;
lsetup ( 4, 6 ) lc1 ;
lsetup ( FIR_filter, bottom_of_FIR_filter ) lc1 ;
lsetup ( 4, 8 ) lc0 = p1 ;
lsetup ( 4, 8 ) lc0 = p1>>1 ;
```

Control Code Bit Management Operations

These operations provide control code bit operations needed to support condtional operations:

- Compute Move CC to a D Register (CCToDreg)
- Move CC To/From ASTAT (CCToStat16)
- Move Status to CC (MvToCC)
- Move Status to CC (MvToCC_STAT)
- 32-Bit Register Compare and Set CC (CompRegisters)
- Accumulator Compare and Set CC (CompAccumulators)
- 32-Bit Pointer Register Compare and Set CC (CCFlagP)

Compute Move CC to a D Register (CCToDreg)

General Form

Move CC conditional bit, to and from dreg (CC2Dreg)
DREG Register Type = cc
DREG Register Type = !cc

Abstract

This instruction moves CC to a 32-bit D Register. The register will either be 1 on 0.

CCToDreg Description

The move CC to data register instruction moves *either* the the status of the control code (CC) bit *or* moves the negated status of the CC bit to a data register.

When copying the CC bit into a 32-bit register, the operation moves the CC bit into the least significant bit of the register, zero-extended to 32 bits. The two cases are as follows.

- If CC = 0, the data register becomes 0x00000000.
- If CC = 1, the data register becomes 0x00000001.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

CCToDreg Example

```
r0 = cc ;
r1 =! cc ;
```

Move CC To/From ASTAT (CCToStat16)

General Form

Copy CC conditional bit, from status (CC2Stat)
CBIT = cc
CBIT |= cc
CBIT &= cc
CBIT ^= cc

Abstract

This instruction moves CC to another ASTAT bit. It is illegal to use the CC bit as source and destination in the same instruction, i.e., CC=CC or CC&=CC.

See Also (Move Status to CC (MvToCC), Move Status to CC (MvToCC_STAT))

CCToStat16 Description

The move CC to arithmetic status register instruction sets or clears status bits based on the logic operations and the status of the control code (CC) bit.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	*AC1*	*AC0*	...	...	...	RND_ MOD	...	*AQ*	*CC*	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CCToStat16 Example

```
az = cc ; /* status bit equals cc */
an |= cc ; /* status bit equals status bit OR cc */
ac0 &= cc ; /* status bit equals status bit AND cc */
av0 ^= cc ; /* status bit equals status bit XOR cc */
```

Move Status to CC (MvToCC)

General Form

Move CC conditional bit, to and from dreg (CC2Dreg)
cc = DREG Register Type
cc = !cc

Abstract

This instruction moves a status bit or LSB of a register to CC. It is illegal to use the CC bit as source and destination in the same instruction (for example, CC=CC or CC&=CC are illegal).

See Also (Move CC To/From ASTAT (CCToStat16), Move Status to CC (MvToCC_STAT))

MvToCC Description

The move data register to CC instruction *either* moves an OR of all bits in the data register *or* moves the negated state of the control code (CC) bit to the CC bit. When copying a data register to the CC bit, the operation sets the CC bit to 1 if any bit in the source data register is set; that is, if the register is nonzero. Otherwise, the operation clears the CC bit.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvToCC Example

```
cc = r4 ;
cc = !cc ;
```

Move Status to CC (MvToCC_STAT)

General Form

Copy CC conditional bit, from status (CC2Stat)
cc = CBIT
cc |= CBIT
cc &= CBIT
cc ^= CBIT

Abstract

This instruction moves a status bit or LSB of a register to CC. It is illegal to use the CC bit as source and destination in the same instruction (for example, CC=CC or CC&=CC are illegal).

See Also (Move CC To/From ASTAT (CCToStat16), Move Status to CC (MvToCC))

MvToCC_STAT Description

The move status bit to CC instruction sets or clears the control code (CC) bit based on the logic operations and the status of the status bits.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

MvToCC_STAT Example

```
cc = av1 ;  /* cc equals status bit */
cc |= aq ;  /* cc equals cc OR status bit */
cc &= an ;  /* cc equals cc AND status bit */
cc ^= ac1 ; /* cc equals cc XOR status bit */
```

32-Bit Pointer Register Compare and Set CC (CCFlagP)

General Form

Set CC conditional bit (CCFlag)
cc = PREG Register Type == PREG Register Type
cc = PREG Register Type == imm3 Register Type
cc = PREG Register Type < PREG Register Type
cc = PREG Register Type < imm3 Register Type
cc = PREG Register Type <= PREG Register Type
cc = PREG Register Type <= imm3 Register Type
cc = PREG Register Type < PREG Register Type (iu)
cc = PREG Register Type < uimm3 Register Type (iu)
cc = PREG Register Type <= PREG Register Type (iu)
cc = PREG Register Type <= uimm3 Register Type (iu)

Abstract

This instruction compares two pointer registers.

See Also (32-Bit Register Compare and Set CC (CompRegisters), Accumulator Compare and Set CC (CompAccumulators))

CCFlagP Description

The compare pointer and move CC instruction sets or clears the control code (CC) bit based on a comparison of two values. The input operands are pointer registers (Preg).

The compare operations are nondestructive on the input operands and affect only the CC bit and the status bits. The value of the CC bit determines all subsequent conditional branching.

The various forms of the compare pointer instruction perform 32-bit signed compare operations on the input operands or an unsigned compare operation (if the `(IU)` optional mode is appended). The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CCFlagP Example

```
cc = p3 == p2 ; /* equal, register, signed */
cc = p0 == 1 ; /* equal, immediate, signed */
cc = p0 < p3 ; /* less than, register, signed */
cc = p2 < -4 ; /* less than, immediate, signed */
cc = p1 <= p0 ; /* less than or equal, register, signed */
cc = p4 <= 3 ; /* less than or equal, immediate, signed */
cc = p5 < p3 (iu) ; /* less than, register, unsigned */
cc = p1 < 0x7 (iu) ; /* less than, immediate, unsigned */
cc = p2 <= p0 (iu) ; /* less than or equal, register, unsigned */
cc = p3 <= 2 (iu) ; /* less than or equal, immediate unsigned */
```

Accumulator Compare and Set CC (CompAccumulators)

General Form

Set CC conditional bit (CCFlag)
cc = a0 == a1
cc = a0 < a1
cc = a0 <= a1

Abstract

This instruction compares the two accumulators ands sets CC.

See Also (32-Bit Register Compare and Set CC (CompRegisters), 32-Bit Pointer Register Compare and Set CC (CCFlagP))

CompAccumulators Description

The Compare Accumulator instruction sets the Control Code (CC) bit based on a comparison of two values. The input operands are Accumulators.

These instructions perform 40-bit signed compare operations on the Accumulators. The compare operations perform the subtraction A0–A1 and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

No unsigned compare operations or immediate compare operations are performed for the Accumulators. The compare operations are nondestructive on the input operands, and affect only the CC bit and the status bits. All subsequent conditional branching is based on the value of the CC bit.

The Compare Accumulator instruction uses the values shown in the ***Compare Accumulator Instruction Values*** table in compare operations after the A0–A1 subtraction is performed.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Table 8-20: Compare Accumulator Instruction Values

Comparison	**Signed**
Equal	AZ =1
Less than	AN =1
Less than or equal	AN or AZ =1

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CompAccumulators Example

```
cc = a0 == a1 ; /* equal, signed */
cc = a0 < a1 ;  /* less than, accumulator, signed */
cc = a0 <= a1 ; /* less than or equal, accumulator, signed */
```

32-Bit Register Compare and Set CC (CompRegisters)

General Form

Set CC conditional bit (CCFlag)
cc = DREG Register Type == DREG Register Type
cc = DREG Register Type == imm3 Register Type
cc = DREG Register Type < DREG Register Type
cc = DREG Register Type < imm3 Register Type

cc = DREG Register Type <= DREG Register Type
cc = DREG Register Type <= imm3 Register Type
cc = DREG Register Type < DREG Register Type (iu)
cc = DREG Register Type < uimm3 Register Type (iu)
cc = DREG Register Type <= DREG Register Type (iu)
cc = DREG Register Type <= uimm3 Register Type (iu)

Abstract

This instruction compares two 32-bit registers and sets CC or sets CC if a register is non-zero.

See Also (Accumulator Compare and Set CC (CompAccumulators), 32-Bit Pointer Register Compare and Set CC (CCFlagP))

CompRegisters Description

The Compare Data Register instruction sets the Control Code (CC) bit based on a comparison of two values. The input operands are D-registers.

The compare operations are nondestructive on the input operands and affect only the CC bit and the status bits. The value of the CC bit determines all subsequent conditional branching.

The various forms of the Compare Data Register instruction perform 32-bit signed compare operations on the input operands or an unsigned compare operation, if the (IU) optional mode is appended. The compare operations perform a subtraction and discard the result of the subtraction without affecting user registers. The compare operation that you specify determines the value of the CC bit.

The Compare Data Register instruction uses the values shown in the ***Compare Data Register Values*** table in signed and unsigned compare operations.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Table 8-21: Compare Data Register Values

Comparison	Signed	Unsigned
Equal	AZ=1	n/a
Less than	AN=1	AC0=0
Less than or equal	AN or AZ=1	AC0=0 or AZ=1

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	*AC0*	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

CompRegisters Example

```
cc = r3 == r2 ; /* equal, register, signed */
cc = r7 == 1 ; /* equal, immediate, signed */
/* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001, then the signed operation . . . */
cc = r0 < r3 ; /* less than, register, signed */
/* . . . produces cc = 1, because r0 is treated as a negative value */
cc = r2 < -4 ; /* less than, immediate, signed */
cc = r6 <= r1 ; /* less than or equal, register, signed */
cc = r4 <= 3 ; /* less than or equal, immediate, signed */
/* If r0 = 0x8FFF FFFF and r3 = 0x0000 0001,then the unsigned operation . . . */
cc = r0 < r3 (iu) ; /* less than, register, unsigned */
/* . . . produces CC = 0, because r0 is treated as a large unsigned value */
cc = r1 < 0x7 (iu) ; /* less than, immediate, unsigned */
cc = r2 <= r0 (iu) ; /* less than or equal, register, unsigned (a) */
cc = r3 <= 2 (iu) ; /* less than or equal, immediate unsigned (a) */
```

Event Management Operations

These operations provide interrupt and exception related operations:

- Interrupt Control (IMaskMv)
- Sequencer Mode (Mode)
- Raise Interrupt (Raise)

Interrupt Control (IMaskMv)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
cli DREG Register Type
sti DREG Register Type

Abstract

The CLI instruction disables or clears general interrupts, and the STI instruction enables interrupts.

IMaskMv Description

The enable interrupts instruction (`sti`) globally enables interrupts by restoring the previous state of the interrupt system from a data register into the `IMASK` register.

The disable interrupts instruction (`cli`) globally disables general interrupts by clearing the `IMASK` register to all zeros. In addition, the instruction copies the previous contents of `IMASK` into a user-specified register in order to save the state of the interrupt system. The disable interrupts instruction does not mask NMI, reset, exceptions, and emulation.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

The enable interrupts and disable interrupts instructions executes only in ***Supervisor mode***. If execution is attempted in User mode, the instruction produces an Illegal Use of Protected Resource exception.

These instructions have some *special applications*. The clear interrupts instruction is often issued immediately *before* an `idle` instruction, so it stores the interrupt state before entering the idle state. The enable interrupts instruction is often located *after* an `idle` instruction, so it executes after a wake-up event from the idle state.

IMaskMv Example

```
sti r3 ; /* previous state of IMASK restored from Dreg */
cli r3 ; /* previous state of IMASK moved to Dreg (a) */
```

Sequencer Mode (Mode)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
emuexcpt

Abstract

The SEI instruction vectors to a fixed location in security firmware. The TRAP instruction raises interrupt 15 to notify the operating system that the user code needs a system service. The EMUEXCPT instruction allows processor to enter emulation mode.

Mode Description

The force emulation instruction forces an emulation exception, allowing the processor to enter emulation mode. When emulation is enabled, the processor immediately takes an exception into emulation mode. When emulation is disabled, `EMUEXCPT` behaves the same as a `NOP` instruction. The emulation exception is the highest priority event in processor.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***

Mode Example

```
emuexcpt ;
```

Raise Interrupt (Raise)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
raise uimm4 Register Type
excpt uimm4 Register Type

Abstract

The EXCPT instruction forces the specified exception (range 0 through 15).

Raise Description

The force interrupt / reset / exception instruction forces a specified interrupt or reset or exception to occur. Typically, it is a software method of invoking a hardware event for debug purposes.

When the `RAISE` instruction is issued, the processor sets a bit in the `ILAT` register corresponding to the interrupt vector specified by the *`uimm4`* constant in the instruction. The interrupt executes when its priority is high enough to be recognized by the processor. The `RAISE` instruction causes these events to occur given the uimm4 arguments shown in the ***uimm4 Arguments and Events*** table.

When the `EXCPT` instruction is issued, the sequencer vectors to the exception handler that the user provides. Application-level code uses the force exception instruction for operating system calls. The instruction does not set the `EVSW` bit (bit 3) of the `ILAT` register.

Table 8-22: uimm4 Arguments and Events

uimm4	Event
0	reserved
1	RST
2	NMI
3	reserved
4	reserved
5	IVHW
6	IVTMR
7	IVG7

Table 8-22: uimm4 Arguments and Events (Continued)

uimm4	Event
8	IVG8
9	IVG9
10	IVG10
11	IVG11
12	IVG12
13	IVG13
14	IVG14
15	IVG15

The `RAISE` instruction cannot invoke exception (EXC) or emulation (EMU) events. Use the `EXCPT` and `EMUEXCPT` instructions, respectively, for those events.

The `RAISE` instruction does not take effect before the write-back stage in the pipeline.

This ***16-bit instruction*** takes up less memory space (over a 32-bit encoded instruction), but may not be issued in parallel with other instructions.

The force interrupt / reset / exception instruction executes only in *Supervisor mode*. If execution is attempted in User mode, the force interrupt / reset instruction produces an Illegal Use of Protected Resource exception.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Raise Example

```
raise 1 ; /* Invoke RST */
raise 6 ; /* Invoke IVTMR timer interrupt */
excpt 4 ;
```

Stack Operations

These operations provide memory stack management operations:

- Linkage (Linkage)

- Stack Pop (Pop)
- Stack Push (Push)
- Stack Push/Pop Multiple Registers (PushPopMul16)

Linkage (Linkage)

General Form

Save/restore registers and link/unlink frame, multiple cycles (Linkage)
link uimm16s4 Register Type
unlink

Abstract

The linkage instruction controls the stack frame space on the stack and the Frame Pointer (FP) for that space. LINK allocates the space and UNLINK de-allocates the space.

Linkage Description

The linkage instruction controls the stack frame space on the stack and the frame pointer (`FP`) for that space. `LINK` allocates the space and `UNLINK` de-allocates the space.

`LINK` saves the current `RETS` and `FP` registers to the stack, loads the `FP` register with the new frame address, then decrements the stack pointer (`SP`) by the user-supplied frame size value.

Typical applications follow the `LINK` instruction with a push multiple instruction to save pointer and data registers to the stack.

The user-supplied argument for `LINK` determines the size of the allocated stack frame. `LINK` always saves `RETS` and `FP` on the stack, so the minimum frame size is 2 words when the argument is zero. The maximum stack frame size is 218 + 8 = 262152 bytes in 4-byte increments.

`UNLINK` performs the reciprocal of `LINK`, de-allocating the frame space by moving the current value of `FP` into `SP` and restoring previous values into `FP` and `RETS` from the stack.

The `UNLINK` instruction typically follows a pop multiple instruction that restores pointer and data registers previously saved to the stack.

The frame values remain on the stack until a subsequent push, push multiple or `LINK` operation overwrites them.

To preserve stack integrity, the `FP` *must not be modified* by user code between `LINK` and `UNLINK` execution.

Neither `LINK` nor `UNLINK` may be interrupted. Exceptions that occur while either of these instructions are executing cause the instruction to abort. For example, a load operation or a store operation might cause a protection violation while `LINK` is executing. In that case, `SP` and `FP` are reset to their original values prior to the execution of this instruction. This measure ensures that the instruction can be restarted after the exception.

Note that when a `LINK` operation aborts due to an exception, the stack memory may already be changed due to stores that have already completed before the exception. Similarly, an aborted `UNLINK` operation may leave the `FP` and `RETS` registers changed because of a load that has already completed before the interruption.

The series of illustrations show how the stack contents change. After executing a `LINK` instruction, the stack contains (for example) the contents shown in the ***Stack After Link Executes*** figure.

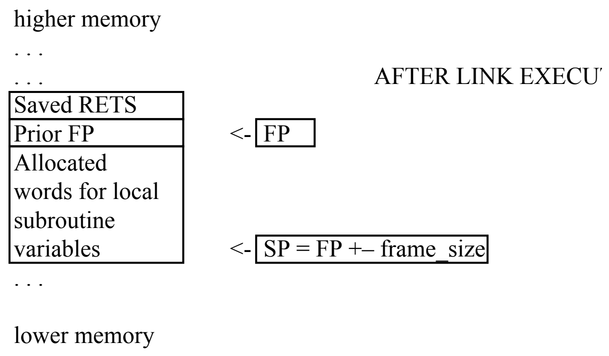


Figure 8-10: Stack After Link Executes

Following the `LINK` and a push multiple instruction, the stack contains (for example) the contents shown in the ***Stack After Push Multiple Executes*** figure.

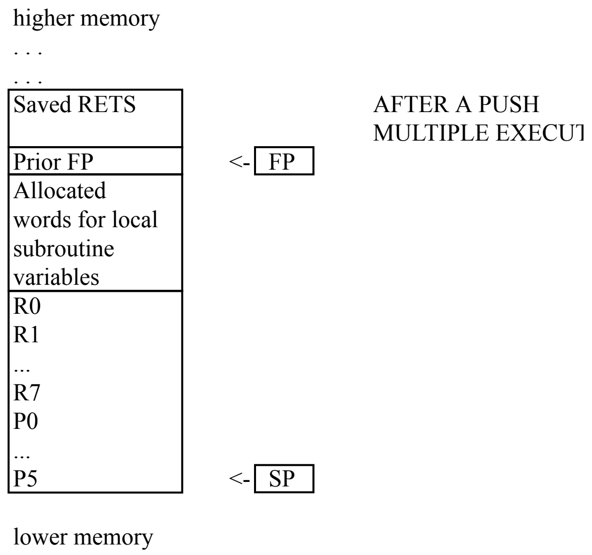


Figure 8-11: Stack After Push Multiple Executes

The stack pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts, as described above.

This ***32-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Linkage Example

```
link 8 ; /* establish frame with 8 words allocated for local variables */
[ -- sp ] = (r7:0, p5:0) ; /* save D- and P-registers */
(r7:0, p5:0) = [ sp ++ ] ; /* restore D- and P-registers */
unlink ; /* close the frame* /
```

Stack Pop (Pop)

General Form

Push or Pop register, to and from the stack pointed to by sp (PushPopReg)
POPREG = [sp++]

Abstract

Thisp instruction loads the contents of the stack indexed by the current stack pointer into a specified register.

See Also (Stack Push (Push), Stack Push/Pop Multiple Registers (PushPopMul16))

Pop Description

The pop instruction loads the contents of the stack—indexed by the current stack pointer (SP)—into a specified register. The instruction post-increments the stack pointer to the next occupied location in the stack before concluding.

The stack grows down from high memory to low memory, therefore the decrement operation is used for pushing, and the increment operation is used for popping values. The stack pointer always points to the last used location. When a pop operation is issued, the value pointed to by the stack pointer is transferred and the SP is replaced by SP + 4.

The following series of illustrations show what the stack would look like when a pop such as R3 = [SP ++] occurs.

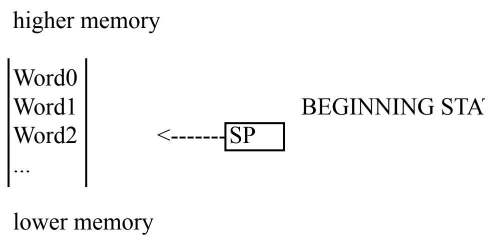


Figure 8-12: Stack Beginning State

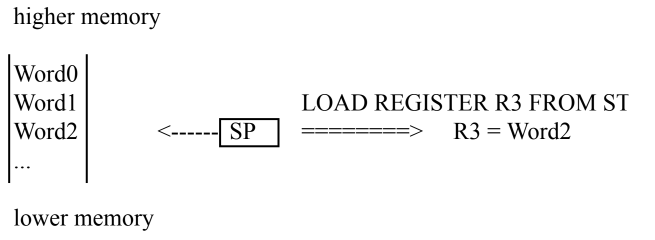


Figure 8-13: Load Register From Stack

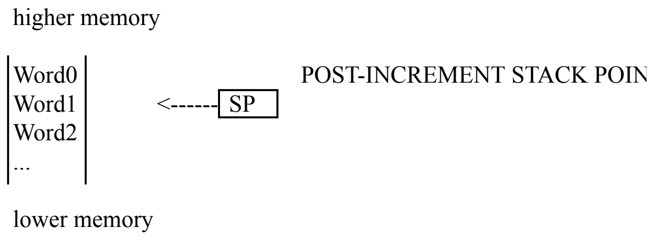


Figure 8-14: Post-Increment Stack Pointer

This ***16-bit instruction*** may be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode*** for most cases, but explicit access to `USP`, `SEQSTAT`, `SYSCFG`, `RETI`, `RETX`, `RETN`, `RETE`, and `EMUDAT` requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.

The `ASTAT = [SP++]` version of this instruction explicitly affects arithmetic status bits. Status bits are not affected by other versions of this instruction.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	*AV1S*	*AV1*	*AV0S*	*AV0*
...	...	*AC1*	*AC0*	...	...	...	RND_ MOD	...	*AQ*	*CC*	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Pop Example

```
r0 = [sp++] ; /* Load Data Register instruction */
p4 = [sp++] ; /* Load Pointer Register instruction */
i1 = [sp++] ; /* Pop instruction */
reti = [sp++] ; /* Pop instruction; supervisor mode required */
```

Stack Push (Push)

General Form

Push or Pop register, to and from the stack pointed to by sp (PushPopReg)
[--sp] = PUSHREG

Abstract

This instruction stores the contents of a specified register in the stack.

See Also (Stack Pop (Pop), Stack Push/Pop Multiple Registers (PushPopMul16))

Push Description

The push instruction stores the contents of a specified register in the stack. The instruction pre-decrements the stack pointer (`SP`) to the next available location in the stack first. Push and push multiple are the only instructions that perform pre-modify functions.

The stack grows down from high memory to low memory. Consequently, the decrement operation is used for pushing, and the increment operation is used for popping values. The stack pointer always points to the last used location. Therefore, the effective address of the push is `SP - 4`.

The following illustration shows what the stack would look like when a series of pushes occur.

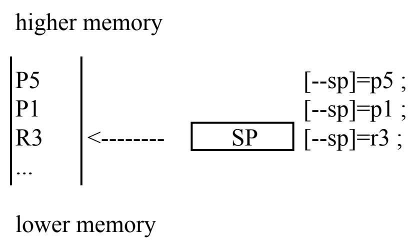


Figure 8-15: Stack Following a Series of Pushes

The stack pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts.

Push/pop on `RETS` has no effect on the interrupt system.

Push/pop on `RETI` does affect the interrupt system.

Pushing `RETI` enables the interrupt system, whereas popping `RETI` disables the interrupt system.

Pushing the stack pointer is meaningless since it cannot be retrieved from the stack. Using the stack pointer as the destination of a pop instruction (as in the fictional instruction `SP = [SP++]`) causes an undefined instruction exception.

This ***16-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode*** for most cases, but explicit access to USP, SEQSTAT, SYSCFG, RETI, RETX, RETN, RETE, and EMUDAT requires Supervisor mode. A protection violation exception results if any of these registers are explicitly accessed from User mode.

Push Example

```
[ -- sp ] = r0 ;
[ -- sp ] = r1 ;
[ -- sp ] = p0 ;
[ -- sp ] = i0 ;
```

Stack Push/Pop Multiple Registers (PushPopMul16)

General Form

Push or Pop Multiple contiguous registers (PushPopMult)
(PREG_RANGE Register Type) = [sp++]
(DREG_RANGE Register Type) = [sp++]
(DREG_RANGE Register Type, PREG_RANGE Register Type) = [sp++]
[--sp] = (PREG_RANGE Register Type)
[--sp] = (DREG_RANGE Register Type)
[--sp] = (DREG_RANGE Register Type, PREG_RANGE Register Type)

Abstract

This instruction pushes or pops the contents of multiple data and/or pointer registers to or from the stack.

See Also (Stack Pop (Pop), Stack Push (Push))

PushPopMul16 Description

The push multiple instruction saves the contents of multiple data and/or pointer registers to the stack, and the pop multiple instruction restores the contents of multiple data and/or pointer registers from the stack. The range of registers to be pushed (saved) or popped (restored) always includes the highest index data register (R7) and/or highest index pointer register (P5) plus any contiguous lower index registers specified by the user down to and including R0 and/or P0.

Push Multiple Instruction Operations

The push multiple instruction operations start by saving the register having the lowest index then advance to the register with the highest index. The index of the first register saved in the stack is specified by the user in the instruction syntax. Data registers are pushed before Pointer registers if both are specified in one instruction.

The push multiple instruction pre-decrements the stack pointer to the next available location in the stack first.

NOTE: Push and Push Multiple are the only instructions that perform pre-modify functions.

The stack grows down from high memory to low memory, therefore the decrement operation is the same used for pushing, and the increment operation is used for popping values. The stack pointer always points to the last used location, making the effective address of the push is `SP - 4`.

The ***Stack Following a Push Multiple*** illustration shows what the stack would look like when a push multiple occurs.

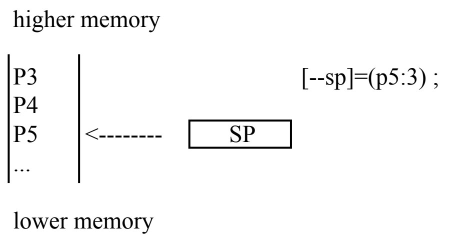


Figure 8-16: Stack Following a Push Multiple

Because the push multiple instruction always saves the lowest-indexed registers first, it is advisable that a run-time system be defined to have its compiler scratch registers as the lowest- indexed registers. For instance, data registers `R0`, `P0` would be the return value registers for a simple calling convention.

Pop Multiple Instruction Operations

The pop multiple instruction operations start by restoring the register having the highest index then descend to the register with the lowest index. The index of the last register restored from the stack is specified by the user in the instruction syntax. Pointer registers are popped before data registers, if both are specified in the same instruction.

The instruction post-increments the stack pointer to the next occupied location in the stack before concluding.

The stack grows down from high memory to low memory, therefore the decrement operation is used for pushing, and the increment operation is used for popping values. The Stack Pointer always points to the last used location. When a pop operation is issued, the value pointed to by the Stack Pointer is transferred and the `SP` is replaced by `SP + 4`.

The series of illustrations show how the stack appears when a pop multiple such as `(R7:5) = [SP++]` occurs.

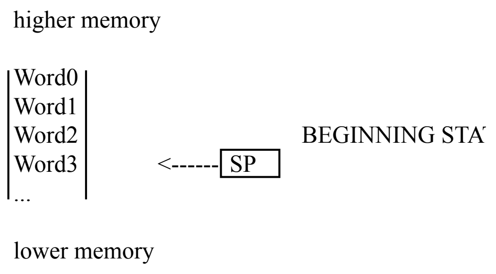


Figure 8-17: Stack Beginning State of Pop Multiple

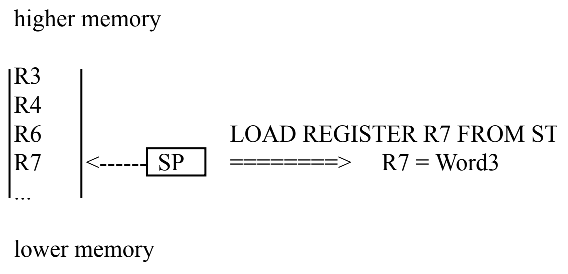


Figure 8-18: Load Register R7 From Stack during Pop Multiple

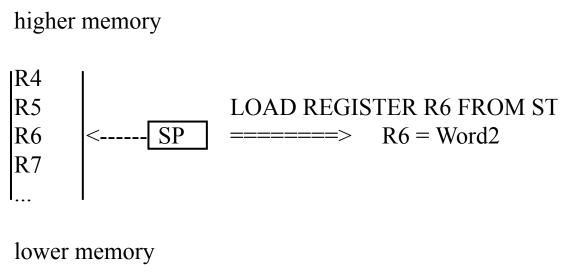


Figure 8-19: Load Register R6 From Stack during Pop Multiple

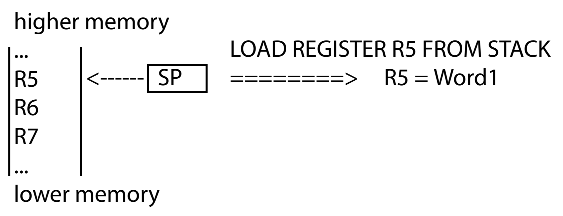


Figure 8-20: Load Register R5 From Stack during Pop Multiple

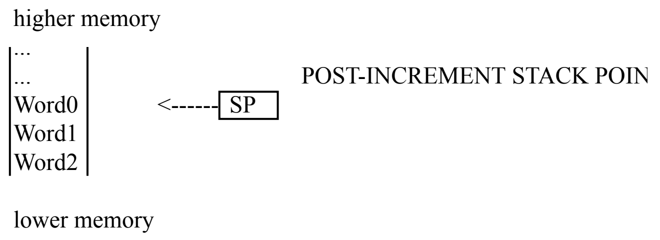


Figure 8-21: Post-Increment Stack Pointer after Pop Multiple

The value(s) just popped remain on the stack until another push instruction overwrites it.

The intended usage for the pop multiple instruction is to recover register values that were previously pushed onto the stack. The user must exercise programming discipline to restore the stack values back to their intended registers from the first-in, last-out structure of the stack. Pop exactly the same registers that were pushed onto the stack, but pop them in the opposite order.

Push Multiple and Pop Multiple Common Features

Although the PushPopMul16 instruction takes a variable amount of time to complete (depending on the number of registers to be saved/restored), the instruction reduces compiled code size.

This instruction is not interruptible. Interrupts asserted after the first issued stack write operation (for push) or stack read operation (for pop) are appended until all the writes or reads complete. However, exceptions that occur while this instruction is executing cause it to abort gracefully. For example:

- While push multiple is executing, a load/store operation might cause a protection violation. In that case, the `SP` is reset to its value before the execution of this instruction. Note that when a push multiple operation is aborted due to an exception, the memory state is changed by the stores that have already completed before the exception.
- While pop multiple is executing, a load/store operation might cause a protection violation In that case, the `SP` is reset to its original value prior to the execution of this instruction. Note that when a pop multiple operation aborts due to an exception, some of the destination registers are changed as a result of loads that have already completed before the exception.

These measures ensures that the instruction can be restarted after the exception.

The stack pointer must already be 32-bit aligned to use this instruction. If an unaligned memory access occurs, an exception is generated and the instruction aborts.

Only data register (`R7−0`) and pointer registers (`P5−0`) may be operands for this instruction. The `SP` and `FP` registers may not be used as operands for this instruction.

This ***16-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

PushPopMul16 Example

```
/* push multiple examples */
[ -- sp ] = (r7:5, p5:0) ; /* D-registers R4:0 excluded */
[ -- sp ] = (r7:5, p5:0) ; /* D-registers R4:0 excluded */
[ -- sp ] = (r7:2) ; /* R1:0 excluded */
[ -- sp ] = (p5:4) ; /* P3:0 excluded */

/* pop multiple examples */
(p5:4) = [ sp ++ ] ; /* P3 through P0 excluded */
(r7:2) = [ sp ++ ] ; /* R1 through R0 excluded */
(r7:5, p5:0) = [ sp ++ ] ; /* D-registers R4 through R0 optionally excluded */
```

Synchronization Operations

These operations provide processor synchronization operations:

- Cache Control (CacheCtrl)
- Sync (Sync)
- SyncExcl (SyncExcl)
- NOP (NOP)
- 32-Bit No Operation (NOP32)
- TestSet (TestSet)

Cache Control (CacheCtrl)

General Form

Cache Control (CacheCtrl)
prefetch [PREGA]
flushinv [PREGA]
flush [PREGA]
iflush [PREGA]

Abstract

These instructions provide the ability to manipulate the caches. The prefetch causes the data cache to prefetch the cache line associated with the effective address provided as the contents of the p-register.

See Also (Sync (Sync))

CacheCtrl Description

These instructions provide the ability to manipulate the cachesa;

- `prefetch` causes the data cache to prefetch the cache line associated with the effective address provided as the contents of the p-register.
- `flushinv` causes the data cache to invalidate a particular line in the cache.
- `flush` causes the a line of data in the cache to be syncronized with higher levels of memory.
- `iflush` causes the instruction cache to invalidate a particular line in the cache.

prefetch Instruction Operations

The Data Cache Prefetch instruction causes the data cache to prefetch the cache line that is associated with the effective address in the P-register. The operation causes the line to be fetched if it is not currently in the data cache and if the address is cacheable (that is, if bit CPLB_L1_CHBL = 1). If the line is already in the cache or if the cache is already fetching a line, the prefetch instruction performs no action, like a NOP.

This instruction may generate CPLB exceptions. For example, exception 0x26 can be generated upon execution of the PREFETCH[P0] instruction if P0 points to an invalid memory location. However, external memory will not be accessed when any of these exceptions are generated.

The instruction can post-increment the line pointer by the cache line size.

flushinv Instruction Operations

The Data Cache Line Invalidate instruction causes the data cache to invalidate a specific line in the cache. The contents of the P-register specify the line to invalidate. If the line is in the cache and dirty, the cache line is written out to the next level of memory in the hierarchy. If the line is not in the cache, the instruction performs no action, like a NOP.

This instruction may generate CPLB exceptions. For example, exception 0x26 can be generated upon execution of the FLUSHINV[P0] instruction if P0 points to an invalid memory location. However, external memory will not be accessed when any of these exceptions are generated. The instruction can post-increment the line pointer by the cache line size.

flush Instruction Operations

The Data Cache Flush instruction causes the data cache to synchronize the specified cache line with higher levels of memory. This instruction selects the cache line corresponding to the effective address contained in the P-register. If the cached data line is dirty, the instruction writes the line out and marks the line clean in the data cache. If the specified data cache line is already clean or the cache does not contain the address in the P-register, this instruction performs no action, like a NOP.

This instruction may generate CPLB exceptions. For example, exception 0x26 can be generated upon execution of the FLUSH[P0] instruction if P0 points to an invalid memory location. However, external memory will not be accessed when any of these exceptions are generated.

The instruction can post-increment the line pointer by the cache line size.

iflush Instruction Operations

The Instruction Cache Flush instruction causes the instruction cache to invalidate a specific line in the cache. The contents of the P-register specify the line to invalidate. The instruction cache contains no dirty bit. Consequently, the contents of the instruction cache are never flushed to higher levels.

This instruction does not cause address exception violations. If a protection violation associated with the address occurs, the instruction acts as a NOP and does not cause a protection violation exception.

The instruction can post-increment the line pointer by the cache line size.

CacheCtrl Instruction Common Features

These instructions have post-modify versions, where the value in the pointer register (Preg) used as address for the prefetch and flush is incremented by the cache-line size.

These instructions do not cause address exception violations. If the effective address is misaligned or outside the allowed memory region, these instructions have no effect.

This ***16-bit instruction*** may not be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either ***User or Supervisor mode***.

CacheCtrl Example

```
prefetch [ p2 ] ;
prefetch [ p0 ++ ] ;
flushinv [ p2 ] ;
flushinv [ p0 ++ ] ;
flush [ p2 ] ;
flush [ p0 ++ ] ;
iflush [ p2 ] ;
iflush [ p0 ++ ] ;
```

Sync (Sync)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
idle
csync
ssync
sti idle DREG Register Type

Abstract

The instructions are DSYNC (Data Sync), SSYNC (System Sync), CSYNC (Core Sync), IDLE, and STI IDLE.

See Also (Cache Control (CacheCtrl))

Sync Description

The sync instructions (`CSYNC`, `SSYNC`, `DSYNC`, `IDLE`, and `STI IDLE`) provide the means to synchronize core, system, and data operations across all clock domains of the processor.

idle Instruction Operations

Typically, the `IDLE` instruction is part of a sequence to place the Blackfin+ processor in a quiescent state so that the external system can switch between core clock frequencies.

The first instruction following the `IDLE` is the first instruction to execute when the processor recovers from idle mode.

NOTE: Blackfin+ processors (unlike previous on previous Blackfin processors) an `IDLE` instruction is not required immediately following an `SSYNC` instruction.

csync Instruction Operations

The core synchronize (`CSYNC`) instruction ensures resolution of all pending core operations and the flushing of the core store buffer before proceeding to the next instruction. Pending core operations include any speculative states (for example, branch prediction) or exceptions. The core store buffer lies between the processor and the L1 cache memory.

`CCYNC` is typically used after core memory-mapped register writes to prevent imprecise behavior, unless otherwise specified in Blackfin+ Processor Programming Reference. For example, an `SSYNC` instruction is required to follow some core memory-mapped register accesses, such as when `IMEM_CONTROL` is written to while enabling cache.

Use `CSYNC` to enforce a strict execution sequence on loads and stores or to conclude all transitional core states before reconfiguring the core modes. For example, issue `CSYNC` before configuring memory-mapped registers (MMRs). `CSYNC` should also be issued after stores to memory-mapped registers to make sure the data reaches the memory-mapped register before the next instruction is fetched.

Typically, the Blackfin+ processor executes all load instructions strictly in the order that they are issued and all store instructions in the order that they are issued. However, for performance reasons, the architecture relaxes ordering between load and store operations. It usually allows load operations to access memory out of order with respect to store operations. Further, it usually allows loads to access memory speculatively. The core may later cancel or restart speculative loads. By using the core synchronize or system synchronize instructions and managing interrupts appropriately, you can restrict out-of-order and speculative behavior.

NOTE: Stores never access memory speculatively.

ssync Instruction Operations

The system synchronize (`SSYNC`) instruction forces all speculative, transient states in the core and system to complete before processing continues. Until `SSYNC` completes, no further instructions can be issued to the pipeline.

The `SSYNC` instruction performs the same function as core synchronize (`CSYNC`). In addition, `SSYNC` flushes any write buffers (between the L1 memory and the system interface) and generates a sync request signal to the external system. The operation requires an acknowledgement by the system before completing the instruction.

An SSYNC instruction should be used when ordering is required between a memory write and a memory read. For more information about these operations, see the memory or pointer instructions.

When strict ordering of instruction execution is required, by design, the Blackfin+ processor architecture allows reads to take priority over writes when there are no dependencies between the address that are accessed. In general, this execution order allows for increased performance. However, when an asynchronous memory device is mapped to a Blackfin+ processor, it is sometimes necessary to ensure the write occurs before the read. But, the Blackfin+ processor re-orders loads over stores if there is not a data dependency. In this case, an SSYNC between the write and read will ensure proper ordering is preserved.

NOTE: Blackfin+ processors (unlike previous on previous Blackfin processors) an IDLE instruction is not required immediately following an SSYNC instruction.

dsync Instruction Operations

The DSYNC instruction (data sync) ensures that all writes have completed to final destinations before any other writes commit.

sti idle Instruction Operations

The STI IDLE instruction (enable interrupts, then idle) performs an implicit SSYNC, then simultaneously restores IMASK from the specifed data register and enters idle mode.

Sync Instruction Common Features

These ***16-bit instructions*** may not be issued in parallel with certain other 16-bit instructions.

These instructions may be used in either ***User or Supervisor mode***.

Sync Example

Example code sequence for IDLE

```
idle ;
```

Example code sequence for CSYNC---In this example, the CSYNC instruction ensures that the load instruction is not executed speculatively. CSYNC ensures that the conditional branch is resolved and any entries in the processor store buffer have been flushed. In addition, all speculative states or exceptions complete processing before CSYNC completes.

```
if cc jump away_from_here ;
/* produces speculative branch prediction */
csync ;
r0 = [p0] ; /* load */
```

Example code sequence for SSYNC---In this example, SSYNC ensures that the load instruction will not be executed speculatively. The instruction ensures that the conditional branch is resolved and any entries in the processor store buffer and write buffer have been flushed. In addition, all exceptions complete processing before SSYNC completes.

```
if cc jump away_from_here ;
/* produces speculative branch prediction */
ssync ;
r0 = [p0] ; /* load */
```

Example code sequence for `DSYNC`---In this example, `DSYNC` ensures that the load instruction will not be executed speculatively. The instruction ensures that the conditional branch is resolved and any entries in the processor store buffer and write buffer have been flushed. In addition, all exceptions complete processing before `DSYNC` completes.

```
if cc jump away_from_here ;
/* produces speculative branch prediction */
dsync ;
r0 = [p0] ; /* load */
```

Example code sequence for `STI IDLE`

```
sti idle r0 ;
```

SyncExcl (SyncExcl)

General Form

Long Load/Store with indexed addressing (LdStExcl)
syncexcl

Abstract

This instruction synchronizes the processor state with the exclusive state, capturing any pending write response and releasing exclusive memory access to a memory location.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

NOP (NOP)

General Form

16-bit Slot Nop (NOP16)
nop

Abstract

This instruction increments the PC (and does nothing else).

See Also (32-Bit No Operation (NOP32))

NOP Description

The No Op instruction increments the PC and does nothing else.

Typically, the No Op instruction allows previous instructions time to complete before continuing with subsequent instructions. Other uses are to produce specific delays in timing loops or to act as hardware event timers and rate generators when no timers and rate generators are available.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

NOP Example

```
nop ;
```

32-Bit No Operation (NOP32)

General Form

32-bit Slot Nop (NOP32)
mnop

Abstract

This instruction increments the PC (and does nothing else).

See Also (NOP (NOP))

NOP32 Description

The No Op instruction increments the PC and does nothing else.

Typically, the No Op instruction allows previous instructions time to complete before continuing with subsequent instructions. Other uses are to produce specific delays in timing loops or to act as hardware event timers and rate generators when no timers and rate generators are available.

MNOP can be used to issue loads or store instructions in parallel without invoking a 32-bit MAC or ALU operation.

This ***32-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

NOP32 Example

```
mnop ;
mnop || /* a 16-bit instr. */ || /* a 16-bit instr. */ ;
```

TestSet (TestSet)

General Form

Basic Program Sequencer Control Functions (ProgCtrl)
testset (PREGP Register Type)

Abstract

This instruction loads a byte, tests whether it is zero, then sets the most significant bit of the byte in memory. CC is set if the byte is originally zero, and cleared if the byte is originally nonzero. The sequence of memory transactions are atomic.

TestSet Description

The `testset` instruction is an atomic operation. (This sequence may be aborted by an interrupt, but will restart from the beginning upon return from interrupt. A byte protected in this manner may be used as a semaphore.) This instruction is primarily provided for backward compatability and it is recomended to use exclusive load and store instructions which make more efficient use of system resources (if that is possible). The `testset` instruction reads an indirectly addressed memory byte, tests whether it is zero, and then writes the byte back to memory with the most significant bit (MSB) set, all as one indivisible operation. If the byte is originally zero, the instruction sets the `CC` bit. If the byte is originally nonzero, the instruction clears the `CC` bit.

The `TESTSET` instruction is never executed speculatively. It is supported by bus-locked memory transactions on the system bus, so no other user of the bus, such as another core, can access memory between the test and set portions of this instruction. The `TESTSET` instruction can be interrupted by the core. If this happens the system bus is released and the `TESTSET` instruction is executed again upon return from the interrupt.

`TESTSET` should not be used in L1 SRAM or cacheable memory as its behavior in these regions varies between different derivatives. `TESTSET` must not be used with MMRs, I/O Device space or extended data access addresses.

This ***16-bit instruction*** may not be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either ***User or Supervisor mode***.

TestSet Example

```
testset (P3) ; /* test and set byte addressed by P3 */
```

Memory or Pointer Instructions

The memory and pointer instructions provide operations, which execute on the ***address arithmetic unit*** in the processor core. Users can take advantage of these instructions to load registers with data from memory locations, to store register data to memory locations, and to execute arithmetic operation using pointer registers.

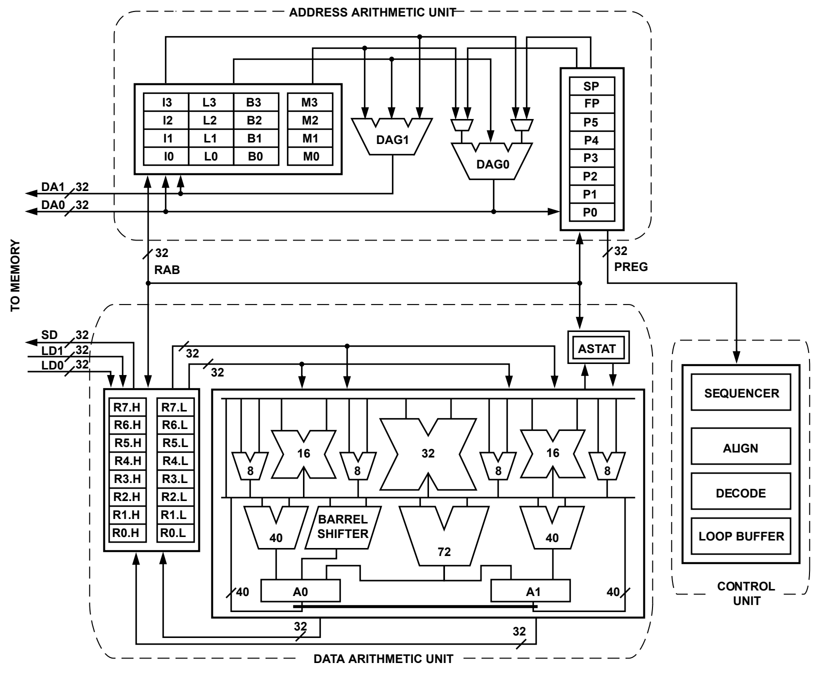


Figure 8-22: Blackfin+ Core Block Diagram

The operation types of memory or pointer instructions include:

- Memory Load Operations
- Memory Store Operations
- Pointer Math Operations

Load from Immediate (Value) Operations

These operations provide register load operations on register and immediate value operands:

- 32-Bit Accumulator Register (.x) Initialization (LdImmToAxX)
- 32-Bit Accumulator Register (.w) Initialization (LdImmToAxW)
- Accumulator Register Initialization (LdImmToAx)
- 16-Bit Register Initialization (LdImmToDregHL)
- 32-Bit Register Initialization (LdImmToReg)
- Dual Accumulator 0 and 1 Registers Initialization (LdImmToAxDual)

Accumulator Register Initialization (LdImmToAx)

General Form

ALU Operations (Dsp32Alu)
a0 = 0
a1 = 0

Abstract

This instruction loads the accumulator register with the immediate value 0 (initializes the result register).

See Also (32-Bit Accumulator Register (.x) Initialization (LdImmToAxX), 32-Bit Accumulator Register (.w) Initialization (LdImmToAxW))

LdImmToAx Description

The load immediate to accumulator instruction loads an immediate value (0) into an accumulator register. This operation initializes (or clears) the accumulator register.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdImmToAx Example

```
a0 = 0 ;
a1 = 0 ;
```

32-Bit Accumulator Register (.w) Initialization (LdImmToAxW)

General Form

Load Immediate Word (LdImm)
a0.w = imm32 Register Type
a1.w = imm32 Register Type

Abstract

This instruction initializes the lower 32-bits (.w) of the accumulator register from a 32-bit immediate value.

See Also (32-Bit Accumulator Register (.x) Initialization (LdImmToAxX), Accumulator Register Initialization (LdImmToAx))

LdImmToAxW Description

The load immediate to accumulator 32-bit section instruction loads a immediate value, or explicit constant, into the the `A0.w` or `A1.w` register.

The instruction loads the 32-bit accumulator section from a 32-bit quantity, depending on the size of the immediate data.

The load operation uses the 32 bits of the input immediate value and leaves the unspecified portion of the accumulator register intact.

This ***64-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdImmToAxW Example

```
a0.w = 0x7FFFFFFF ;
a1.w = 0x80000000 ;
a0.w = MyResult ;
a1.w = MyOtherResult ;
```

32-Bit Accumulator Register (.x) Initialization (LdImmToAxX)

General Form

Load Immediate Word (LdImm)
a0.x = imm32 Register Type
a1.x = imm32 Register Type

Abstract

This instruction initializes the upper 8-bit (.x)of the accumulator register from a 32-bit immediate value.

See Also (32-Bit Accumulator Register (.w) Initialization (LdImmToAxW), Accumulator Register Initialization (LdImmToAx))

LdImmToAxX Description

The load immediate to accumulator 8-bit section instruction loads a immediate value, or explicit constant, into the the `A0.x` or `A1.x` register.

The instruction loads the 8-bit accumulator section from a 32-bit quantity, depending on the size of the immediate data.

The load operation uses the least significant 8 bits of the input immediate value and leaves the unspecified portion of the accumulator register intact.

This ***64-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

LdImmToAxX Example

```
a0.x = 0x7FFFFFFF ;
a1.x = 0x80000000 ;
a0.x = MyResult ;
a1.x = MyOtherResult ;
```

16-Bit Register Initialization (LdImmToDregHL)

General Form

Load Immediate Half Word (LdImmHalf)
DST_L = imm16 Register Type
DST_H = imm16 Register Type

Abstract

This instruction loads a low-half register or a high-half register with a 16-bit immediate value.

LdImmToDregHL Description

The load immediate to high/low half register instruction loads an immediate value, or explicit constant, into a high or low half register. The instruction loads a 16-bit quantity, depending on the size of the immediate data.

The 16-bit half-words are be loaded into either the high half or low half of a register. The load operation leaves the unspecified half of the register intact.

Loading a 32-bit value into a register using this load immediate instruction requires two separate instructions—one for the high and one for the low half. For example, to load the address `foo` into register `P3`, write:

```
p3.h = foo ;
p3.l = foo ;
```

The assembler automatically selects the correct half-word portion of the 32-bit literal for inclusion in the instruction word.

This ***32-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

LdImmToDregHL Example

```
r7.h = 63 ;
p3.l = 12 ;
i0.l = 4 ;
m2.h = 8 ;
```

```
l3.h = 0xbcde ;
```

32-Bit Register Initialization (LdImmToReg)

General Form

Destructive Binary Operations, dreg with 7bit immediate (CompI2opD)
DREG Register Type = imm7 Register Type (x)
Destructive Binary Operations, preg with 7bit immediate (CompI2opP)
PREG Register Type = imm7 Register Type (x)
Load Immediate Half Word (LdImmHalf)
DST = imm16 Register Type (x)
DST = rimm16 Register Type (z)
Load Immediate Word (LdImm)
DREG Register Type = imm32 Register Type
PREG Register Type = imm32 Register Type
IREG Register Type = imm32 Register Type
MREG Register Type = imm32 Register Type
BREG Register Type = imm32 Register Type
LREG Register Type = imm32 Register Type
astat = imm32 Register Type
rets = imm32 Register Type
SYSREG2 Register Type = imm32 Register Type
SYSREG3 Register Type = imm32 Register Type

Abstract

This instruction initializes a 32-bit register to an immediate value. For the smaller instructions, where the immediate is less than 32, you can specify if you want the immediate value sign or zero extended to fill the register.

LdImmToReg Description

The load immediate to register instruction loads an immediate value, or explicit constants, into a register.

The instruction loads a 7-, 16-, or 32-bit quantity, depending on the size of the immediate data.

The zero-extended `(z)` versions of this instruction fill the upper bits of the destination register with zeros. The sign-extended `(x)` versions of this instruction fill the upper bits with the sign of the constant value.

The instruction opcode size varies with the immediate value size as follows:

- Load immediate to register of 7-bit data encodes as a ***16-bit instruction***.
- Load immediate to register of 16-bit data encodes as a ***32-bit instruction***.
- Load immediate to register of 7-bit data encodes as a ***64-bit instruction***.

These load immediate instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdImmToReg Example

```
r7 = 63 (z) ;
p3 = 12 (z) ;
r0 = -344 (x) ;
r7 = 436 (z) ;
m2 = 0x89ab (z) ;
p1 = 0x1234 (z) ;
m3 = 0x3456 (x) ;
```

Dual Accumulator 0 and 1 Registers Initialization (LdImmToAxDual)

General Form

ALU Operations (Dsp32Alu)
a1 = a0 = 0

Abstract

This instruction loads the accumulator 0 and 1 registers (A0, A1) with the immediate value 0 (initializes both result registers).

LdImmToAxDual Description

The dual load immediate to accumulator instruction loads an immediate value (0) into both accumulator registers. This operation initializes (or clears) both of the accumulator registers.

This ***32-bit instruction*** can sometimes save execution time (over a 16-bit encoded instruction) because it can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdImmToAxDual Example

```
a1 = a0 = 0 ;
```

Memory Load Operations

These operations provide memory load operations on register and immediate value operands:

- 8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
- 16-Bit Load from Memory (LdM16bitToDregH)
- 16-Bit Load from Memory (LdM16bitToDregL)
- 32-Bit Load from Memory (LdM32bitToDreg)
- 16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
- 32-Bit Register Initialization (LdImmToReg)

8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)

General Form

Load/Store (LdSt)
DREG Register Type = b[PREG Register Type++] (z)
DREG Register Type = b[PREG Register Type--] (z)
DREG Register Type = b[PREG Register Type] (z)
DREG Register Type = b[PREG Register Type++] (x)
DREG Register Type = b[PREG Register Type--] (x)
DREG Register Type = b[PREG Register Type] (x)
Long Load/Store with indexed addressing (LdStIdxI)
DREG Register Type = b[PREG Register Type + imm16reloc Register Type] (z)
DREG Register Type = b[PREG Register Type + imm16reloc Register Type] (x)
Load/Store 32-bit Absolute Address (LdStAbs)
DREG Register Type = b[uimm32 Register Type] (z)
DREG Register Type = b[uimm32 Register Type] (x)

Abstract

This instruction loads a register with an 8-bit value from memory. The value is sign or zero extended in the register.

See Also (32-Bit Load from Memory (LdM32bitToDreg), 16-Bit Load from Memory (LdM16bitToDregH), 16-Bit Load from Memory (LdM16bitToDregL), 16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg))

LdM08bitToDreg Description

The load byte to data register instruction loads an 8-bit byte value from a memory location into a 32-bit data register. The address of the memory location is identified with a pointer register, a pointer plus an offset, or a 32-bit absolute address. The byte value is sign-extended `(x)` or zero-extended `(z)` to 32 bits in the destination data register. The address used in this instruction has no restrictions for memory address alignment. This instruction supports the following options.

- Post-increment the source pointer by 1 byte `[Preg ++]`
- Post-decrement the source pointer by 1 byte `[Preg --]`
- Offset the source pointer with a 16-bit signed constant `[Preg + Offset]`

The instruction opcode size varies with the address type as follows:

- Load byte to register using a pointer register for the address encodes as a ***16-bit instruction***.
- Load byte to register using a pointer register with 16-bit offset for the address encodes as a ***32-bit instruction***.
- Load byte to register using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit load byte instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load byte instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdM08bitToDreg Example

```
r3 = b [ p0 ] (z) ;
r7 = b [ p1 ++ ] (z) ;
r2 = b [ sp -- ] (z) ;
r0 = b [ p4 + 0xFFFF800F ] (z) ;
r3 = b [ p0 ] (x) ;
r7 = b [ p1 ++ ](x) ;
r2 = b [ sp -- ] (x) ;
r0 = b [ p4 + 0xFFFF800F ](x) ;
```

16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)

General Form

Load/Store postmodify addressing, pregister based (LdStPmod)
DREG Register Type = w[PREG Register Type ++ PREG Register Type] (z)
DREG Register Type = w[PREG Register Type ++ PREG Register Type] (x)
Load/Store (LdSt)
DREG Register Type = w[PREG Register Type++] (z)
DREG Register Type = w[PREG Register Type--] (z)
DREG Register Type = w[PREG Register Type] (z)
DREG Register Type = w[PREG Register Type++] (x)

DREG Register Type = w[PREG Register Type--] (x)
DREG Register Type = w[PREG Register Type] (x)
Load/Store indexed with small immediate offset (LdStII)
DREG Register Type = w[PREG Register Type + uimm4s2 Register Type] (z)
DREG Register Type = w[PREG Register Type + uimm4s2 Register Type] (x)
Long Load/Store with indexed addressing (LdStIdxI)
DREG Register Type = w[PREG Register Type + imm16s2 Register Type] (z)
DREG Register Type = w[PREG Register Type + imm16s2 Register Type] (x)
Load/Store 32-bit Absolute Address (LdStAbs)
DREG Register Type = w[uimm32 Register Type] (z)
DREG Register Type = w[uimm32 Register Type] (x)

Abstract

This instruction loads a register with a 16-bit value from memory. The value is sign or zero extended in the register.

See Also (8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg), 32-Bit Load from Memory (LdM32bit-ToDreg), 16-Bit Load from Memory (LdM16bitToDregH), 16-Bit Load from Memory (LdM16bitToDregL))

LdM16bitToDreg Description

The load word to data register instruction loads a 16-bit value from a memory location into a 32-bit data register. The address of the memory location is identified with a pointer register, a pointer plus an offset, or a 32-bit absolute address. The word value is sign-extended `(x)` or zero-extended `(z)` to 32 bits in the destination data register. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source pointer by 2 bytes `[Preg ++]`
- Post-decrement the source pointer by 2 bytes `[Preg --]`
- Offset the source pointer with a 5-bit signed constant `[Preg + SmallOffset]`
- Offset the source pointer with a 16-bit signed constant `[Preg + LargeOffset]`
- Offset the source pointer with second pointer `[Preg ++ Preg]`

The syntax of the form:

```
Dest = w [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest = w [Src_1] ; /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- `Dest` is the destination register. (`Dreg` in the syntax example).
- `Src_1` is the first source register on the right-hand side of the equation.
- `Src_2` is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0 = w [p2 ++ p2] (z) ;
```

functions as:

```
r0 = w [p2] (z) ;
```

The instruction opcode size varies with the address type as follows:

- Load word to register using a pointer register for the address or using a pointer register with small offset for the address encodes as a ***16-bit instruction***.
- Load word to register using a pointer register with 16-bit offset for the address or using a pointer register offset by a second pointer for the address encodes as a ***32-bit instruction***.
- Load word to register using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdM16bitToDreg Example

```
r3 = w [ p0 ] (z) ;
r7 = w [ p1 ++ ] (z) ;
r2 = w [ sp -- ] (z) ;
r6 = w [ p2 + 12 ] (z) ;
r0 = w [ p4 + 0x8004 ] (z) ;
r1 = w [ p0 ++ p1 ] (z) ;
r3 = w [ p0 ] (x) ;
r7 = w [ p1 ++ ] (x) ;
r2 = w [ sp -- ] (x) ;
r6 = w [ p2 + 12 ] (x) ;
r0 = w [ p4 + 0x800E ] (x) ;
r1 = w [ p0 ++ p1 ] (x) ;
```

16-Bit Load from Memory (LdM16bitToDregH)

General Form

Load/Store postmodify addressing, pregister based (LdStPmod)
DREG_H Register Type = w[PREG Register Type ++ PREG Register Type]
Load/Store (DspLdSt)
DREG_H Register Type = w[IREG Register Type++]
DREG_H Register Type = w[IREG Register Type--]
DREG_H Register Type = w[IREG Register Type]
Load/Store 32-bit Absolute Address (LdStAbs)
DREG_H Register Type = w[uimm32 Register Type]

Abstract

This instruction loads a high-half register with a 16-bit value from memory.

See Also (8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg), 32-Bit Load from Memory (LdM32bitToDreg), 16-Bit Load from Memory (LdM16bitToDregL), 16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg))

LdM16bitToDregH Description

The load word to high-half data register instruction loads a 16-bit value from a memory location into a 16-bit high-half data register. The operation does not affect the related low-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes `[`*`Ireg`*` ++]`
- Post-decrement the source index by 2 bytes `[`*`Ireg`*` --]`
- Offset the source pointer with second pointer `[`*`Preg`*` ++ `*`Preg`*`]`

The instruction versions that explicitly modify an index register (*`Ireg`*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*`Lreg`*) corresponding to the *`Ireg`* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *`Lreg`* beforehand can result in unexpected *`Ireg`* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
Dest_hi = w [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest_hi = w [Src_1] ; /* load the 16-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- `Dest_hi` is the destination high-half register. (`Dreg_hi` in the syntax example).
- `Src_1` is the first source register on the right-hand side of the equation.
- `Src_2` is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0.h = w [p2 ++ p2] ;
```

functions as:

```
r0.h = w [p2] ;
```

The instruction opcode size varies with the address type as follows:

- Load word to high-half register using an index register or a pointer register for the address encodes as a ***16-bit instruction***.
- Load word to high-half register using a pointer register offset by a second pointer for the address encodes as a ***16-bit instruction***.
- Load word to high-half register using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdM16bitToDregH Example

```
r3.h = w [ i1 ] ;
r7.h = w [ i3 ++ ] ;
r1.h = w [ i0 -- ] ;
r2.h = w [ p4 ] ;
r5.h = w [ p2 ++ p0 ] ;
```

16-Bit Load from Memory (LdM16bitToDregL)

General Form

Load/Store postmodify addressing, pregister based (LdStPmod)
DREG_L Register Type = w[PREG Register Type ++ PREG Register Type]
Load/Store (DspLdSt)
DREG_L Register Type = w[IREG Register Type++]
DREG_L Register Type = w[IREG Register Type--]
DREG_L Register Type = w[IREG Register Type]
Load/Store 32-bit Absolute Address (LdStAbs)
DREG_L Register Type = w[uimm32 Register Type]

Abstract

This instruction loads a low-half register with a 16-bit value from memory.

See Also (8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg), 32-Bit Load from Memory (LdM32bitToDreg), 16-Bit Load from Memory (LdM16bitToDregH), 16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg))

LdM16bitToDregL Description

The load word to low-half data register instruction loads a 16-bit value from a memory location into a 16-bit low-half data register. The operation does not affect the related high-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes `[`*`Ireg`*` ++]`
- Post-decrement the source index by 2 bytes `[`*`Ireg`*` --]`
- Offset the source pointer with second pointer `[`*`Preg`*` ++ `*`Preg`*`]`

The instruction versions that explicitly modify an index register (*`Ireg`*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*`Lreg`*) corresponding to the *`Ireg`* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *`Lreg`* beforehand can result in unexpected *`Ireg`* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
Dest_lo = w [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest_lo = w [Src_1] ; /* load the 16-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

• `Dest_lo` is the destination low-half register. (`Dreg_lo` in the syntax example).

• `Src_1` is the first source register on the right-hand side of the equation.

• `Src_2` is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0.l = w [p2 ++ p2] ;
```

functions as:

```
r0.l = w [p2] ;
```

The instruction opcode size varies with the address type as follows:

- Load word to low-half register using an index register or a pointer register for the address encodes as a ***16-bit instruction***.
- Load word to low-half register using a pointer register offset by a second pointer for the address encodes as a ***16-bit instruction***.
- Load word to low-half register using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdM16bitToDregL Example

```
r3.l = w[ i1 ] ;
r7.l = w[ i3 ++ ] ;
r1.l = w[ i0 -- ] ;
r2.l = w[ p4 ] ;
r5.l = w[ p2 ++ p0 ] ;
```

32-Bit Load from Memory (LdM32bitToDreg)

General Form

Load/Store postmodify addressing, pregister based (LdStPmod)
DREG Register Type = [PREG Register Type ++ PREG Register Type]
Load/Store (DspLdSt)
DREG Register Type = [IREG Register Type++]
DREG Register Type = [IREG Register Type--]
DREG Register Type = [IREG Register Type]
DREG Register Type = [IREG Register Type ++ MREG Register Type]
Load/Store (LdSt)
DREG Register Type = [PREG Register Type++]
DREG Register Type = [PREG Register Type--]
DREG Register Type = [PREG Register Type]
Load/Store indexed with small immediate offset FP (LdStIIFP)
DREG Register Type = [fp - imm5nzs4negpos Register Type]
Load/Store indexed with small immediate offset FP (LdpIIFP)
PREG Register Type = [fp - imm5nzs4negpos Register Type]
Load/Store indexed with small immediate offset (LdStII)
DREG Register Type = [PREG Register Type + uimm4s4 Register Type]
Long Load/Store with indexed addressing (LdStIdxI)
DREG Register Type = [PREG Register Type + imm16s4 Register Type]
Load/Store 32-bit Absolute Address (LdStAbs)
DREG Register Type = [uimm32 Register Type]
PREG Register Type = [uimm32 Register Type]

Abstract

This instruction loads a register with a 32-bit value from memory.

See Also (8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg), 16-Bit Load from Memory (LdM16bit-ToDregH), 16-Bit Load from Memory (LdM16bitToDregL), 16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg))

LdM32bitToDreg Description

The load 32-bit data to data register instruction loads a 32-bit value from a memory location into a data register. The address of the memory location is identified with an index register, an index register plus an offset, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 4 bytes `[`*`Ireg`*` ++]`
- Post-decrement the source index by 4 bytes `[`*`Ireg`*` --]`
- Offset the source index with a modifier `[`*`Ireg`*` ++ `*`Mreg`*`]`
- Post-increment the source pointer by 4 bytes `[`*`Preg`*` ++]`
- Post-decrement the source pointer by 4 bytes `[`*`Preg`*` --]`
- Offset the source frame pointer with a 5-bit signed constant `[FP - `*`SmallOffset`*`]`
- Offset the source pointer with a 5-bit signed constant `[`*`Preg`*` + `*`SmallOffset`*`]`
- Offset the source pointer with a 16-bit signed constant `[`*`Preg`*` + `*`LargeOffset`*`]`
- Offset the source pointer with second pointer `[`*`Preg`*` ++ `*`Preg`*`]`

The instruction versions that explicitly modify an index register (*`Ireg`*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*`Lreg`*) corresponding to the *`Ireg`* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *`Lreg`* beforehand can result in unexpected *`Ireg`* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
Dest = [ Src_1 ++ Src_2 ] ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
Dest = [Src_1] ; /* load the 32-bit destination, indirect*/
Src_1 += Src_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *`Dest`* is the destination high-half register. (*`Dreg`* in the syntax example).
- *`Src_1`* is the first source register on the right-hand side of the equation.
- *`Src_2`* is the second source register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
r0 = [p2 ++ p2] ;
```

functions as:

```
r0 = [p2] ;
```

The instruction opcode size varies with the address type as follows:

- Load 32-bit data to register using an index register or a pointer register for the address encodes as a ***16-bit instruction***.
- Load 32-bit data to register using an index register offset by a modifier register or a pointer register offset by a second pointer for the address encodes as a ***16-bit instruction***.
- Load 32-bit data to register using a pointer or frame pointer register with a small offset for the address encodes as a ***16-bit instruction***.
- Load 32-bit data to register using a pointer register with a large offset for the address encodes as a ***32-bit instruction***.
- Load 32-bit data to register using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit load 32-bit data to register instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load 32-bit data to register instructions may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

LdM32bitToDreg Example

```
r3 = [ p0 ] ;
r7 = [ p1 ++ ] ;
r2 = [ sp -- ] ;
r6 = [ p2 + 12 ] ;
r0 = [ p4 + 0x800C ] ;
r1 = [ p0 ++ p1 ] ;
r5 = [ fp -12 ] ;
r2 = [ i2 ] ;
r0 = [ i0 ++ ] ;
r0 = [ i0 -- ] ;
/* Before indirect post-increment indexed addressing*/
r7 = 0 ;
i3 = 0x4000 ; /* Memory location contains 15, for example.*/
m0 = 4 ;
r7 = [i3 ++ m0] ;
  /* Afterwards . . .*/
  /* r7 = 15 from memory location 0x4000*/
  /* i3 = i3 + m0 = 0x4004*/
  /* m0 still equals 4*/
```

32-Bit Pointer Load from Memory (LdM32bitToPreg)

General Form

Load/Store (Ldp)
PREG Register Type = [PREG Register Type++]
PREG Register Type = [PREG Register Type--]
PREG Register Type = [PREG Register Type]
Load/Store indexed with small immediate offset (LdpII)
PREG Register Type = [PREG Register Type + uimm4s4 Register Type]
Long Load/Store with indexed addressing (LdStIdxI)
PREG Register Type = [PREG Register Type + imm16s4 Register Type]

Abstract

This instruction loads a pointer register with 7-bit immediate value.

LdM32bitToPreg Description

The load 32-bit data to pointer register instruction loads a 32-bit value from a memory location into a pointer register. The address of the memory location is identified with a pointer register or a pointer plus an offset. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source pointer by 4 bytes `[Preg ++]`
- Post-decrement the source pointer by 4 bytes `[Preg --]`
- Offset the source pointer with a 5-bit signed constant `[Preg + SmallOffset]`
- Offset the source pointer with a 16-bit signed constant `[Preg + LargeOffset]`

The instruction opcode size varies with the address type as follows:

- Load 32-bit data to register using a pointer register for the address encodes as a ***16-bit instruction***.
- Load 32-bit data to register using a pointer register with a small offset for the address encodes as a ***16-bit instruction***.
- Load 32-bit data to register using a pointer register with a large offset for the address encodes as a ***32-bit instruction***.

The 16-bit load 32-bit data to pointer instructions may be issued in parallel with certain other instructions. The 32-bit load 32-bit data to pointer instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

LdM32bitToPreg Example

```
p3 = [ p2 ] ;
p5 = [ p0 ++ ] ;
p2 = [ sp -- ] ;
p3 = [ p2 + 8 ] ;
p0 = [ p2 + 0x4008 ] ;
p1 = [ fp - 16 ] ;
```

Memory Load (Exclusive) Operations

The memory load (exclusive) operations read data from memory (similar to a non-exclusive, "regular" memory load operation) and establish exclusive access to the memory location. When the memory location is in *non-shareable* memory, the memory load (exclusive) operation loads through the memory management unit in exactly the same manner as a regular memory load from the same memory location. When the memory location is in *shareable* memory, the memory load (exclusive) performs an exclusive read on the memory bus. For more information about illegal, non-shareable, and shareable memory regions, see the Exclusive Loads and Stores section of the Memory chapter.

The memory load (exclusive) operations include the following instructions:

- 8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg)
- 32-Bit Load from Memory (LdX32bitToDreg)
- 16-Bit Load from Memory (LdX16bitToDregH)
- 16-Bit Load from Memory (LdX16bitToDregL)
- 16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg)

When the memory management unit successfully completes an memory load (exclusive), the destination data register is updated with the loaded value and the `SEQSTAT.XMONITOR` bit is set, as shown in the ***Exclusive Related Bits in Status Register (SEQSTAT)*** table.

Table 8-23: Exclusive Related Bits in Status Register (SEQSTAT)

Name	Description	Condition
`SEQSTAT.XMONITOR`	Excl. monitor (0=open, 1=exclusive)	Always updated
	=1 after completion of exclusive load	

When the memory management unit cannot complete an memory load (exclusive) a number of exceptions and errors may be issued, in addition to those that may be caused by a regular memory load. The ***Exceptions/Errors from Unsuccessful Memory Load (Exclusive) Operations*** table lists these exceptions and errors.

Table 8-24: Exceptions/Errors from Unsuccessful Memory Load (Exclusive) Operations

Condition	Exception or Hardware Error
Access to misaligned address	The data access generated a misaligned address violation exception. The address for the exclusive access must be aligned. This restriction holds even if misaligned accesses are supported generally.
Access to core MMR	The data access attempted an illegal use of a supervisor resource.
Access to I/O device space	The data access generated a CPLB protection violation exception. This exception occurs when the access is to memory marked as I/O device space in the CPLB.
Access to non-exclusive slave	A load exclusive from a shareable memory region accessed a memory device without hardware support for exclusive accesses. The program which recieves this exception should not execute the store exclusive to this address. The store exclusive will be unsuccessful causing the exclusive instruction sequence to be retried and the program to loop. This condition has to be treated as a programming error and the program restructured to place the semaphore in another memory location, or use the TESTSET instruction.

8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg)

General Form

Long Load/Store with indexed addressing (LdStExcl)
DREG Register Type = b[PREG Register Type] (z,excl)
DREG Register Type = b[PREG Register Type] (x,excl)

Abstract

This instruction loads a register with an 8-bit value from memory, using an exclusive memory read. The value is sign or zero extended in the register.

See Also (32-Bit Load from Memory (LdX32bitToDreg), 16-Bit Load from Memory (LdX16bitToDregH), 16-Bit Load from Memory (LdX16bitToDregL), 16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg))

LdX08bitToDreg Description

The load data register from memory (8-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the least significant byte of the register with zero- or sign-extension) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX08bitToDreg Example

```
r1 = b[p4] (x,excl);   /* load exclusive 8-bits sign sign to D-register */
```

16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg)

General Form

Long Load/Store with indexed addressing (LdStExcl)
DREG Register Type = w[PREG Register Type] (z,excl)
DREG Register Type = w[PREG Register Type] (x,excl)

Abstract

This instruction loads a register with a 16-bit value from memory, using an exclusive memory read. The value is sign or zero extended in the register.

See Also (8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg), 32-Bit Load from Memory (LdX32bitToDreg), 16-Bit Load from Memory (LdX16bitToDregH), 16-Bit Load from Memory (LdX16bitToDregL))

LdX16bitToDreg Description

The load data register from memory (16-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the least significant 16-bits of the register with zero- or sign-extension) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX16bitToDreg Example

```
r2 = w[p4] (z,excl);   /* load exclusive 16-bits zero extend to D-register */
r3 = w[p4] (x,exc;)   /* load exclusive 16-bits sign extend to D-register */
```

16-Bit Load from Memory (LdX16bitToDregH)

General Form

Long Load/Store with indexed addressing (LdStExcl)
DREG_H Register Type = w[PREG Register Type] (excl)

Abstract

This instruction loads a high-half register with a 16-bit value from memory, using an exclusive memory read.

See Also (8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg), 32-Bit Load from Memory (LdX32bitToDreg), 16-Bit Load from Memory (LdX16bitToDregL), 16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg))

LdX16bitToDregH Description

The load high half data register from memory (16-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the half register) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX16bitToDregH Example

```
r1.h = w[p4] (excl);   /* load exclusive 16-bits zero sign to high D-register half */
```

16-Bit Load from Memory (LdX16bitToDregL)

General Form

Long Load/Store with indexed addressing (LdStExcl)
DREG_L Register Type = w[PREG Register Type] (excl)

Abstract

This instruction loads a low-half register with a 16-bit value from memory, using an exclusive memory read.

See Also (8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg), 32-Bit Load from Memory (LdX32bitToDreg), 16-Bit Load from Memory (LdX16bitToDregH), 16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg))

LdX16bitToDregL Description

The load low half data register from memory (16-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the half register) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX16bitToDregL Example

```
r1.l = w[p4] (excl);   /* load exclusive 16-bits zero sign to low D-register half */
```

32-Bit Load from Memory (LdX32bitToDreg)

General Form

Long Load/Store with indexed addressing (LdStExcl)
DREG Register Type = [PREG Register Type] (excl)

Abstract

This instruction loads a register with a 32-bit value from memory, using an exclusive memory read.

See Also (8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg), 16-Bit Load from Memory (LdX16bitToDregH), 16-Bit Load from Memory (LdX16bitToDregL), 16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg))

LdX32bitToDreg Description

The load data register from memory (32-bit transfer) checks for exclusive access to a memory location and reads a data value (loading it into the register) from the location only if exclusive access is still held.

For more information about memory load (exclusive) operations, see Memory Load (Exclusive) Operations.

LdX32bitToDreg Example

```
r0 = [p4] (excl);      /* load exclusive 32-bits to D-register */
```

Pack Operations

These operations provide byte packing and unpacking operations on register and register pair operands:

- Pack 8-Bit to 32-Bit (BytePack)
- Spread 8-Bit to 16-Bit (ByteUnPack)
- Pack 16-Bit to 32-Bit (Pack16Vec)

Pack 8-Bit to 32-Bit (BytePack)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = bytepack (DREG Register Type, DREG Register Type)

Abstract

This instruction takes the low bytes from each 16-bit register half of two registers and combines them to create a single 32-bit register. Used to re-order data.

See Also (Spread 8-Bit to 16-Bit (ByteUnPack), Pack 16-Bit to 32-Bit (Pack16Vec))

BytePack Description

The Quad 8-Bit Pack instruction packs four 8-bit values, half-word aligned, contained in two source registers into one register, byte aligned. The ***Source Registers Contain*** figure and ***Destination Register Receives*** figure show the packing pattern.

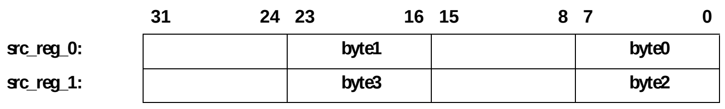


Figure 8-23: Source Registers Contain

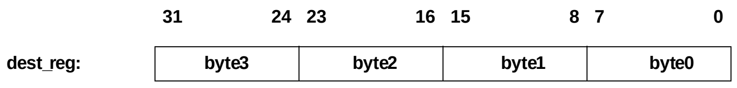


Figure 8-24: Destination Register Receives

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

This ***16-bit instruction*** may be issued in parallel with other 16-bit instructions.

This instruction may be used in either ***User or Supervisor mode***.

BytePack Example

```
r2 = bytepack (r4,r5) ;
/* Assume the following: ... */
/*    R4 = 0xFEED FACE */
/*    R5 = 0xBEEF BADD */
/* Then, this instruction returns: ... */
/*    R2 = 0xEFDD EDCE */
```

Spread 8-Bit to 16-Bit (ByteUnPack)

General Form

ALU Operations (Dsp32Alu)
(DREG Register Type, DREG Register Type) = byteunpack PAIR0 RS

Abstract

This instruction spreads four bytes to four zero extended 16-Bit values. The lower two bits of I0 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair.

See Also (Pack 8-Bit to 32-Bit (BytePack), Pack 16-Bit to 32-Bit (Pack16Vec))

ByteUnPack Description

The Quad 8-Bit Unpack instruction copies four contiguous bytes from a pair of source registers, adjusting for byte alignment. The instruction loads the selected bytes into two arbitrary data registers on half-word alignment. The two LSBs of the `I0` register determine the source byte alignment, as shown in the ***I-register Bits and the Byte Alignment, no (r) option*** figure. This figure shows the default source order case---not the `(r)` syntax---and the data contained in the source register pair. This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Figure 8-25: I-register Bits and the Byte Alignment, no (r) option

The `(r)` syntax reverses the order of the source registers within the pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The `(r)` option causes the low order bytes to come from the high register. In the optional reverse source order case (for example, using the `(r)` syntax), the only difference is the source registers swap places in their byte ordering. Assume the source register pair contains the data shown in the ***I-register Bits and the Byte Alignment, with (r) option*** figure.

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Figure 8-26: I-register Bits and the Byte Alignment, with (r) option

The four bytes, now byte aligned, are copied into the destination registers on half-word alignment, as shown in the ***Source Register Contains*** figure and the ***Destination Registers Receive*** figure.

	31 24	23 16	15 8	7 0
Aligned bytes :	byte_D	byte_C	byte_B	byte_A

Figure 8-27: Source Register Contains

	31 24	23 16	15 8	7 0
dest_reg_0:		byte_B		byte_A
dest_reg_1:		byte_D		byte_C

Figure 8-28: Destination Registers Receive

Only register pairs `R1:0` and `R3:2` are valid sources for this instruction, and the destination registers must be unique. Misaligned access exceptions are disabled during this instruction.

This ***16-bit instruction*** may be issued in parallel with other 16-bit instructions.

This instruction may be used in either ***User or Supervisor mode***.

ByteUnPack Example

```
(r6,r5) = byteunpack r1:0 ; /* non-reversing sources */
```

```
/* Assume the following: */
/* ... register I0’s two LSBs = 00b,  */
/* ... R1 = 0xFEED FACE  */
/* ... R0 = 0xBEEF BADD  */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00BE 00EF  */
/* ... R5 = 0x00BA 00DD  */
/* Assume the following: */
/* ... register I0’s two LSBs = 01b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00CE 00BE */
/* ... R5 = 0x00EF 00BA */
/* Assume the following: */
/* ... register I0’s two LSBs = 10b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00FA 00CE */
/* ... R5 = 0x00BE 00EF */
/* Assume the following: */
/* ... register I0’s two LSBs = 11b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00ED 00FA */
/* ... R5 = 0x00CE 00BE */

(r6,r5) = byteunpack r1:0 (R) ; /* reversing sources case */
/* Assume the following: */
/* ... register I0’s two LSBs = 00b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00FE 00ED */
/* ... R5 = 0x00FA 00CE */
/* Assume the following: */
/* ... register I0’s two LSBs = 01b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00DD 00FE */
/* ... R5 = 0x00ED 00FA */
/* Assume the following: */
/* ... register I0’s two LSBs = 10b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00BA 00DD */
/* ... R5 = 0x00FE 00ED */
/* Assume the following: */
/* ... register I0's two LSBs = 11b, */
/* ... R1 = 0xFEED FACE */
/* ... R0 = 0xBEEF BADD */
/* ... Then, this instruction returns: */
/* ... R6 = 0x00EF 00BA */
/* ... R5 = 0x00DD 00FE */
```

Pack 16-Bit to 32-Bit (Pack16Vec)

General Form

Shift (Dsp32Shf)
DREG Register Type = pack (DREG_L Register Type, DREG_L Register Type)
DREG Register Type = pack (DREG_L Register Type, DREG_H Register Type)
DREG Register Type = pack (DREG_H Register Type, DREG_L Register Type)
DREG Register Type = pack (DREG_H Register Type, DREG_H Register Type)

Abstract

This instruction packs two 16-bit half registers into one 32-bit register.

See Also (Pack 8-Bit to 32-Bit (BytePack), Spread 8-Bit to 16-Bit (ByteUnPack))

Pack16Vec Description

The vector pack instruction packs two 16-bit half-word numbers into the halves of a 32-bit data register as shown in the *Source Registers Contain* figure and the *Destination Register Contains* figure.

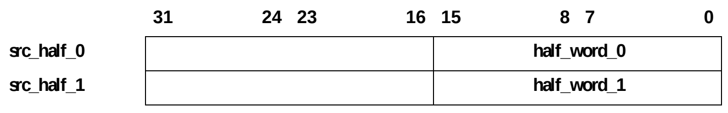


Figure 8-29: Source Registers Contain

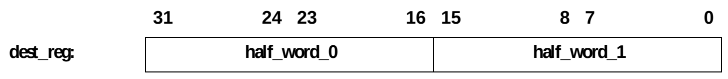


Figure 8-30: Destination Register Contains

This *16-bit instruction* may be issued in parallel with certain other 16-bit instructions.

This instruction may be used in either *User or Supervisor mode*.

Pack16Vec Example

```
r3=pack(r4.l, r5.l) ; /* pack low / low half-words */
```

```
r1=pack(r6.l, r4.h) ; /* pack low / high half-words */
r0=pack(r2.h, r4.l) ; /* pack high / low half-words */
r5=pack(r7.h, r2.h) ; /* pack high / high half-words */

/* Special Applications */
/* If r4.l = 0xDEAD and r5.l = 0xBEEF, then . . . */
r3 = pack (r4.l, r5.l) ;
/* . . . produces r3 = 0xDEAD BEEF */
/* example needed here */
```

Memory Store Operations

These operations provide memory store operations on register and immediate value operands:

- 8-Bit Store to Memory (StDregToM08bit)
- 16-Bit Store to Memory (StDregLToM16bit)
- 16-Bit Store to Memory (StDregHToM16bit)
- 32-Bit Store to Memory (StDregToM32bit)
- Store Pointer (StPregToM32bit)

16-Bit Store to Memory (StDregHToM16bit)

General Form

Load/Store postmodify addressing, pregister based (LdStPmod)
w[PREG Register Type ++ PREG Register Type] = DREG_H Register Type
Load/Store (DspLdSt)
w[IREG Register Type++] = DREG_H Register Type
w[IREG Register Type--] = DREG_H Register Type
w[IREG Register Type] = DREG_H Register Type
Load/Store 32-bit Absolute Address (LdStAbs)
w[uimm32 Register Type] = DREG_H Register Type

Abstract

This instruction stores the most significant 16-bit value from a register to memory.

See Also (8-Bit Store to Memory (StDregToM08bit), 32-Bit Store to Memory (StDregToM32bit), 16-Bit Store to Memory (StDregLToM16bit))

StDregHToM16bit Description

The store word from high-half data register instruction stores a 16-bit value from a high-half data register to a memory location. The operation does not affect the related low-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes `[Ireg ++]`
- Post-decrement the source index by 2 bytes `[Ireg --]`
- Offset the source pointer with second pointer `[Preg ++ Preg]`

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
w [ Dest_1 ++ Dest_2 ] = Src_hi ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
w [Dest_1] = Src_hi ; /* store to the 16-bit destination, indirect*/
Dest_1 += Dest_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Src_hi* is the source high-half register. (*Dreg_hi* in the syntax example).
- *Dest_1* is the first destination register on the left-hand side of the equation.
- *Dest_2* is the second destination register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
w [p2 ++ p2] = r0.h ;
```

functions as:

```
w [p2] = r0.h ;
```

The instruction opcode size varies with the address type as follows:

- Store word to memory using an index register or a pointer register for the address encodes as a ***16-bit instruction***.
- Store word to memory using a pointer register offset by a second pointer for the address encodes as a ***16-bit instruction***.
- Store word to memory using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

StDregHToM16bit Example

```
w[ i1 ] = r3.h ;
w[ i3 ++ ] = r7.h ;
w[ i0 -- ] = r1.h ;
w[ p4 ] = r2.h ;
w[ p2 ++ p0 ] = r5.h ;
```

16-Bit Store to Memory (StDregLToM16bit)

General Form

Load/Store postmodify addressing, pregister based (LdStPmod)
w[PREG Register Type ++ PREG Register Type] = DREG_L Register Type
Load/Store (DspLdSt)
w[IREG Register Type++] = DREG_L Register Type
w[IREG Register Type--] = DREG_L Register Type
w[IREG Register Type] = DREG_L Register Type
Load/Store (LdSt)
w[PREG Register Type++] = DREG Register Type
w[PREG Register Type--] = DREG Register Type
w[PREG Register Type] = DREG Register Type
Load/Store indexed with small immediate offset (LdStII)
w[PREG Register Type + uimm4s2 Register Type] = DREG Register Type
Long Load/Store with indexed addressing (LdStIdxI)
w[PREG Register Type + imm16s2 Register Type] = DREG Register Type
Load/Store 32-bit Absolute Address (LdStAbs)
w[uimm32 Register Type] = DREG Register Type

Abstract

This instruction stores the least significant 16-bit value from a register to memory.

See Also (8-Bit Store to Memory (StDregToM08bit), 32-Bit Store to Memory (StDregToM32bit), 16-Bit Store to Memory (StDregHToM16bit))

StDregLToM16bit Description

The store word from low-half data register instruction stores a 16-bit value *either* from a low-half data register *or* from the least significant 16 bits of a data register to a memory location. The operation does not affect the related high-half register. The address of the memory location is identified with an index register, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (2-byte half-word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 2 bytes `[Ireg ++]`
- Post-decrement the source index by 2 bytes `[Ireg --]`
- Offset the source pointer with second pointer `[Preg ++ Preg]`

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
w [ Dest_1 ++ Dest_2 ] = Src_hi ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
w [Dest_1] = Src_hi ; /* store to the 16-bit destination, indirect*/
Dest_1 += Dest_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Src_hi* is the source high-half register. (*Dreg_hi* in the syntax example).
- *Dest_1* is the first destination register on the left-hand side of the equation.
- *Dest_2* is the second destination register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
w [p2 ++ p2] = r0.h ;
```

functions as:

```
w [p2] = r0.h ;
```

The instruction opcode size varies with the address type as follows:

- Store word to memory using an index register or a pointer register for the address encodes as a ***16-bit instruction***.
- Store word to memory using a pointer register offset by a second pointer for the address encodes as a ***16-bit instruction***.
- Store word to memory using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit load word instructions may be issued in parallel with certain other instructions. The 64-bit load word instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

StDregLToM16bit Example

```
w [ i1 ] = r3.l ;
w [ p0 ] = r3 ;
w [ i3 ++ ] = r7.l ;
w [ i0 -- ] = r1.l ;
w [ p4 ] = r2.l ;
w [ p1 ++ ] = r7 ;
w [ sp -- ] = r2 ;
w [ p2 + 12 ] = r6 ;
w [ p4 - 0x200C ] = r0 ;
w [ p2 ++ p0 ] = r5.l ;
```

8-Bit Store to Memory (StDregToM08bit)

General Form

Load/Store (LdSt)
b[PREG Register Type++] = DREG Register Type
b[PREG Register Type--] = DREG Register Type
b[PREG Register Type] = DREG Register Type
Long Load/Store with indexed addressing (LdStIdxI)
b[PREG Register Type + imm16reloc Register Type] = DREG Register Type

Load/Store 32-bit Absolute Address (LdStAbs)
b[uimm32 Register Type] = DREG Register Type

Abstract

This instruction stores the least significant 8-bit value from a register to memory.

See Also (32-Bit Store to Memory (StDregToM32bit), 16-Bit Store to Memory (StDregLToM16bit), 16-Bit Store to Memory (StDregHToM16bit))

StDregToM08bit Description

The store byte from data register instruction stores the least significant 8 bits from a 32-bit data register byte to a memory location. The address of the memory location is identified with a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction has no restrictions for memory address alignment. This instruction supports the following options.

- Post-increment the source pointer by 1 byte `[Preg ++]`
- Post-decrement the source pointer by 1 byte `[Preg --]`
- Offset the source pointer with a 16-bit signed constant `[Preg + Offset]`

The instruction opcode size varies with the address type as follows:

- Store byte to memory using a pointer register for the address encodes as a ***16-bit instruction***.
- Store byte to memory using a pointer register with 16-bit offset for the address encodes as a ***32-bit instruction***.
- Store byte to memory using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit store byte instructions may be issued in parallel with certain other instructions. The 32- and 64-bit load byte instructions may not be issued in parallel with other instructions.

This instruction may be used in either *User or Supervisor mode*.

StDregToM08bit Example

```
b [ p0 ] = r3 ;
b [ p1 ++ ] = r7 ;
b [ sp -- ] = r2 ;
b [ p4 + 0x100F ] = r0 ;
b [ p4 - 0x53F ] = r0 ;
```

32-Bit Store to Memory (StDregToM32bit)

General Form

Load/Store postmodify addressing, pregister based (LdStPmod)

[PREG Register Type ++ PREG Register Type] = DREG Register Type
Load/Store (DspLdSt)
[IREG Register Type++] = DREG Register Type
[IREG Register Type--] = DREG Register Type
[IREG Register Type] = DREG Register Type
[IREG Register Type ++ MREG Register Type] = DREG Register Type
Load/Store (LdSt)
[PREG Register Type++] = DREG Register Type
[PREG Register Type--] = DREG Register Type
[PREG Register Type] = DREG Register Type
Load/Store indexed with small immediate offset FP (LdStIIFP)
[fp - imm5nzs4negpos Register Type] = DREG Register Type
[fp - imm5nzs4negpos Register Type] = PREG Register Type
Load/Store indexed with small immediate offset (LdStII)
[PREG Register Type + uimm4s4 Register Type] = DREG Register Type
Long Load/Store with indexed addressing (LdStIdxI)
[PREG Register Type + imm16s4 Register Type] = DREG Register Type
Load/Store 32-bit Absolute Address (LdStAbs)
[uimm32 Register Type] = DREG Register Type
[uimm32 Register Type] = PREG Register Type

Abstract

This instruction stores the 32-bit value from a register to memory.

See Also (8-Bit Store to Memory (StDregToM08bit), 16-Bit Store to Memory (StDregLToM16bit), 16-Bit Store to Memory (StDregHToM16bit))

StDregToM32bit Description

The store 32-bit data from data register instruction stores a 32-bit value from a data register into a memory location. The address of the memory location is identified with an index register, an index register plus an offset, a pointer register, a pointer plus an offset, or a 32-bit absolute address. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source index by 4 bytes `[`*`Ireg`*` ++]`
- Post-decrement the source index by 4 bytes `[`*`Ireg`*` --]`
- Offset the source index with a modifier `[`*`Ireg`*` ++ `*`Mreg`*`]`

- Post-increment the source pointer by 4 bytes `[Preg ++]`
- Post-decrement the source pointer by 4 bytes `[Preg --]`
- Offset the source frame pointer with a 5-bit signed constant `[FP - SmallOffset]`
- Offset the source pointer with a 5-bit signed constant `[Preg + SmallOffset]`
- Offset the source pointer with a 16-bit signed constant `[Preg + LargeOffset]`
- Offset the source pointer with second pointer `[Preg ++ Preg]`

The instruction versions that explicitly modify an index register (*Ireg*) support optional circular buffering. See the description of Automatic Circular Addressing in the Address Arithmetic Unit chapter for more information.

NOTE: Unless circular buffering is desired, disable it prior to issuing this instruction by clearing the length register (*Lreg*) corresponding to the *Ireg* used in this instruction. For example, if you use `I2` to increment your address pointer, first clear `L2` to disable circular buffering. Failure to explicitly clear *Lreg* beforehand can result in unexpected *Ireg* values. The circular address buffer registers (index, length, and base) are not initialized automatically by reset. Typically, user software clears all the circular address buffer registers during boot-up to disable circular buffering, then initializes them later, if needed

The syntax of the form:

```
[ Dest_1 ++ Dest_2 ] = Src ;
```

is indirect, post-increment index addressing. The form is shorthand for the following sequence.

```
[Dest_1] = Src ; /* store to the 32-bit destination, indirect*/
Dest_1 += Dest_2 ; /* post-increment Src_1 by a quantity indexed by Src_2 */
```

where:

- *Src* is the source register. (*Dreg* in the syntax example).
- *Dest_1* is the first destination register on the right-hand side of the equation.
- *Dest_2* is the second destination register.

Indirect and post-increment index addressing supports customized indirect address cadence. The indirect, post-increment index version must have separate pointer registers for the input operands. If a common pointer is used for the inputs, the instruction functions as a simple, non-incrementing load. For example,

```
[p2 ++ p2] = r0 ;
```

functions as:

```
[p2] = r0 ;
```

The instruction opcode size varies with the address type as follows:

- Store 32-bit data to memory using an index register or a pointer register for the address encodes as a ***16-bit instruction***.

- Store 32-bit data to memory using an index register offset by a modifier register or a pointer register offset by a second pointer for the address encodes as a ***16-bit instruction***.
- Store 32-bit data to memory using a pointer or frame pointer register with a small offset for the address encodes as a ***16-bit instruction***.
- Store 32-bit data to memory using a pointer register with a large offset for the address encodes as a ***32-bit instruction***.
- Store 32-bit data to memory using a 32-bit absolute address encodes as a ***64-bit instruction***.

The 16-bit store 32-bit data to register instructions may be issued in parallel with certain other instructions. The 32- and 64-bit store 32-bit data to register instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

StDregToM32bit Example

```
[ p0 ] = r3 ;
[ p1 ++ ] = r7 ;
[ sp -- ] = r2 ;
[ p2 + 12 ] = r6 ;
[ p4 - 0x1004 ] = r0 ;
[ p0 ++ p1 ] = r1 ;
[ fp - 28 ] = r5 ;
[ i2 ] = r2 ;
[ i0 ++ ] = r0 ;
[ i0 -- ] = r0 ;
[ i3 ++ m0 ] = r7 ;
```

Store Pointer (StPregToM32bit)

General Form

Load/Store (LdSt)
[PREG Register Type++] = PREG Register Type
[PREG Register Type--] = PREG Register Type
[PREG Register Type] = PREG Register Type
Load/Store indexed with small immediate offset (LdStII)
[PREG Register Type + uimm4s4 Register Type] = PREG Register Type
Long Load/Store with indexed addressing (LdStIdxI)
[PREG Register Type + imm16s4 Register Type] = PREG Register Type

Abstract

This instruction stores the 32-bit value from a pointer register to memory.

StPregToM32bit Description

The store 32-bit data from pointer register instruction stores a 32-bit value from a pointer register into a memory location. The address of the memory location is identified with a pointer register or a pointer plus an offset. The address used in this instruction is restricted to even memory address alignment (4-byte word address alignment). Failure to maintain proper alignment causes a misaligned memory access exception. This instruction supports the following options.

- Post-increment the source pointer by 4 bytes `[`*`Preg`*` ++]`
- Post-decrement the source pointer by 4 bytes `[`*`Preg`*` --]`
- Offset the source pointer with a 5-bit signed constant `[`*`Preg`*` + `*`SmallOffset`*`]`
- Offset the source pointer with a 16-bit signed constant `[`*`Preg`*` + `*`LargeOffset`*`]`

The instruction opcode size varies with the address type as follows:

- Store 32-bit data to memory using a pointer register for the address encodes as a ***16-bit instruction***.
- Store 32-bit data to memory using a pointer register with a small offset for the address encodes as a ***16-bit instruction***.
- Store 32-bit data to memory using a pointer register with a large offset for the address encodes as a ***32-bit instruction***.

The 16-bit store 32-bit data from pointer instructions may be issued in parallel with certain other instructions. The 32-bit store 32-bit data from pointer instructions may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

StPregToM32bit Example

```
[ p2 ] = p3 ;
[ sp ++ ] = p5 ;
[ p0 -- ] = p2 ;
[ p2 + 8 ] = p3 ;
[ p2 + 0x4444 ] = p0 ;
[ fp -12 ] = p1 ;
```

Memory Store (Exclusive) Operations

The memory store (exclusive) instruction forms the second part of the exclusive instruction sequence, in which the instruction has to establish exclusive access to a memory location. The store exclusive instruction only modifies memory if the current task (for example, core or thread) still has exclusive access to that location. An intervening load exclusive by another task, a write to the memory location, or a `SYNCEXCL` instruction may have caused the exclusive access to have been lost.

Checking for exclusive access is a two stage process. First the `SEQSTAT.XMONITOR` bit is tested. If this is 0 the store exclusive instruction terminates immediately. If `SEQSTAT.XMONITOR` is 1, the store exclusive instruction attempts to write data to the memory location. When the memory location is in *non-shareable* memory, the instruction is considered to have exclusive access to the location simply based on the SEQSTAT.XMONITOR bit, and the store exclusive instruction performs the write in exactly the same manner as a regular memory store to the same memory location. When the memory location is in *shareable* memory, the store exclusive instruction performs the write with an exclusive transaction on the memory bus. This transaction may itself fail to update the location if another core has established exclusive access or written to the location since the current task executed the prior load exclusive instruction. For more information about illegal, non-shareable, and shareable memory regions, see the Exclusive Loads and Stores section of the Memory chapter.

The memory store (exclusive) operations include the following instructions:

- 8-Bit Store to Memory (StDregToX08bit)
- 32-Bit Store to Memory (StDregToX32bit)
- 16-Bit Store to Memory (StDregLToX16bit)
- 16-Bit Store to Memory (StDregHToX16bit)

The store exclusive instruction terminates before the success or failure of the write transaction is known. The state of the write transaction is tracked in the SEQSTAT.XWACTIVE and SEQSTAT.XWAVAIL bits which are updated asynchronously to the core pipeline once the write response has been received from the system. These bits should not be tested directly, instead the SYNCEXCL (Synchronize Exclusive State) instruction should be used to waits for the write to complete and set ASTAT.CC according to whether the store exclusive instruction successfully wrote to the memory location. The ***Exclusive Related Bits in Status Registers (SEQSTAT and ASTAT)*** table shows how exclusive access status changes during an exclusive memory store operation.

Table 8-25: Exclusive Related Bits in Status Registers (SEQSTAT and ASTAT)

Name	Description	Condition
SEQSTAT.XMONITOR	Exclusive monitor (0=open, 1=exclusive)	Not updated
	=0 on start of instruction, CC=0	
	=1 on start of instruction, attempt update (Preg,val), ASTAT.CC=1, SEQSTAT.XWACTIVE=1	
SEQSTAT.XWACTIVE	Exclusive write active (0=no status[*1], 1=active)	Always updated
	=0 on completion of instruction	
	=1 while active	
ASTAT.CC	Condition Code (0=no write attempted, 1=write attempted)	Always updated
SEQSTAT.XWAVAIL	Exclusive write resp. (0=no status, 1=available)	Always updated
	=1 on completion of write transaction	

*1 If XWACTIVE is not 0 when the instruction starts, the instruction throws an exception.

When the memory management unit cannot complete an memory store (exclusive) a number of exceptions and errors may be issued, in addition to those that may be caused by a regular memory store. The ***Exceptions/Errors from Unsuccessful Memory Store (Exclusive) Operations*** table lists these exceptions and errors.

Table 8-26: Exceptions/Errors from Unsuccessful Memory Store (Exclusive) Operations

Condition	Exception or Hardware Error
Access to misaligned address	The data access generated a misaligned address violation exception. The address for the exclusive access must be aligned. This restriction holds even if misaligned accesses are supported generally.
Access to core MMR	The data access attempted an illegal use of a supervisor resource.
Access to I/O device space	The data access generated a CPLB protection violation exception. This exception occurs when the access is to memory marked as I/O device space in the CPLB.
Access during in progress exclusive operation	The data access (exclusive write on memory bus) occurred while the `SEQSTAT.XWACTIVE` bit =1 or the `SEQSTAT.XWAVAIL` bit =1 before the new memory store (exclusive) instruction started.

16-Bit Store to Memory (StDregHToX16bit)

General Form

Long Load/Store with indexed addressing (LdStExcl)
cc = (w[PREG Register Type] = DREG_H Register Type) (excl)

Abstract

This instruction stores the most significant 16-bit value from a register to memory, using an exclusive memory write.

See Also (8-Bit Store to Memory (StDregToX08bit), 32-Bit Store to Memory (StDregToX32bit), 16-Bit Store to Memory (StDregLToX16bit))

StDregHToX16bit Description

The store high half data register to memory (16-bit transfer) checks for exclusive access to a memory location and writes a data value (contents of half register) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0

...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregHToX16bit Example

```
CC = (W[P4] = R3.H)(EXCL);  /* store exclusive from high 16-bits of a D-register */
```

16-Bit Store to Memory (StDregLToX16bit)

General Form

Long Load/Store with indexed addressing (LdStExcl)
cc = (w[PREG Register Type] = DREG Register Type) (excl)

Abstract

This instruction stores the least significant 16-bit value from a register to memory, using an exclusive memory write.

See Also (8-Bit Store to Memory (StDregToX08bit), 32-Bit Store to Memory (StDregToX32bit), 16-Bit Store to Memory (StDregHToX16bit))

StDregLToX16bit Description

The store low half data register to memory (16-bit transfer) checks for exclusive access to a memory location and writes a data value (contents of half register) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregLToX16bit Example

```
CC = (W[P4] = R3)(EXCL);  /* store exclusive from low 16-bits of a D-register */
CC = (W[P4] = R3.L)(EXCL);  /* alternate syntax for same instruction */
```

8-Bit Store to Memory (StDregToX08bit)

General Form

Long Load/Store with indexed addressing (LdStExcl)
cc = (b[PREG Register Type] = DREG Register Type) (excl)

Abstract

This instruction stores the least significant 8-bit value from a register to memory, using an exclusive memory write.

See Also (32-Bit Store to Memory (StDregToX32bit), 16-Bit Store to Memory (StDregLToX16bit), 16-Bit Store to Memory (StDregHToX16bit))

StDregToX08bit Description

The store data register to memory (8-bit transfer) checks for exclusive access to a memory location and writes a data value (least significant byte of data register contents) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregToX08bit Example

```
CC = (B[P4] = R6)(EXCL);  /* store exclusive from low 8-bits of a D-register */
```

32-Bit Store to Memory (StDregToX32bit)

General Form

Long Load/Store with indexed addressing (LdStExcl)
cc = ([PREG Register Type] = DREG Register Type) (excl)

Abstract

This instruction stores the 32-bit value from a register to memory, using an exclusive memory write.

See Also (8-Bit Store to Memory (StDregToX08bit), 16-Bit Store to Memory (StDregLToX16bit), 16-Bit Store to Memory (StDregHToX16bit))

StDregToX32bit Description

The store data register to memory (32-bit transfer) checks for exclusive access to a memory location and writes a data value (contents of register) to the location only if exclusive access is still held.

For more information about memory store (exclusive) operations, see Memory Store (Exclusive) Operations.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

StDregToX32bit Example

```
CC = ([P2] = R0)(EXCL);  /* store exclusive all 32-bits of a D-register */
```

Specialized Compute Instructions

The specialized compute instructions provide operations, which execute on the ***data arithmetic unit*** in the processor core. Users can take advantage of these instructions to detect exponents, add/subtract with pre-scale, divide, execute bitwise XOR, execute vector operations, execute Viterbi operations, and execute video related operations. These instructions are considered specialized because (unlike the arithmetic operation instructions) these instructions tend to provide features that are uniquely required to optimize specialized applications.

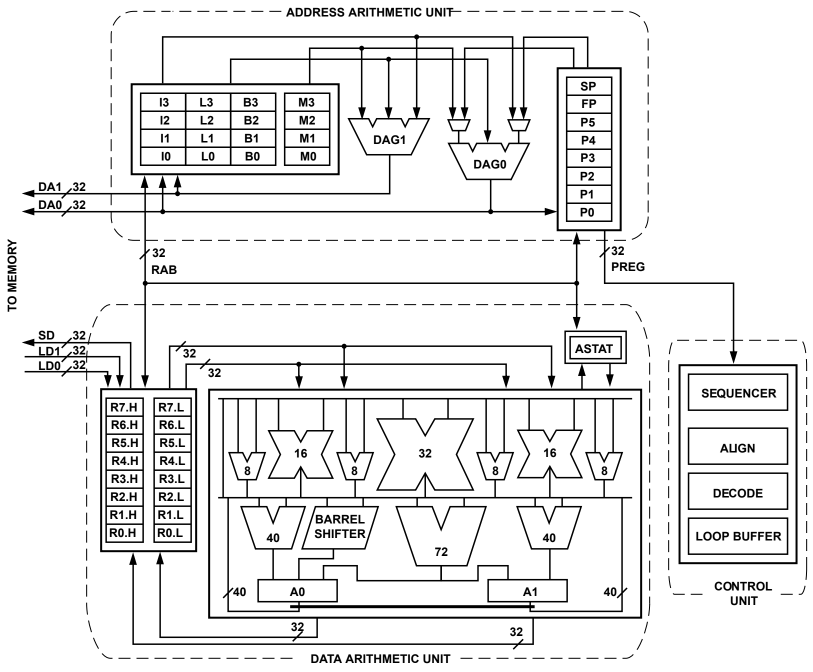


Figure 8-31: Blackfin+ Core Block Diagram

The operation types of specialized compute instructions include:

- Block Floating Point Operations
- DCT Operations
- Divide Operations
- Linear Feedback Shift Register LFSR Operations
- Video Operations
- Viterbi Operations

Block Floating Point Operations

These operations provide exponent adjustment for floating-point operations on register operands:

- Exponent Detection (Shift_ExpAdj32)

Exponent Detection (Shift_ExpAdj32)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = expadj (DREG Register Type, DREG_L Register Type)
DREG_L Register Type = expadj (DREG Register Type, DREG_L Register Type) (v)

DREG_L Register Type = expadj (DREG_L Register Type, DREG_L Register Type)
DREG_L Register Type = expadj (DREG_H Register Type, DREG_L Register Type)

Abstract

This instruction identifies the largest magnitude of a fractional number (YOP) and a reference exponent and returns the smaller of the two exponents. The exponent is the number of sign bits minus one. Exponents are unsigned integers. The input values can be a 32-bit register, a 16-bit half register, or a 16-bit vector.

Shift_ExpAdj32 Description

The exponent detection instruction identifies the largest magnitude of two or three fractional numbers based on their exponents. It compares the magnitude of one or two sample values to a reference exponent and returns the smallest of the exponents.

The exponent is the number of sign bits minus one. In other words, the exponent is the number of redundant sign bits in a signed number. Exponents are unsigned integers. The exponent detection instruction accommodates the two special cases (0 and –1) and always returns the smallest exponent for each case.

The reference exponent and destination exponent are 16-bit half-word unsigned values. The sample number can be either a word or half-word. The exponent detection instruction does not implicitly modify input values. The *dest_reg* and *exponent_register* can be the same data register. Doing this explicitly modifies the *exponent_register*.

The valid range of exponents is 0 through 31, with 31 representing the smallest 32-bit number magnitude and 15 representing the smallest 16-bit number magnitude.

Exponent detection supports three types of samples—one 32-bit sample, one 16-bit sample (either upper-half or lower-half word), and two 16-bit samples that occupy the upper-half and lower-half words of a single 32-bit register.

One special application of EXPADJ is to use this instruction to detect the exponent of the largest magnitude number in an array. The detected value may then be used to normalize the array on a subsequent pass with a shift operation. Typically, use this feature to implement block floating-point capabilities.

This ***16-bit instruction*** may be issued in parallel with certain other 16-bit other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Shift_ExpAdj32 Example

```
r5.l = expadj (r4, r2.l) ;
/* ...  Assume R4 = 0x0000 0052 and R2.L = 12. Then R5.L becomes 12. */
/* ...  Assume R4 = 0xFFFF 0052 and R2.L = 12. Then R5.L becomes 12. */
/* ...  Assume R4 = 0x0000 0052 and R2.L = 27. Then R5.L becomes 24. */
/* ...  Assume R4 = 0xF000 0052 and R2.L = 27. Then R5.L becomes 3. */

r5.l = expadj (r4.l, r2.l) ;
```

```
/* ...  Assume R4.L = 0x0765 and R2.L = 12. Then R5.L becomes 4. */
/* ...  Assume R4.L = 0xC765 and R2.L = 12. Then R5.L becomes 1. */

r5.l = expadj (r4.h, r2.l) ;
/* ...  Assume R4.H = 0x0765 and R2.L = 12. Then R5.L becomes 4. */
/* ...  Assume R4.H = 0xC765 and R2.L = 12. Then R5.L becomes 1. */

r5.l = expadj (r4, r2.l)(v) ;
/* ...  Assume R4.L = 0x0765, R4.H = 0xFF74 and R2.L = 12. Then R5.L becomes 4. */
/* ...  Assume R4.L = 0x0765, R4.H = 0xE722 and R2.L = 12. Then R5.L becomes 2. */
```

DCT Operations

These operations provide addition and/or subtract operations with prescale and rounding on register operands:

- 32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12)
- 32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20)

32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12)

General Form

ALU Operations (Dsp32Alu)
DDST0_HL = DREG Register Type + DREG Register Type (rnd12)
DDST0_HL = DREG Register Type - DREG Register Type (rnd12)

Abstract

This instruction shifts then adds or subtracts two 32-bit numbers, then it extracts sixteen bits of result. The instruciont pre-shifts the operands four bits to the left, then it does the add/sub, round, and extract. The instruction supports only biased rounding, which adds a half LSB (bit 15) before truncating bits 15-0. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Also (32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20))

AddSubRnd12 Description

The add/subtract prescale up instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale up both input operand values by shifting them four places to the left
- Add or subtract the operands, depending on the instruction version used
- Round and saturate the upper 16 bits of the result
- Extract the upper 16 bits to the *dest_reg*

The instruction supports only biased rounding. The `RND_MOD` bit in the `ASTAT` register has no bearing on the rounding behavior of this instruction. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

A special applications of the add/subtract prescale up instruction is to use this instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	*VS*	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubRnd12 Example

```
r1.l = r6+r7(rnd12) ;
r1.l = r6-r7(rnd12) ;
r1.h = r6+r7(rnd12) ;
r1.h = r6-r7(rnd12) ;
```

32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20)

General Form

ALU Operations (Dsp32Alu)
DDST0_HL = DREG Register Type + DREG Register Type (rnd20)
DDST0_HL = DREG Register Type - DREG Register Type (rnd20)

Abstract

This instruction shifts then adds or subtracts two 32-bit numbers, then it extracts sixteen bits of result. The instruction arithmetically pre-shifts the operands four bits to the right. It adds or subtracts them, rounds the upper 16-Bits of the result then extracts the upper 16-Bits to the result. The instruction supports only biased rounding, which adds a half LSB (bit 15) before truncating bits 15-0. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Also (32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12))

AddSubRnd20 Description

The add/subtract prescale down instruction combines two 32-bit values to produce a 16-bit result as follows:

- Prescale down both input operand values by arithmetically shifting them four places to the right
- Add or subtract the operands, depending on the instruction version used
- Round the upper 16 bits of the result
- Extract the upper 16 bits to the `dest_reg`

The instruction supports only biased rounding. The `RND_MOD` bit in the `ASTAT` register has no bearing on the rounding behavior of this instruction. See the *Saturation* topic in the *Introduction* chapter for a description of saturation behavior.

This ***32-bit instruction*** can be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

A special applications of the add/subtract prescale down instruction is to use this instruction to provide an IEEE 1180–compliant 2D 8x8 inverse discrete cosine transform.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	*V*	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_MOD	...	AQ	CC	...	...	...	*AN*	*AZ*
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AddSubRnd20 Example

```
r1.l = r6+r7(rnd20) ;
r1.l = r6-r7(rnd20) ;
r1.h = r6+r7(rnd20) ;
r1.h = r6-r7(rnd20) ;
```

Divide Operations

These operations provide division primitive operations on register operands:

- DIVS and DIVQ Divide Primitives (Divide)

DIVS and DIVQ Divide Primitives (Divide)

General Form

ALU Binary Operations (ALU2op)
divq (DREG Register Type, DREG Register Type)
divs (DREG Register Type, DREG Register Type)

Abstract

The DIVQ instruction is a simple non-restoring divide primitive. It takes two operands, src and dst, where the source operand is the denominator and dst is the numerator. The denominator or divisor is subtracted or added repeatedly from the numerator which becomes the dividend. The algorithm uses a status bit AQ (quotient bit), which determines how the ALU will compute the next bit of the quotient. If the AQ bit is 1 then an add is performed otherwise the dividend is subtracted from the partial remainder. The DIVS instruction is the initializing instruction for DIVQ. It sets the AQ flag based on the signs of the 32-bit dividend and the 16-bit divisor, left shifts the dividend one bit, then copies AQ into the dividend LSB.

Divide Description

The Divide Primitive instruction versions are the foundation elements of a nonrestoring conditional add-subtract division algorithm. See “Example” on page 15-24 for such a routine.

The dividend (numerator) is a 32-bit value. The divisor (denominator) is a 16-bit value in the lower half of divisor_register. The high-order half-word of divisor_register is ignored entirely.

The division can either be signed or unsigned, but the dividend and divisor must both be of the same type. The divisor cannot be negative. A signed division operation, where the dividend may be negative, begins the sequence with the DIVS (“divide-sign”) instruction, followed by repeated execution of the DIVQ (“divide-quotient”) instruction. An unsigned division omits the DIVS instruction. In that case, the user must manually clear the AQ status bit of the ASTAT register before issuing the DIVQ instructions.

Up to 16 bits of signed quotient resolution can be calculated by issuing DIVS once, then repeating the DIVQ instruction 15 times. A 16-bit unsigned quotient is calculated by omitting DIVS, clearing the AQ status bit, then issuing 16 DIVQ instructions.

Less quotient resolution is produced by executing fewer DIVQ iterations.

The result of each successive addition or subtraction appears in dividend_register, aligned and ready for the next addition or subtraction step. The contents of divisor_register are not modified by this instruction.

The final quotient appears in the low-order half-word of dividend_register at the end of the successive add/subtract sequence.

DIVS computes the sign bit of the quotient based on the signs of the dividend and divisor. DIVS initializes the AQ status bit based on that sign, and initializes the dividend for the first addition or subtraction. DIVS performs no addition or subtraction.

DIVQ either adds (dividend + divisor) or subtracts (dividend – divisor) based on the AQ status bit, then reinitializes the AQ status bit and dividend for the next iteration. If AQ is 1, addition is performed; if AQ is 0, subtraction is performed.

See “Status Bits Affected” on page 15-4 for the conditions that set and clear the AQ status bit.

Both instruction versions align the dividend for the next iteration by left shifting the dividend one bit to the left (without carry). This left shift accomplishes the same function as aligning the divisor one bit to the right, such as one would do in manual binary division.

The format of the quotient for any numeric representation can be determined by the format of the dividend and divisor. Let:

- NL represent the number of bits to the left of the binal point of the dividend, and
- NR represent the number of bits to the right of the binal point of the dividend (numerator);
- DL represent the number of bits to the left of the binal point of the divisor, and
- DR represent the number of bits to the right of the binal point of the divisor (denominator).

Then the quotient has NL – DL + 1 bits to the left of the binal point and NR – DR – 1 bits to the right of the binal point. See the following example.

```
Dividend (numerator)      BBBB B .        BBB BBBB BBBB BBBB BBBB BBBB BBBB
                          NL bits         NR bits

Divisor (denominator)     BB .            BB BBBB BBBB BBBB
                          DL bits         DR bits

Quotient                  BBBB .          BBBB BBBB BBBB

                          NL - DL +1      NR - DR - 1
                          (5 - 2 + 1)     (27 - 14 - 1)

                          4.12 format
```

Figure 8-32: 4.12 Format

Some format manipulation may be necessary to guarantee the validity of the quotient. For example, if both operands are signed and fully fractional (dividend in 1.31 format and divisor in 1.15 format), the result is fully fractional (in 1.15 format) and therefore the upper 16 bits of the dividend must have a smaller magnitude than the divisor to avoid a quotient overflow beyond 16 bits. If an overflow occurs, AV0 is set. User software is able to detect the overflow, rescale the operand, and repeat the division.

Dividing two integers (32.0 dividend by a 16.0 divisor) results in an invalid quotient format because the result will not fit in a 16-bit register. To divide two integers (dividend in 32.0 format and divisor in 16.0 format) and produce an integer quotient (in 16.0 format), one must shift the dividend one bit to the left (into 31.1 format) before dividing. This requirement to shift left limits the usable dividend range to 31 bits. Violations of this range produce an invalid result of the division operation.

The algorithm overflows if the result cannot be represented in the format of the quotient as calculated above, or when the divisor is zero or less than the upper 16 bits of the dividend in magnitude (which is tantamount to multiplication).

It is important to understand error conditions related to this instruction. Two special cases can produce invalid or inaccurate results. Software can trap and correct both cases.

No signed division by a negative divisor

The Divide Primitive instructions do not support signed division by a negative divisor. Attempts to divide by a negative divisor result in a quotient that is, in most cases, one LSB less than the correct value. If division by a negative divisor is required, follow the steps below.

1. Before performing the division, save the sign of the divisor in a scratch register.
2. Calculate the absolute value of the divisor and use that value as the divisor operand in the Divide Primitive instructions.
3. After the divide sequence concludes, multiply the resulting quotient by the original divisor sign.
4. The quotient then has the correct magnitude and sign.

No unsigned division by a divisor greater than 0x7FFF

The Divide Primitive instructions do not support unsigned division by a divisor greater than 0x7FFF. If such divisions are necessary, prescale both operands by shifting the dividend and divisor one bit to the right prior to division. The resulting quotient will be correctly aligned. Of course, prescaling the operands decreases their resolution, and may introduce one LSB of error in the quotient. Such error can be detected and corrected by the following steps.

1. Save the original (unscaled) dividend and divisor in scratch registers.
2. Prescale both operands as described and perform the division as usual.
3. Multiply the resulting quotient by the unscaled divisor. Do not corrupt the quotient by the multiplication step.
4. Subtract the product from the unscaled dividend. This step produces an error value.
5. Compare the error value to the unscaled divisor.
6. If error > divisor, add one LSB to the quotient.
7. If error < divisor, subtract one LSB from the quotient.
8. If error = divisor, do nothing.

This ***16-bit instruction*** may not be issued in parallel with other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	*AQ*	CC	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Divide Example

```
/* Evaluate given a signed integer dividend and divisor */
p0 = 15 ; /* Evaluate the quotient to 16 bits. */
r0 = 70 ; /* Dividend, or numerator */
r1 = 5 ; /* Divisor, or denominator */
r0 <<= 1 ; /* Left shift dividend by 1 needed for integer division */
divs (r0, r1) ; /* Evaluate quotient MSB. Initialize AQ status bit and dividend for the
DIVQ loop. */
loop .div_prim lc0=p0 ; /* Evaluate DIVQ p0=15 times. */
loop_begin .div_prim ;
divq (r0, r1) ;
loop_end .div_prim ;
r0 = r0.l (x) ; /* Sign extend the 16-bit quotient to 32bits. */
/* r0 contains the quotient (70/5 = 14). */
```

Linear Feedback Shift Register LFSR Operations

These operations provide LFSR related operations on register operands:

- 40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF)
- 40-Bit BXOR LSFR with Feedback to a Register (BXOR)
- 32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF)

40-Bit BXOR LSFR with Feedback to a Register (BXOR)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = cc = bxor (a0, a1, cc)

Abstract

This instruction (linear feedback shift register, LFSR) provides a bit-wise XOR reduction of A0 logically AND'ed with A1 and the feedback bit (CC). The result is placed into both the CC flag and the least significant bit of the destination register. The Accumulator is not modified by this operation.

See Also (40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF), 32-Bit BXOR or BXOR-Shift LSFR without Feedback (BXOR_NF))

BXOR Description

Four Bit-Wise Exclusive-OR (BXOR) instructions support two different types of linear feedback shift register (LFSR) implementations. The Type I LFSRs (no feedback) applies a 32-bit registered mask to a 40-bit state residing in Accumulator A0, followed by a bit-wise XOR reduction operation. The result is placed in CC and a destination register half. The Type I LFSRs (with feedback) applies a 40-bit mask in Accumulator A1 to a 40-bit state residing in A0. The result is shifted into A0. In the following circuits describing the BXOR instruction group, a bit-wise XOR reduction is defined as:

$$\mathbf{Out} = (((((B_0 \oplus B_1) \oplus B_2) \oplus B_3) \oplus \ldots) \oplus B_{n-1})$$

Figure 8-33: BXOR Instruction Group

where B0 through BN–1 represent the N bits that result from masking the contents of Accumulator A0 with the polynomial stored in either A1 or a 32-bit register. The instruction descriptions are shown in Figure 12-1.

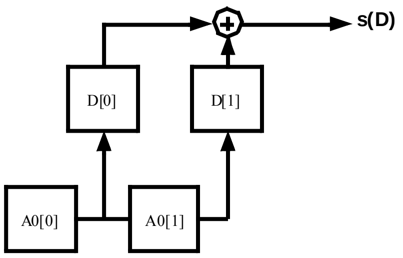


Figure 8-34: Bit-Wise Exclusive-OR Reduction

In the figure above, the bits A0 bit 0 and A0 bit 1 are logically AND’ed with bits D[0] and D[1]. The result from this operation is XOR reduced according to the following formula.

$$s(D) = (A0[0] \& D[0]) \oplus (A0[1] \& D[1])$$

Figure 8-35: Bit-Wise Exclusive-OR Reduction Result

Modified Type I LFSR (without feedback) Two instructions support the LSFR with no feedback. Dreg_lo = CC = BXORSHIFT(A0, dreg) Dreg_lo = CC = BXOR(A0, dreg) In the first instruction the Accumulator A0 is left-shifted by 1 prior to the XOR reduction. This instruction provides a bit-wise XOR of A0 logically AND’ed with a dreg. The result of the operation is placed into both the CC status bit and the least significant bit of the destination register. The operation is shown in Figure 12-2. The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN after the operation.

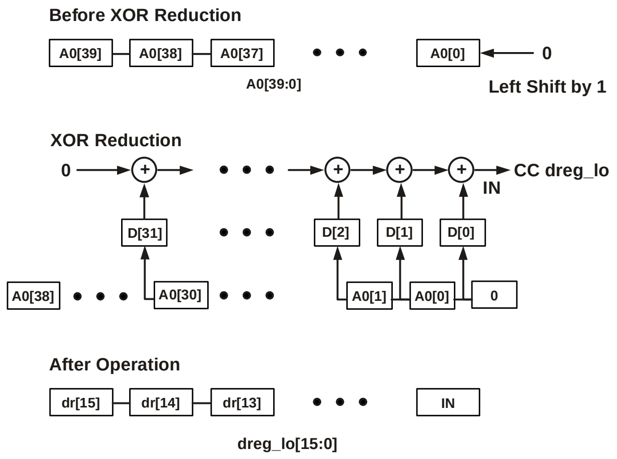


Figure 8-36: A0 Left-Shifted by 1 Followed by XOR Reduction

The second instruction in this class performs a bit-wise XOR of A0 logically AND'ed with the dreg. The output is placed into the least significant bit of the destination register and into the CC bit. The Accumulator A0 is not modified by this operation. This operation is illustrated in Figure 12-3. The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN after the operation.

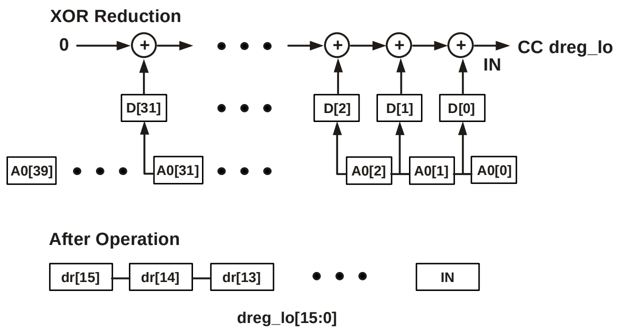


Figure 8-37: XOR of A0, Logical AND with the D-Register

Modified Type I LFSR (with feedback) Two instructions support the LFSR with feedback. A0 = BXORSHIFT(A0, A1, CC) Dreg_lo = CC = BXOR(A0, A1, CC)

The first instruction provides a bit-wise XOR of A0 logically AND'ed with A1. The resulting intermediate bit is XOR'ed with the CC status bit. The result of the operation is left-shifted into the least significant bit of A0 following the operation. This operation is illustrated in Figure 12-4. The CC bit is not modified by this operation.

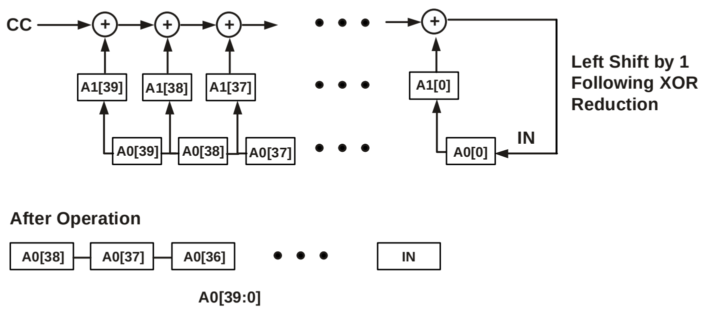


Figure 8-38: XOR of A0 AND A1, Left-Shifted into LSB of A0

The second instruction in this class performs a bit-wise XOR of A0 logically AND'ed with A1. The resulting intermediate bit is XOR'ed with the CC status bit. The result of the operation is placed into both the CC status bit and the least significant bit of the destination register.

This operation is illustrated in Figure 12-5.

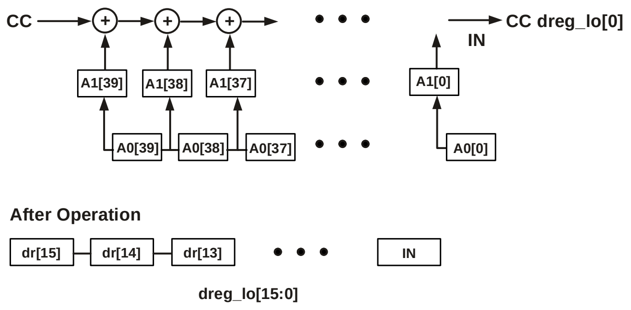


Figure 8-39: XOR of A0 AND A1, to CC Bit and LSB of Dest Register

The Accumulator A0 is not modified by this operation. The upper 15 bits of dreg_lo are overwritten with zero, and dr[0] = IN.

Special Applications Linear feedback shift registers (LFSRs) can multiply and divide polynomials and are often used to implement cyclical encoders and decoders.

LFSRs use the set of Bit-Wise XOR instructions to compute bit XOR reduction from a state masked by a polynomial.

When implementing a CRC algorithm, it is known that there is an equivalence between polynomial division and LFSR circuits. For example, CRC is defined as the remainder of the division of a message polynomial appended with n zeros by the code generator polynomial:

$C_n(x)=\{M_k(x)x^n\}\mathrm{mod}G_n(x)$

Where:

- $M_{k-1}(x)$ is the message polynomial of length k:

  $M_{k-1}(x)=m_{k-1}x^{k-1}+m_{k-2}x^{k-2}+\ldots+m_0x^0$

- $G_n(x)$ is the CRC generating polynomial of degree n, and n is also the CRC field length in bits:

  $G_n(x)=x^n+g_{n-1}x^{n-1}+\ldots+g_0x^0$

- $C_n(x)$ is the calculated CRC polynomial of degree n:

  $C_n(x)=x^n+c_{n-1}x^{n-1}+\ldots+c_0x^0$

The division is performed modulo-2 over Galois field GF2. In the above equation, the message stream Mk is post-fixed by n zeros before the actual division. This equation can be implemented by one of two types of n taps LFSR's. The more familiar type of LFSR is called Type II (or internal) LFSR of the form:

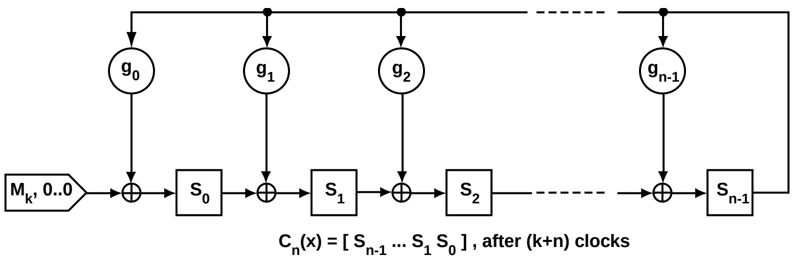

g_0
g_1
g_2
g_{n-1}
M_k, 0..0
S_0
S_1
S_2
S_{n-1}
$C_n(x)$ = [S_{n-1} ... S_1 S_0] , after (k+n) clocks

Figure 8-40: Internal LFSR (Type II)

The other type of LFSR, Type I (or external LFSR) has the form:

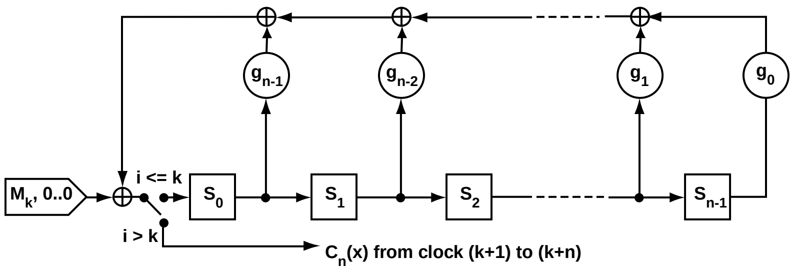


Figure 8-41: External LFSR (Type I)

The two are equivalent, and the simple rule for conversion from Type II to Type I is:

1. While keeping the LFSR flow direction, flip the order of the feedback taps.

2. After the first k clocks, feed the first tap (S0) with n zeros and read the n output bits (which are the required CRC) as the sum of the feedback and the input.

For example, consider the following equivalent implementations of the polynomial G5(x) = x5+x4+x2+1:

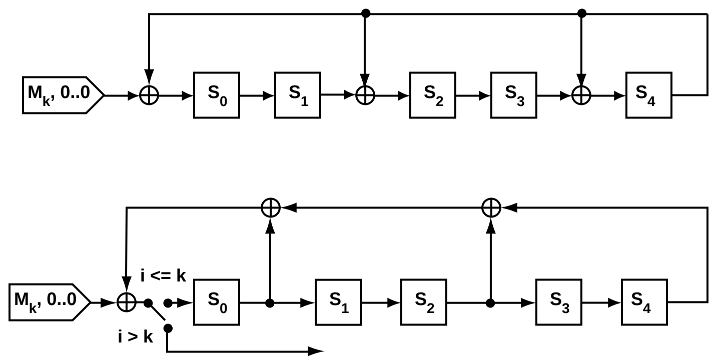


Figure 8-42: Internal (Type II) Versus External (Type I) LFSR

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BXOR Example

The BXOR and BXORSHIFT instructions let you calculate a Type I CRC at a rate of two cycles per input bit, as in the following example program.

```
// _CRC_BXOR - calculate CRC value of a message polynomial
// for a given generator polynomial.
#define MSG_LEN 32 // bits
#define CRC_LEN 16 // bits
_CRC_BXOR:
   a1 = a0 = 0;
   r1 = 0x8408 (z); // LFSR polynomial, reversed:
   // x^16 + x^12 + x^5 + 1
   a1.w = r1; // initialize LFSR mask
   r2.h = 0xd065; // r2 = message
   r2.l = 0x86c9;
   p1 = MSG_LEN (z);
   loop _MSG_loop lc0 = p1;
   loop_begin _MSG_loop;
      r2 = rot r2 by 1;
      a0 = bxorshift(a0, a1, cc);
   loop_end _MSG_loop;
   r0 = 0; // initialize CRC
   r2.l = cc = bxor(a0, r1);
   r0 = rot r0 by 1;
   p1 = CRC_LEN-1 (z);
   loop _CRC_loop lc0 = p1;
   loop_begin _CRC_loop;
      r2.l = cc = bxorshift(a0, r1);
      r0 = rot r0 by 1;
   loop_end _CRC_loop;
   // r0.l now contains the CRC
_CRC_BXOR.end:
```

40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF)

General Form

Shift (Dsp32Shf)
a0 = bxorshift (a0, a1, cc)

Abstract

This instruction (linear feedback shift register, LFSR) provides a bit-wise XOR reduction of A0 logically AND'ed with A1 and the feedback bit (CC). The result is left-shifted into the least significant bit of A0. The CC bit is not modified.

See Also (40-Bit BXOR LSFR with Feedback to a Register (BXOR), 32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF))

BXORShift_NF Description

Linear feedback shift register (LFSR) instruction. Provides a bit-wise XOR reduction of A0 logically AND'ed with A1 and the feedback bit (CC). The result is left-shifted into the least significant bit of A0. The CC bit is not modified.

The bit-wise XOR reduction is defined as:

```
   out = ((((CC &oplus B0) &oplus B1) &oplus B2) &oplus ...)  &oplus Bn-1)
```

For more information about this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

BXORShift_NF Example

For examples using this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = cc = bxorshift (a0, DREG Register Type)
DREG_L Register Type = cc = bxor (a0, DREG Register Type)

Abstract

This instruction (linear feedback shift register, LFSR) provides a bit-wise XOR reduction of A0 or A0 shifted left one, logically AND'ed with a 32-bit data register. The result is placed into both the CC flag and the least significant bit of the destination register.

See Also (40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF), 40-Bit BXOR LSFR with Feedback to a Register (BXOR))

BXOR_NF Description

Linear feedback shift register (LFSR) instruction. Provides a bit-wise XOR reduction of A0 or A0 shifted left one, logically AND'ed with a 32-bit data register. The result is placed into both the CC flag and the least significant bit of the destination register.

A bit-wise XOR reduction is defined as:

out = %%(((((%%B_0 &oplus B_1) &oplus B_2) &oplus B_3) &oplus ...) &oplus B_{n-1})

For more information about this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

ASTAT Flags

The table shows the affected ASTAT flags. For more information, see Arithmetic Status Register .

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
.	...	...	...	...	...	VS	V	...	...	...	...	AV1S	AV1	AV0S	AV0
...	...	AC1	AC0	...	...	...	RND_ MOD	...	AQ	*CC*	...	...	...	AN	AZ
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

BXOR_NF Example

For examples using this instruction, see 40-Bit BXOR LSFR with Feedback to a Register (BXOR).

Video Operations

These operations provide video application specific operations on register operands:

- Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8)
- Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip)
- Disable Alignment Exception (DisAlignExcept)
- Quad Byte Average (Byteop2P) (Avg4x8Vec)
- Vector Byte Average (Byteop1P) (Avg8Vec)
- Dual Accumulator Extraction with Addition (AddAccHalf)
- Vectored 8-Bit Sum of Absolute Differences (SAD8Vec)

Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = byteop3p (PAIR0, PAIR1) (lo RSC)
DREG Register Type = byteop3p (PAIR0, PAIR1) (hi RSC)

Abstract

This instruction adds two 8-bit unsigned values to two 16-bit signed values, then it clips the result back to the 8-bit unsigned range. The instruction either adds Y[3] & Y[1] or Y[2] & Y[0] to the two 16-bt X values. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. The results are written back to either the lower or higher bytes of each 16-bit result half. The unused bytes are filled with zeros.

See Also (Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8))

AddClip Description

The dual 16-bit add/clip instruction adds two 8-bit unsigned values to two 16-bit signed values, then limits (or “clips”) the result to the 8-bit unsigned range 0 through 255, inclusive. The instruction loads the results as bytes on half-word boundaries in one 32-bit destination register. Some syntax options load the upper byte in the half-word and others load the lower byte, as shown in the next few figures.

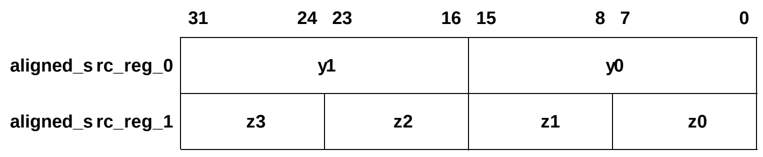


Figure 8-43: The source registers contain:

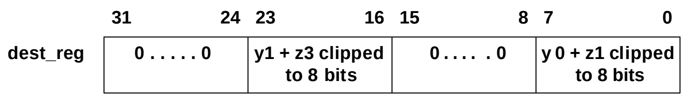
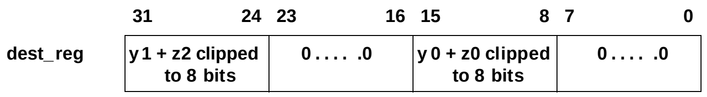


Figure 8-44: The versions that load the result into the lower byte (LO) produce:



	31 24	23 16	15 8	7 0
dest_reg	y1 + z2 clipped to 8 bits	0 0	y0 + z0 clipped to 8 bits	0 0

Figure 8-45: The versions that load the result into the higher byte (HI) produce:

In either case, the unused bytes in the destination register are filled with 0x00. The 8-bit and 16-bit addition is performed as a signed operation. The 16-bit operand is sign-extended to 32 bits before adding.

The only valid input source register pairs are R1:0 and R3:2.

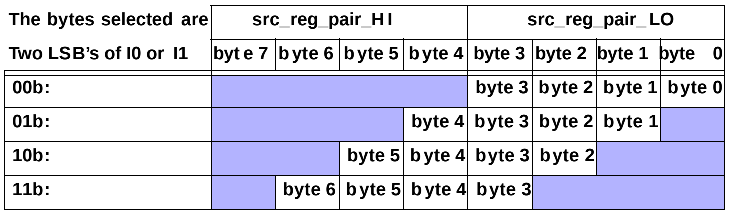
The dual 16-bit add/clip instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated in the ***I-register Bits and the Byte Alignment (no reverse)*** figure.

In the default source order case (for example, not the (– , R) syntax), assuming a source register pair contains the following.

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.



The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Figure 8-46: I-register Bits and the Byte Alignment (no reverse)

Options The (– , R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (– , R) option causes the low order bytes to come from the high register.

In the optional reverse source order case (for example, using the (– , R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in the ***I-register Bits and the Byte Alignment (with reverse)*** figure.

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Figure 8-47: I-register Bits and the Byte Alignment (with reverse)

A special application of this instruction is support for video motion compensation algorithms. The instruction supports the addition of the residual to a video pixel value, followed by unsigned byte saturation.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

AddClip Example

```
r3 = byteop3p (r1:0, r3:2) (lo) ;
r3 = byteop3p (r1:0, r3:2) (hi) ;
r3 = byteop3p (r1:0, r3:2) (lo, r) ;
r3 = byteop3p (r1:0, r3:2) (hi, r) ;
```

Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8)

General Form

ALU Operations (Dsp32Alu)
(DREG Register Type, DREG Register Type) = byteop16p (PAIR0, PAIR1) RS
(DREG Register Type, DREG Register Type) = byteop16m (PAIR0, PAIR1) RS

Abstract

This instruction (Byteop16M and ByteOp16P) adds or subtracts two unsigned quad byte vectors, adjusting for byte alignment. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair

See Also (Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip))

AddSub4x8 Description

The quad 8-bit add instruction adds two unsigned quad byte number sets byte-wise, adjusting for byte alignment. It then loads the byte-wise results as 16-bit, zero-extended, half-words in two destination registers, as shown in the next figures.

	31 24	23 16	15 8	7 0
aligned_s rc_reg_0	y3	y2	y1	y0
aligned_s rc_reg_1	z3	z2	z1	z0

Figure 8-48: Source Registers Contain

	31 24 23 16	15 8 7 0
dest_reg_0:	y1 + z1	y0 + z0
dest_reg_1:	y3 + z3	y2 + z2

Figure 8-49: Destination Registers Receive

The only valid input source register pairs are R1:0 and R3:2, and the two destination registers must be unique.

The Quad 8-Bit Add instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated in Table 18-1.

In the default source order case (for example, not the (R) syntax), assume that a source register pair contains the data shown in the ***I-register Bits and the Byte Alignment (No Reverse)*** figure.

The bytes selected are	src_reg_pair_H I				src_reg_pair_ LO			
Two LSB's of I0 or I1	b yte 7	b yte 6	b yte 5	b yte 4	byte 3	byte 2	byte 1	byte 0
00b:	▨	▨	▨	▨	byte 3	byte 2	byte 1	byte 0
01b:	▨	▨	▨	byte 4	byte 3	byte 2	byte 1	▨
10b:	▨	▨	byte 5	b yte 4	byte 3	byte 2	▨	▨
11b:	▨	byte 6	b yte 5	b yte 4	byte 3	▨	▨	▨

Figure 8-50: I-register Bits and the Byte Alignment (No Reverse)

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Options

The (R) syntax reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in the ***I-register Bits and the Byte Alignment (With Reverse)*** figure.

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:	▨	▨	▨	▨	byte 3	byte 2	byte 1	byte 0
01b:	▨	▨	▨	byte 4	byte 3	byte 2	byte 1	▨
10b:	▨	▨	byte 5	byte 4	byte 3	byte 2	▨	▨
11b:	▨	byte 6	byte 5	byte 4	byte 3	▨	▨	▨

Figure 8-51: I-register Bits and the Byte Alignment (With Reverse)

The mnemonic derives its name from the fact that the operands are bytes, the result is 16 bits, and the arithmetic operation is "plus" for addition.

A special application of this instruction provides packed data arithmetic typical of video and image processing applications.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

AddSub4x8 Example

```
(r1,r2)= byteop16p (r3:2,r1:0) ;
(r1,r2)= byteop16p (r3:2,r1:0) (r) ;
```

Disable Alignment Exception (DisAlignExcept)

General Form

ALU Operations (Dsp32Alu)
disalgnexcpt

Abstract

This instruction disables alignment excptions. This instruction only affects misaligned loads that use I registers. The address is forced to be 32-bit aligned.

See Also (Byte Align (Shift_Align))

DisAlignExcept Description

The disable alignment exception for load (DISALGNEXCPT) instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel. This instruction only affects misaligned 32-bit load instructions that use I-register indirect addressing.

In order to force address alignment to a 32-bit boundary, the two LSBs of the address are cleared before being sent to the memory system. The I-register is not modified by the DISALIGNEXCPT instruction. Also, any modifications performed to the I-register by a parallel instruction are not affected by the DISALIGNEXCPT instruction.

A special applications of this instruction is to use the DISALGNEXCPT instruction when priming data registers for Quad 8-Bit single-instruction, multiple-data (SIMD) instructions.

Quad 8-Bit SIMD instructions require as many as sixteen 8-bit operands, four D-registers worth, to be preloaded with operand data. The operand data is 8 bits and not necessarily word aligned in memory. Thus, use DISALGNEXCPT to prevent spurious exceptions for these potentially misaligned accesses.

During execution, when Quad 8-Bit SIMD instructions perform 8-bit boundary accesses, they automatically prevent exceptions for misaligned accesses. No user intervention is required.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

DisAlignExcept Example

```
disalgnexcpt || r1 = [i0++] || r3 = [i1++] ;
/* three instructions in parallel */
disalgnexcpt || [p0 ++ p1] = r5 || r3 = [i1++] ;
/* alignment exception is prevented only for the load */
disalgnexcpt || r0 = [p2++] || r3 = [i1++] ;
/* alignment exception is prevented only for the I-reg load */
```

Byte Align (Shift_Align)

General Form

Shift (Dsp32Shf)
DREG Register Type = align8 (DREG Register Type, DREG Register Type)
DREG Register Type = align16 (DREG Register Type, DREG Register Type)
DREG Register Type = align24 (DREG Register Type, DREG Register Type)

Abstract

This instruction copies four contiguous bytes from a register pair. The bytes are offset by 8, 16, or 24 bits in the register pair.

See Also (Disable Alignment Exception (DisAlignExcept))

Shift_Align Description

The Byte Align instruction copies a contiguous four-byte unaligned word from a combination of two data registers. The instruction version determines the bytes that are copied; in other words, the byte alignment of the copied word. Alignment options are shown in Table 18-1.

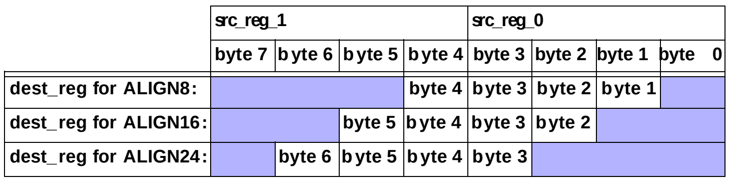


Figure 8-52: Byte Alignment Options

The ALIGN16 version performs the same operation as the Vector Pack instruction using the syntax:

```
dest_reg = PACK ( Dreg_lo, Dreg_hi )
```

Use the Byte Align instruction to align data bytes for subsequent single- instruction, multiple-data (SIMD) instructions.

The input values are not implicitly modified by this instruction. The destination register can be the same D-register as one of the source registers. Doing this explicitly modifies that source register.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Shift_Align Example

```
// If r3 = 0x0011 2233 and r4 = 0x4455 6677, then . . .
r0 = align8 (r3, r4) ; /* produces r0 = 0x3344 5566, */
r0 = align16 (r3, r4) ; /* produces r0 = 0x2233 4455, and */
r0 = align24 (r3, r4) ; /* produces r0 = 0x1122 3344, */
```

Quad Byte Average (Byteop2P) (Avg4x8Vec)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = byteop2p (PAIR0, PAIR1) (rndl RSC)
DREG Register Type = byteop2p (PAIR0, PAIR1) (rndh RSC)
DREG Register Type = byteop2p (PAIR0, PAIR1) (tl RSC)
DREG Register Type = byteop2p (PAIR0, PAIR1) (th RSC)

Abstract

This instruction averages the upper two bytes and the lower two bytes of two two unsigned quad byte vectors adjusting for byte alignment. The lower two bits of I0 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. Note that this operation only uses I0, which is different than all the other byteop instructions. If you specify round (RND), a round bit is added prior to the shift. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction. It returns the two averages in either Dest[3] & Dest[1] or Dest[2] & Dest[0].

See Also (Vector Byte Average (Byteop1P) (Avg8Vec))

Avg4x8Vec Description

The quad 8-bit average half-word instruction finds the arithmetic average of two unsigned quad byte number sets byte wise, adjusting for byte alignment. This instruction averages four bytes together. The instruction loads the results as bytes on half-word boundaries in one 32-bit destination register. Some syntax options load the upper byte in the half-word and others load the lower byte, as shown in the next few figures.

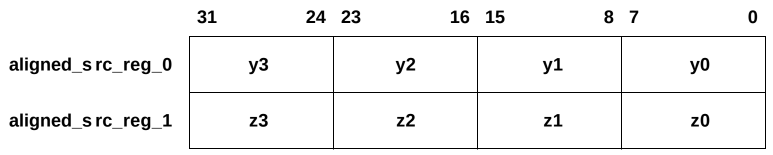


Figure 8-53: Source Registers Contain

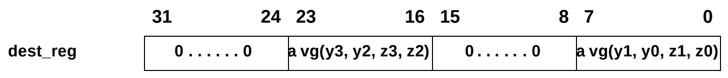


Figure 8-54: The versions that load the result into the lower byte – RNDL and TL – produce:

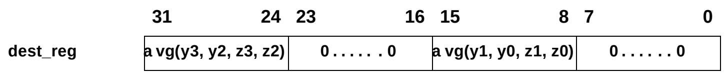


Figure 8-55: The versions that load the result into the higher byte – RNDH and TH – produce:

In either case, the unused bytes in the destination register are filled with 0x00.

Arithmetic average (or mean) is calculated by summing the four byte operands, then shifting right two places to divide by four.

When the intermediate sum is not evenly divisible by 4, precision may be lost.

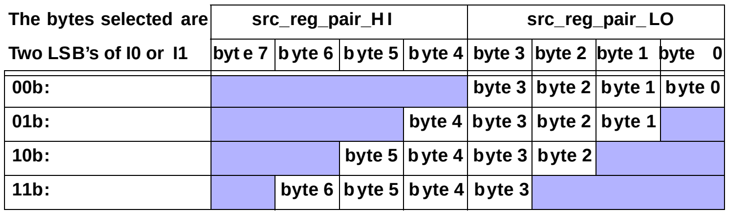
The user has two options to bias the result–truncation or biased rounding.

The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

The only valid input source register pairs are R1:0 and R3:2.

The quad 8-bit average half-word instruction provides byte alignment directly in the source register pairs src_reg_0 (typically R1:0) and src_reg_1 (typically R3:2) based only on the I0 register. The byte alignment in both source registers must be identical since only one register specifies the byte alignment for them both.

The relationship between the I-register bits and the byte alignment is shown in the ***I-register Bits and the Byte Alignment (no reverse)*** figure. In the default source order case (for example, not the (R) syntax), assume a source register pair contains the data shown in the figure.



The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:	▨	▨	▨	▨	byte 3	byte 2	byte 1	byte 0
01b:	▨	▨	▨	byte 4	byte 3	byte 2	byte 1	▨
10b:	▨	▨	byte 5	byte 4	byte 3	byte 2	▨	▨
11b:	▨	byte 6	byte 5	byte 4	byte 3	▨	▨	▨

Figure 8-56: I-register Bits and the Byte Alignment (no reverse)

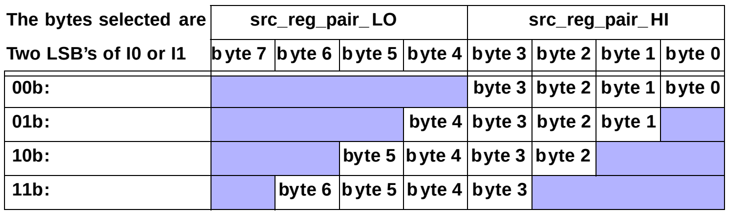
This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

The quad 8-bit average half-word instruction supports the options shown in the ***Options for Quad 8-Bit Average -- Half-Word*** table.

Table 8-27: Options for Quad 8-Bit Average -- Half-Word

Option	Description
(RND—)	Rounds up the arithmetic mean.
(T—)	Truncates the arithmetic mean.
(—L)	Loads the results into the lower byte of each destination half-word.
(—H)	Loads the results into the higher byte of each destination half-word.
(—,R)	Reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

When used together, the order of the options in the syntax makes no difference. In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in the ***I-register Bits and the Byte Alignment (with reverse)*** figure.



The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Figure 8-57: I-register Bits and the Byte Alignment (with reverse)

The mnemonic derives its name from the fact that the operands are bytes, the result is two half-words, and the basic arithmetic operation is “plus” for addition. The single destination register indicates that averaging is performed.

A special applications of this instruction is support for binary interpolation used in fractional motion search and motion compensation algorithms.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Avg4x8Vec Example

```
r3 = byteop2p (r1:0, r3:2) (rndl) ;
r3 = byteop2p (r1:0, r3:2) (rndh) ;
r3 = byteop2p (r1:0, r3:2) (tl) ;
r3 = byteop2p (r1:0, r3:2) (th) ;
r3 = byteop2p (r1:0, r3:2) (rndl, r) ;
r3 = byteop2p (r1:0, r3:2) (rndh, r) ;
r3 = byteop2p (r1:0, r3:2) (tl, r) ;
r3 = byteop2p (r1:0, r3:2) (th, r) ;
```

Vector Byte Average (Byteop1P) (Avg8Vec)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = byteop1p (PAIR0, PAIR1) RS
DREG Register Type = byteop1p (PAIR0, PAIR1) (t RSC)

Abstract

This instruction computes the vector average of two unsigned quad byte vectors adjusting for byte alignment. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. By default, this instruction rounds by adding a one prior to shifting. If you specify truncate, the round bit is not added. The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction.

See Also (Quad Byte Average (Byteop2P) (Avg4x8Vec))

Avg8Vec Description

The quad 8-bit average byte instruction computes the arithmetic average of two unsigned quad byte number sets byte wise, adjusting for byte alignment. This instruction loads the byte-wise results as concatenated bytes in one 32-bit destination register, as shown in the next figures.

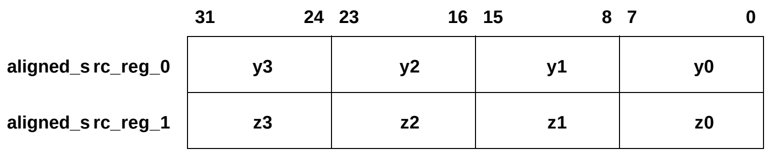


Figure 8-58: Source Registers Contain

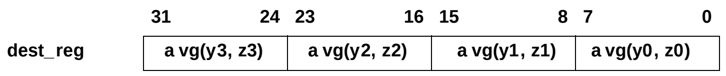


Figure 8-59: Destination Registers Receive

Arithmetic average (or mean) is calculated by summing the two operands, then shifting right one place to divide by two.

The user has two options to bias the result–truncation or rounding up. By default, the architecture rounds up the mean when the sum is odd. However, the syntax supports optional truncation.

See “Rounding and Truncating” on page 1-19 for a description of biased rounding and truncating behavior.

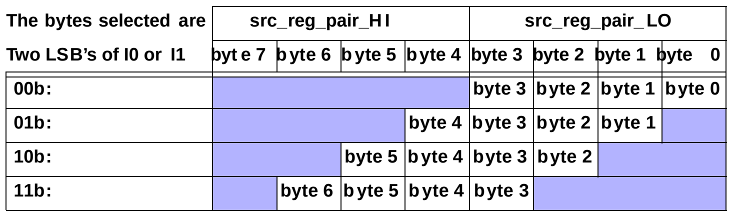
The RND_MOD bit in the ASTAT register has no bearing on the rounding behavior of this instruction. The only valid input source register pairs are R1:0 and R3:2.

The Quad 8-Bit Average – Byte instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is illustrated in the ***I-register Bits and the Byte Alignment (no reverse)*** figure.

In the default source order case (for example, not the (R) syntax), assume a source register pair contains the data shown in the figure.



The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Figure 8-60: I-register Bits and the Byte Alignment (no reverse)

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

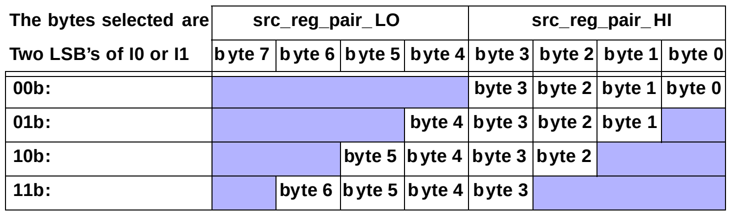
Options

The quad 8-bit average byte instruction supports the options shown in the ***Options for Quad 8-Bit Average – Byte*** table.

Table 8-28: Options for Quad 8-Bit Average – Byte

Option	Description
Default	Rounds up the arithmetic mean.
(T)	Truncates the arithmetic mean.
(R)	Reverses the order of the source registers within each register pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.
(T, R)	Combines both of the above options.

In the optional reverse source order case (for example, using the (R) syntax), the only difference is the source registers swap places within the register pair in their byte ordering. Assume a source register pair contains the data shown in the ***I-register Bits and the Byte Alignment (with reverse)*** figure.



The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:					byte 3	byte 2	byte 1	byte 0
01b:				byte 4	byte 3	byte 2	byte 1	
10b:			byte 5	byte 4	byte 3	byte 2		
11b:		byte 6	byte 5	byte 4	byte 3			

Figure 8-61: I-register Bits and the Byte Alignment (with reverse)

The mnemonic derives its name from the fact that the operands are bytes, the result is one word, and the basic arithmetic operation is "plus" for addition. The single destination register indicates that averaging is performed.

A special application of this instruction is support for binary interpolation used in fractional motion search and motion compensation algorithms.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Avg8Vec Example

```
r3 = byteop1p (r1:0, r3:2) ;
r3 = byteop1p (r1:0, r3:2) (r) ;
r3 = byteop1p (r1:0, r3:2) (t) ;
r3 = byteop1p (r1:0, r3:2) (t,r) ;
```

Dual Accumulator Extraction with Addition (AddAccHalf)

General Form

ALU Operations (Dsp32Alu)
DREG Register Type = a1.l + a1.h, DREG Register Type = a0.l + a0.h

Abstract

This instruction adds the accumulator half words together, then it extracts the result to a register. Each half word is sign extended to 32-bits before being added. This operation is used to sum the results of the video [[.:SAD8Vec|Sum of Absolute Differences]] instruction.

See Also (Vectored 8-Bit Sum of Absolute Differences (SAD8Vec))

AddAccHalf Description

The dual 16-bit accumulator eExtraction with addition instruction adds together the upper half-words (bits 31through 16) and lower half-words (bits 15 through 0) of each Accumulator and loads each result into a 32-bit destination register.

Each 16-bit half-word in each Accumulator is sign extended before being added together.

A special application of this instruction is to use the dual 16-bit accumulator extraction with addition instruction for motion estimation algorithms in conjunction with the quad 8-bit subtract-absolute-accumulate instruction.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

AddAccHalf Example

```
r4=a1.l+a1.h, r7=a0.l+a0.h ;
```

Vectored 8-Bit Sum of Absolute Differences (SAD8Vec)

General Form

ALU Operations (Dsp32Alu)
saa (PAIR0, PAIR1) RS

Abstract

This instruction does a vector 8-bit subtract, takes the absolute value of the differences and accumulates them adjusting for byte alignment. The lower two bits of I0 and I1 are used to [[extractBytes | extract four contiguous bytes]] from the input register pair. The four 16-bit results are stored in A1.h, A1.l, A0.h and A0.l These will saturate if the unsigned 16-Bit sections of the accumulator overflow.

See Also (Dual Accumulator Extraction with Addition (AddAccHalf))

SAD8Vec Description

The quad 8-bit subtract-absolute-accumulate instruction subtracts four pairs of values, takes the absolute value of each difference, and accumulates each result into a 16-bit Accumulator half. The results are placed in the upper- and lower-half Accumulators A0.H, A0.L, A1.H, and A1.L.

Saturation is performed if an operation overflows a 16-bit Accumulator half.

This instruction supports the following byte-wise Sum of Absolute Difference (SAD) calculations.

$$SAD = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a(i,j) - b(i,j)$$

Figure 8-62: Absolute Difference (SAD) Calculations

Typical values for N are 8 and 16, corresponding to the video block size of 8x8 and 16x16 pixels, respectively. The 16-bit Accumulator registers limit the pixel region or block size to 32x32 pixels.

The SAA instruction behavior is shown in the ***SAA Instruction Behavior*** figure.

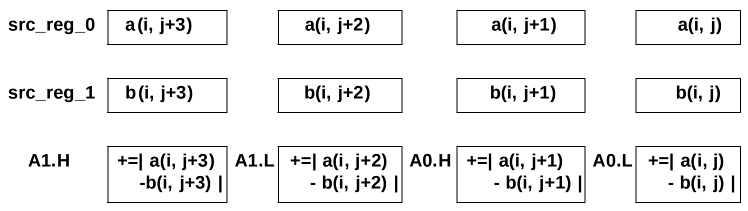


Figure 8-63: SAA Instruction Behavior

The Quad 8-Bit Subtract-Absolute-Accumulate instruction provides byte alignment directly in the source register pairs src_reg_0 and src_reg_1 based on index registers I0 and I1.

- The two LSBs of the I0 register determine the byte alignment for source register pair src_reg_0 (typically R1:0).
- The two LSBs of the I1 register determine the byte alignment for source register pair src_reg_1 (typically R3:2).

The relationship between the I-register bits and the byte alignment is shown in the ***I-register Bits and the Byte Alignment (no reverse)*** figure.

In the default source order case (for example, not the (R) syntax), assume a source register pair contain the data shown in the figure.

The bytes selected are	src_reg_pair_HI				src_reg_pair_LO			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:	▨	▨	▨	▨	byte 3	byte 2	byte 1	byte 0
01b:	▨	▨	▨	byte 4	byte 3	byte 2	byte 1	▨
10b:	▨	▨	byte 5	byte 4	byte 3	byte 2	▨	▨
11b:	▨	byte 6	byte 5	byte 4	byte 3	▨	▨	▨

Figure 8-64: I-register Bits and the Byte Alignment (no reverse)

This instruction prevents exceptions that would otherwise be caused by misaligned 32-bit memory loads issued in parallel.

Options

The (R) syntax reverses the order of the source registers within each pair. Typical high performance applications cannot afford the overhead of reloading both register pair operands to maintain byte order for every calculation. Instead, they alternate and load only one register pair operand each time and alternate between the forward and reverse byte order versions of this instruction. By default, the low order bytes come from the low register in the register pair. The (R) option causes the low order bytes to come from the high register.

When reversing source order by using the (R) syntax, the source registers swap places within the register pair in their byte ordering. If a source register pair contains the data shown in the ***I-register Bits and the Byte Alignment (with reverse)*** figure, then the SAA instruction computes 12 pixel operations simultaneously–the three-operation subtract-absolute-accumulate on four pairs of operand bytes in parallel.

The bytes selected are	src_reg_pair_LO				src_reg_pair_HI			
Two LSB's of I0 or I1	byte 7	byte 6	byte 5	byte 4	byte 3	byte 2	byte 1	byte 0
00b:	▨	▨	▨	▨	byte 3	byte 2	byte 1	byte 0
01b:	▨	▨	▨	byte 4	byte 3	byte 2	byte 1	▨
10b:	▨	▨	byte 5	byte 4	byte 3	byte 2	▨	▨
11b:	▨	byte 6	byte 5	byte 4	byte 3	▨	▨	▨

Figure 8-65: I-register Bits and the Byte Alignment (with reverse)

A special application of this instruction is to use the quad 8-bit subtract-absolute-accumulate instruction for block-based video motion estimation algorithms using block sum of absolute difference (SAD) calculations to measure distortion.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

SAD8Vec Example

```
saa (r1:0, r3:2) || r0 = [i0++] || r2 = [i1++] ; /* parallel fill instructions */
saa (r1:0, r3:2) (R) || r1 = [i0++] || r3 = [i1++] ; /* reverse, parallel fill instructions
*/
saa (r1:0, r3:2) ; /* last SAA in a loop, no more fill required */
```

Viterbi Operations

These operations provide Viterbi application specific operations on register operands:

- 16-Bit Add on Sign (AddOnSign)
- 16-Bit Modulo Maximum with History (Shift_VitMax)
- Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax)

16-Bit Add on Sign (AddOnSign)

General Form

ALU Operations (Dsp32Alu)
DREG_H Register Type = DREG_L Register Type = sign(DREG_H Register Type) * DREG_H Register Type + sign(DREG_L Register Type) * DREG_L Register Type

Abstract

This instruction does a vector multiply of the signs SRC0.h and SRC0.l by the values of SRC1.h and SRC1.l Then, it adds the two results and stores it in both of the destination half registers. This instruction does not saturate if the result is greater than 16-Bits.

See Also (16-Bit Modulo Maximum with History (Shift_VitMax), Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax))

AddOnSign Description

The Add on Sign instruction performs a two step function, as follows.

Step 1

Multiply the arithmetic sign of a 16-bit half-word number in src0 by the corresponding half-word number in src1. The arithmetic sign of src0 is either (+1) or (–1), depending on the sign bit of src0. The instruction performs this operation on the upper and lower half-words of the same data registers.

The results of this step obey the signed multiplication rules summarized in the ***Signed Multiplication Rules*** table. Y is the number in src0, and Z is the number in src1. The numbers in src0 and src1 may be positive or negative. Note the result always bears the magnitude of Z with only the sign affected.

Table 8-29: Signed Multiplication Rules

SRC0	SRC1	Adjusted
+Y	+Z	+Z
+Y	–Z	–Z
–Y	+Z	–Z
–Y	–Z	+Z

Step 2

Add the sign-adjusted src1 upper and lower half-word results together and store the same 16-bit sum in the upper and lower halves of the destination register, as shown in the next figures.

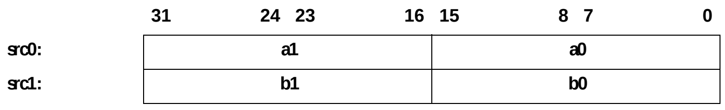


Figure 8-66: Source Registers Contain

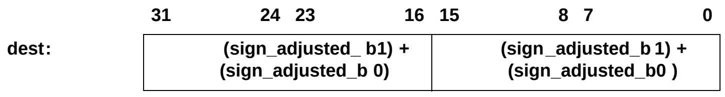

31
24
23
16
15
8
7
0
dest:
(sign_adjusted_ b1) + (sign_adjusted_b 0)
(sign_adjusted_b 1) + (sign_adjusted_b0)

Figure 8-67: Destination Register Receives

The sum is not saturated if the addition exceeds 16 bits.

A special application of this instruction is to use the Sum on Sign instruction to compute the branch metric used by each Viterbi Butterfly.

This ***16-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

AddOnSign Example

```
r7.h = r7.l = sign(r2.h) * r3.h + sign(r2.l) * r3.l ;
```

- If

```
R2.H = 2
R3.H = 23
R2.L = 2001
R3.L = 1234
```

  then

```
R7.H = 1257 (or 1234 + 23)
R7.L = 1257
```

- If

```
R2.H = -2
R3.H = 23
R2.L = 2001
R3.L = 1234
```

then

```
R7.H = 1211 (or 1234 - 23)
R7.L = 1211
```

- If

```
R2.H = 2
R3.H = 23
R2.L = -2001
R3.L = 1234
```

then

```
R7.H = -1211 (or (-1234) + 23)
R7.L = -1211
```

- If

```
R2.H = -2
R3.H = 23
R2.L = -2001
R3.L = 1234
```

then

```
R7.H = -1257 (or (-1234) - 23)
R7.L = -1257
```

Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax)

General Form

Shift (Dsp32Shf)
DREG Register Type = vit_max (DREG Register Type, DREG Register Type) (asl)
DREG Register Type = vit_max (DREG Register Type, DREG Register Type) (asr)

Abstract

This instruction performs maximum value selection and history update. It is used to implement the selection function of a Viterbi decoder. It performs a dual maximum value selection storing the two results in one destination register.

See Also (16-Bit Add on Sign (AddOnSign), 16-Bit Modulo Maximum with History (Shift_VitMax))

Shift_DualVitMax Description

Maximum value selection and history update. This instruction is used to implement the selection function of a Viterbi decoder. It performs a dual maximum value selection storing the two results in one destination register. In addition shifts left A0 by two bit positions, and stores two bits in A0 representing the result of the two maximum value selections in bit1 and bit0 of A0. No attempt to correct the selection on overflow should be made. This ensures that overflowed path metrics compare correctly, as long as they are close to each other in magnitude.

```
Reg1:  PM3 PM2      Reg0: PM1 PM0
        |   |              |   |
       MAX----+  +--------MAX
              |  |
              v  v
RegOut:       H  L
```

If the user specifies ASR or ASL this will shift in two bits into the accumulator specifying which 16-bit half register was the max. For ASR it will shift the history bits right, for ASL it will shift them left. To compute the maximum this instruction uses a form of modulo arithmetic where 0x8000 > 0x7fff > 0 > 0xffff > 0x8000.

For more information, see the 16-Bit Modulo Maximum with History (Shift_VitMax) instruction.

Shift_DualVitMax Example

For examples using this instruction, see the 16-Bit Modulo Maximum with History (Shift_VitMax) instruction.

16-Bit Modulo Maximum with History (Shift_VitMax)

General Form

Shift (Dsp32Shf)
DREG_L Register Type = vit_max (DREG Register Type) (asl)
DREG_L Register Type = vit_max (DREG Register Type) (asr)

Abstract

If the user specifies ASR or ASL, this instruction shifts in a bit into the accumulator specifying which 16-bit half register was the max. For ASR, it will shift the history bits right. For ASL, it will shift them left. To compute the maximum, this instruction uses a form of [[.:modulo_comparisons|modulo arithmetic]] where 0x8000 > 0x7fff > 0 > 0xffff > 0x8000 .

See Also (16-Bit Add on Sign (AddOnSign), Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax))

Shift_VitMax Description

The Compare-Select (VIT_MAX) instruction selects the maximum values of pairs of 16-bit operands, returns the largest values to the destination register, and serially records in A0.W the source of the maximum.This operation performs signed operations. The operands are compared as two's-complements.

The Accumulator extension bits (bits 39–32) must be cleared before executing this instruction.

Dual 16-Bit Operand Behavior

Versions are available for dual and single 16-bit operations. Whereas the dual versions compare four operands to return two maxima, the single versions compare only two operands to return one maximum.

This operation is illustrated in Table 19-4 and Table 19-5.

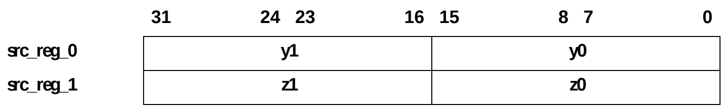


Figure 8-68: Source Registers Contain (Dual)

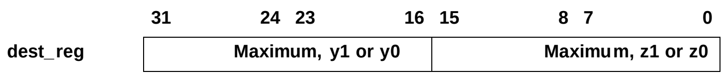


Figure 8-69: Destination Register Contains (Dual)

The ASL version shifts A0 left two bit positions and appends two LSBs to indicate the source of each maximum as shown in Table 19-6 and Table 19-7.

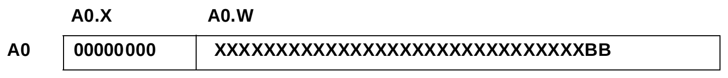


Figure 8-70: ASL Version Shifts (Dual)

Table 8-30: Where ...

BB	Indicates
00	z0 and y0 are maxima
01	z0 and y1 are maxima
10	z1 and y0 are maxima
11	z1 and y1 are maxima

Conversely, the ASR version shifts A0 right two bit positions and appends two MSBs to indicate the source of each maximum as shown in Table 19-8 and Table 19-9.

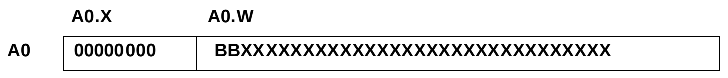


Figure 8-71: ASR Version Shifts (Dual)

Table 8-31: Where ...

BB	Indicates
00	y0 and z0 are maxima
01	y0 and z1 are maxima
10	y1 and z0 are maxima
11	y1 and z1 are maxima

Notice that the history bit code depends on the A0 shift direction. The bit for src_reg_1 is always shifted onto A0 first, followed by the bit for src_reg_0. The single operand versions behave similarly.

Single 16-Bit Operand Behavior

If the dual source register contains the data shown in Table 19-10 the destination register receives the data shown in Table 19-11.

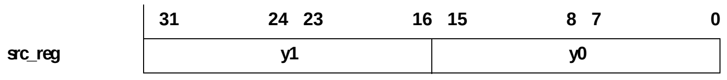


Figure 8-72: Source Registers Contain (Single)

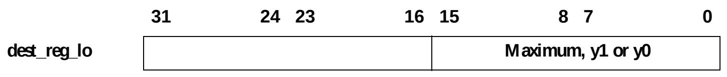


Figure 8-73: Destination Register Contains (Single)

The ASL version shifts A0 left one bit position and appends an LSB to indicate the source of the maximum.

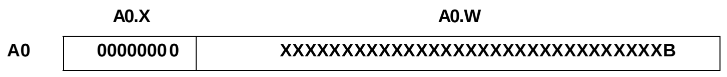


Figure 8-74: ASL Version Shifts (Single)

Conversely, the ASR version shifts A0 right one bit position and appends an MSB to indicate the source of the maximum.

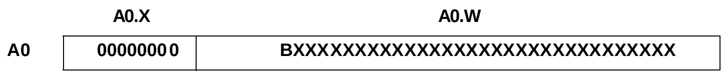


Figure 8-75: ASR Version Shifts (Single)

Table 8-32: Where ...

B	Indicates
0	y0 is the maximum
1	y1 is the maximum

The path metrics are allowed to overflow, and maximum comparison is done on the two's-complement circle. Such comparison gives a better indication of the relative magnitude of two large numbers when a small number is added/ subtracted to both.

A special application of this instruction is to use the Compare-Select (VIT_MAX) instruction as a key element of the Add-Compare-Select (ACS) function for Viterbi decoders. Combine it with a Vector Add instruction to calculate a trellis butterfly used in ACS functions.

This ***32-bit instruction*** may be issued in parallel with certain other instructions.

This instruction may be used in either ***User or Supervisor mode***.

Shift_VitMax Example

- For:

  ```
  r5 = vit_max(r3, r2)(asl) ; /* shift left, dual operation */
  ```

  Assume:

  ```
  R3 = 0xFFFF 0000
  R2 = 0x0000 FFFF
  A0 = 0x00 0000 0000
  ```

  This example produces:

  ```
  R5 = 0x0000 0000
  A0 = 0x00 0000 0002
  ```

- For:

  ```
  r7 = vit_max (r1, r0) (asr) ; /* shift right, dual operation */
  ```

  Assume:

  ```
  R1 = 0xFEED BEEF
  R0 = 0xDEAF 0000
  A0 = 0x00 0000 0000
  ```

  This example produces:

  ```
  R7 = 0xFEED 0000
  A0 = 0x00 8000 0000
  ```

- For:

  ```
  r3.l = vit_max (r1)(asl) ; /* shift left, single operation */
  ```

  Assume:

  ```
  R1 = 0xFFFF 0000
  A0 = 0x00 0000 0000
  ```

  This example produces:

  ```
  R3.L = 0x0000
  ```

```
A0 = 0x00 0000 0000
```

- For:

```
r3.l = vit_max (r1)(asr) ; /* shift right, single operation */
```

Assume:

```
R1 = 0x1234 FADE
A0 = 0x00 FFFF FFFF
```

This example produces:

```
R3.L = 0x1234
A0 = 0x00 FFFF FFFF
```

Instruction Page Tables

The instruction page tables provide definitions of:

- Instruction types (including opcodes for each instruction)
- Constant types (including immediate value types used in all instructions)
- Register types

These pages are organized alphabetically, with instruction types first, followed by constant types, then register types.

ALU Binary Operations (ALU2op)

ALU2op Instruction Syntax

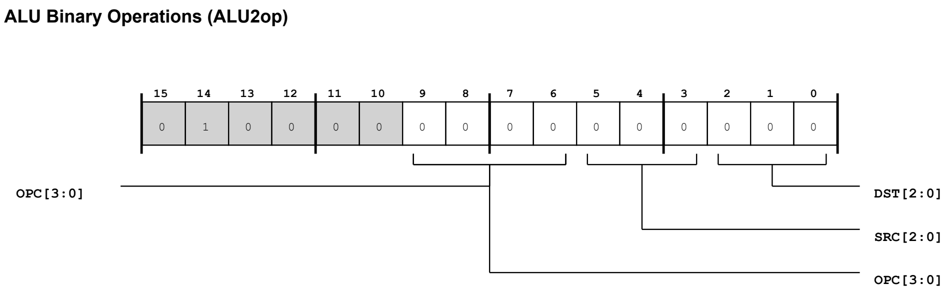


Figure 8-76: ALU2op Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
0000	DREG Register Type >>>= DREG Register Type	32-Bit Arithmetic Shift (AShift32)

OPC	Syntax	Instruction
0001	DREG Register Type >>= DREG Register Type	32-Bit Logical Shift (LShift)
0010	DREG Register Type <<= DREG Register Type	32-Bit Logical Shift (LShift)
0011	DREG Register Type *= DREG Register Type	32 x 32-Bit Multiply, Integer (MultInt)
0100	DREG Register Type = (DREG Register Type + DREG Register Type) << 1	32-bit Add then Shift (AddSubShift)
0101	DREG Register Type = (DREG Register Type + DREG Register Type) << 2	32-bit Add then Shift (AddSubShift)
1000	divq (DREG Register Type, DREG Register Type)	DIVS and DIVQ Divide Primitives (Divide)
1001	divs (DREG Register Type, DREG Register Type)	DIVS and DIVQ Divide Primitives (Divide)
1010	DREG Register Type = DREG_L Register Type (x)	Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg)
1011	DREG Register Type = DREG_L Register Type (z)	Pass 16-Bit to 32-Bit Register Expansion (MvDregLToDreg)
1100	DREG Register Type = DREG_B Register Type (x)	Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg)
1101	DREG Register Type = DREG_B Register Type (z)	Pass 8-Bit to 32-Bit Register Expansion (MvDregBToDreg)
1110	DREG Register Type = -DREG Register Type	32-Bit Negate (Neg32)
1111	DREG Register Type = ~DREG Register Type	32-Bit One's Complement (Not32)

Conditional Branch PC relative on CC (BrCC)

BrCC Instruction Syntax

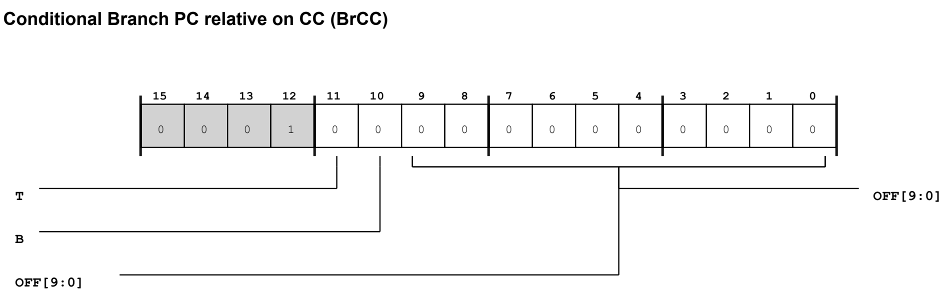


Figure 8-77: BrCC Instruction

The following table provides the opcode field values (T, B), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

T	B	Syntax	Instruction
0	0	if !cc jump imm10s2 Register Type	Conditional Jump Immediate (BrCC)
0	1	if !cc jump imm10s2 Register Type (bp)	Conditional Jump Immediate (BrCC)
1	0	if cc jump imm10s2 Register Type	Conditional Jump Immediate (BrCC)
1	1	if cc jump imm10s2 Register Type (bp)	Conditional Jump Immediate (BrCC)

Move CC conditional bit, to and from dreg (CC2Dreg)

CC2Dreg Instruction Syntax

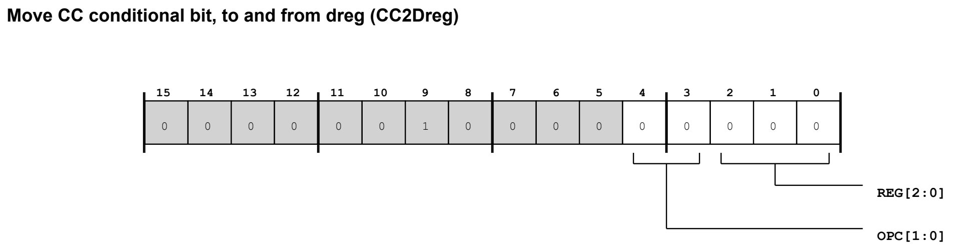


Figure 8-78: CC2Dreg Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction	Rev
00	DREG Register Type = cc	Compute Move CC to a D Register (CCToDreg)	
01	cc = DREG Register Type	Move Status to CC (MvToCC)	
10	DREG Register Type = !cc	Compute Move CC to a D Register (CCToDreg)	2.0
11	cc = !cc	Move Status to CC (MvToCC)	

Copy CC conditional bit, from status (CC2Stat)

CC2Stat Instruction Syntax

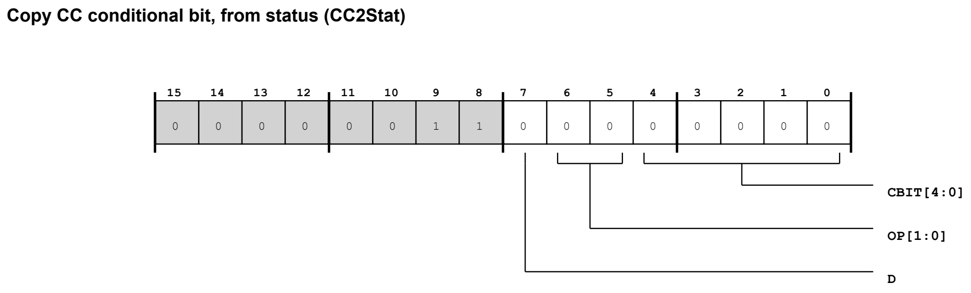


Figure 8-79: CC2Stat Instruction

The following table provides the opcode field values (D, OP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

D	OP	Syntax	Instruction
0	00	cc = CBIT	Move Status to CC (MvToCC_STAT)
0	01	cc |= CBIT	Move Status to CC (MvToCC_STAT)
0	10	cc &= CBIT	Move Status to CC (MvToCC_STAT)
0	11	cc ^= CBIT	Move Status to CC (MvToCC_STAT)
1	00	CBIT = cc	Move CC To/From ASTAT (CCToStat16)
1	01	CBIT |= cc	Move CC To/From ASTAT (CCToStat16)
1	10	CBIT &= cc	Move CC To/From ASTAT (CCToStat16)
1	11	CBIT ^= cc	Move CC To/From ASTAT (CCToStat16)

CBIT

CBIT Encode Table

CBIT	Syntax
00000	az
00001	an

CBIT	Syntax
00110	aq
01000	rnd_mod
01100	ac0
01101	ac1
10000	av0
10001	av0s
10010	av1
10011	av1s
11000	v
11001	vs

Set CC conditional bit (CCFlag)

CCFlag Instruction Syntax

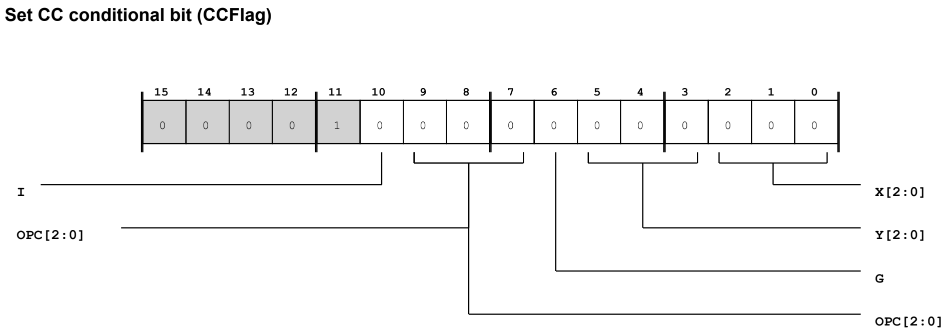


Figure 8-80: CCFlag Instruction

The following table provides the opcode field values (OPC, G, I), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	G	I	Syntax	Instruction
000	0	0	cc = DREG Register Type == DREG Register Type	32-Bit Register Compare and Set CC (CompRegisters)
000	0	1	cc = DREG Register Type == imm3 Register Type	32-Bit Register Compare and Set CC (CompRegisters)
000	1	0	cc = PREG Register Type == PREG Register Type	32-Bit Pointer Register Compare and Set CC (CCFlagP)

OPC	G	I	Syntax	Instruction
000	1	1	cc = PREG Register Type == imm3 Register Type	32-Bit Pointer Register Compare and Set CC (CCFlagP)
001	0	0	cc = DREG Register Type < DREG Register Type	32-Bit Register Compare and Set CC (CompRegisters)
001	0	1	cc = DREG Register Type < imm3 Register Type	32-Bit Register Compare and Set CC (CompRegisters)
001	1	0	cc = PREG Register Type < PREG Register Type	32-Bit Pointer Register Compare and Set CC (CCFlagP)
001	1	1	cc = PREG Register Type < imm3 Register Type	32-Bit Pointer Register Compare and Set CC (CCFlagP)
010	0	0	cc = DREG Register Type <= DREG Register Type	32-Bit Register Compare and Set CC (CompRegisters)
010	0	1	cc = DREG Register Type <= imm3 Register Type	32-Bit Register Compare and Set CC (CompRegisters)
010	1	0	cc = PREG Register Type <= PREG Register Type	32-Bit Pointer Register Compare and Set CC (CCFlagP)
010	1	1	cc = PREG Register Type <= imm3 Register Type	32-Bit Pointer Register Compare and Set CC (CCFlagP)
011	0	0	cc = DREG Register Type < DREG Register Type (iu)	32-Bit Register Compare and Set CC (CompRegisters)
011	0	1	cc = DREG Register Type < uimm3 Register Type (iu)	32-Bit Register Compare and Set CC (CompRegisters)
011	1	0	cc = PREG Register Type < PREG Register Type (iu)	32-Bit Pointer Register Compare and Set CC (CCFlagP)
011	1	1	cc = PREG Register Type < uimm3 Register Type (iu)	32-Bit Pointer Register Compare and Set CC (CCFlagP)
100	0	0	cc = DREG Register Type <= DREG Register Type (iu)	32-Bit Register Compare and Set CC (CompRegisters)
100	0	1	cc = DREG Register Type <= uimm3 Register Type (iu)	32-Bit Register Compare and Set CC (CompRegisters)
100	1	0	cc = PREG Register Type <= PREG Register Type (iu)	32-Bit Pointer Register Compare and Set CC (CCFlagP)
100	1	1	cc = PREG Register Type <= uimm3 Register Type (iu)	32-Bit Pointer Register Compare and Set CC (CCFlagP)
101	0	0	cc = a0 == a1	Accumulator Compare and Set CC (CompAccumulators)
110	0	0	cc = a0 < a1	Accumulator Compare and Set CC (CompAccumulators)

OPC	G	I	Syntax	Instruction
111	0	0	cc = a0 <= a1	Accumulator Compare and Set CC (CompAccumulators)

Conditional Move (CCMV)

CCMV Instruction Syntax

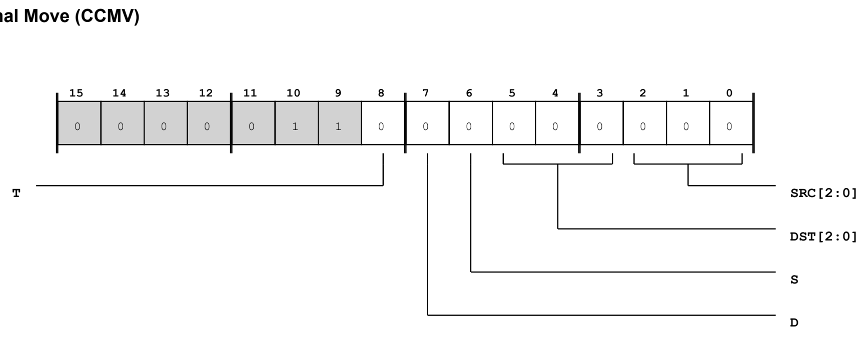


Figure 8-81: CCMV Instruction

The following table provides the opcode field values (T), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

T	Syntax	Instruction
1	if cc GDST = GSRC	Conditional Move Register to Register (MvRegToRegCond)
0	if !cc GDST = GSRC	Conditional Move Register to Register (MvRegToRegCond)

GDST

GDST Encode Table

D	DST	Syntax
0	---	DREG Register Type
1	---	PREG Register Type

GSRC

GSRC Encode Table

S	SRC	Syntax
0	---	DREG Register Type
1	---	PREG Register Type

Cache Control (CacheCtrl)

CacheCtrl Instruction Syntax

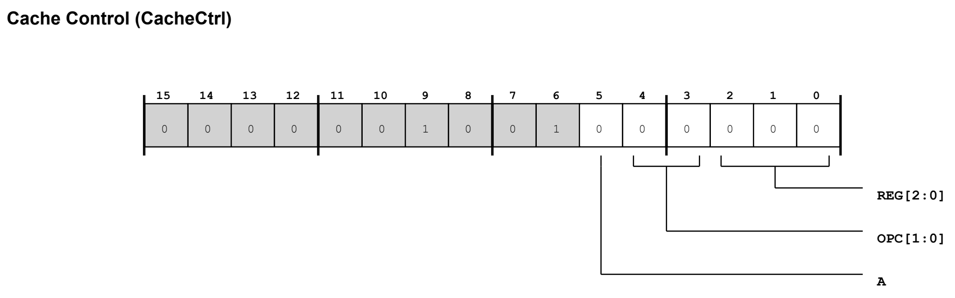


Figure 8-82: CacheCtrl Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
00	prefetch [PREGA]	Cache Control (CacheCtrl)
01	flushinv [PREGA]	Cache Control (CacheCtrl)
10	flush [PREGA]	Cache Control (CacheCtrl)
11	iflush [PREGA]	Cache Control (CacheCtrl)

PREGA

PREGA Encode Table

A	Syntax
0	PREG Register Type
1	PREG Register Type++

Call function with pcrel address (CallA)

CallA Instruction Syntax

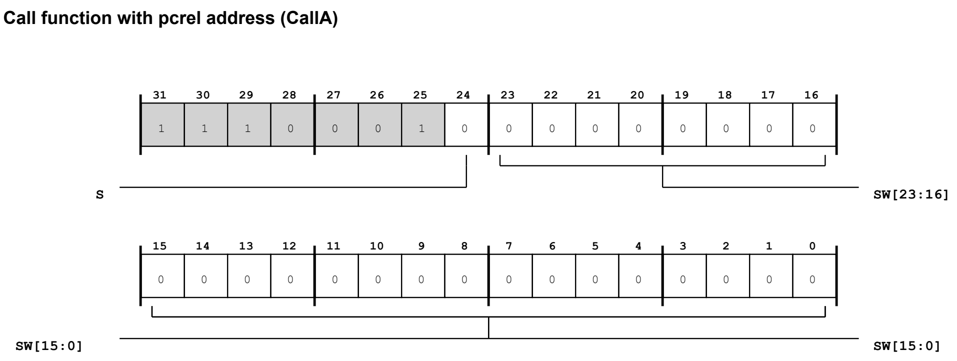


Figure 8-83: CallA Instruction

The following table provides the opcode field values (S), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

S	Syntax	Instruction
0	jump.l imm24s2 Register Type	Jump Immediate (JumpAbs)
1	call imm24nxs2 Register Type	Call (Call)

Compute with 3 operands (Comp3op)

Comp3op Instruction Syntax

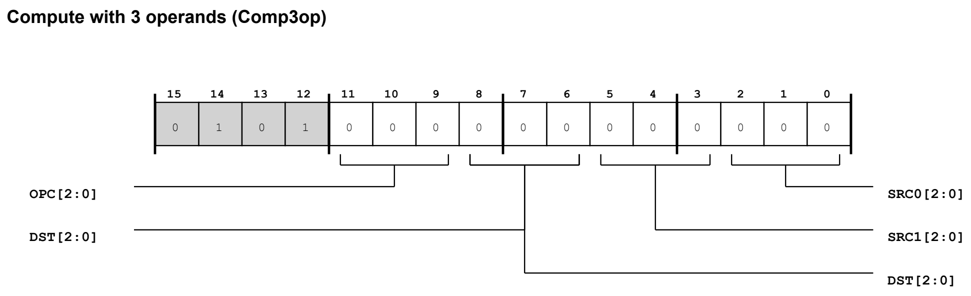


Figure 8-84: Comp3op Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
000	DREG Register Type = DREG Register Type + DREG Register Type	32-bit Add or Subtract (AddSub32)

OPC	Syntax	Instruction
001	DREG Register Type = DREG Register Type - DREG Register Type	32-bit Add or Subtract (AddSub32)
010	DREG Register Type = DREG Register Type & DREG Register Type	32-Bit Logic Operations (Logic32)
011	DREG Register Type = DREG Register Type | DREG Register Type	32-Bit Logic Operations (Logic32)
100	DREG Register Type = DREG Register Type ^ DREG Register Type	32-Bit Logic Operations (Logic32)
101	PREG Register Type = PREG Register Type + PREG Register Type	32-bit Add or Subtract (DagAdd32)
110	PREG Register Type = PREG Register Type + (PREG Register Type << 1)	32-bit Add Shifted Pointer (PtrOp)
111	PREG Register Type = PREG Register Type + (PREG Register Type << 2)	32-bit Add Shifted Pointer (PtrOp)

Destructive Binary Operations, dreg with 7bit immediate (CompI2opD)

CompI2opD Instruction Syntax

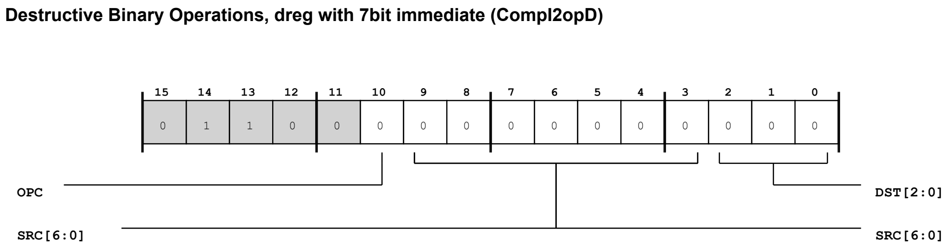


Figure 8-85: CompI2opD Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
0	DREG Register Type = imm7 Register Type (x)	32-Bit Register Initialization (LdImmToReg)
1	DREG Register Type += imm7 Register Type	32-bit Add Constant (AddImm)

Destructive Binary Operations, preg with 7bit immediate (CompI2opP)

CompI2opP Instruction Syntax

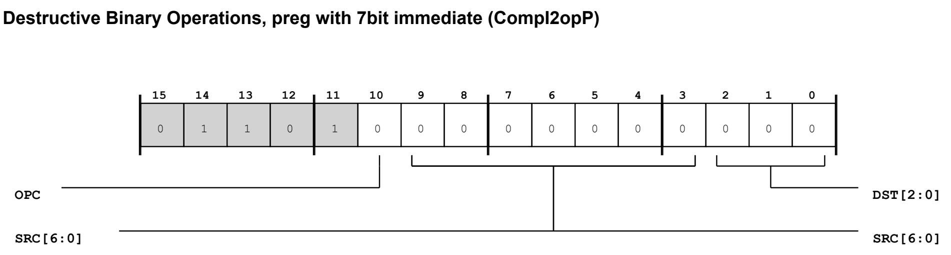


Figure 8-86: CompI2opP Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
0	PREG Register Type = imm7 Register Type (x)	32-Bit Register Initialization (LdImmToReg)
1	PREG Register Type += imm7 Register Type	32-bit Add or Subtract Constant (DagAddImm)

DAG Arithmetic (DAGModIk)

DAGModIk Instruction Syntax

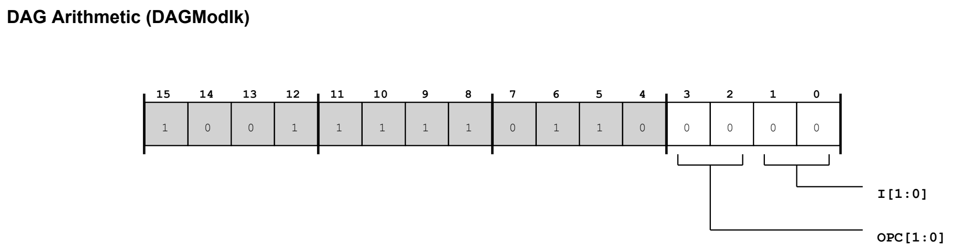


Figure 8-87: DAGModIk Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
00	IREG Register Type += 2	32-bit Add or Subtract Constant (DagAddImm)
01	IREG Register Type -= 2	32-bit Add or Subtract Constant (DagAddImm)

OPC	Syntax	Instruction
10	IREG Register Type += 4	32-bit Add or Subtract Constant (DagAd-dImm)
11	IREG Register Type -= 4	32-bit Add or Subtract Constant (DagAd-dImm)

DAG Arithmetic (DAGModIm)

DAGModIm Instruction Syntax

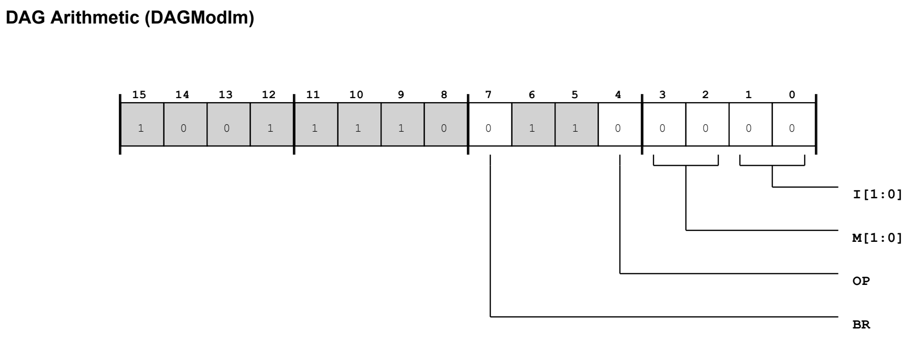


Figure 8-88: DAGModIm Instruction

The following table provides the opcode field values (OP, BR), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OP	BR	Syntax	Instruction
0	0	IREG Register Type += MREG Register Type	32-bit Add or Subtract (DagAdd32)
0	1	IREG Register Type += MREG Register Type (brev)	32-bit Add or Subtract (DagAdd32)
1	0	IREG Register Type -= MREG Register Type	32-bit Add or Subtract (DagAdd32)

ALU Operations (Dsp32Alu)

Dsp32Alu Instruction Syntax

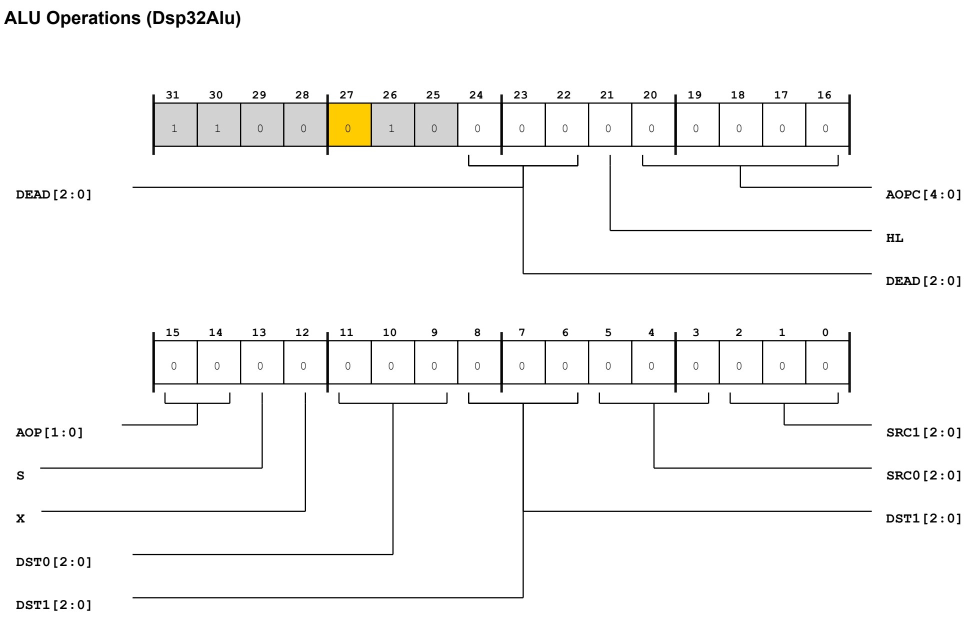


Figure 8-89: Dsp32Alu Instruction

The following table provides the opcode field values (AOPC, AOP, HL, S, X), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
00000	--	-	-	-	DREG Register Type = DREG Register Type AOPL DREG Register Type SX	Vectored 16-Bit Add or Subtract (AddSubVec16)	
00001	--	0	-	-	DREG Register Type = DREG Register Type +|+ DREG Register Type, DREG Register Type = DREG Register Type -|- DREG Register Type SXA	Vectored 16-Bit Add or Subtract (AddSubVec16)	
00001	--	1	-	-	DREG Register Type = DREG Register Type +|- DREG Register Type, DREG Register Type = DREG Register Type -|+ DREG Register Type SXA	Vectored 16-Bit Add or Subtract (AddSubVec16)	
00010	00	-	-	0	DDST0_HL = DREG_L Register Type + DREG_L Register Type SAT2	16-Bit Add or Subtract (AddSub16)	
00010	01	-	-	0	DDST0_HL = DREG_L Register Type + DREG_H Register Type SAT2	16-Bit Add or Subtract (AddSub16)	
00010	10	-	-	0	DDST0_HL = DREG_H Register Type + DREG_L Register Type SAT2	16-Bit Add or Subtract (AddSub16)	

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
00010	11	-	-	0	DDST0_HL = DREG_H Register Type + DREG_H Register Type SAT2	16-Bit Add or Subtract (AddSub16)	
00011	00	-	-	0	DDST0_HL = DREG_L Register Type - DREG_L Register Type SAT2	16-Bit Add or Subtract (AddSub16)	
00011	01	-	-	0	DDST0_HL = DREG_L Register Type - DREG_H Register Type SAT2	16-Bit Add or Subtract (AddSub16)	
00011	10	-	-	0	DDST0_HL = DREG_H Register Type - DREG_L Register Type SAT2	16-Bit Add or Subtract (AddSub16)	
00011	11	-	-	0	DDST0_HL = DREG_H Register Type - DREG_H Register Type SAT2	16-Bit Add or Subtract (AddSub16)	
00100	00	0	-	0	DREG Register Type = DREG Register Type + DREG Register Type NSAT	32-bit Add or Subtract (AddSub32)	
00100	01	0	-	0	DREG Register Type = DREG Register Type - DREG Register Type NSAT	32-bit Add or Subtract (AddSub32)	
00100	10	0	-	0	DREG Register Type = DREG Register Type + DREG Register Type, DREG Register Type = DREG Register Type - DREG Register Type SAT	32-bit Add and Subtract (AddSub32Dual)	
00101	00	-	0	0	DDST0_HL = DREG Register Type + DREG Register Type (rnd12)	32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12)	
00101	01	-	0	0	DDST0_HL = DREG Register Type - DREG Register Type (rnd12)	32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12)	
00101	10	-	0	1	DDST0_HL = DREG Register Type + DREG Register Type (rnd20)	32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20)	
00101	11	-	0	1	DDST0_HL = DREG Register Type - DREG Register Type (rnd20)	32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20)	
00110	00	0	0	0	DREG Register Type = max(DREG Register Type, DREG Register Type) (v)	Vectored 16-Bit Maximum (Max16Vec)	
00110	01	0	0	0	DREG Register Type = min(DREG Register Type, DREG Register Type) (v)	Vectored 16-Bit Minimum (Min16Vec)	
00110	10	0	0	0	DREG Register Type = abs DREG Register Type (v)	Vectored 16-Bit Absolute Value (Abs2x16)	
00111	00	0	0	0	DREG Register Type = max(DREG Register Type, DREG Register Type)	32-bit Maximum (Max32)	
00111	01	0	0	0	DREG Register Type = min(DREG Register Type, DREG Register Type)	32-Bit Minimum (Min32)	
00111	10	0	0	0	DREG Register Type = abs DREG Register Type	32-bit Absolute Value (Abs32)	

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
00111	11	0	-	0	DREG Register Type = -DREG Register Type NSAT	32-Bit Negate (Neg32)	
01000	00	0	0	0	a0 = 0	Accumulator Register Initialization (LdImm-ToAx)	
01000	00	0	1	0	a0 = a0 (s)	Accumulator0 32-Bit Saturate (ALU_Sa-tAcc0)	
01000	01	0	0	0	a1 = 0	Accumulator Register Initialization (LdImm-ToAx)	
01000	01	0	1	0	a1 = a1 (s)	Accumulator1 32-Bit Saturate (ALU_Sa-tAcc1)	
01000	10	0	0	0	a1 = a0 = 0	Dual Accumulator 0 and 1 Registers Initiali-zation (LdImmToAxDual)	
01000	10	0	1	0	a1 = a1 (s), a0 = a0 (s)	Dual Accumulator 32-Bit Saturate (ALU_SatAccDual)	
01000	11	0	0	0	a0 = a1	Move Register to Accumulator0 (MvAx-ToAx)	
01000	11	0	1	0	a1 = a0	Move Register to Accumulator0 (MvAx-ToAx)	
01001	00	-	0	0	A0_HL = DSRC0_HL	Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL)	
01001	00	0	1	-	a0 = DREG Register Type XMODE	Move Register to Accumulator1 (MvDreg-ToAx)	
01001	01	0	0	0	a0.x = DREG_L Register Type	Move Register Half (LSBs) to 8-Bit Accu-mulator Section (MvDregLToAxX)	
01001	10	-	0	0	A1_HL = DSRC0_HL	Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL)	
01001	10	0	1	-	a1 = DREG Register Type XMODE	Move Register to Accumulator1 (MvDreg-ToAx)	
01001	11	0	0	0	a1.x = DREG_L Register Type	Move Register Half (LSBs) to 8-Bit Accu-mulator Section (MvDregLToAxX)	
01010	00	0	0	0	DREG_L Register Type = a0.x	Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL)	
01010	01	0	0	0	DREG_L Register Type = a1.x	Move 8-Bit Accumulator Section to Register Half (MvAxXToDregL)	
01011	00	0	0	0	DREG Register Type = (a0 += a1)	Accumulator Add and Extract (AddAccExt)	
01011	01	-	0	0	DDST0_HL = (a0 += a1)	Accumulator Add and Extract (AddAccExt)	
01011	10	0	0	0	a0 += a1	Accumulator Add or Subtract (AddSubAcc)	
01011	10	0	1	0	a0 += a1 (w32)	Accumulator Add or Subtract (AddSubAcc)	

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
01011	11	0	0	0	a0 -= a1	Accumulator Add or Subtract (AddSubAcc)	
01011	11	0	1	0	a0 -= a1 (w32)	Accumulator Add or Subtract (AddSubAcc)	
01100	00	0	0	0	DREG_H Register Type = DREG_L Register Type = sign(DREG_H Register Type) * DREG_H Register Type + sign(DREG_L Register Type) * DREG_L Register Type	16-Bit Add on Sign (AddOnSign)	
01100	01	0	0	0	DREG Register Type = a1.l + a1.h, DREG Register Type = a0.l + a0.h	Dual Accumulator Extraction with Addition (AddAccHalf)	
01100	11	-	0	0	DDST0_HL = DREG Register Type (rnd)	Fractional 32-bit to 16-Bit Conversion (Pass32Rnd16)	
01101	00	0	0	0	(DREG Register Type, DREG Register Type) = search DREG Register Type (gt)	Vectored 16-Bit Search (Search)	
01101	01	0	0	0	(DREG Register Type, DREG Register Type) = search DREG Register Type (ge)	Vectored 16-Bit Search (Search)	
01101	10	0	0	0	(DREG Register Type, DREG Register Type) = search DREG Register Type (lt)	Vectored 16-Bit Search (Search)	
01101	11	0	0	0	(DREG Register Type, DREG Register Type) = search DREG Register Type (le)	Vectored 16-Bit Search (Search)	
01110	00	0	0	0	a0 = -a0	Accumulator0 Negate (NegAcc0)	
01110	00	1	0	0	a1 = -a0	Accumulator1 Negate (NegAcc1)	
01110	01	0	0	0	a0 = -a1	Accumulator0 Negate (NegAcc0)	
01110	01	1	0	0	a1 = -a1	Accumulator1 Negate (NegAcc1)	
01110	11	0	0	0	a1 = -a1, a0 = -a0	Dual Accumulator Negate (NegAccDual)	
01111	11	0	0	0	DREG Register Type = -DREG Register Type (v)	Vectored 16-bit Negate (Neg16Vec)	
10000	00	0	0	0	a0 = abs a0	Accumulator0 Absolute Value (AbsAcc0)	
10000	00	1	0	0	a1 = abs a0	Accumulator Absolute Value (AbsAcc1)	
10000	01	0	0	0	a0 = abs a1	Accumulator0 Absolute Value (AbsAcc0)	
10000	01	1	0	0	a1 = abs a1	Accumulator Absolute Value (AbsAcc1)	
10000	11	0	0	0	a1 = abs a1, a0 = abs a0	Accumulator Absolute Value (AbsAccDual)	
10000	11	1	-	-	a1 = DREG Register Type SMODE, a0 = DREG Register Type XMODE	Move Register to Accumulator0 & Accumulator1 (MvDregToAxDual)	2.1
10001	00	0	-	0	DREG Register Type = a1 + a0, DREG Register Type = a1 - a0 SAT	Dual Accumulator Add and Subtract to Registers (AddSubAccExt)	
10001	01	0	-	0	DREG Register Type = a0 + a1, DREG Register Type = a0 - a1 SAT	Dual Accumulator Add and Subtract to Registers (AddSubAccExt)	

AOPC	AOP	HL	S	X	Syntax	Instruction	Rev
10010	00	0	-	0	saa (PAIR0, PAIR1) RS	Vectored 8-Bit Sum of Absolute Differences (SAD8Vec)	
10010	11	0	0	0	disalgnexcpt	Disable Alignment Exception (DisAlignExcept)	
10100	00	0	-	0	DREG Register Type = byteop1p (PAIR0, PAIR1) RS	Vector Byte Average (Byteop1P) (Avg8Vec)	
10100	01	0	-	0	DREG Register Type = byteop1p (PAIR0, PAIR1) (t RSC)	Vector Byte Average (Byteop1P) (Avg8Vec)	
10101	00	0	-	0	(DREG Register Type, DREG Register Type) = byteop16p (PAIR0, PAIR1) RS	Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8)	
10101	01	0	-	0	(DREG Register Type, DREG Register Type) = byteop16m (PAIR0, PAIR1) RS	Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8)	
10110	00	0	-	0	DREG Register Type = byteop2p (PAIR0, PAIR1) (rndl RSC)	Quad Byte Average (Byteop2P) (Avg4x8Vec)	
10110	00	1	-	0	DREG Register Type = byteop2p (PAIR0, PAIR1) (rndh RSC)	Quad Byte Average (Byteop2P) (Avg4x8Vec)	
10110	01	0	-	0	DREG Register Type = byteop2p (PAIR0, PAIR1) (tl RSC)	Quad Byte Average (Byteop2P) (Avg4x8Vec)	
10110	01	1	-	0	DREG Register Type = byteop2p (PAIR0, PAIR1) (th RSC)	Quad Byte Average (Byteop2P) (Avg4x8Vec)	
10111	00	0	-	0	DREG Register Type = byteop3p (PAIR0, PAIR1) (lo RSC)	Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip)	
10111	00	1	-	0	DREG Register Type = byteop3p (PAIR0, PAIR1) (hi RSC)	Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip)	
11000	00	0	0	0	DREG Register Type = bytepack (DREG Register Type, DREG Register Type)	Pack 8-Bit to 32-Bit (BytePack)	
11000	01	0	-	0	(DREG Register Type, DREG Register Type) = byteunpack PAIR0 RS	Spread 8-Bit to 16-Bit (ByteUnPack)	
11001	00	0	-	0	DREG Register Type = DREG Register Type + DREG Register Type + ac0 SAT	32-Bit Add or Subtract with Carry (AddSubAC0)	2.0
11001	01	0	-	0	DREG Register Type = DREG Register Type - DREG Register Type + ac0 - 1 SAT	32-Bit Add or Subtract with Carry (AddSubAC0)	2.0

A0_HL

A0_HL Encode Table

HL	Syntax
0	a0.l

HL	Syntax
1	a0.h

A1_HL

A1_HL Encode Table

HL	Syntax
0	a1.l
1	a1.h

AOPL

AOPL Encode Table

AOP	Syntax
00	+|+
01	+|-
10	-|+
11	-|-

DDST0_HL

DDST0_HL Encode Table

HL	Syntax
0	DREG_L Register Type
1	DREG_H Register Type

DSRC0_HL

DSRC0_HL Encode Table

HL	Syntax
0	DREG_L Register Type
1	DREG_H Register Type

NSAT

NSAT Encode Table

S	Syntax	Description
0	(ns)	The `(ns)` option directs the ALU not to saturate the result.
1	(s)	The `(s)` option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.

PAIR0

PAIR0 Encode Table

SRC0	Syntax
00-	r1:0
01-	r3:2

PAIR1

PAIR1 Encode Table

SRC1	Syntax
00-	r1:0
01-	r3:2

RS

RS Encode Table

S	Syntax	Description
0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
1	(r)	The `(r)` option directs the ALU to reverse the order of the source registers within each register pair.

RSC

RSC Encode Table

S	Syntax	Description
0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
1	,r	The `,r` option directs the ALU to reverse the order of the source registers within each register pair.

SAT

SAT Encode Table

S	Syntax	Description
0	(ns)	The `(ns)` option directs the ALU not to saturate the result.
1	(s)	The `(s)` option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.

SAT2

SAT2 Encode Table

S	Syntax	Description
0	(ns)	The `(ns)` option directs the ALU not to saturate the result.
1	(s)	The `(s)` option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.

SMODE

SMODE Encode Table

S	Syntax	Description	Rev
0	(x)	The `(x)` option directs the ALU to sign-extend the result.	2.1
1	(z)	The `(z)` option directs the ALU to zero extend the result.	2.1

SX

SX Encode Table

S	X	Syntax	Description
0	0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
0	1	(co)	The `(co)` option directs the ALU to swap (cross order) the order of the results in the destination register.
1	0	(s)	The `(s)` option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.
1	1	(sco)	The `(sco)` option directs the ALU to apply the combination of the `(co)` and `(s)` options.

SXA

SXA Encode Table

AOP	S	X	Syntax	Description
00	0	0		The default (no option select) operation directs the ALU to execute the operation with no optional modification to the result.
00	0	1	(co)	The `(co)` option directs the ALU to swap (cross order) the order of the results in the destination register.
00	1	0	(s)	The `(s)` option directs the ALU to saturate the result at 16 or 32 bits, depending on the operand size.
00	1	1	(sco)	The `(sco)` option directs the ALU to apply the combination of the `(co)` and `(s)` options.
10	0	0	(asr)	The `(asr)` option directs the ALU to arithmetic shift right, halving the result (divide by 2) before storing in the destination register.
10	0	1	(co, asr)	The `(co, asr)` option directs the ALU to to apply the combination of the `(asr)` and `(co)` options
10	1	0	(s, asr)	The `(s, asr)` option directs the ALU to arithmetic shift right (halving the result; divide by 2) then saturate before storing in the destination register.
10	1	1	(sco, asr)	The `(sco, asr)` option directs the ALU to apply the combination of the `(asr)` and `(sco)` options
11	0	0	(asl)	The `(asl)` option directs the ALU to arithmetic shift left, doubling the result (multiply by 2, truncated) before storing in the destination register.
11	0	1	(co, asl)	The `(co, asl)` option directs the ALU to to apply the combination of the `(asl)` and `(co)` options

AOP	S	X	Syntax	Description
11	1	0	(s, asl)	The `(s, asl)` option directs the ALU to arithmetic shift left (doubling the result; multiply by 2, truncated) then saturate before storing in the destination register.
11	1	1	(sco, asl)	The `(sco, asl)` option directs the ALU to apply the combination of the `(asl)` and `(sco)` options

XMODE

XMODE Encode Table

X	Syntax	Description	Rev
0	(x)	The `(x)` option directs the ALU to sign-extend the result.	2.1
1	(z)	The `(z)` option directs the ALU to zero extend the result.	2.1

Multiply Accumulate (Dsp32Mac)

Dsp32Mac Instruction Syntax

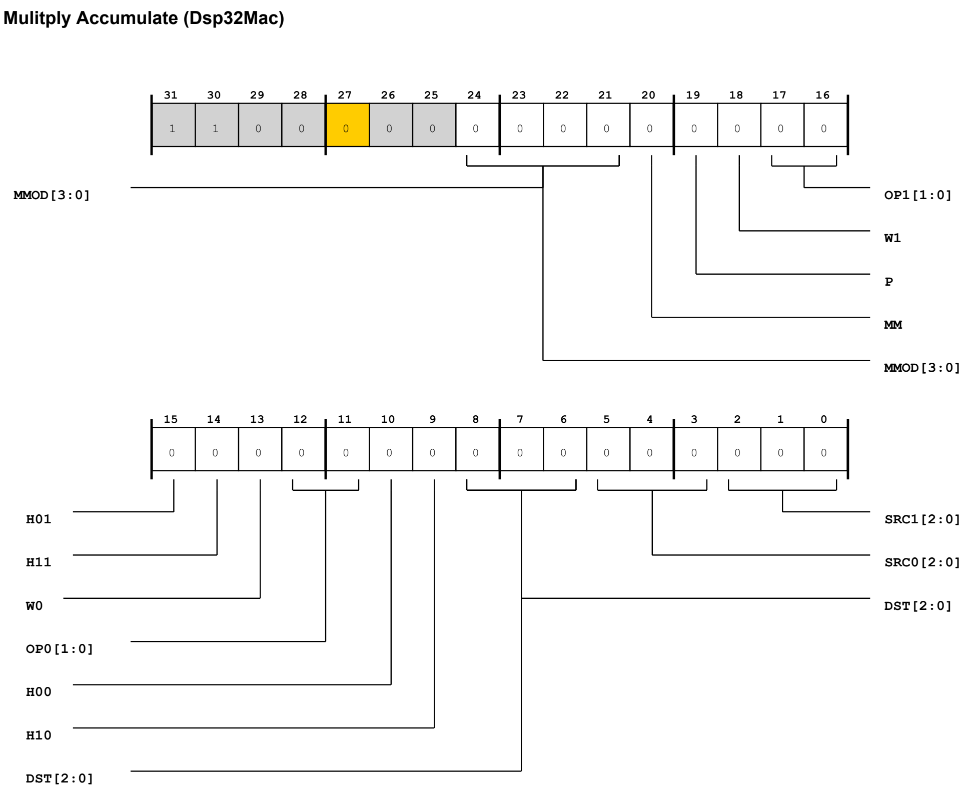


Figure 8-90: Dsp32Mac Instruction

The following table provides the opcode field values (MMOD), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

MMOD	Syntax
0---	TRADMAC
10--	TRADMAC
1100	TRADMAC
1101	CMPLXMAC
111-	CMPLXMAC

CMODE

CMODE Encode Table

MMOD	Syntax	Description
1101		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1111	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

CMPLXMAC

CMPLXMAC Encode Table

W0	P	OP0	Syntax Instruction	Rev
0	0	00	a1:0 = CMPLXOP CMODE	2.0
0	0	01	a1:0 += CMPLXOP CMODE	2.0
0	0	10	a1:0 -= CMPLXOP CMODE	2.0
1	0	00	DREG Register Type = (a1:0 = CMPLXOP) NARROWING_CMODE	2.0
1	0	01	DREG Register Type = (a1:0 += CMPLXOP) NARROWING_CMODE	2.0
1	0	10	DREG Register Type = (a1:0 -= CMPLXOP) NARROWING_CMODE	2.0
1	0	11	DREG Register Type = CMPLXOP NARROWING_CMODE	2.0
1	1	00	DREG_PAIR Register Type = (a1:0 = CMPLXOP) CMODE	2.0
1	1	01	DREG_PAIR Register Type = (a1:0 += CMPLXOP) CMODE	2.0
1	1	10	DREG_PAIR Register Type = (a1:0 -= CMPLXOP) CMODE	2.0
1	1	11	DREG_PAIR Register Type = CMPLXOP CMODE	2.0

CMPLXOP

CMPLXOP Encode Table

OP1	Syntax
00	cmul(DREG Register Type, DREG Register Type)
01	cmul(DREG Register Type, DREG Register Type*)
10	cmul(DREG Register Type*, DREG Register Type*)

MAC0

MAC0 Encode Table

OP0	Syntax
00	a0 = MAC0S
01	a0 += MAC0S

OP0	Syntax
10	a0 -= MAC0S

MAC0S

MAC0S Encode Table

H00	H10	Syntax
0	0	DREG_L Register Type * DREG_L Register Type
0	1	DREG_L Register Type * DREG_H Register Type
1	0	DREG_H Register Type * DREG_L Register Type
1	1	DREG_H Register Type * DREG_H Register Type

MAC1

MAC1 Encode Table

OP1	Syntax
00	a1 = MAC1S
01	a1 += MAC1S
10	a1 -= MAC1S

MAC1S

MAC1S Encode Table

H01	H11	Syntax
0	0	DREG_L Register Type * DREG_L Register Type
0	1	DREG_L Register Type * DREG_H Register Type
1	0	DREG_H Register Type * DREG_L Register Type
1	1	DREG_H Register Type * DREG_H Register Type

MML

MML Encode Table

MM	Syntax	Description
0		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1	(m)	The `(m)` option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.

MMLMMOD0

MMLMMOD0 Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0011	(w32)	The `(w32)` option directs the MAC to use signed fraction with 32-bit saturation. Multiply 1.15 x 1.15 to produce 1.31 format data after shift correction. Sign extend the result to 9.31 format before passing it to the accumulator. Saturate the accumulator after copying or

MM	MMOD	Syntax	Description
			accumulating at bit 31 to maintain 1.31 precision. Result is between minimum -1 and maximum 1-2^{-31} (or, expressed in hex, between minimum 0xFF 8000 0000 and maximum 0x00 7FFF FFFF).
0	0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum 1-2^{-32} (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1	0000	(m)	The `(m)` option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0011	(m,w32)	The `(m,w32)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction (with 32-bit saturation) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	1000	(m,is)	The `(m,is)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.

MMLMMOD1

MMLMMOD1 Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0001	(s2rnd)	The `(s2rnd)` option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)
0	0010	(t)	The `(t)` option directs the MAC to use signed fraction format with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum

MM	MMOD	Syntax	Description
			0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum 1-2^{-32} (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	0110	(tfu)	The `(tfu)` option directs the MAC to use unsigned fraction format with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate Accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0	1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1001	(iss2)	The `(iss2)` option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1011	(ih)	The `(ih)` option directs the MAC to use signed integer format, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round accumulator 40.0 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the

MM	MMOD	Syntax	Description
			rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	1100	(iu)	The `(iu)` option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1	0000	(m)	The `(m)` option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.
1	0010	(m,t)	The `(m,t)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	0110	(m,tfu)	The `(m,tfu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format (with truncation) operation.
1	1000	(m,is)	The `(m,is)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The `(m,iss2)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.
1	1011	(m,ih)	The `(m,ih)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format (high word extract) operation.
1	1100	(m,iu)	The `(m,iu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned integer format operation.

MMLMMODE

MMLMMODE Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0001	(s2rnd)	The `(s2rnd)` option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)
0	0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format

MM	MMOD	Syntax	Description
			before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1001	(iss2)	The `(iss2)` option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1100	(iu)	The `(iu)` option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1	0000	(m)	The `(m)` option directs the MAC to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	1000	(m,is)	The `(m,is)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The `(m,iss2)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.
1	1100	(m,iu)	The `(m,iu)` option directs the MAC to use mixed mode multiply format (valid only for MAC1) and unsigned integer format operation.

MMOD0

MMOD0 Encode Table

MMOD	Syntax	Description
0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(w32)	The `(w32)` option directs the MAC to use signed fraction with 32-bit saturation. Multiply 1.15 x 1.15 to produce 1.31 format data after shift correction. Sign extend the result to 9.31 format before passing it to the accumulator. Saturate the accumulator after copying or accumulating at bit 31 to maintain 1.31 precision. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0xFF 8000 0000 and maximum 0x00 7FFF FFFF).
0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD1

MMOD1 Encode Table

MMOD	Syntax	Description
0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum 1-2^{-31} (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0001	(s2rnd)	The `(s2rnd)` option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum 1-2^{-31} (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)
0010	(t)	The `(t)` option directs the MAC to use signed fraction format with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum

MMOD	Syntax	Description
		0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum 1-2^{-32} (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0110	(tfu)	The `(tfu)` option directs the MAC to use unsigned fraction format with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate Accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The `(iss2)` option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1011	(ih)	The `(ih)` option directs the MAC to use signed integer format, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to Accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round accumulator 40.0 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the

MMOD	Syntax	Description
		rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
1100	(iu)	The `(iu)` option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

MMODE

MMODE Encode Table

MMOD	Syntax	Description
0000		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0001	(s2rnd)	The `(s2rnd)` option directs the MAC to use signed fraction format with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF)

MMOD	Syntax	Description
0100	(fu)	The `(fu)` option directs the MAC to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
1000	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The `(iss2)` option directs the MAC to use signed integer format with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1100	(iu)	The `(iu)` option directs the MAC to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

NARROWING_CMODE

NARROWING_CMODE Encode Table

MMOD	Syntax	Description
1101		The default (no option) operation directs the MAC to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to Accumulator. Then, saturate Accumulator to maintain 9.31 precision; Accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round Accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1110	(t)	The `(t)` option directs the MAC to use signed fraction format with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
1111	(is)	The `(is)` option directs the MAC to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

TRADMAC

TRADMAC Encode Table

P	W1	W0	OP1	OP0	Syntax	Instruction
0	0	0	11	0-	MAC0 MMOD0	16 x 16-Bit MAC (Mac16)
0	0	0	11	10	MAC0 MMOD0	16 x 16-Bit MAC (Mac16)

P	W1	W0	OP1	OP0	Syntax	Instruction
0	0	0	0-	11	MAC1 MMLMMOD0	16 x 16-Bit MAC (Mac16)
0	0	0	10	11	MAC1 MMLMMOD0	16 x 16-Bit MAC (Mac16)
0	0	0	0-	0-	MAC1 MML, MAC0 MMOD0	Dual 16 x 16-Bit MAC (ParaMac16AndMac16)
0	0	0	0-	10	MAC1 MML, MAC0 MMOD0	Dual 16 x 16-Bit MAC (ParaMac16AndMac16)
0	0	0	10	0-	MAC1 MML, MAC0 MMOD0	Dual 16 x 16-Bit MAC (ParaMac16AndMac16)
0	0	0	10	10	MAC1 MML, MAC0 MMOD0	Dual 16 x 16-Bit MAC (ParaMac16AndMac16)
0	0	1	11	11	DREG_L Register Type = a0 MMOD1	Move 16-Bit Accumulator Section to Low Half Register (MvA0ToDregL)
0	0	1	11	0-	DREG_L Register Type = (MAC0) MMOD1	16 x 16-Bit MAC with Move to Register (Mac16WithMv)
0	0	1	11	10	DREG_L Register Type = (MAC0) MMOD1	16 x 16-Bit MAC with Move to Register (Mac16WithMv)
0	0	1	0-	11	MAC1 MML, DREG_L Register Type = a0 MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)
0	0	1	10	11	MAC1 MML, DREG_L Register Type = a0 MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)
0	0	1	0-	0-	MAC1 MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)
0	0	1	0-	10	MAC1 MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)
0	0	1	10	0-	MAC1 MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)
0	0	1	10	10	MAC1 MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMac16WithMv)
0	1	0	11	11	DREG_H Register Type = a1 MMLMMOD1	Move 16-Bit Accumulator Section to High Half Register (MvA1ToDregH)
0	1	0	11	0-	DREG_H Register Type = a1 MML, MAC0 MMOD1	Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)
0	1	0	11	10	DREG_H Register Type = a1 MML, MAC0 MMOD1	Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)

P	W1	W0	OP1	OP0	Syntax	Instruction
0	1	0	0-	11	DREG_H Register Type = (MAC1) MMLMMOD1	16 x 16-Bit MAC with Move to Register (Mac16WithMv)
0	1	0	10	11	DREG_H Register Type = (MAC1) MMLMMOD1	16 x 16-Bit MAC with Move to Register (Mac16WithMv)
0	1	0	0-	0-	DREG_H Register Type = (MAC1) MML, MAC0 MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16With-MvAndMac16)
0	1	0	0-	10	DREG_H Register Type = (MAC1) MML, MAC0 MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16With-MvAndMac16)
0	1	0	10	0-	DREG_H Register Type = (MAC1) MML, MAC0 MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16With-MvAndMac16)
0	1	0	10	10	DREG_H Register Type = (MAC1) MML, MAC0 MMOD1	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16With-MvAndMac16)
0	1	1	11	11	DREG_H Register Type = a1 MML, DREG_L Register Type = a0 MMOD1	Dual Move Accumulators to Half Registers (ParaMvA1ToDregH-withMvA0ToDregL)
0	1	1	11	0-	DREG_H Register Type = a1 MML, DREG_L Register Type = (MAC0) MMOD1	Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)
0	1	1	11	10	DREG_H Register Type = a1 MML, DREG_L Register Type = (MAC0) MMOD1	Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)
0	1	1	0-	11	DREG_H Register Type = (MAC1) MML, DREG_L Register Type = a0 MMOD1	Dual 16 x 16-Bit MAC with Moves to Registers (Para-Mac16WithMvAndMv)
0	1	1	10	11	DREG_H Register Type = (MAC1) MML, DREG_L Register Type = a0 MMOD1	Dual 16 x 16-Bit MAC with Moves to Registers (Para-Mac16WithMvAndMv)
0	1	1	0-	0-	DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Moves to Registers (Para-Mac16WithMvAnd-Mac16WithMv)
0	1	1	10	0-	DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Moves to Registers (Para-Mac16WithMvAnd-Mac16WithMv)

P	W1	W0	OP1	OP0	Syntax	Instruction
0	1	1	0-	10	DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Moves to Registers (Para-Mac16WithMvAndMac16WithMv)
0	1	1	10	10	DREG_H Register Type = (MAC1) MML, DREG_L Register Type = (MAC0) MMOD1	Dual 16 x 16-Bit MAC with Moves to Registers (Para-Mac16WithMvAnd-Mac16WithMv)
1	0	1	11	11	DREG_E Register Type = a0 MMODE	Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE)
1	0	1	11	0-	DREG_E Register Type = (MAC0) MMODE	32 x 32-Bit MAC with Move to Register (Mac32WithMv)
1	0	1	11	10	DREG_E Register Type = (MAC0) MMODE	32 x 32-Bit MAC with Move to Register (Mac32WithMv)
1	0	1	0-	11	MAC1 MML, DREG_E Register Type = a0 MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)
1	0	1	0-	0-	MAC1 MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16And-Mac16WithMv)
1	0	1	0-	10	MAC1 MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16And-Mac16WithMv)
1	0	1	10	11	MAC1 MML, DREG_E Register Type = a0 MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16AndMv)
1	0	1	10	0-	MAC1 MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16And-Mac16WithMv)
1	0	1	10	10	MAC1 MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16And-Mac16WithMv)
1	1	0	11	11	DREG_O Register Type = a1 MMLMMODE	Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO)
1	1	0	11	0-	DREG_O Register Type = a1 MML, MAC0 MMODE	Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)
1	1	0	11	10	DREG_O Register Type = a1 MML, MAC0 MMODE	Dual Move to Register and 16 x 16-Bit MAC (ParaMvAndMac16)
1	1	0	0-	11	DREG_O Register Type = (MAC1) MMLMMODE	32 x 32-Bit MAC with Move to Register (Mac32WithMv)
1	1	0	10	11	DREG_O Register Type = (MAC1) MMLMMODE	32 x 32-Bit MAC with Move to Register (Mac32WithMv)

P	W1	W0	OP1	OP0	Syntax	Instruction
1	1	0	0-	0-	DREG_O Register Type = (MAC1) MML, MAC0 MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16)
1	1	0	0-	10	DREG_O Register Type = (MAC1) MML, MAC0 MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16)
1	1	0	10	0-	DREG_O Register Type = (MAC1) MML, MAC0 MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16)
1	1	0	10	10	DREG_O Register Type = (MAC1) MML, MAC0 MMODE	Dual 16 x 16-Bit MAC with Move to Register (ParaMac16WithMvAndMac16)
1	1	1	11	11	DREG_O Register Type = a1 MML, DREG_E Register Type = a0 MMODE	Dual Move Accumulators to Register (ParaMvA1ToDregOwithMvA0ToDregE)
1	1	1	11	0-	DREG_O Register Type = a1 MML, DREG_E Register Type = (MAC0) MMODE	Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)
1	1	1	11	10	DREG_O Register Type = a1 MML, DREG_E Register Type = (MAC0) MMODE	Dual Move to Register and 16 x 16-Bit MAC with Move to Register (ParaMvAndMac16WithMv)
1	1	1	0-	11	DREG_O Register Type = (MAC1) MML, DREG_E Register Type = a0 MMODE	Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMv)
1	1	1	10	11	DREG_O Register Type = (MAC1) MML, DREG_E Register Type = a0 MMODE	Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMv)
1	1	1	0-	0-	DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)
1	1	1	0-	10	DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)
1	1	1	10	0-	DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)

P	W1	W0	OP1	OP0	Syntax	Instruction
1	1	1	10	10	DREG_O Register Type = (MAC1) MML, DREG_E Register Type = (MAC0) MMODE	Dual 16 x 16-Bit MAC with Moves to Registers (ParaMac16WithMvAndMac16WithMv)

Multiply with 3 operands (Dsp32Mult)

Dsp32Mult Instruction Syntax

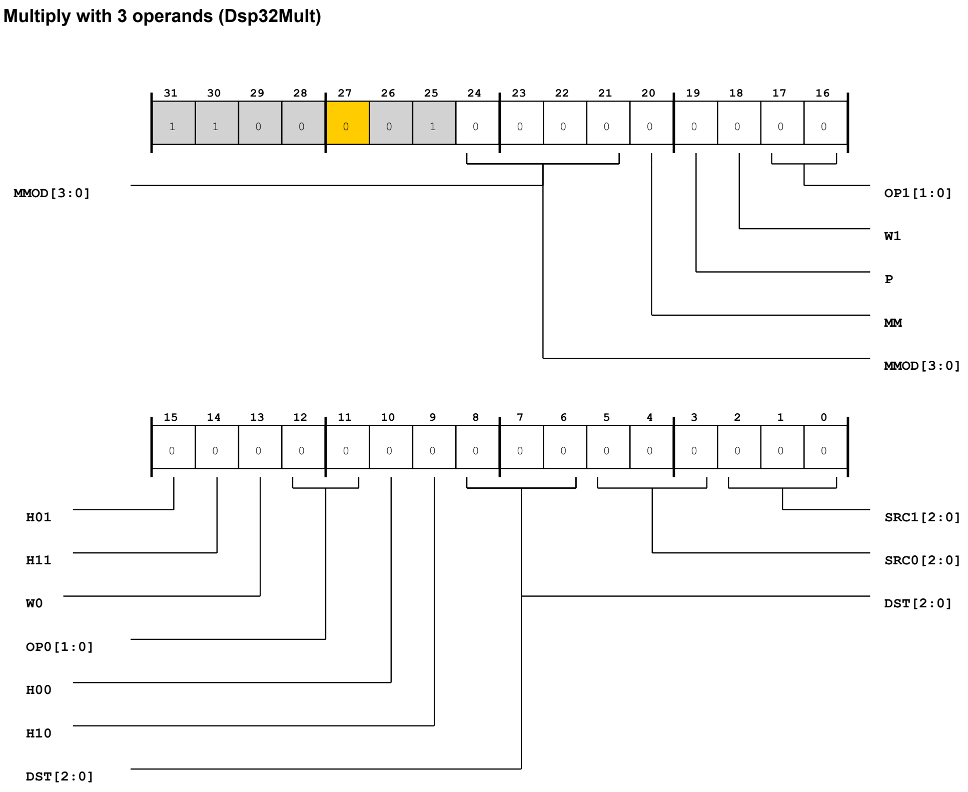


Figure 8-91: Dsp32Mult Instruction

The following table provides the opcode field values (OP1, OP0, MM, P, W1, W0), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OP1	OP0	MM	P	W1	W0	Syntax	Instruction	Rev
00	00	0	0	0	1	DREG_L Register Type = MUL0 MMOD1	16 x 16-Bit Multiply (Mult16)	
00	00	-	0	1	0	DREG_H Register Type = MUL1 MMLMMOD1	16 x 16-Bit Multiply (Mult16)	
00	00	-	0	1	1	DREG_H Register Type = MUL1 MML, DREG_L Register Type = MUL0 MMOD1	Dual 16 x 16-Bit Multiply (ParaMult16AndMult16)	

OP1	OP0	MM	P	W1	W0	Syntax	Instruction	Rev
00	00	0	1	0	1	DREG_E Register Type = MUL0 MMODE	16 x 16-Bit Multiply (Mult16)	
00	00	-	1	1	0	DREG_O Register Type = MUL1 MMLMMODE	16 x 16-Bit Multiply (Mult16)	
00	00	-	1	1	1	DREG_O Register Type = MUL1 MML, DREG_E Register Type = MUL0 MMODE	Move Accumulator to Register (MvAxToDreg)	
01	00	0	0	0	0	a1:0 = DREG Register Type * DREG Register Type M32MMOD	32 x 32-Bit MAC (Mac32)	2.1
01	01	0	0	0	0	a1:0 += DREG Register Type * DREG Register Type M32MMOD	32 x 32-Bit MAC (Mac32)	2.1
01	10	0	0	0	0	a1:0 -= DREG Register Type * DREG Register Type M32MMOD	32 x 32-Bit MAC (Mac32)	2.1
01	00	0	0	0	1	DREG Register Type = (a1:0 = DREG Register Type * DREG Register Type) M32MMOD1	32 x 32-Bit MAC with Move to Register (Mac32WithMv)	2.1
01	01	0	0	0	1	DREG Register Type = (a1:0 += DREG Register Type * DREG Register Type) M32MMOD1	32 x 32-Bit MAC with Move to Register (Mac32WithMv)	2.1
01	10	0	0	0	1	DREG Register Type = (a1:0 -= DREG Register Type * DREG Register Type) M32MMOD1	32 x 32-Bit MAC with Move to Register (Mac32WithMv)	2.1
01	11	0	0	0	1	DREG Register Type = a1:0 M32MMOD2	Move Accumulator to Register (MvAxToDreg)	2.1
01	00	0	1	0	1	DREG_PAIR Register Type = (a1:0 = DREG Register Type * DREG Register Type) M32MMOD	32 x 32-Bit MAC with Move to Register (Mac32WithMv)	2.1
01	01	0	1	0	1	DREG_PAIR Register Type = (a1:0 += DREG Register Type * DREG Register Type) M32MMOD	32 x 32-Bit MAC with Move to Register (Mac32WithMv)	2.1
01	10	0	1	0	1	DREG_PAIR Register Type = (a1:0 -= DREG Register Type * DREG Register Type) M32MMOD	32 x 32-Bit MAC with Move to Register (Mac32WithMv)	2.1
01	11	0	1	0	1	DREG_PAIR Register Type = a1:0 M32MMOD	Move Accumulator to Register (MvAxToDreg)	2.1
01	00	1	0	0	1	DREG Register Type = DREG Register Type * DREG Register Type M32MMOD2	32 x 32-bit Multiply (Mult32)	2.1
01	00	1	1	0	1	DREG_PAIR Register Type = DREG Register Type * DREG Register Type M32MMOD	32 x 32-bit Multiply (Mult32)	2.1

M32MMOD

M32MMOD Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0010	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(is,ns)	The `(is,ns)` option directs the multiplier to
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0110	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0

MMOD	Syntax	Description
		precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0111	(iu,ns)	The `(iu,ns)` option directs the multiplier to
1000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1010	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1011	(m,is,ns)	The `(m,is,ns)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (with no saturation) operation.

M32MMOD1

M32MMOD1 Encode Table

MMOD	Syntax	Description
0001	(t)	The `(t)` option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0010	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(is,ns)	The `(is,ns)` option directs the multiplier to

MMOD	Syntax	Description
0101	(tfu)	The `(tfu)` option directs the multiplier to use unsigned fraction with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0110	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}\text{-}1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0111	(iu,ns)	The `(iu,ns)` option directs the multiplier to
1001	(m,t)	The `(m,t)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1010	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1011	(m,is,ns)	The `(m,is,ns)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (with no saturation) operation.

M32MMOD2

M32MMOD2 Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
0001	(t)	The `(t)` option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0010	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0011	(is,ns)	The `(is,ns)` option directs the multiplier to
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0101	(tfu)	The `(tfu)` option directs the multiplier to use unsigned fraction with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

MMOD	Syntax	Description
0110	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0111	(iu,ns)	The `(iu,ns)` option directs the multiplier to
1000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1001	(m,t)	The `(m,t)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1010	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1011	(m,is,ns)	The `(m,is,ns)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (with no saturation) operation.

MML

MML Encode Table

MM	Syntax	Description
0		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum 1-2^{-31} (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer

MM	Syntax	Description
		mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.

MMLMMOD1

MMLMMOD1 Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0001	(s2rnd)	The `(s2rnd)` option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0010	(t)	The `(t)` option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1

MM	MMOD	Syntax	Description
			and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum 1-2^{-32} (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	0110	(tfu)	The `(tfu)` option directs the multiplier to use unsigned fraction with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
0	1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register.

MM	MMOD	Syntax	Description
			Result is between minimum -2^{31} and maximum 2^{31}-1 (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1011	(ih)	The `(ih)` option directs the multiplier to use signed integer, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round accumulator 40.0 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum 2^{15}-1 (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0	1100	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum 2^{16}-1 (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1	0000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.
1	0010	(m,t)	The `(m,t)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with truncation) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	0110	(m,tfu)	The `(m,tfu)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format (with truncation) operation.
1	1000	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The `(m,iss2)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.
1	1011	(m,ih)	The `(m,ih)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format (high word extract) operation.
1	1100	(m,iu)	The `(m,iu)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned integer format operation.

MMLMMODE

MMLMMODE Encode Table

MM	MMOD	Syntax	Description
0	0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum 1-2^{-31} (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0001	(s2rnd)	The `(s2rnd)` option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum 1-2^{-15} (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum 1-2^{-31} (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum 1-2^{-16} (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum 1-2^{-32} (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0	1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format

MM	MMOD	Syntax	Description
			before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0	1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}-1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1	0000	(m)	The `(m)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1). When issued in a fraction mode instruction (with default, FU, T, TFU, or S2RND mode), multiply 1.15 * 0.16 to produce 1.31 results. When issued in an integer mode instruction (with IS, ISS2, or IH mode), multiply 16.0 * 16.0 (signed * unsigned) to produce 32.0 results. No shift correction in either case. *`Src_reg_0`* is the signed operand and *`Src_reg_1`* is the unsigned operand. Accumulation and extraction proceed according to the other mode selection or default.
1	0001	(m,s2rnd)	The `(m,s2rnd)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed fraction format (with scaling and rounding) operation.
1	0100	(m,fu)	The `(m,fu)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and unsigned fraction format operation.
1	1000	(m,is)	The `(m,is)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format operation.
1	1001	(m,iss2)	The `(m,iss2)` option directs the multiplier to use mixed mode multiply format (valid only for MAC1) and signed integer format with scaling operation.

MMOD1

MMOD1 Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0001	(s2rnd)	The `(s2rnd)` option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1\text{-}2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0010	(t)	The `(t)` option directs the multiplier to use signed fraction with truncation. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To extract to half-register, truncate accumulator 9.31 format value at bit 16. (Perform no rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1\text{-}2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator.

MMOD	Syntax	Description
		Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
0110	(tfu)	The `(tfu)` option directs the multiplier to use unsigned fraction with truncation. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. (Same as the FU mode.) Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To extract to half-register, truncate accumulator 8.32 format value at bit 16. (Perform no rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).
1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1011	(ih)	The `(ih)` option directs the multiplier to use signed integer, high word extract. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 32.0 precision; accumulator result is between minimum 0x00 8000 0000 and maximum 0x00 7FFF FFFF. To extract to half-register, round

MMOD	Syntax	Description
		accumulator 40.0 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate to 32.0 result. Copy the upper 16 bits of that value to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}-1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF).
1100	(iu)	The `(iu)` option directs the multiplier to use unsigned integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Zero extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. Extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum 0 and maximum $2^{16}-1$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF).

MMODE

MMODE Encode Table

MMOD	Syntax	Description
0000		The default (no option selected) operation directs the multiplier to use signed fraction format. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate the result to 1.31 precision and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
0001	(s2rnd)	The `(s2rnd)` option directs the multiplier to use signed fraction with scaling and rounding. Multiply 1.15 * 1.15 formats to produce 1.31 results after shift correction. The special case of 0x8000 * 0x8000 is saturated to 0x7FFF FFFF to fit the 1.31 result. (Same as the default mode.) Sign extend 1.31 result to 9.31 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 9.31 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, shift the accumulator contents one place to the left (multiply x 2). Round accumulator 9.31 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 1.15 precision and copy it to the destination register half. Result is between minimum -1 and maximum $1-2^{-15}$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result to 1.31 precision, and copy it to the destination register. Result is between minimum -1 and maximum $1-2^{-31}$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MMOD	Syntax	Description
0100	(fu)	The `(fu)` option directs the multiplier to use unsigned fraction format. Multiply 0.16* 0.16 formats to produce 0.32 results. No shift correction. The special case of 0x8000 * 0x8000 yields 0x4000 0000. No saturation is necessary since no shift correction occurs. Zero extend 0.32 result to 8.32 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 8.32 precision; accumulator result is between minimum 0x00 0000 0000 and maximum 0xFF FFFF FFFF. To ***extract to half register***, round accumulator 8.32 format value at bit 16. (The `ASTAT.RND_MOD` bit controls the rounding.) Saturate the result to 0.16 precision and copy it to the destination register half. Result is between minimum 0 and maximum $1\text{-}2^{-16}$ (or, expressed in hex, between minimum 0x0000 and maximum 0xFFFF). To ***extract to full register***, saturate the result to 0.32 precision and copy it to the destination register. Result is between minimum 0 and maximum $1\text{-}2^{-32}$ (or, expressed in hex, between minimum 0x0000 0000 and maximum 0xFFFF FFFF).
1000	(is)	The `(is)` option directs the multiplier to use signed integer format. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Saturate for 16.0 precision and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, saturate for 32.0 precision and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).
1001	(iss2)	The `(iss2)` option directs the multiplier to use signed integer with scaling. Multiply 16.0 * 16.0 formats to produce 32.0 results. No shift correction. (Same as the IS mode.) Sign extend 32.0 result to 40.0 format before copying or accumulating to accumulator. Then, saturate Accumulator to maintain 40.0 precision; accumulator result is between minimum 0x80 0000 0000 and maximum 0x7F FFFF FFFF. To ***extract to half register***, extract the lower 16 bits of the accumulator. Shift them one place to the left (multiply x 2). Saturate the result for 16.0 format and copy to the destination register half. Result is between minimum -2^{15} and maximum $2^{15}\text{-}1$ (or, expressed in hex, between minimum 0x8000 and maximum 0x7FFF). To ***extract to full register***, shift the accumulator contents one place to the left (multiply x 2), saturate the result for 32.0 format, and copy to the destination register. Result is between minimum -2^{31} and maximum $2^{31}\text{-}1$ (or, expressed in hex, between minimum 0x8000 0000 and maximum 0x7FFF FFFF).

MUL0

MUL0 Encode Table

H00	H10	Syntax
0	0	DREG_L Register Type * DREG_L Register Type

H00	H10	Syntax
0	1	DREG_L Register Type * DREG_H Register Type
1	0	DREG_H Register Type * DREG_L Register Type
1	1	DREG_H Register Type * DREG_H Register Type

MUL1

MUL1 Encode Table

H01	H11	Syntax
0	0	DREG_L Register Type * DREG_L Register Type
0	1	DREG_L Register Type * DREG_H Register Type
1	0	DREG_H Register Type * DREG_L Register Type
1	1	DREG_H Register Type * DREG_H Register Type

Shift (Dsp32Shf)

Dsp32Shf Instruction Syntax

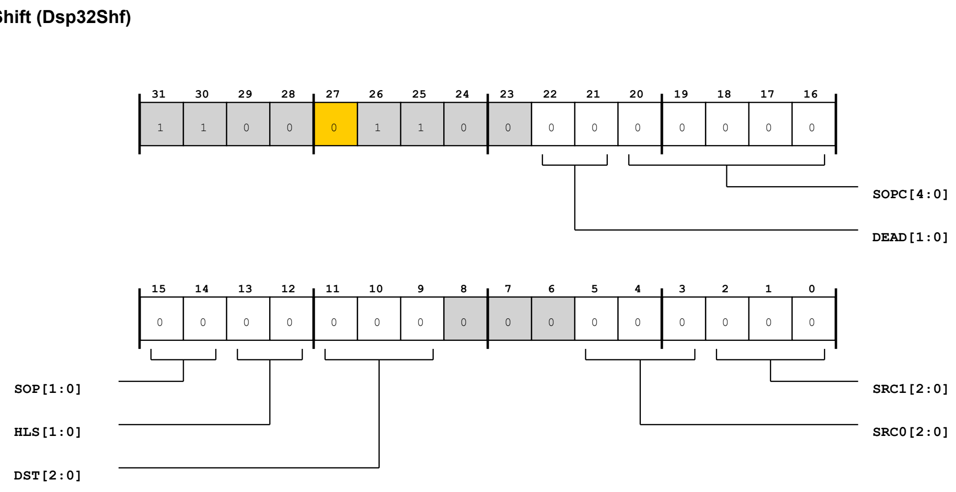


Figure 8-92: Dsp32Shf Instruction

The following table provides the opcode field values (SOPC, SOP, HLS), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

SOPC	SOP	HLS	Syntax	Instruction
00000	00	00	DREG_L Register Type = ashift DREG_L Register Type by DREG_L Register Type	16-Bit Arithmetic Shift (AShift16)
00000	00	01	DREG_L Register Type = ashift DREG_H Register Type by DREG_L Register Type	16-Bit Arithmetic Shift (AShift16)
00000	00	10	DREG_H Register Type = ashift DREG_L Register Type by DREG_L Register Type	16-Bit Arithmetic Shift (AShift16)
00000	00	11	DREG_H Register Type = ashift DREG_H Register Type by DREG_L Register Type	16-Bit Arithmetic Shift (AShift16)
00000	01	00	DREG_L Register Type = ashift DREG_L Register Type by DREG_L Register Type (s)	16-Bit Arithmetic Shift (AShift16)
00000	01	01	DREG_L Register Type = ashift DREG_H Register Type by DREG_L Register Type (s)	16-Bit Arithmetic Shift (AShift16)
00000	01	10	DREG_H Register Type = ashift DREG_L Register Type by DREG_L Register Type (s)	16-Bit Arithmetic Shift (AShift16)
00000	01	11	DREG_H Register Type = ashift DREG_H Register Type by DREG_L Register Type (s)	16-Bit Arithmetic Shift (AShift16)
00000	10	00	DREG_L Register Type = lshift DREG_L Register Type by DREG_L Register Type	16-Bit Logical Shift (LShift16)
00000	10	01	DREG_L Register Type = lshift DREG_H Register Type by DREG_L Register Type	16-Bit Logical Shift (LShift16)
00000	10	10	DREG_H Register Type = lshift DREG_L Register Type by DREG_L Register Type	16-Bit Logical Shift (LShift16)
00000	10	11	DREG_H Register Type = lshift DREG_H Register Type by DREG_L Register Type	16-Bit Logical Shift (LShift16)
00001	00	00	DREG Register Type = ashift DREG Register Type by DREG_L Register Type (v)	Vectored 16-Bit Arithmetic (AShift16Vec)
00001	01	00	DREG Register Type = ashift DREG Register Type by DREG_L Register Type (v,s)	Vectored 16-Bit Arithmetic (AShift16Vec)
00001	10	00	DREG Register Type = lshift DREG Register Type by DREG_L Register Type (v)	Vectored 16-Bit Logical Shift (LShift16Vec)
00010	00	00	DREG Register Type = ashift DREG Register Type by DREG_L Register Type	32-Bit Arithmetic Shift (AShift32)
00010	01	00	DREG Register Type = ashift DREG Register Type by DREG_L Register Type (s)	32-Bit Arithmetic Shift (AShift32)
00010	10	00	DREG Register Type = lshift DREG Register Type by DREG_L Register Type	32-Bit Logical Shift (LShift)

SOPC	SOP	HLS	Syntax	Instruction
00010	11	00	DREG Register Type = rot DREG Register Type by DREG_L Register Type	32-Bit Rotate (Shift_Rot32)
00011	00	00	a0 = ashift a0 by DREG_L Register Type	Accumulator Arithmetic Shift (AShiftAcc)
00011	00	01	a1 = ashift a1 by DREG_L Register Type	Accumulator Arithmetic Shift (AShiftAcc)
00011	01	00	a0 = lshift a0 by DREG_L Register Type	Accumulator Logical Shift (LShiftA)
00011	01	01	a1 = lshift a1 by DREG_L Register Type	Accumulator Logical Shift (LShiftA)
00011	10	00	a0 = rot a0 by DREG_L Register Type	Accumulator Rotate (Shift_RotAcc)
00011	10	01	a1 = rot a1 by DREG_L Register Type	Accumulator Rotate (Shift_RotAcc)
00100	00	00	DREG Register Type = pack (DREG_L Register Type, DREG_L Register Type)	Pack 16-Bit to 32-Bit (Pack16Vec)
00100	01	00	DREG Register Type = pack (DREG_L Register Type, DREG_H Register Type)	Pack 16-Bit to 32-Bit (Pack16Vec)
00100	10	00	DREG Register Type = pack (DREG_H Register Type, DREG_L Register Type)	Pack 16-Bit to 32-Bit (Pack16Vec)
00100	11	00	DREG Register Type = pack (DREG_H Register Type, DREG_H Register Type)	Pack 16-Bit to 32-Bit (Pack16Vec)
00101	00	00	DREG_L Register Type = signbits DREG Register Type	Redundant Sign Bits (Shift_SignBits32)
00101	01	00	DREG_L Register Type = signbits DREG_L Register Type	Redundant Sign Bits (Shift_SignBits32)
00101	10	00	DREG_L Register Type = signbits DREG_H Register Type	Redundant Sign Bits (Shift_SignBits32)
00110	00	00	DREG_L Register Type = signbits a0	Redundant Sign Bits (Shift_SignBitsAcc)
00110	01	00	DREG_L Register Type = signbits a1	Redundant Sign Bits (Shift_SignBitsAcc)
00110	11	00	DREG_L Register Type = ones DREG Register Type	Ones Count (Shift_Ones)
00111	00	00	DREG_L Register Type = expadj (DREG Register Type, DREG_L Register Type)	Exponent Detection (Shift_ExpAdj32)
00111	01	00	DREG_L Register Type = expadj (DREG Register Type, DREG_L Register Type) (v)	Exponent Detection (Shift_ExpAdj32)
00111	10	00	DREG_L Register Type = expadj (DREG_L Register Type, DREG_L Register Type)	Exponent Detection (Shift_ExpAdj32)
00111	11	00	DREG_L Register Type = expadj (DREG_H Register Type, DREG_L Register Type)	Exponent Detection (Shift_ExpAdj32)
01000	00	00	bitmux (DREG Register Type, DREG Register Type, a0) (asr)	Bit Mux (BitMux)

SOPC	SOP	HLS	Syntax	Instruction
01000	01	00	bitmux (DREG Register Type, DREG Register Type, a0) (asl)	Bit Mux (BitMux)
01001	00	00	DREG_L Register Type = vit_max (DREG Register Type) (asl)	16-Bit Modulo Maximum with History (Shift_VitMax)
01001	01	00	DREG_L Register Type = vit_max (DREG Register Type) (asr)	16-Bit Modulo Maximum with History (Shift_VitMax)
01001	10	00	DREG Register Type = vit_max (DREG Register Type, DREG Register Type) (asl)	Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax)
01001	11	00	DREG Register Type = vit_max (DREG Register Type, DREG Register Type) (asr)	Dual 16-Bit Modulo Maximum with History (Shift_DualVitMax)
01010	00	00	DREG Register Type = extract (DREG Register Type, DREG_L Register Type) (z)	Extract Bits (Shift_Extract)
01010	01	00	DREG Register Type = extract (DREG Register Type, DREG_L Register Type) (x)	Extract Bits (Shift_Extract)
01010	10	00	DREG Register Type = deposit (DREG Register Type, DREG Register Type)	Deposit Bits (Shift_Deposit)
01010	11	00	DREG Register Type = deposit (DREG Register Type, DREG Register Type) (x)	Deposit Bits (Shift_Deposit)
01011	00	00	DREG_L Register Type = cc = bxorshift (a0, DREG Register Type)	32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF)
01011	01	00	DREG_L Register Type = cc = bxor (a0, DREG Register Type)	32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF)
01100	00	00	a0 = bxorshift (a0, a1, cc)	40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF)
01100	01	00	DREG_L Register Type = cc = bxor (a0, a1, cc)	40-Bit BXOR LSFR with Feedback to a Register (BXOR)
01101	00	00	DREG Register Type = align8 (DREG Register Type, DREG Register Type)	Byte Align (Shift_Align)
01101	01	00	DREG Register Type = align16 (DREG Register Type, DREG Register Type)	Byte Align (Shift_Align)
01101	10	00	DREG Register Type = align24 (DREG Register Type, DREG Register Type)	Byte Align (Shift_Align)

Shift Immediate (Dsp32ShfImm)

Dsp32ShfImm Instruction Syntax

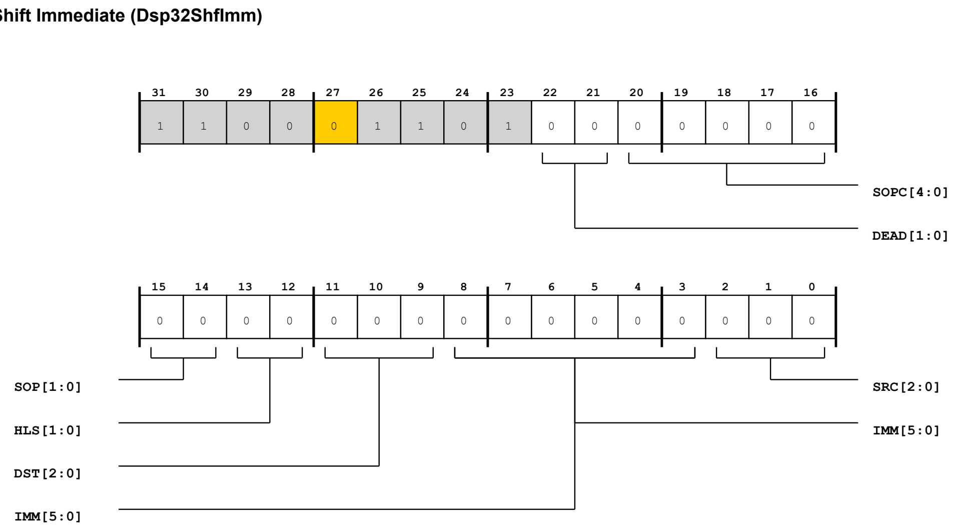


Figure 8-93: Dsp32ShfImm Instruction

The following table provides the opcode field values (SOPC, SOP, HLS), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

SOPC	SOP	HLS	Syntax	Instruction
00000	00	00	DREG_L Register Type = DREG_L Register Type AHSH4	16-Bit Arithmetic Shift (AShift16)
00000	00	01	DREG_L Register Type = DREG_H Register Type AHSH4	16-Bit Arithmetic Shift (AShift16)
00000	00	10	DREG_H Register Type = DREG_L Register Type AHSH4	16-Bit Arithmetic Shift (AShift16)
00000	00	11	DREG_H Register Type = DREG_H Register Type AHSH4	16-Bit Arithmetic Shift (AShift16)
00000	01	00	DREG_L Register Type = DREG_L Register Type AHSH4S	16-Bit Arithmetic Shift (AShift16)
00000	01	01	DREG_L Register Type = DREG_H Register Type AHSH4S	16-Bit Arithmetic Shift (AShift16)
00000	01	10	DREG_H Register Type = DREG_L Register Type AHSH4S	16-Bit Arithmetic Shift (AShift16)
00000	01	11	DREG_H Register Type = DREG_H Register Type AHSH4S	16-Bit Arithmetic Shift (AShift16)
00000	10	00	DREG_L Register Type = DREG_L Register Type LHSH4	16-Bit Logical Shift (LShift16)

SOPC	SOP	HLS	Syntax	Instruction
00000	10	01	DREG_L Register Type = DREG_H Register Type LHSH4	16-Bit Logical Shift (LShift16)
00000	10	10	DREG_H Register Type = DREG_L Register Type LHSH4	16-Bit Logical Shift (LShift16)
00000	10	11	DREG_H Register Type = DREG_H Register Type LHSH4	16-Bit Logical Shift (LShift16)
00001	00	00	DREG Register Type = DREG Register Type AHSH4 (v)	Vectored 16-Bit Arithmetic (AShift16Vec)
00001	01	00	DREG Register Type = DREG Register Type AHSH4VS	Vectored 16-Bit Arithmetic (AShift16Vec)
00001	10	00	DREG Register Type = DREG Register Type LHSH4 (v)	Vectored 16-Bit Logical Shift (LShift16Vec)
00010	00	00	DREG Register Type = DREG Register Type ASH5	32-Bit Arithmetic Shift (AShift32)
00010	01	00	DREG Register Type = DREG Register Type ASH5S	32-Bit Arithmetic Shift (AShift32)
00010	10	00	DREG Register Type = DREG Register Type LSH5	32-Bit Logical Shift (LShift)
00010	11	00	DREG Register Type = rot DREG Register Type by imm6 Register Type	32-Bit Rotate (Shift_Rot32)
00011	00	00	a0 = a0 ASH5	Accumulator Arithmetic Shift (AShiftAcc)
00011	00	01	a1 = a1 ASH5	Accumulator Arithmetic Shift (AShiftAcc)
00011	01	00	a0 = a0 LSH5	Accumulator Logical Shift (LShiftA)
00011	01	01	a1 = a1 LSH5	Accumulator Logical Shift (LShiftA)
00011	10	00	a0 = rot a0 by imm6 Register Type	Accumulator Rotate (Shift_RotAcc)
00011	10	01	a1 = rot a1 by imm6 Register Type	Accumulator Rotate (Shift_RotAcc)

AHSH4

AHSH4 Encode Table

IMM	Syntax	Rev
000000	<<< 0	2.1.1
00----	<<< uimm4nz Register Type	2.1.1
11----	>>> uimm4nznegpos Register Type	

AHSH4S

AHSH4S Encode Table

IMM	Syntax	Rev
000000	<< 0 (s)	
00----	<< uimm4nz Register Type (s)	
11----	>>> uimm4nznegpos Register Type (s)	2.1.1

AHSH4VS

AHSH4VS Encode Table

IMM	Syntax	Rev
000000	<< 0 (v,s)	
00----	<< uimm4nz Register Type (v,s)	
11----	>>> uimm4nznegpos Register Type (v,s)	2.1.1

ASH5

ASH5 Encode Table

IMM	Syntax	Rev
000000	<<< 0	2.1.1
0-----	<<< uimm5nz Register Type	2.1.1
1-----	>>> uimm5nznegpos Register Type	

ASH5S

ASH5S Encode Table

IMM	Syntax	Rev
000000	<< 0 (s)	

IMM	Syntax	Rev
0-----	<< uimm5nz Register Type (s)	
1-----	>>> uimm5nznegpos Register Type (s)	2.1.1

LHSH4

LHSH4 Encode Table

IMM	Syntax
000000	<< 0
00----	<< uimm4nz Register Type
11----	>> uimm4nznegpos Register Type

LSH5

LSH5 Encode Table

IMM	Syntax
000000	<< 0
0-----	<< uimm5nz Register Type
1-----	>> uimm5nznegpos Register Type

Load/Store (DspLdSt)

DspLdSt Instruction Syntax

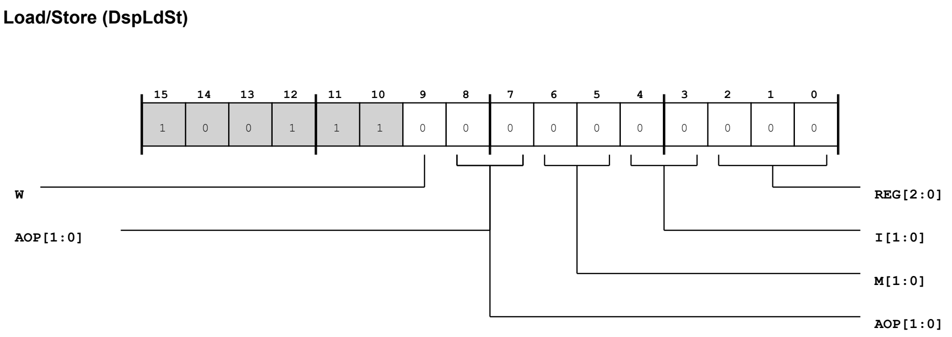


Figure 8-94: DspLdSt Instruction

The following table provides the opcode field values (W, M, AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	M	AOP	Syntax	Instruction
0	00	00	DREG Register Type = [IREG Register Type ++]	32-Bit Load from Memory (LdM32bitToDreg)
0	00	01	DREG Register Type = [IREG Register Type--]	32-Bit Load from Memory (LdM32bitToDreg)
0	00	10	DREG Register Type = [IREG Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
0	01	00	DREG_L Register Type = w[IREG Register Type++]	16-Bit Load from Memory (LdM16bitToDregL)
0	01	01	DREG_L Register Type = w[IREG Register Type--]	16-Bit Load from Memory (LdM16bitToDregL)
0	01	10	DREG_L Register Type = w[IREG Register Type]	16-Bit Load from Memory (LdM16bitToDregL)
0	10	00	DREG_H Register Type = w[IREG Register Type++]	16-Bit Load from Memory (LdM16bitToDregH)
0	10	01	DREG_H Register Type = w[IREG Register Type--]	16-Bit Load from Memory (LdM16bitToDregH)
0	10	10	DREG_H Register Type = w[IREG Register Type]	16-Bit Load from Memory (LdM16bitToDregH)
0	--	11	DREG Register Type = [IREG Register Type ++ MREG Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
1	00	00	[IREG Register Type++] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)
1	00	01	[IREG Register Type--] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)
1	00	10	[IREG Register Type] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)
1	01	00	w[IREG Register Type++] = DREG_L Register Type	16-Bit Store to Memory (StDregLToM16bit)
1	01	01	w[IREG Register Type--] = DREG_L Register Type	16-Bit Store to Memory (StDregLToM16bit)
1	01	10	w[IREG Register Type] = DREG_L Register Type	16-Bit Store to Memory (StDregLToM16bit)
1	10	00	w[IREG Register Type++] = DREG_H Register Type	16-Bit Store to Memory (StDregHToM16bit)
1	10	01	w[IREG Register Type--] = DREG_H Register Type	16-Bit Store to Memory (StDregHToM16bit)

W	M	AOP	Syntax	Instruction
1	10	10	w[IREG Register Type] = DREG_H Register Type	16-Bit Store to Memory (StDregHToM16bit)
1	--	11	[IREG Register Type ++ MREG Register Type] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)

Jump/Call to 32-bit Immediate (Jump32)

Jump32 Instruction Syntax

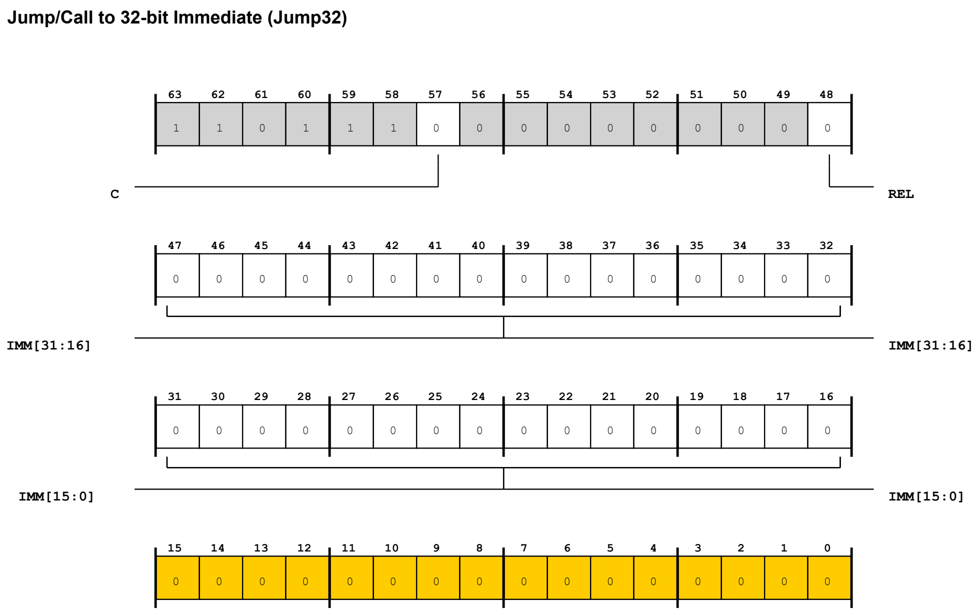


Figure 8-95: Jump32 Instruction

The following table provides the opcode field values (C, REL), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

C	REL	Syntax	Instruction
0	0	jump.a buimm32 Register Type	Jump Immediate (JumpAbs)
0	1	jump bimm32 Register Type	Jump Immediate (JumpAbs)
1	0	call.a buimm32 Register Type	Call (Call)
1	1	call bimm32 Register Type	Call (Call)

Load Immediate Word (LdImm)

LdImm Instruction Syntax

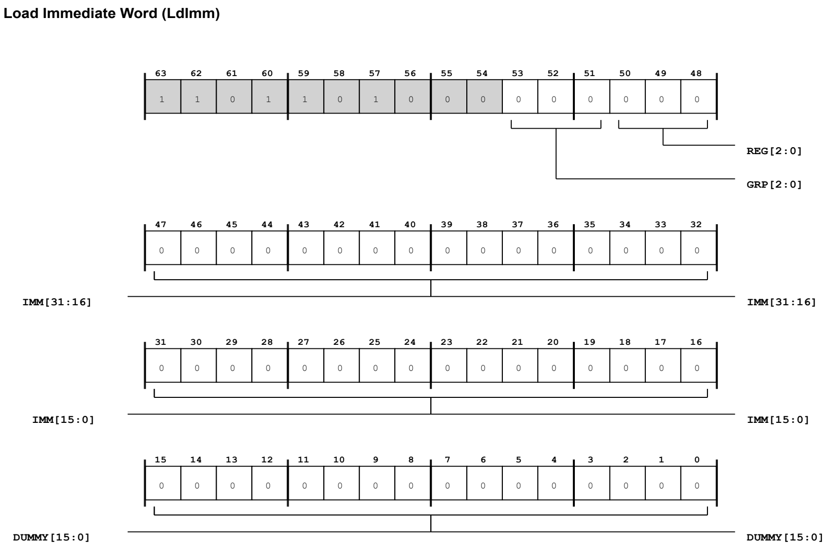


Figure 8-96: LdImm Instruction

The following table provides the opcode field values (GRP, REG), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

GRP	REG	Syntax	Instruction
000	---	DREG Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
001	---	PREG Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
010	0--	IREG Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
010	1--	MREG Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
011	0--	BREG Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
011	1--	LREG Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
100	000	a0.x = imm32 Register Type	32-Bit Accumulator Register (.x) Initialization (LdImm-ToAxX)
100	001	a0.w = imm32 Register Type	32-Bit Accumulator Register (.w) Initialization (LdImm-ToAxW)

GRP	REG	Syntax	Instruction
100	010	a1.x = imm32 Register Type	32-Bit Accumulator Register (.x) Initialization (LdImm-ToAxX)
100	011	a1.w = imm32 Register Type	32-Bit Accumulator Register (.w) Initialization (LdImm-ToAxW)
100	110	astat = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
100	111	rets = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
110	---	SYSREG2 Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)
111	---	SYSREG3 Register Type = imm32 Register Type	32-Bit Register Initialization (LdImmToReg)

Load Immediate Half Word (LdImmHalf)

LdImmHalf Instruction Syntax

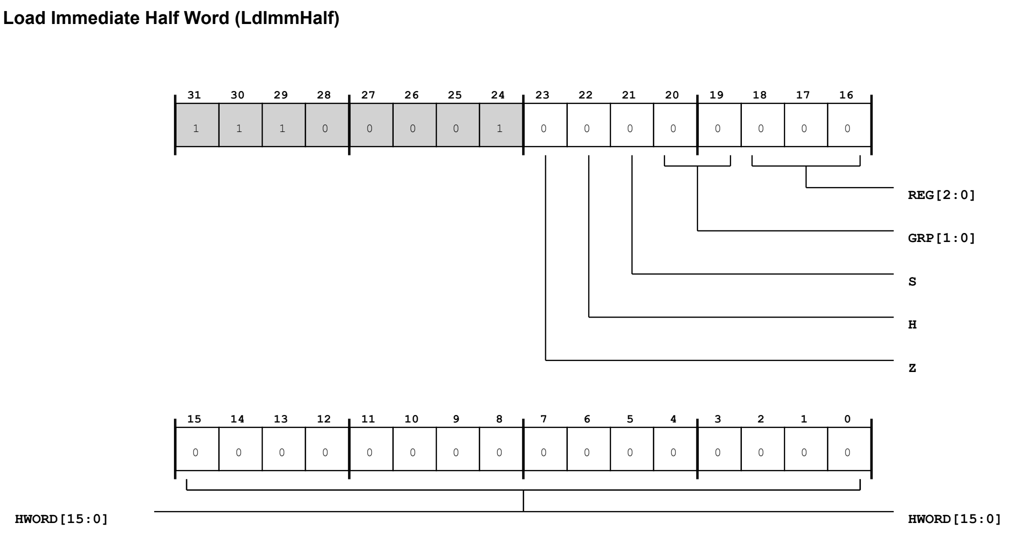


Figure 8-97: LdImmHalf Instruction

The following table provides the opcode field values (H, Z, S), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

H	Z	S	Syntax	Instruction
0	0	0	DST_L = imm16 Register Type	16-Bit Register Initialization (LdImmTo-DregHL)

H	Z	S	Syntax	Instruction
0	0	1	DST = imm16 Register Type (x)	32-Bit Register Initialization (LdImmToReg)
0	1	0	DST = rimm16 Register Type (z)	32-Bit Register Initialization (LdImmToReg)
1	0	0	DST_H = imm16 Register Type	16-Bit Register Initialization (LdImmToDregHL)

DST

DST Encode Table

GRP	REG	Syntax
00	---	DREG Register Type
01	---	PREG Register Type
10	0--	IREG Register Type
10	1--	MREG Register Type
11	0--	BREG Register Type
11	1--	LREG Register Type

DST_H

DST_H Encode Table

GRP	REG	Syntax
00	---	DREG_H Register Type
01	---	PREG_H Register Type
10	0--	IREG_H Register Type
10	1--	MREG_H Register Type
11	0--	BREG_H Register Type
11	1--	LREG_H Register Type

DST_L

DST_L Encode Table

GRP	REG	Syntax
00	---	DREG_L Register Type
01	---	PREG_L Register Type

GRP	REG	Syntax
10	0--	IREG_L Register Type
10	1--	MREG_L Register Type
11	0--	BREG_L Register Type
11	1--	LREG_L Register Type

Load/Store (LdSt)

LdSt Instruction Syntax

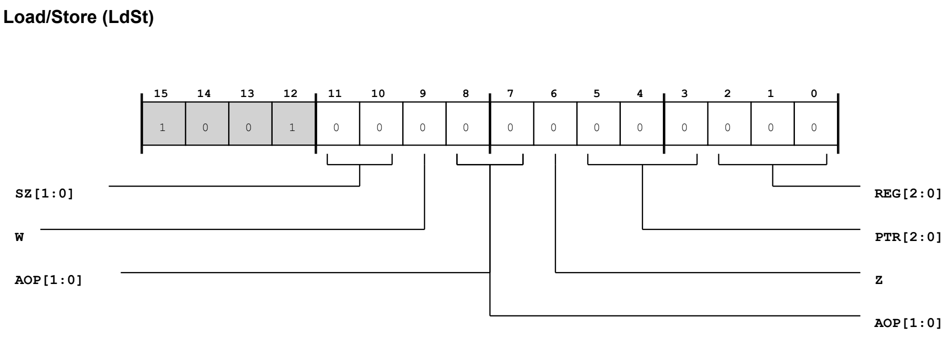


Figure 8-98: LdSt Instruction

The following table provides the opcode field values (W, SZ, Z, AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	SZ	Z	AOP	Syntax	Instruction
0	00	0	00	DREG Register Type = [PREG Register Type++]	32-Bit Load from Memory (LdM32bitToDreg)
0	00	0	01	DREG Register Type = [PREG Register Type--]	32-Bit Load from Memory (LdM32bitToDreg)
0	00	0	10	DREG Register Type = [PREG Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
0	01	0	00	DREG Register Type = w[PREG Register Type++] (z)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	01	0	01	DREG Register Type = w[PREG Register Type--] (z)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	01	0	10	DREG Register Type = w[PREG Register Type] (z)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	01	1	00	DREG Register Type = w[PREG Register Type++] (x)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)

W	SZ	Z	AOP	Syntax	Instruction
0	01	1	01	DREG Register Type = w[PREG Register Type--] (x)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	01	1	10	DREG Register Type = w[PREG Register Type] (x)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	10	0	00	DREG Register Type = b[PREG Register Type++] (z)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
0	10	0	01	DREG Register Type = b[PREG Register Type--] (z)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
0	10	0	10	DREG Register Type = b[PREG Register Type] (z)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
0	10	1	00	DREG Register Type = b[PREG Register Type++] (x)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
0	10	1	01	DREG Register Type = b[PREG Register Type--] (x)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
0	10	1	10	DREG Register Type = b[PREG Register Type] (x)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
1	00	0	00	[PREG Register Type++] = DREG Register Type	32-Bit Store to Memory (StDreg-ToM32bit)
1	00	0	01	[PREG Register Type--] = DREG Register Type	32-Bit Store to Memory (StDreg-ToM32bit)
1	00	0	10	[PREG Register Type] = DREG Register Type	32-Bit Store to Memory (StDreg-ToM32bit)
1	00	1	00	[PREG Register Type++] = PREG Register Type	Store Pointer (StPregToM32bit)
1	00	1	01	[PREG Register Type--] = PREG Register Type	Store Pointer (StPregToM32bit)
1	00	1	10	[PREG Register Type] = PREG Register Type	Store Pointer (StPregToM32bit)
1	01	0	00	w[PREG Register Type++] = DREG Register Type	16-Bit Store to Memory (StDregL-ToM16bit)
1	01	0	01	w[PREG Register Type--] = DREG Register Type	16-Bit Store to Memory (StDregL-ToM16bit)
1	01	0	10	w[PREG Register Type] = DREG Register Type	16-Bit Store to Memory (StDregL-ToM16bit)
1	10	0	00	b[PREG Register Type++] = DREG Register Type	8-Bit Store to Memory (StDreg-ToM08bit)
1	10	0	01	b[PREG Register Type--] = DREG Register Type	8-Bit Store to Memory (StDreg-ToM08bit)

W	SZ	Z	AOP	Syntax	Instruction
1	10	0	10	b[PREG Register Type] = DREG Register Type	8-Bit Store to Memory (StDreg-ToM08bit)

Load/Store 32-bit Absolute Address (LdStAbs)

LdStAbs Instruction Syntax

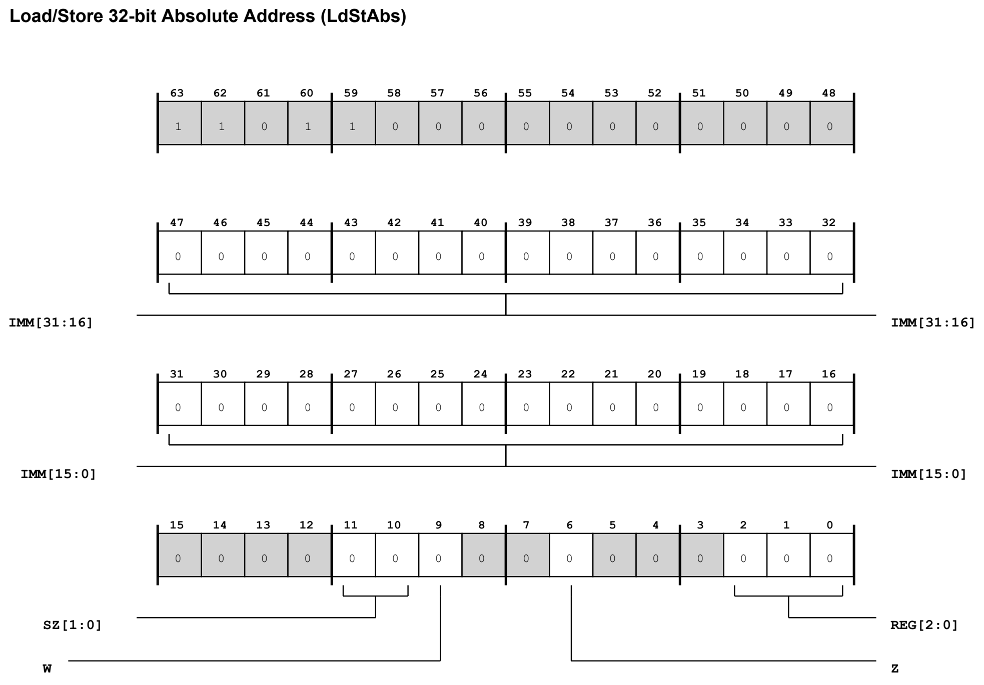


Figure 8-99: LdStAbs Instruction

The following table provides the opcode field values (W, SZ, Z), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	SZ	Z	Syntax	Instruction
0	00	0	DREG Register Type = [uimm32 Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
0	00	1	PREG Register Type = [uimm32 Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
0	01	0	DREG Register Type = w[uimm32 Register Type] (z)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	01	1	DREG Register Type = w[uimm32 Register Type] (x)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	10	0	DREG Register Type = b[uimm32 Register Type] (z)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)

W	SZ	Z	Syntax	Instruction
0	10	1	DREG Register Type = b[uimm32 Register Type] (x)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
0	11	0	DREG_L Register Type = w[uimm32 Register Type]	16-Bit Load from Memory (LdM16bitToDregL)
0	11	1	DREG_H Register Type = w[uimm32 Register Type]	16-Bit Load from Memory (LdM16bitToDregH)
1	00	0	[uimm32 Register Type] = DREG Register Type	32-Bit Store to Memory (StDreg-ToM32bit)
1	00	1	[uimm32 Register Type] = PREG Register Type	32-Bit Store to Memory (StDreg-ToM32bit)
1	01	0	w[uimm32 Register Type] = DREG Register Type	16-Bit Store to Memory (StDregL-ToM16bit)
1	10	0	b[uimm32 Register Type] = DREG Register Type	8-Bit Store to Memory (StDreg-ToM08bit)
1	11	1	w[uimm32 Register Type] = DREG_H Register Type	16-Bit Store to Memory (StDregH-ToM16bit)

Long Load/Store with indexed addressing (LdStExcl)

LdStExcl Instruction Syntax

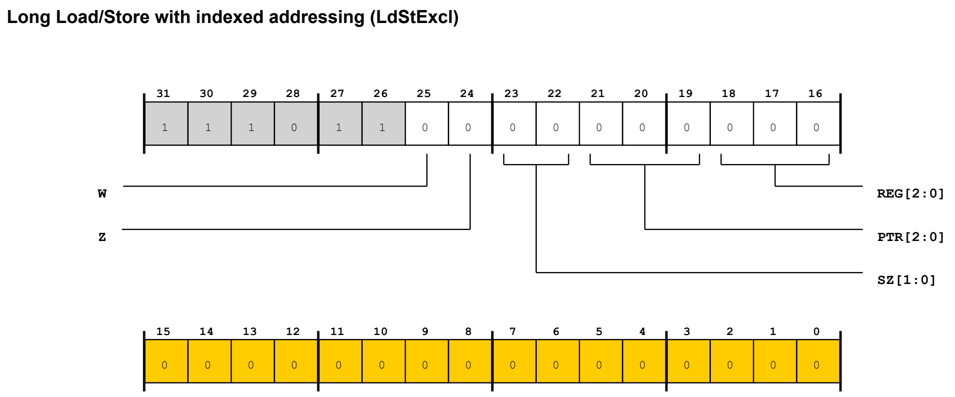


Figure 8-100: LdStExcl Instruction

The following table provides the opcode field values (W, SZ, Z), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	SZ	Z	Syntax	Instruction	Rev
0	00	0	DREG Register Type = [PREG Register Type] (excl)	32-Bit Load from Memory (LdX32bitTo-Dreg)	2.2

W	SZ	Z	Syntax	Instruction	Rev
0	01	0	DREG Register Type = w[PREG Register Type] (z,excl)	16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg)	2.2
0	01	1	DREG Register Type = w[PREG Register Type] (x,excl)	16-Bit Load from Memory to 32-Bit Register (LdX16bitToDreg)	2.2
0	10	0	DREG Register Type = b[PREG Register Type] (z,excl)	8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg)	2.2
0	10	1	DREG Register Type = b[PREG Register Type] (x,excl)	8-Bit Load from Memory to 32-bit Register (LdX08bitToDreg)	2.2
0	11	0	DREG_L Register Type = w[PREG Register Type] (excl)	16-Bit Load from Memory (LdX16bitToDregL)	2.2
0	11	1	DREG_H Register Type = w[PREG Register Type] (excl)	16-Bit Load from Memory (LdX16bitToDregH)	2.2
1	00	0	cc = ([PREG Register Type] = DREG Register Type) (excl)	32-Bit Store to Memory (StDregToX32bit)	2.2
1	01	0	cc = (w[PREG Register Type] = DREG Register Type) (excl)	16-Bit Store to Memory (StDregLToX16bit)	2.2
1	10	0	cc = (b[PREG Register Type] = DREG Register Type) (excl)	8-Bit Store to Memory (StDregToX08bit)	2.2
1	11	0	cc = (w[PREG Register Type] = DREG_H Register Type) (excl)	16-Bit Store to Memory (StDregHToX16bit)	2.2
1	11	1	syncexcl	SyncExcl (SyncExcl)	2.2

Load/Store indexed with small immediate offset (LdStII)

LdStII Instruction Syntax

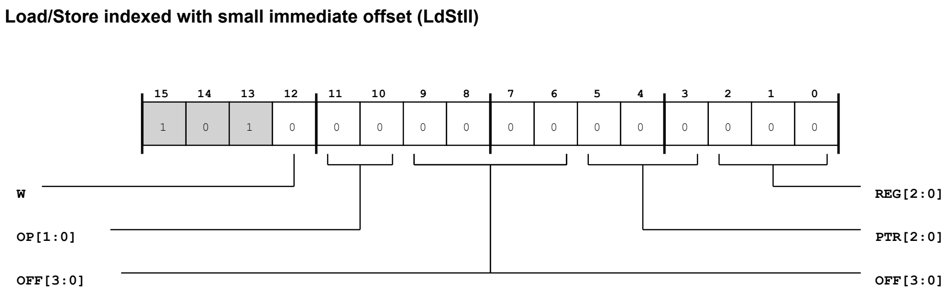


Figure 8-101: LdStII Instruction

The following table provides the opcode field values (W, OP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	OP	Syntax	Instruction
0	00	DREG Register Type = [PREG Register Type + uimm4s4 Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
0	01	DREG Register Type = w[PREG Register Type + uimm4s2 Register Type] (z)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	10	DREG Register Type = w[PREG Register Type + uimm4s2 Register Type] (x)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
1	00	[PREG Register Type + uimm4s4 Register Type] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)
1	01	w[PREG Register Type + uimm4s2 Register Type] = DREG Register Type	16-Bit Store to Memory (StDregLToM16bit)
1	11	[PREG Register Type + uimm4s4 Register Type] = PREG Register Type	Store Pointer (StPregToM32bit)

Load/Store indexed with small immediate offset FP (LdStIIFP)

LdStIIFP Instruction Syntax

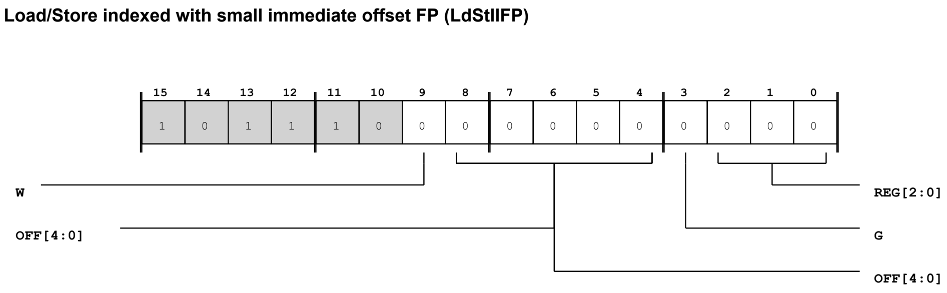


Figure 8-102: LdStIIFP Instruction

The following table provides the opcode field values (W, G), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	G	Syntax	Instruction
0	0	DREG Register Type = [fp - imm5nzs4negpos Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
1	0	[fp - imm5nzs4negpos Register Type] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)
1	1	[fp - imm5nzs4negpos Register Type] = PREG Register Type	32-Bit Store to Memory (StDregToM32bit)

Long Load/Store with indexed addressing (LdStIdxI)

LdStIdxI Instruction Syntax

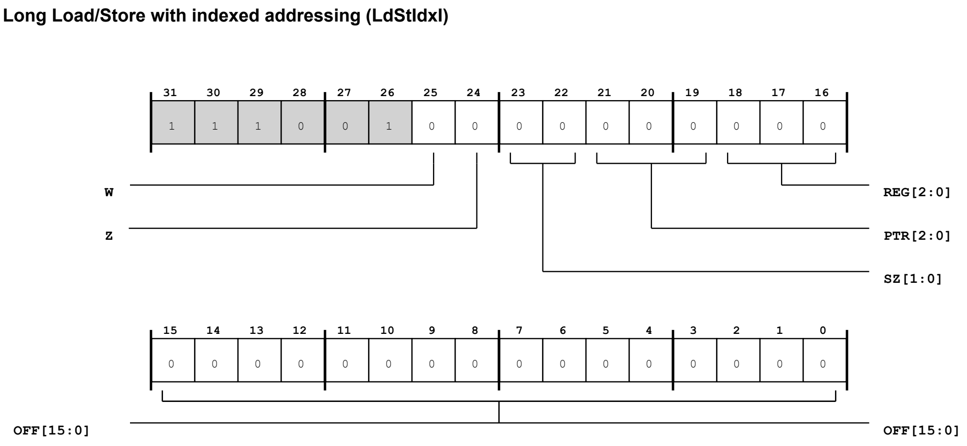


Figure 8-103: LdStIdxI Instruction

The following table provides the opcode field values (W, SZ, Z), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	SZ	Z	Syntax	Instruction
0	00	0	DREG Register Type = [PREG Register Type + imm16s4 Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
0	00	1	PREG Register Type = [PREG Register Type + imm16s4 Register Type]	32-Bit Pointer Load from Memory (LdM32bitToPreg)
0	01	0	DREG Register Type = w[PREG Register Type + imm16s2 Register Type] (z)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	01	1	DREG Register Type = w[PREG Register Type + imm16s2 Register Type] (x)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
0	10	0	DREG Register Type = b[PREG Register Type + imm16reloc Register Type] (z)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
0	10	1	DREG Register Type = b[PREG Register Type + imm16reloc Register Type] (x)	8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg)
1	00	0	[PREG Register Type + imm16s4 Register Type] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)
1	00	1	[PREG Register Type + imm16s4 Register Type] = PREG Register Type	Store Pointer (StPregToM32bit)
1	01	0	w[PREG Register Type + imm16s2 Register Type] = DREG Register Type	16-Bit Store to Memory (StDregLToM16bit)
1	10	0	b[PREG Register Type + imm16reloc Register Type] = DREG Register Type	8-Bit Store to Memory (StDregToM08bit)

Load/Store postmodify addressing, pregister based (LdStPmod)

LdStPmod Instruction Syntax

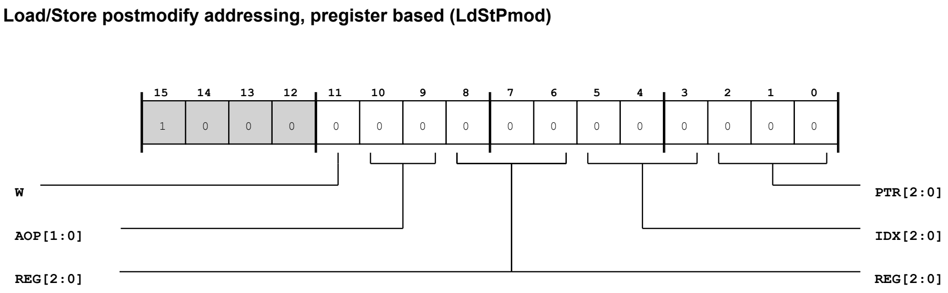


Figure 8-104: LdStPmod Instruction

The following table provides the opcode field values (W, AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	AOP	Syntax	Instruction
0	00	DREG Register Type = [PREG Register Type ++ PREG Register Type]	32-Bit Load from Memory (LdM32bitToDreg)
0	01	DREG_L Register Type = w[PREG Register Type ++ PREG Register Type]	16-Bit Load from Memory (LdM16bitToDregL)
0	10	DREG_H Register Type = w[PREG Register Type ++ PREG Register Type]	16-Bit Load from Memory (LdM16bitToDregH)
0	11	DREG Register Type = w[PREG Register Type ++ PREG Register Type] (z)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)
1	00	[PREG Register Type ++ PREG Register Type] = DREG Register Type	32-Bit Store to Memory (StDregToM32bit)
1	01	w[PREG Register Type ++ PREG Register Type] = DREG_L Register Type	16-Bit Store to Memory (StDregLToM16bit)
1	10	w[PREG Register Type ++ PREG Register Type] = DREG_H Register Type	16-Bit Store to Memory (StDregHToM16bit)
1	11	DREG Register Type = w[PREG Register Type ++ PREG Register Type] (x)	16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg)

Load/Store (Ldp)

Ldp Instruction Syntax

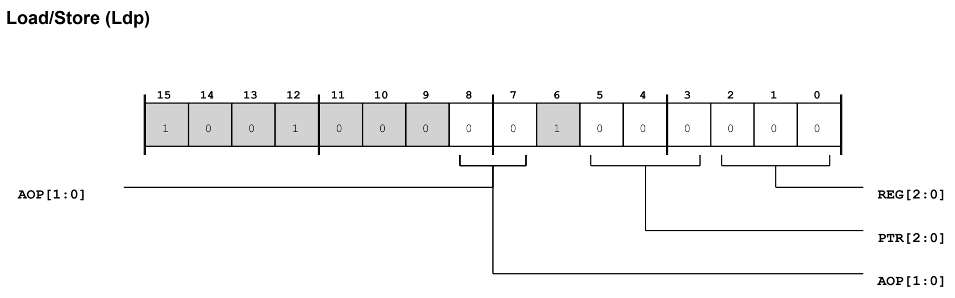


Figure 8-105: Ldp Instruction

The following table provides the opcode field values (AOP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

AOP	Syntax	Instruction
00	PREG Register Type = [PREG Register Type++]	32-Bit Pointer Load from Memory (LdM32bitToPreg)
01	PREG Register Type = [PREG Register Type--]	32-Bit Pointer Load from Memory (LdM32bitToPreg)
10	PREG Register Type = [PREG Register Type]	32-Bit Pointer Load from Memory (LdM32bitToPreg)

Load/Store indexed with small immediate offset (LdpII)

LdpII Instruction Syntax

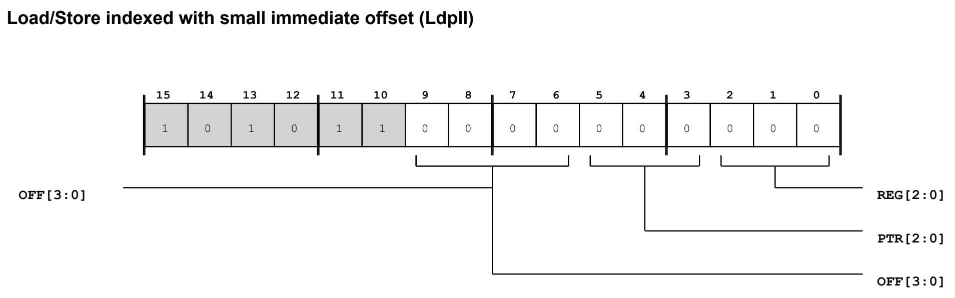


Figure 8-106: LdpII Instruction

The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

Syntax	Instruction
PREG Register Type = [PREG Register Type + uimm4s4 Register Type]	32-Bit Pointer Load from Memory (LdM32bitToPreg)

Load/Store indexed with small immediate offset FP (LdpIIFP)

LdpIIFP Instruction Syntax

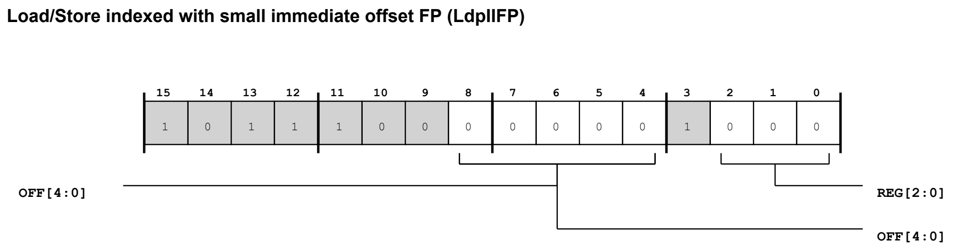


Figure 8-107: LdpIIFP Instruction

The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

Syntax	Instruction
PREG Register Type = [fp - imm5nzs4neg-pos Register Type]	32-Bit Load from Memory (LdM32bitTo-Dreg)

Save/restore registers and link/unlink frame, multiple cycles (Linkage)

Linkage Instruction Syntax

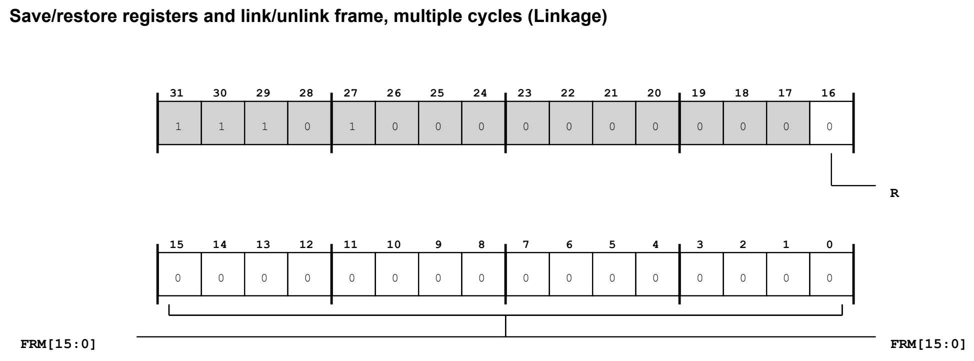


Figure 8-108: Linkage Instruction

The following table provides the opcode field values (R), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

R	Syntax	Instruction
0	link uimm16s4 Register Type	Linkage (Linkage)
1	unlink	Linkage (Linkage)

Logic Binary Operations (Logi2Op)

Logi2Op Instruction Syntax

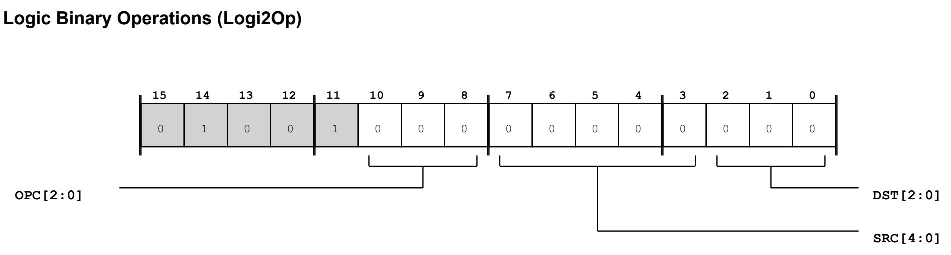


Figure 8-109: Logi2Op Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
000	cc = !bittst (DREG Register Type, uimm5 Register Type)	Bit Test (Shift_BitTst)
001	cc = bittst (DREG Register Type, uimm5 Register Type)	Bit Test (Shift_BitTst)
010	bitset (DREG Register Type, uimm5 Register Type)	Bit Modify (Shift_BitMod)
011	bittgl (DREG Register Type, uimm5 Register Type)	Bit Modify (Shift_BitMod)
100	bitclr (DREG Register Type, uimm5 Register Type)	Bit Modify (Shift_BitMod)
101	DREG Register Type >>>= uimm5 Register Type	32-Bit Arithmetic Shift (AShift32)
110	DREG Register Type >>= uimm5 Register Type	32-Bit Logical Shift (LShift)
111	DREG Register Type <<= uimm5 Register Type	32-Bit Logical Shift (LShift)

Virtually Zero Overhead Loop Mechanism (LoopSetup)

LoopSetup Instruction Syntax

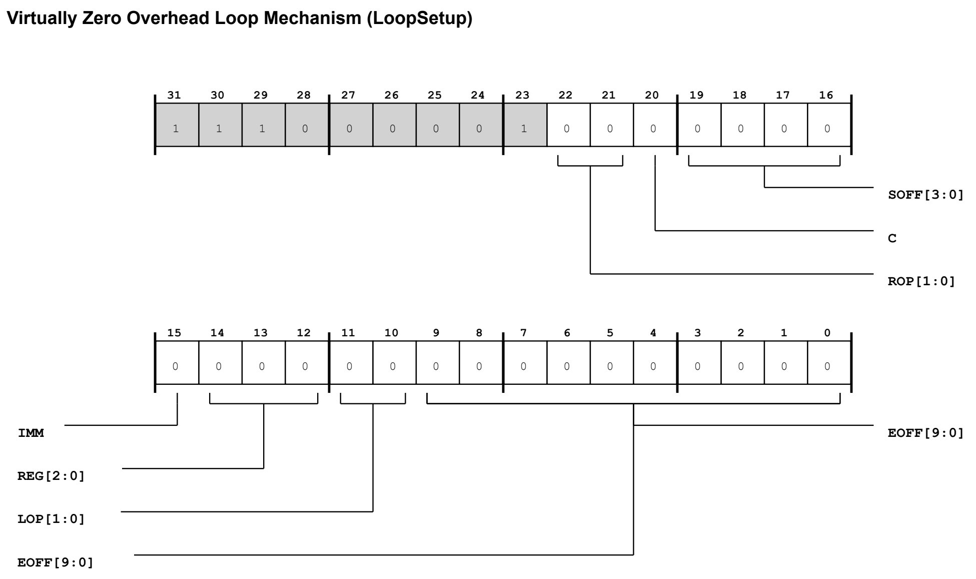


Figure 8-110: LoopSetup Instruction

The following table provides the opcode field values (LOP, ROP), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

LOP	ROP	Syntax	Instruction	Rev
00	00	lsetup (uimm4s2o4 Register Type, uimm10s2o4 Register Type) LC	Hardware Loop Set Up (LoopSetup)	
00	01	lsetup (uimm4s2o4 Register Type, uimm10s2o4 Register Type) LC = PREG Register Type	Hardware Loop Set Up (LoopSetup)	
--	10	lsetup (uimm10s2o4 Register Type) LC = uimm10 Register Type	Hardware Loop Set Up (LoopSetup)	2.0
00	11	lsetup (uimm4s2o4 Register Type, uimm10s2o4 Register Type) LC = PREG Register Type >> 1	Hardware Loop Set Up (LoopSetup)	
01	01	lsetupz (uimm10s2o4 Register Type) LC = PREG Register Type	Hardware Loop Set Up (LoopSetup)	2.0
01	11	lsetupz (uimm10s2o4 Register Type) LC = PREG Register Type >> 1	Hardware Loop Set Up (LoopSetup)	2.0
10	01	lsetuplez (uimm10s2o4 Register Type) LC = PREG Register Type	Hardware Loop Set Up (LoopSetup)	2.0
10	11	lsetuplez (uimm10s2o4 Register Type) LC = PREG Register Type >> 1	Hardware Loop Set Up (LoopSetup)	2.0

LC

LC Encode Table

C	Syntax
0	lc0
1	lc1

64-bit Instruction Shell (Multi)

Multi Instruction Syntax

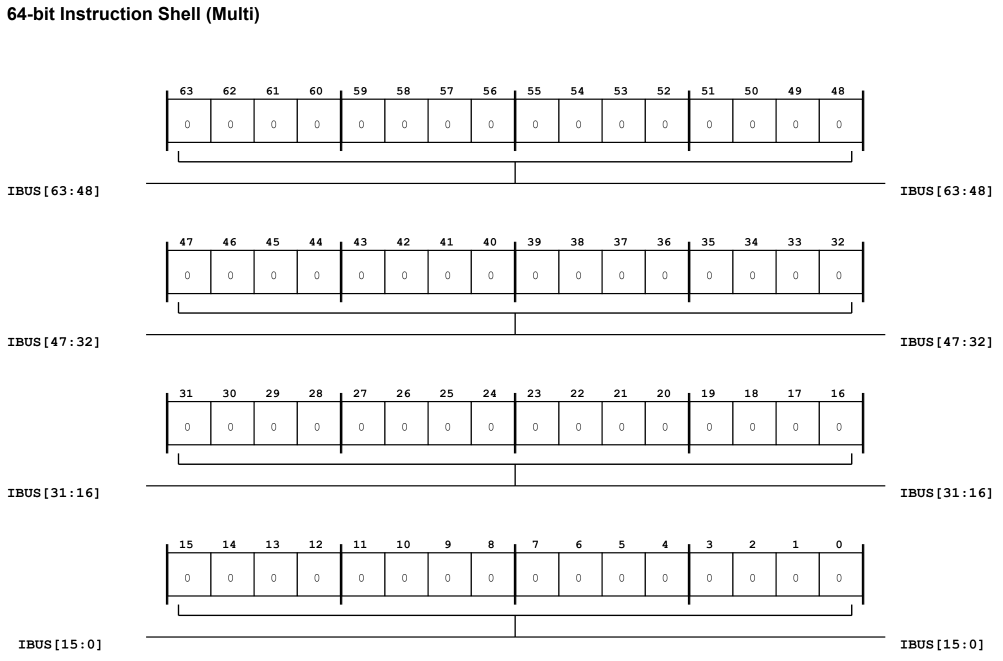


Figure 8-111: Multi Instruction

The following table provides the opcode field values (IBUS[63:59]), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

IBUS[63:59]	Syntax	Rev
0----	MAIN16A;	
10---	MAIN16B;	
110-0	MAIN32A;	
111--	MAIN32B;	
11011	MAIN64;	2.0
11001	SLOTM || SLOT0 || SLOT1;	

16-bit Slot Nop (NOP16)

NOP16 Instruction Syntax

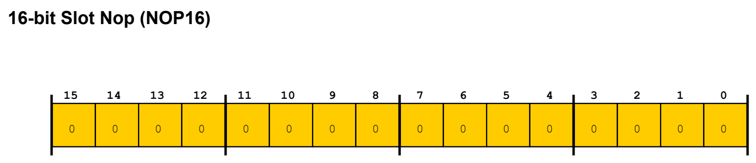


Figure 8-112: NOP16 Instruction

The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

Syntax	Instruction
nop	NOP (NOP)

32-bit Slot Nop (NOP32)

NOP32 Instruction Syntax

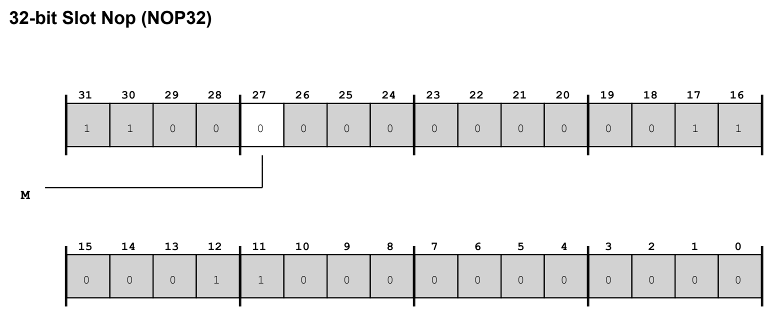


Figure 8-113: NOP32 Instruction

The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

Syntax	Instruction
mnop	32-Bit No Operation (NOP32)

Basic Program Sequencer Control Functions (ProgCtrl)

ProgCtrl Instruction Syntax

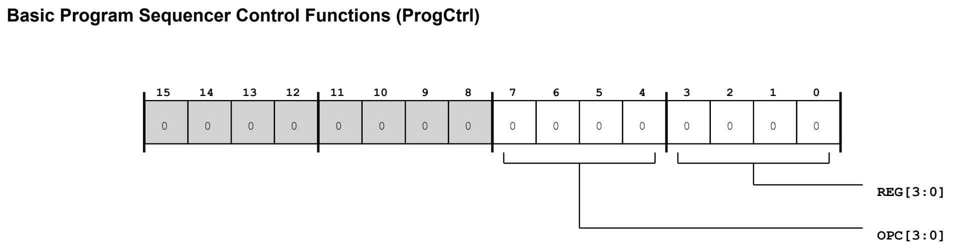


Figure 8-114: ProgCtrl Instruction

The following table provides the opcode field values (OPC, REG), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	REG	Syntax	Instruction	Rev
0001	0000	rts	Return from Branch (Return)	
0001	0001	rti	Return from Branch (Return)	
0001	0010	rtx	Return from Branch (Return)	
0001	0011	rtn	Return from Branch (Return)	
0001	0100	rte	Return from Branch (Return)	
0010	0000	idle	Sync (Sync)	
0010	0011	csync	Sync (Sync)	
0010	0100	ssync	Sync (Sync)	
0010	0101	emuexcpt	Sequencer Mode (Mode)	
0011	0---	cli DREG Register Type	Interrupt Control (IMaskMv)	
0100	0---	sti DREG Register Type	Interrupt Control (IMaskMv)	
0101	0---	jump (PREG Register Type)	Jump (Jump)	
0110	0---	call (PREG Register Type)	Call (Call)	
0111	0---	call (pc+PREG Register Type)	Call (Call)	
1000	0---	jump (pc+PREG Register Type)	Jump (Jump)	
1001	----	raise uimm4 Register Type	Raise Interrupt (Raise)	
1010	----	excpt uimm4 Register Type	Raise Interrupt (Raise)	
1011	0---	testset (PREGP Register Type)	TestSet (TestSet)	
1100	0---	sti idle DREG Register Type	Sync (Sync)	2.1.1

Pointer Arithmetic Operations (Ptr2op)

Ptr2op Instruction Syntax

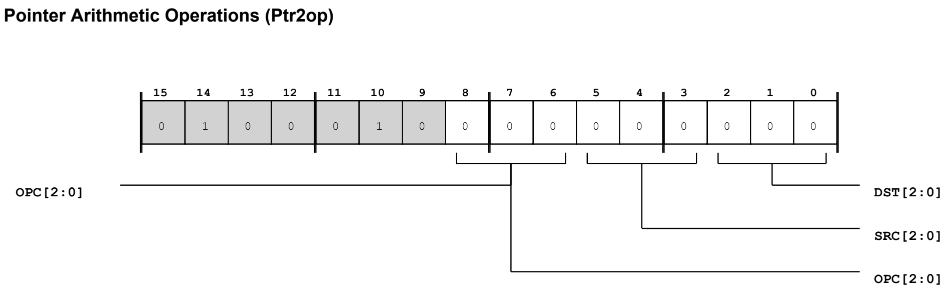


Figure 8-115: Ptr2op Instruction

The following table provides the opcode field values (OPC), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

OPC	Syntax	Instruction
000	PREG Register Type -= PREG Register Type	32-bit Add or Subtract (DagAdd32)
001	PREG Register Type = PREG Register Type << 2	Pointer Logical Shift (LShiftPtr)
010	PREG Register Type = PREG Register Type << 1	Pointer Logical Shift (LShiftPtr)
011	PREG Register Type = PREG Register Type >> 2	Pointer Logical Shift (LShiftPtr)
100	PREG Register Type = PREG Register Type >> 1	Pointer Logical Shift (LShiftPtr)
101	PREG Register Type += PREG Register Type (brev)	32-bit Add or Subtract (DagAdd32)
110	PREG Register Type = (PREG Register Type + PREG Register Type) << 1	32-bit Add then Shift (DagAddSubShift)
111	PREG Register Type = (PREG Register Type + PREG Register Type) << 2	32-bit Add then Shift (DagAddSubShift)

Push or Pop Multiple contiguous registers (PushPopMult)

PushPopMult Instruction Syntax

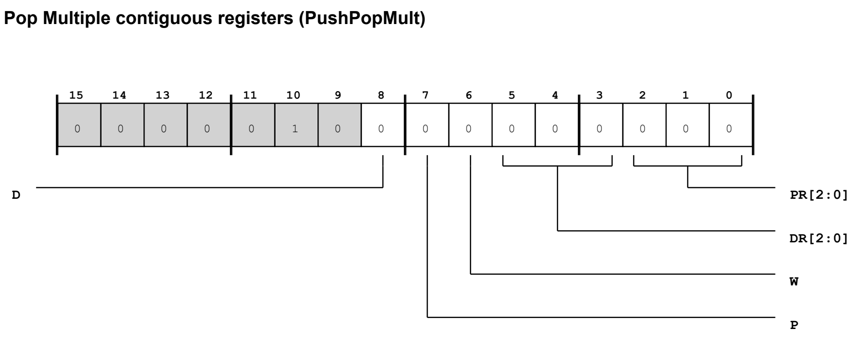


Figure 8-116: PushPopMult Instruction

The following table provides the opcode field values (W, D, P), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	D	P	Syntax	Instruction
0	0	1	(PREG_RANGE Register Type) = [sp++]	Stack Push/Pop Multiple Registers (PushPopMul16)
0	1	0	(DREG_RANGE Register Type) = [sp++]	Stack Push/Pop Multiple Registers (PushPopMul16)
0	1	1	(DREG_RANGE Register Type, PREG_RANGE Register Type) = [sp++]	Stack Push/Pop Multiple Registers (PushPopMul16)
1	0	1	[--sp] = (PREG_RANGE Register Type)	Stack Push/Pop Multiple Registers (PushPopMul16)
1	1	0	[--sp] = (DREG_RANGE Register Type)	Stack Push/Pop Multiple Registers (PushPopMul16)
1	1	1	[--sp] = (DREG_RANGE Register Type, PREG_RANGE Register Type)	Stack Push/Pop Multiple Registers (PushPopMul16)

Push or Pop register, to and from the stack pointed to by sp (PushPopReg)

PushPopReg Instruction Syntax

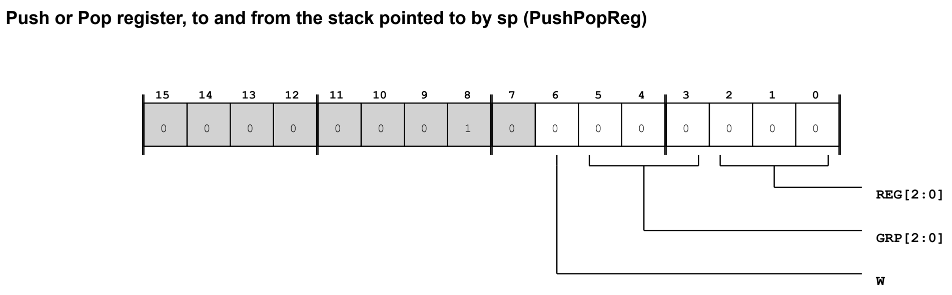


Figure 8-117: PushPopReg Instruction

The following table provides the opcode field values (W), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

W	Syntax	Instruction
0	POPREG = [sp++]	Stack Pop (Pop)
1	[--sp] = PUSHREG	Stack Push (Push)

POPREG

POPREG Encode Table

GRP	REG	Syntax
010	0--	IREG Register Type
010	1--	MREG Register Type
011	0--	BREG Register Type
011	1--	LREG Register Type
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat
100	111	rets
110	---	SYSREG2 Register Type
111	---	SYSREG3 Register Type

PUSHREG

PUSHREG Encode Table

GRP	REG	Syntax
000	---	DREG Register Type
001	---	PREG Register Type
010	0--	IREG Register Type
010	1--	MREG Register Type
011	0--	BREG Register Type
011	1--	LREG Register Type
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat
100	111	rets
110	---	SYSREG2 Register Type
111	---	SYSREG3 Register Type

Register to register transfer operation (RegMv)

RegMv Instruction Syntax

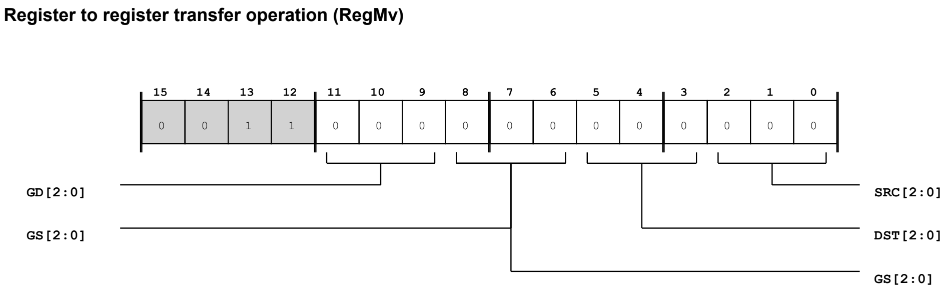


Figure 8-118: RegMv Instruction

The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

Syntax	Instruction
GDST = GSRC	Move Register to Register (MvRegToReg)

GDST

GDST Encode Table

GD	DST	Syntax
000	---	DREG Register Type
001	---	PREG Register Type
010	0--	IREG Register Type
010	1--	MREG Register Type
011	0--	BREG Register Type
011	1--	LREG Register Type
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat
100	111	rets
110	---	SYSREG2 Register Type
111	---	SYSREG3 Register Type

GSRC

GSRC Encode Table

GS	SRC	Syntax
000	---	DREG Register Type
001	---	PREG Register Type
010	0--	IREG Register Type
010	1--	MREG Register Type
011	0--	BREG Register Type
011	1--	LREG Register Type
100	000	a0.x
100	001	a0.w
100	010	a1.x
100	011	a1.w
100	110	astat

GS	SRC	Syntax
100	111	rets
110	---	SYSREG2 Register Type
111	---	SYSREG3 Register Type

Unconditional Branch PC relative with 12bit offset (UJump)

UJump Instruction Syntax

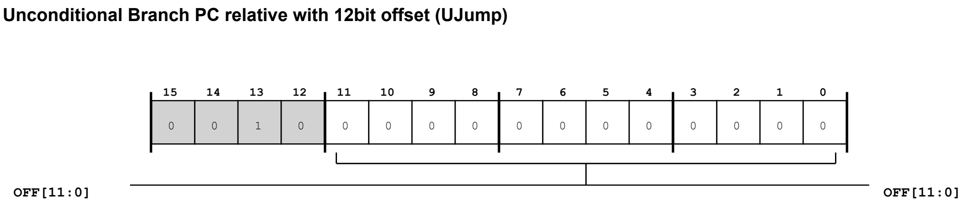


Figure 8-119: UJump Instruction

The following table provides the opcode field values (), the instruction syntax overview (Syntax), and a link to the corresponding instruction reference page (Instruction)

Syntax	Instruction
jump.s imm12nxs2 Register Type	Jump Immediate (JumpAbs)

bimm32 Register Type

bimm32 Attributes

range	allow_label
-0x80000000:0x7fffffff	true

buimm32 Register Type

buimm32 Attributes

range	allow_label
0x0:0xffffffff	true

huimm16 Register Type

huimm16 Attributes

range
0x0:0xffff

imm10s2 Register Type

imm10s2 Attributes

range	allow_label
-0x400:0x3fe:2	true

imm12nxs2 Register Type

imm12nxs2 Attributes

range	allow_label
-0x1000:0xffe:2	true

imm12s2 Register Type

imm12s2 Attributes

range	allow_label
-0x1000:0xffe:2	true

imm12xs2 Register Type

imm12xs2 Attributes

range	allow_label
-0x1000:0xffe:2	true

imm16 Register Type

imm16 Attributes

range
-0x8000:0x7fff

imm16negpos Register Type

imm16negpos Attributes

range	negated
0x1:0x8000	true

imm16reloc Register Type

imm16reloc Attributes

range
-0x8000:0x7fff

imm16s2 Register Type

imm16s2 Attributes

range
-0x10000:0xfffe:2

imm16s2negpos Register Type

imm16s2negpos Attributes

range	negated
0x2:0x10000:2	true

imm16s4 Register Type

imm16s4 Attributes

range
-0x20000:0x1fffc:4

imm16s4negpos Register Type

imm16s4negpos Attributes

range	negated
0x4:0x20000:4	true

imm24nxs2 Register Type

imm24nxs2 Attributes

range
-0x1000000:0xfffffe:2

imm24s2 Register Type

imm24s2 Attributes

range
-0x1000000:0xfffffe:2

imm24xs2 Register Type

imm24xs2 Attributes

range
-0x1000000:0xfffffe:2

imm3 Register Type

imm3 Attributes

range
-0x4:0x3

imm32 Register Type

imm32 Attributes

range
-0x80000000:0x7fffffff

imm5nzs4negpos Register Type

imm5nzs4negpos Attributes

range	negated
0x4:0x80:4	true

imm6 Register Type

imm6 Attributes

range
-0x20:0x1f

imm7 Register Type

imm7 Attributes

range
-0x40:0x3f

luimm16 Register Type

luimm16 Attributes

range
0x0:0xffff

negimm5s4 Register Type

negimm5s4 Attributes

range
-0x80:-0x4:4

rimm16 Register Type

rimm16 Attributes

range
0x0:0xffff

uimm10 Register Type

uimm10 Attributes

range	iencode
0x1:0x3ff,0xffffffff	0xffffffff:0

uimm10s2o4 Register Type

uimm10s2o4 Attributes

range	allow_label
0x4:0x7fe:2	true

uimm16s4 Register Type

uimm16s4 Attributes

range
0x0:0x3fffc:4

uimm3 Register Type

uimm3 Attributes

range
0x0:0x7

uimm32 Register Type

uimm32 Attributes

range	allow_label
0x0:0xffffffff	true

uimm4 Register Type

uimm4 Attributes

range
0x0:0xf

uimm4nz Register Type

uimm4nz Attributes

range
0x1:0xf

uimm4nznegpos Register Type

uimm4nznegpos Attributes

range	negated
0x1:0xf	true

uimm4s2 Register Type

uimm4s2 Attributes

range
0x0:0x1e:2

uimm4s2o4 Register Type

uimm4s2o4 Attributes

range	allow_label
0x4:0x1e:2	true

uimm4s4 Register Type

uimm4s4 Attributes

range
0x0:0x3c:4

uimm5 Register Type

uimm5 Attributes

range
0x0:0x1f

uimm5nz Register Type

uimm5nz Attributes

range
0x1:0x1f

uimm5nznegpos Register Type

uimm5nznegpos Attributes

range	negated
0x1:0x1f	true

BREG Register Type

BREG Syntax

Code	Syntax
00	b0
01	b1
10	b2
11	b3

BREG_H Register Type

BREG_H Syntax

Code	Syntax
00	b0.h
01	b1.h
10	b2.h
11	b3.h

BREG_L Register Type

BREG_L Syntax

Code	Syntax
00	b0.l
01	b1.l
10	b2.l
11	b3.l

DREG Register Type

DREG Syntax

Code	Syntax
000	r0
001	r1
010	r2
011	r3
100	r4
101	r5
110	r6
111	r7

DREG_B Register Type

DREG_B Syntax

Code	Syntax
000	r0.b
001	r1.b
010	r2.b
011	r3.b
100	r4.b
101	r5.b
110	r6.b
111	r7.b

DREG_E Register Type

DREG_E Syntax

Code	Syntax
000	r0
010	r2
100	r4
110	r6

DREG_H Register Type

DREG_H Syntax

Code	Syntax
000	r0.h
001	r1.h
010	r2.h
011	r3.h
100	r4.h
101	r5.h
110	r6.h
111	r7.h

DREG_L Register Type

DREG_L Syntax

Code	Syntax
000	r0.l
001	r1.l
010	r2.l
011	r3.l
100	r4.l
101	r5.l
110	r6.l
111	r7.l

DREG_O Register Type

DREG_O Syntax

Code	Syntax
000	r1
010	r3
100	r5
110	r7

DREG_PAIR Register Type

DREG_PAIR Syntax

Code	Syntax
000	r1:0
010	r3:2
100	r5:4
110	r7:6

DREG_RANGE Register Type

DREG_RANGE Syntax

Code	Syntax
000	r7:0
001	r7:1
010	r7:2
011	r7:3
100	r7:4
101	r7:5
110	r7:6
111	r7:7

IREG Register Type

IREG Syntax

Code	Syntax
00	i0
01	i1
10	i2
11	i3

IREG_H Register Type

IREG_H Syntax

Code	Syntax
00	i0.h
01	i1.h
10	i2.h
11	i3.h

IREG_L Register Type

IREG_L Syntax

Code	Syntax
00	i0.l
01	i1.l
10	i2.l
11	i3.l

LREG Register Type

LREG Syntax

Code	Syntax
00	l0
01	l1
10	l2

Code	Syntax
11	l3

LREG_H Register Type

LREG_H Syntax

Code	Syntax
00	l0.h
01	l1.h
10	l2.h
11	l3.h

LREG_L Register Type

LREG_L Syntax

Code	Syntax
00	l0.l
01	l1.l
10	l2.l
11	l3.l

MREG Register Type

MREG Syntax

Code	Syntax
00	m0
01	m1
10	m2
11	m3

MREG_H Register Type

MREG_H Syntax

Code	Syntax
00	m0.h
01	m1.h
10	m2.h
11	m3.h

MREG_L Register Type

MREG_L Syntax

Code	Syntax
00	m0.l
01	m1.l
10	m2.l
11	m3.l

PREG Register Type

PREG Syntax

Code	Syntax
000	p0
001	p1
010	p2
011	p3
100	p4
101	p5
110	sp
111	fp

PREGP Register Type

PREGP Syntax

Code	Syntax
000	p0
001	p1
010	p2
011	p3
100	p4
101	p5

PREG_H Register Type

PREG_H Syntax

Code	Syntax
000	p0.h
001	p1.h
010	p2.h
011	p3.h
100	p4.h
101	p5.h
110	sp.h
111	fp.h

PREG_L Register Type

PREG_L Syntax

Code	Syntax
000	p0.l
001	p1.l
010	p2.l
011	p3.l
100	p4.l
101	p5.l

Code	Syntax
110	sp.l
111	fp.l

PREG_RANGE Register Type

PREG_RANGE Syntax

Code	Syntax
000	p5:0
001	p5:1
010	p5:2
011	p5:3
100	p5:4
101	p5:5

SYSREG2 Register Type

SYSREG2 Syntax

Code	Syntax
000	lc0
001	lt0
010	lb0
011	lc1
100	lt1
101	lb1
110	cycles
111	cycles2

SYSREG3 Register Type

SYSREG3 Syntax

Code	Syntax
000	usp

Code	Syntax
001	seqstat
010	syscfg
011	reti
100	retx
101	retn
110	rete
111	emudat

Issuing Parallel Instructions

This chapter discusses the instructions that can be issued in parallel. It identifies supported combinations for parallel issue, parallel issue syntax, 32-bit ALU/MAC instructions, 16-bit instructions, and examples.

The Blackfin processor is not superscalar; it does not execute multiple instructions at once. However, it does permit up to three instructions to be issued in parallel with some limitations. A multi-issue instruction is 64-bits in length and consists of one 32-bit instruction and two 16-bit instructions. All three instructions execute in the same amount of time as the slowest of the three.

Sections in this chapter

- Supported Parallel Combinations
- Parallel Issue Syntax
- 32-Bit ALU/MAC Instructions
- 16-Bit Instructions
- Parallel Operation Examples

Supported Parallel Combinations

The diagram in Supported Parallel Combinations illustrates the combinations for parallel issue that the Blackfin processor supports.

32-bit ALU/MAC instruction	16-bit Instruction	16-bit Instruction

Parallel Issue Syntax

The syntax of a parallel issue instruction is as follows.

- `A 32-bit ALU/MAC instruction || A 16-bit instruction || A 16-bit instruction ;`

  The vertical bar (||) indicates the following instruction is to be issued in parallel with the previous instruction. Note the terminating semicolon appears only at the end of the parallel issue instruction.

It is possible to issue a 32-bit ALU/MAC instruction in parallel with only one 16-bit instruction using the following syntax. The result is still a 64-bit instruction with a 16-bit NOP automatically inserted into the unused 16-bit slot.

- `A 32-bit ALU/MAC instruction || A 16-bit instruction ;`

Alternately, it is also possible to issue two 16-bit instructions in parallel with one another without an active 32-bit ALU/MAC instruction by using the MNOP instruction, shown below. Again, the result is still a 64-bit instruction.

- `MNOP || A 16-bit instruction || A 16-bit instruction ;`

See the MNOP (32-bit NOP) instruction description in NOP (NOP). The MNOP instruction does not have to be explicitly included by the programmer; the software tools prepend it automatically. The MNOP instruction will appear in disassembled parallel 16-bit instructions.

32-Bit ALU/MAC Instructions

The list of 32-bit instructions that can be in a parallel instruction are shown in the ***32-Bit DSP Instructions*** table.

Table 8-33: 32-Bit DSP Instructions

Instruction Name (Description)	Operation Type and Parallel Version Notes
Arithmetic Operations	
32-bit Absolute Value (Abs32) (Absolute Value)	ALU Operations (Dsp32Alu)
32-bit Add or Subtract (AddSub32) (Add or Subtract)	ALU Operations (Dsp32Alu) Note: Only permits parallelism for versions supporting saturation.
32-Bit Prescale Up Add/Sub to 16-bit (AddSubRnd12) (Add/Subtract – Prescale Up)	ALU Operations (Dsp32Alu)
32-Bit Prescale Down Add/Sub to 16-Bit (AddSubRnd20) (Add/Subtract – Prescale Down)	ALU Operations (Dsp32Alu)
Exponent Detection (Shift_ExpAdj32) (Exponent Detection)	Shift (Dsp32Shf)
32-bit Maximum (Max32) (Maximum)	ALU Operations (Dsp32Alu)
32-Bit Minimum (Min32) (Minimum)	ALU Operations (Dsp32Alu)
Accumulator Add or Subtract (AddSubAcc) (Modify Increment/Decrement)	ALU Operations (Dsp32Alu)
32-Bit Negate (Neg32) (Negate, Two's-Complement)	ALU Operations (Dsp32Alu) Note: Only permits parallelism for the accumulator versions.
Fractional 32-bit to 16-Bit Conversion (Pass32Rnd16) (Round to Half-Word)	ALU Operations (Dsp32Alu)
Accumulator0 32-Bit Saturate (ALU_SatAcc0) (Saturate A0)	ALU Operations (Dsp32Alu)
Accumulator1 32-Bit Saturate (ALU_SatAcc1) (Saturate A1)	ALU Operations (Dsp32Alu)

Table 8-33: 32-Bit DSP Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes
Dual Accumulator 32-Bit Saturate (ALU_SatAccDual) (Dual Saturate)	ALU Operations (Dsp32Alu)
Redundant Sign Bits (Shift_SignBits32)	Shift (Dsp32Shf)
Load Store	
Accumulator Register Initialization (LdImmToAx) (Clear A0)	ALU Operations (Dsp32Alu)
Accumulator Register Initialization (LdImmToAx) (Clear A1)	ALU Operations (Dsp32Alu)
Dual Accumulator 0 and 1 Registers Initialization (LdImmToAxDual) (Dual Clear)	ALU Operations (Dsp32Alu)
Bit Operations	
Deposit Bits (Shift_Deposit) (Bit Field Deposit)	Shift (Dsp32Shf)
Extract Bits (Shift_Extract) (Bit Field Extract)	Shift (Dsp32Shf)
Bit Mux (BitMux) (Bit Multiplex)	Shift (Dsp32Shf)
Ones Count (Shift_Ones) (Ones Count)	Shift (Dsp32Shf)
Logical Operations	
40-Bit BXORShift LSFR with Feedback to the Accumulator (BXORShift_NF) ((Bitwise XOR with Feedback)	Shift (Dsp32Shf)
32-Bit BXOR or BXORShift LSFR without Feedback (BXOR_NF) (Bitwise XOR without Feedback)	Shift (Dsp32Shf)
Move	
Move Register to Accumulator1 (MvDregToAx) (Move Register to A0)	ALU Operations (Dsp32Alu)
Move Register to Accumulator1 (MvDregToAx) (Move Register to A1)	ALU Operations (Dsp32Alu)
Move 32-Bit Accumulator Section to Even Register (MvA0ToDregE) (Move A0.X to Register Half)	ALU Operations (Dsp32Alu)
Move 32-Bit Accumulator Section to Odd Register (MvA1ToDregO) (Move A1.X to Register Half)	ALU Operations (Dsp32Alu)
Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL) (Move Register Half to A0)	ALU Operations (Dsp32Alu)
Move Register Half to 16-Bit Accumulator Section (MvDregHLToAxHL) (Move Register Half to A1)	ALU Operations (Dsp32Alu)
Shift / Rotate Operations[*1]	
Accumulator Arithmetic Shift (AShiftAcc) (Arithmetic Shift A0/A1)	Shift (Dsp32Shf) Note: See footnote restrictions.
32-Bit Arithmetic Shift (AShift32) (Arithmetic Shift Register)	Shift (Dsp32Shf)

Table 8-33: 32-Bit DSP Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes
	Note: Only permits parallelism for saturating versions; see footnote restrictions.
16-Bit Logical Shift (LShift16) (Logical Shift Half Register by Half Register or Immediate)	Shift (Dsp32Shf) and Shift (Dsp32Shf) Note See footnote restrictions.
32-Bit Rotate (Shift_Rot32) (Rotate Register)	Shift (Dsp32Shf) Note: See footnote restrictions.
Accumulator Rotate (Shift_RotAcc) (Rotate A0/A1)	Shift (Dsp32Shf) Note: See footnote restrictions.
External Event Management	
32-Bit No Operation (NOP32) (No Operation)	32-Bit No Operation (NOP32) Note: Only permits parallelism for 32-bit MNOP.
Vector Operations	
16-Bit Modulo Maximum with History (Shift_VitMax) (Modulo Maximum with History)	Shift (Dsp32Shf)
16-Bit Add on Sign (AddOnSign) (Add on Sign)	ALU Operations (Dsp32Alu)
16 x 16-Bit MAC (Mac16) (Multiply to Accumulator)	Multiply with 3 operands (Dsp32Mult)
16 x 16-Bit MAC (Mac16) (Multiply-Accumulate to Accumulator)	Multiply with 3 operands (Dsp32Mult)
16 x 16-Bit Multiply (Mult16) (Multiply to Half Register)	Multiply with 3 operands (Dsp32Mult)
16 x 16-Bit MAC (Mac16) (Multiply-Accumulate to Half Register)	Multiply Accumulate (Dsp32Mac)
32 x 32-bit Multiply (Mult32) (Multiply to Register)	Multiply with 3 operands (Dsp32Mult)
16 x 16-Bit MAC (Mac16) (Multiply-Accumulate to Register)	Multiply with 3 operands (Dsp32Mult)
Vectored 16-Bit Absolute Value (Abs2x16) (Absolute Value, Vector)	ALU Operations (Dsp32Alu)
Vectored 16-Bit Add or Subtract (AddSubVec16) (Add or Subtract, Vector)	ALU Operations (Dsp32Alu)
Vectored 16-Bit Arithmetic (AShift16Vec) (Arithmetic Shift Register, Vector)	Shift (Dsp32Shf)
Vectored 16-Bit Logical Shift (LShift16Vec) (Logical Shift Register by Half Register or Immediate, Vector)	Shift (Dsp32Shf)
Vectored 16-Bit Maximum (Max16Vec) (Maximum, Vector)	ALU Operations (Dsp32Alu)
Vectored 16-Bit Minimum (Min16Vec) (Minimum, Vector)	ALU Operations (Dsp32Alu)
16 x 16-Bit Multiply (Mult16) (Multiply 16-Bit Operands)	Multiply with 3 operands (Dsp32Mult)
Vectored 16-bit Negate (Neg16Vec) (Negate, Two's-Complement, Vector)	ALU Operations (Dsp32Alu)
Pack 16-Bit to 32-Bit (Pack16Vec) (Pack, Vector)	Shift (Dsp32Shf)

Table 8-33: 32-Bit DSP Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes
Vectored 16-Bit Search (Search) (Search, Vector)	ALU Operations (Dsp32Alu)
Video Pixel Operations	
Byte Align (Shift_Align) (Byte Align 8, 16, and 24)	Shift (Dsp32Shf)
Disable Alignment Exception (DisAlignExcept) (Disable Alignment Exception for Load)	ALU Operations (Dsp32Alu)
Vectored 8-Bit Sum of Absolute Differences (SAD8Vec) (Sum of Absolute Differences, Vector)	ALU Operations (Dsp32Alu)
Dual Accumulator Extraction with Addition (AddAccHalf) (Dual Half Register Add to A0/A1)	ALU Operations (Dsp32Alu)
Vectored 8-Bit Add or Subtract to 16-Bit (Byteop16P/M) (AddSub4x8) (Quad 8-Bit Add/Subtract)	ALU Operations (Dsp32Alu)
Vector Byte Average (Byteop1P) (Avg8Vec) (Quad 8-Bit Average - Byte)	ALU Operations (Dsp32Alu)
Quad Byte Average (Byteop2P) (Avg4x8Vec) (Quad 8-Bit Average - Half Word)	ALU Operations (Dsp32Alu)
Vectored 8-Bit to 16-Bit Add then Clip to 8-Bit (Byteop3P) (AddClip) (Dual 16-Bit Add/Clip)	ALU Operations (Dsp32Alu)
Pack 8-Bit to 32-Bit (BytePack) (Quad 8-Bit Pack)	ALU Operations (Dsp32Alu)
Spread 8-Bit to 16-Bit (ByteUnPack) (Quad 8-Bit Unpack)	ALU Operations (Dsp32Alu)

*1 Multi-issue may not combine SHIFT/ROTATE with STORE using Preg + Offset operation.

16-Bit Instructions

The two 16-bit instructions in a multi-issue instruction must each be from the instructions shown in the ***Compatible 16-Bit Instructions*** table.

The following additional restrictions also apply to the 16-bit instructions of the multi-issue instruction.

- Only one of the 16-bit instructions can be a store instruction.
- Only one of the 16-bit instructions may load a pointer register. This load must be encoded in DAG slot 0.

Table 8-34: Compatible 16-Bit Instructions

Instruction Name (Description)	Operation Type and Parallel Version Notes
Arithmetic Operations	
32-bit Add or Subtract Constant (DagAddImm) (DAG Add/Subtract Immediate)	Destructive Binary Operations, preg with 7bit immediate (CompI2opP) Note: I-Register versions only.

Table 8-34: Compatible 16-Bit Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes
32-bit Add or Subtract (DagAdd32) (DAG Modify Increment/Decrement)	Pointer Arithmetic Operations (Ptr2op) Note: I-Register versions only.
Load / Store	
32-Bit Register Initialization (LdImmToReg) (Load Pointer Register)	Load/Store (Ldp), Load/Store indexed with small immediate offset (LdpII), and Long Load/Store with indexed addressing (LdStIdxI)
32-Bit Load from Memory (LdM32bitToDreg) (Load Data Register)	Load/Store postmodify addressing, pregister based (LdStPmod), Load/Store (DspLdSt), Load/Store (LdSt), Load/Store indexed with small immediate offset FP (LdStIIFP), Load/Store indexed with small immediate offset FP (LdpIIFP), Load/Store indexed with small immediate offset (LdStII), Long Load/Store with indexed addressing (LdStIdxI), and Load/Store 32-bit Absolute Address (LdStAbs)
16-Bit Load from Memory to 32-Bit Register (LdM16bitToDreg) (Load Half Word Sign/Zero Extended)	Load/Store postmodify addressing, pregister based (LdStPmod), Load/Store (LdSt), Load/Store indexed with small immediate offset (LdStII), Long Load/Store with indexed addressing (LdStIdxI), and Load/Store 32-bit Absolute Address (LdStAbs)
16-Bit Load from Memory (LdM16bitToDregH) (Load High Half Register)	Load/Store postmodify addressing, pregister based (LdStPmod), Load/Store (DspLdSt), and Load/Store 32-bit Absolute Address (LdStAbs)
16-Bit Load from Memory (LdM16bitToDregL) (Load Low Half Register)	Load/Store postmodify addressing, pregister based (LdStPmod), Load/Store (DspLdSt), and Load/Store 32-bit Absolute Address (LdStAbs)
8-Bit Load from Memory to 32-bit Register (LdM08bitToDreg) (Load Byte Sign/Zero Extended)	Load/Store (LdSt), Long Load/Store with indexed addressing (LdStIdxI), and Load/Store 32-bit Absolute Address (LdStAbs)
Store Pointer (StPregToM32bit) (Store Pointer Register)	Load/Store (LdSt)
32-Bit Store to Memory (StDregToM32bit) (Store Data Register)	Load/Store postmodify addressing, pregister based (LdStPmod), Load/Store (DspLdSt), Load/Store (LdSt), Load/Store indexed with small immediate offset FP (LdStIIFP), Load/Store indexed with small immediate offset (LdStII), Long Load/Store with indexed addressing (LdStIdxI), and Load/Store 32-bit Absolute Address (LdStAbs)
16-Bit Store to Memory (StDregHToM16bit) (Store High Half Data Register)	Load/Store postmodify addressing, pregister based (LdStPmod), Load/Store (DspLdSt), and Load/Store 32-bit Absolute Address (LdStAbs)

Table 8-34: Compatible 16-Bit Instructions (Continued)

Instruction Name (Description)	Operation Type and Parallel Version Notes
16-Bit Store to Memory (StDregLToM16bit) (Store Low Half Data Register)	Load/Store postmodify addressing, pregister based (LdStPmod), Load/Store (DspLdSt), Load/Store (LdSt), Load/Store indexed with small immediate offset (LdStII), Long Load/Store with indexed addressing (LdStIdxI), and Load/Store 32-bit Absolute Address (LdStAbs)
8-Bit Store to Memory (StDregToM08bit) (Store Byte)	Load/Store (LdSt), Long Load/Store with indexed addressing (LdStIdxI), and Load/Store 32-bit Absolute Address (LdStAbs)
External Event Management	
NOP (NOP) (No Operation)	16-bit Slot Nop (NOP16) Note: 16-bit NOP only.

Parallel Operation Examples

• Two Parallel Memory Access Instructions

```
/* Subtract-Absolute-Accumulate issued in parallel with the memory access instructions that
fetch the data for the next SAA instruction. This sequence is executed in a loop to flip-
flop back and forth between the data in R1 and R3, then the data in R0 and R2. */
saa (r1:0, r3:2) || r0=[i0++] || r2=[i1++] ;
saa (r1:0, r3:2)(r) || r1=[i0++] || r3=[i1++] ;
mnop || r1 = [i0++] || r3 = [i1++] ;
```

• One `Ireg` and One Memory Access Instruction in Parallel

```
/* Add on Sign while incrementing an Ireg and loading a data register based on the previous
value of the Ireg. */
r7.h=r7.l=sign(r2.h)*r3.h + sign(r2.l)*r3.l || i0+=m3 || r0=[i0] ;
/* Add/subtract two vector values while incrementing an Ireg and loading a data register. */
R2 = R2 +|+ R4, R4 = R2 -|- R4 (ASR) || I0 += M0 (BREV) || R1 = [I0] ;
/* Multiply and accumulate to Accumulator while loading a data register and storing a data
register using an Ireg pointer. */
A1=R2.L*R1.L, A0=R2.H*R1.H || R2.H=W[I2++] || [I3++]=R3 ;
/* Multiply and accumulate while loading two data registers. One load uses an Ireg pointer.
*/
A1+=R0.L*R2.H,A0+=R0.L*R2.L || R2.L=W[I2++] || R0=[I1--] ;
R3.H=(A1+=R0.L*R1.H), R3.L=(A0+=R0.L*R1.L) || R0=[P0++] || R1=[I0] ;
/* Pack two vector values while storing a data register using an Ireg pointer and loading
another data register. */
R1=PACK(R1.H,R0.H) || [I0++]=R0 || R2.L=W[I2++] ;
```

• One `Ireg` Instruction in Parallel

```
/* Multiply-Accumulate to a Data register while incrementing an Ireg. */
r6=(a0+=r3.h*r2.h)(fu) || i2-=m0 ;
```

```
/* which the assembler expands into:
   r6=(a0+=r3.h*r2.h)(fu) || i2-=m0 || nop ; */
```

9 Debug

The Blackfin+ processor's debug functionality is used for software debugging. It also complements some services often found in an operating system (OS) kernel. The functionality is implemented in the processor hardware and is grouped into multiple levels.

A summary of available debug features is shown in the ***Blackfin+ Debug Features*** table.

Table 9-1: Blackfin+ Debug Features

Debug Feature	**Description**
Watchpoints	Specify address ranges and conditions that halt the processor when satisfied.
Cycle Count	Provides functionality for all code profiling functions.
Performance Monitoring	Allows internal resources to be monitored and measured non-intrusively.

Watchpoint Unit

By monitoring the addresses on both the instruction bus and the data bus, the Watchpoint Unit provides several mechanisms for examining program behavior. After counting the number of times a particular address is matched, the unit schedules an event based on this count.

In addition, information that the Watchpoint Unit provides helps in the optimization of code. The unit also makes it easier to maintain executables through code patching.

The Watchpoint Unit contains these memory-mapped registers (MMRs), which are accessible in Supervisor and Emulator modes:

- Watchpoint Status register (`WPSTAT`)
- Six Instruction Watchpoint Address registers (`WPIA[5:0]`)
- Six Instruction Watchpoint Address Count registers (`WPIACNT[5:0]`)
- Instruction Watchpoint Address Control register (`WPIACTL`)
- Two Data Watchpoint Address registers (`WPDA[1:0]`)

- Two Data Watchpoint Address Count registers (WPDACNT[1:0])
- Data Watchpoint Address Control register (WPDACTL)

Two operations implement instruction watchpoints:

- The values in the six Instruction Watchpoint Address registers, WPIA[5:0], are compared to the address on the instruction bus.
- Corresponding count values in the Instruction Watchpoint Address Count registers, WPIACNT[5:0], are decremented each time a match occurs.

The six Instruction Watchpoint Address registers may be further grouped into three ranges of instruction-address-range watchpoints, as defined in the WPIA0/WPIA1, WPIA2/WPIA3, and WPIA4/WPIA5 register pairs:

- WPIA0 <= WPIA1
- WPIA2 <= WPIA3
- WPIA4 <= WPIA5

Two operations implement data watchpoints:

- The values in the two Data Watchpoint Address registers, WPDA[1:0], are compared to the addresses on the data buses.
- Corresponding count values in the Data Watchpoint Address Count registers, WPDACNT[1:0], are decremented each time a match occurs.

The two Data Watchpoint Address registers may be further grouped together into a single data-address-range watchpoint, WPDA[1: 0].

The instruction and data count value registers must be loaded with the number of times the watchpoint must match minus one. After the count value reaches zero, the subsequent watchpoint match results in an exception or emulation event.

An event can also be triggered on a combination of the instruction and data watchpoints. If the WPIACTL.WPAND bit is set, then an event is triggered only when both an instruction address watchpoint matches *and* a data address watchpoint matches. If the WPAND bit is 0, then an event is triggered when any of the enabled watchpoints or watchpoint ranges match.

To enable the Watchpoint Unit, the WPIACTL.PWR bit must be set. If WPIACTL.PWR = 1, then the individual watchpoints and watchpoint ranges may be enabled using the specific enable bits in the WPIACTL and WPDACTL MMRs. If WPIACTL.PWR = 0, then all watchpoint activity is disabled.

Instruction Watchpoints

Each instruction watchpoint is controlled by three bits in the WPIACTL register, as shown in the ***WPIACTL Control Bits*** table.

Table 9-2: WPIACTL Control Bits

Bit Name	Description
EMUSWx	Determines whether an instruction address match causes either an emulation event or an exception event.
WPICNTENx	Enables the 16-bit counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPIAENx	Enables the address watchpoint activity.

When two watchpoints are associated to form a range, two additional bits are used, as shown in the ***WPIACTL Watchpoint Range Control Bits*** table.

Table 9-3: WPIACTL Watchpoint Range Control Bits

Bit Name	Description
WPIRENxy	Indicates the two watchpoints that are to be associated to form a range.
WPIRINVxy	Determines whether an event is caused by an address within the range identified or outside of the range identified.

Code patching allows software to replace sections of existing code with new code. The watchpoint registers are used to trigger an exception at the start addresses of the earlier code. The exception routine then vectors to the location in memory that contains the new code.

On the processor, code patching can be achieved by writing the start address of the earlier code to one of the `WPIAx` registers and setting the corresponding `EMUSWx` bit to trigger an exception. In the exception service routine, the `WPSTAT` register is read to determine which watchpoint triggered the exception. Next, the code writes the start address of the new code in the `RETX` register and then returns from the exception to the new code. Because the exception mechanism is used for code patching, event service routines of the same or higher priority (exception, NMI, and reset routines) cannot be patched.

A write to the `WPSTAT` MMR clears all the sticky status bits, though the data value written is ignored.

WPIAx Registers

When the Watchpoint Unit is enabled, the values in the Instruction Watchpoint Address registers (`WPIAx`) are compared to the address on the instruction bus. Corresponding count values in the Instruction Watchpoint Address Count registers (`WPIACNTx`) are decremented each time a match is identified. For more information, see Watchpoint Instruction Address Register .

Register Name	Memory-Mapped Address
WPIA0	0xFFE0 7040
WPIA1	0xFFE0 7044
WPIA2	0xFFE0 7048
WPIA3	0xFFE0 704C

Register Name	Memory-Mapped Address
WPIA4	0xFFE0 7050
WPIA5	0xFFE0 7054

WPIACNTx Registers

When the Watchpoint Unit is enabled, the count values in the Instruction Watchpoint Address Count registers (`WPIACNT[5:0]`) are decremented each time the address or the address bus matches a value in the `WPIAx` registers. Load the `WPIACNTx` register with a value that is one less than the number of times the watchpoint must match before triggering an event. The `WPIACNTx` register will decrement to 0x0000 when the programmed count expires. For more information, see the Watchpoint Instruction Address Count Register.

Register Name	Memory-Mapped Address
WPIACNT0	0xFFE0 7080
WPIACNT1	0xFFE0 7084
WPIACNT2	0xFFE0 7088
WPIACNT3	0xFFE0 708C
WPIACNT4	0xFFE0 7090
WPIACNT5	0xFFE0 7094

WPIACTL Register

Three bits in the Instruction Watchpoint Address Control register (`WPIACTL`) control each instruction watchpoint. For more information about the bits in this register, see Watchpoint Unit and Watchpoint Instruction Address Control Register.

Data Address Watchpoints

Each data watchpoint is controlled by four bits in the `WPDACTL` register, as shown in the ***Data Address Watchpoints*** table.

Table 9-4: Data Address Watchpoints

Bit Name	Description
WPDACCx	Determines whether the match should be on a read or write access.
WPDSRCx	Determines which DAG the unit should monitor.
WPDCNTENx	Enables the counter that counts the number of address matches. If the counter is disabled, then every match causes an event.
WPDAENx	Enables the data watchpoint activity.

When the two watchpoints are associated to form a range, two additional bits are used. See the ***WPDACTL Watchpoint Control Bits*** table.

Table 9-5: WPDACTL Watchpoint Control Bits

Bit Name	Description
WPDREN01	Indicates the two watchpoints associated to form a range.
WPDRINV01	Determines whether an event is caused by an address within or outside the range identified.

WPDAx Registers

When the Watchpoint Unit is enabled, the values in the Data Watchpoint Address registers (`WPDAx`) are compared to the address on the data buses. Corresponding count values in the Data Watchpoint Address Count registers (`WPDACNTx`) are decremented each time a match is identified. For more information, see the Watchpoint Data Address Register.

WPDACNTx Registers

When the Watchpoint Unit is enabled, the count values in the Data Watchpoint Address Count Value registers (`WPDACNTx`) are decremented each time the address or the address bus matches a value in the `WPDAx` registers. Load this `WPDACNTx` register with a value that is one less than the number of times the watchpoint must match the address bus before triggering an event. The `WPDACNTx` register will decrement to 0x0000 when the programmed count expires. For more information, see the Watchpoint Data Address Count Value Register .

WPDACTL Register

For more information about the bits in the Data Watchpoint Address Control register (`WPDACTL`), see Data Address Watchpoints and Watchpoint Data Address Control Register.

WPSTAT Register

The Watchpoint Status register (`WPSTAT`) monitors the status of the watchpoints. It may be read and written in Supervisor or Emulator modes only. When a watchpoint or watchpoint range matches, this register reflects the source of the watchpoint. The status bits in the `WPSTAT` register are sticky, and all of them are cleared when the register is written (with any value). For more information, see the Watchpoint Status Register.

Performance Monitor Unit (PMU)

The Blackfin+ architecture provides a built-in performance monitor unit (PMU) to non-intrusively monitor the processor's internal resources. The PMU includes a set of processor events that can be counted during program execution.

A subset of these processor events can be counted in terms of the *number of stalls* that occur while the event is active or true. This stall measurement is in core clock cycles and provides an indication of the performance penalty associated with the event. The rest of the processor events can be counted in terms of the *number of occurrences* of the event. This event measurement helps with program debugging and provides an aid to understanding the performance bottlenecks in an application.

Developers can use the PMU to count pipeline and memory stalls. The stall information can be used iteratively to quickly locate areas to focus on during the software optimization process. The highest level of debugging efficiency is achieved when using the PMU while running applications directly on hardware as opposed to predicting these events in a simulation environment.

For example, the PMU can help to detect whether the performance bottleneck is due to L1 data memory access latencies. Using another PMU event, it can be concluded that the memory stall results from simultaneous access by both the core and the DMA controller to the same region of L1 memory, which is not allowed by the architecture, thus causing one access to stall. However, the processor core and the DMA controller can access different subbanks of memory in the same cycle (refer to *Overview of On-Chip Level 1 (L1) Memory* in the Memory chapter for more details on L1 memory arbitration stalls). After identifying an issue like this using the PMU, one of the buffers can be moved to a non-conflicting bank of L1 memory to minimize core versus DMA access conflicts.

Functional Description

The PMU provides two sets of registers (`PFCTRx` and `PFCTL`), which permit non-intrusive monitoring of the processor's internal resources during program execution.

The 32-bit Performance Monitor Counter (`PFCNTR1-0`) registers hold the number of occurrences of a selected event from within a processor core. Each of the counters must be enabled prior to use.

The Performance Control (PFCTL) register provides:

- enable/disable capabilities for the PMU,
- selection of the *event mode*,
- configuration of the *event type* to be monitored, and
- selection of interrupt handling type for a counter overflow condition.

Together, these registers provide feedback indicating the measure of load-balancing between the various resources on the chip. This feedback permits comparison and analysis of expected versus actual resource usage.

PFCNTRx Registers

The ***Performance Monitor Counter Registers*** figure shows the Performance Monitor Counter registers, `PFCNTR[1:0]`. The `PFCNTR0` register contains the count value of Performance Monitor Counter 0, while the `PFCNTR1` register contains the count value of Performance Counter 1. For more information, see Counter 0 Register and Counter 1 Register .

The counter retains its value even after the module is disabled, so the programmer has to clear the counter before using it again. The counter can also be programmed with a non-zero 32-bit value.

PFCTL Register

To enable the PMU, set the `PFPWR` bit in the Performance Monitor Control (`PFCTL`) register. After the unit is enabled, individual Count Enable bits (`PFCENx`) take effect. Use the `PFCENx` bits to enable or disable the performance monitors in User mode, Supervisor mode, or both. Use the EVENTx bits to select the type of event triggered. For more information, see the Control Register .

Count Event Mode

Setting the PFCTL.CNTx bits enables the events that are listed in the ***PFCTL.MONx Event Type (Occurrences)*** table (see Monitor Event Types) to be counted as an "occurrence" of an event. Clearing these bits enables the events listed in the ***PFCTL.MONx Event Type (Stalls)*** table (see Monitor Event Types) to be counted as "number of stalls" while the event is active/true.

Monitor Event Types

Use the PFCTL.MONx[7:0] bits to select the type of event triggered. The ***PFCTL.MONx Event Type (Occurrences)*** table identifies events that cause the Performance Monitor Counter (PFCTL.MON0 or PFCTL.MON1) fields to increment, based on the number of "occurrences" of that particular event. For the events listed in the ***PFCTL.MONx Event Type (Occurrences)*** table, set the corresponding PFCTL.CNTx bit.

The ***PFCTL.MONx Event Type (Stalls)*** table identifies events that cause the Performance Monitor Counter (PFCTL.MON0 or PFCTL.MON1) fields to increment based on the "number of stalls" until the event is true. For the events listed in the ***PFCTL.MONx Event Type (Stalls)*** table, clear the corresponding PFCTL.CNTx bit.

EVENTx - Counter Overflow Condition

The PFCTL.EVENTx bit provides the flexibility for the PMU to generate either a hardware error or an emulation event when it rolls over. When a hardware error is generated, the SEQSTAT.HWERRCAUSE[1:0] Sequencer status bits are set to 0x12. The PMU can also be used to detect an instance of any of the events in the ***PFCTL.MONx Event Type (Occurrences)*** table (see Monitor Event Types) by pre-loading the counter with the maximum value 0xFFFFFFFF. When configured this way, the first time the selected event happens, the counter rolls over and generates a hardware error.

Programming Example

The following code example demonstrates a possible use case of the PMU to track stalls in a particular application.

```
/* L1 data memory address */
I0.L = LO(0xFF801004);
I0.H = HI(0xFF801004);
/* L1 data memory address in same 4K sub-bank */
I1.L = LO(0xFF801244);
I1.H = HI(0xFF801244);
/* reset performance control register */
P0.L = LO(PFCTL);
P0.H = HI(PFCTL);
R0 = 0;
[P0] = R0;

/* reset performance counter 0 */
P0.L = LO(PFCNTR0);
```

```
P0.H = HI(PFCNTR0);
R0 = 0;
[P0] = R0;

/* enable the monitor and counter 0 */
P0.L = LO(PFCTL);
P0.H = HI(PFCTL);
R0.L = 0x0019;
R0.H = 0x0000;

R1 = PFCEN_VALUE;    /* load the event number (0x96) */
R0 = R0 | R1;
[P0] = R0;          /* program performance control register */
/* parallel instruction accessing 2 data memory locations */
R1 = R4.L * R5.H (IS) || R3 = [I0++] || R4 = [I1++];
```

This results in the counter being incremented by one, as there is a one-cycle stall incurred due to a collision in the data bank A sub-bank 1. A simultaneous access will only result in a stall if the accesses are to the same 32-bit word alignment (address bit 2 matches), the same 4 KB sub-bank (address bits 13 and 12 match), the same 16 KB half-bank (address bit 16 matches), and the same bank (address bits 21 and 20 match).

The Hardware Error interrupt can be used in cases where the application needs to be notified of a specific PMU event. To support this, the `EVENTx` bit has to be cleared, and the counter has to be pre-loaded with a value of 0xFFFFFFFF, as follows:

```
P0.L = LO(PFCNTR0);
P0.H = HI(PFCNTR0);
R0.L = 0xFFFF;
R0.H = 0xFFFF;
[P0] = R0;
```

Because the `EVENT0` bit is cleared, the counter overflow that occurs the first time the programmed event occurs results in a hardware error interrupt being generated The Hardware Error interrupt service routine could be set up and populated as follows to enable custom handling of any PMU event:

```
/* LOAD IMASK ADDRESS */
P0.L = LO(IMASK);
P0.H = HI(IMASK);
R0 = [P0];
R1 = IVHW;             /* ENABLE HARDWARE ERROR INTERRUPT */
R0 = R0 | R1;
[P0] = R0;
/* STORE ISR HANDLER ADDRESS */
P0.L = LO(EVT5);
P0.H = HI(EVT5);
R0.L = LO(IVHW_ISR);
R0.H = HI(IVHW_ISR);
[P0] = R0;
/* HARDWARE ERROR INTERRUPT SERVICE ROUTINE */
IVHW_ISR:
     P0 = SEQSTAT;
     R0 = [P0];      /* READ SEQUENCER STATUS REGISTER */
     R1 = 0x12;      /* CHECK FOR PMU EVENT (HWERRCAUSE 0x12) */
     R1 <<= 14;
     CC = R1 == R0;
     IF !CC JUMP HWERR_EXIT;

PFMON_OVERFLOW:
  /* PERFORMANCE MONITOR OVERFLOW HAS BEEN DETECTED */
  /* Add handling code here */
HWERR_EXIT:
  RTI;
```

Cycle Counters

The cycle counter counts CCLK cycles while the program is executing. All cycles- including execution, wait state, interrupts, and events - are counted while the processor is in User or Supervisor mode, but the cycle counter stops counting in Emulator mode.

The 64-bit cycle counter increments every core clock cycle and is tracked in two 32-bit registers, CYCLES and CYCLES2. The least significant 32 bits (LSBs) are stored in CYCLES, and the most significant 32 bits (MSBs) are stored in CYCLES2.

The CYCLES and CYCLES2 registers are read/write in all modes (User, Supervisor, and Emulator) for all Blackfin+ processors. For more information, see Cycle Count (32 LSBs) Register and Cycle Count (32 MSBs) Register .

To enable the cycle counters, set the SYSCFG.CCEN bit. The following example shows how to use the cycle counter to benchmark a piece of code:

```
R2 = 0;          /* Clear the cycle counters */
CYCLES  = R2;
CYCLES2 = R2;

R2 = SYSCFG;
BITSET(R2,1);
SYSCFG = R2;      /* Enable the cycle counters */

/* Insert code to be benchmarked here */
R2 = CYCLES2;
R1 = CYCLES;      /* R2:1 contain 64-bit CYCLES2/CYCLES cycle count */
```

CYCLES and CYCLES2 Registers

The Execution Cycle Count registers (CYCLES and CYCLES2) form a 64-bit counter that increments every CCLK cycle. The CYCLES register contains the least significant 32 bits of the cycle counter's 64-bit count value, while the CYCLES2 register contains the most significant 32 bits. For more information, see Cycle Count (32 LSBs) Register and Cycle Count (32 MSBs) Register .

NOTE: When single-stepping through instructions in a debug environment, the CYCLES register is incremented in a non-uniform fashion due to interaction with the debugger over JTAG.

SYSCFG Register

The System Configuration register (SYSCFG) controls the configuration of the processor. This register is accessible only from Supervisor mode. For more information, see the System Configuration Register .

Product Identification Register

The 32-bit Product Identification register (DSPID) is a core MMR that contains core identification and revision fields for the core.

DSPID Register

The Product Identification register (DSPID) is a read-only register and is part of the processor core. This register format differs depending on whether the processor is a single or dual-core processor. For more information, see the DSP Identification Register .

Blackfin+ DBG Register Descriptions

Debug (DBG) contains the following registers.

Table 9-6: Blackfin+ DBG Register List

Name	Description
DSPID	DSP Identification Register

DSP Identification Register

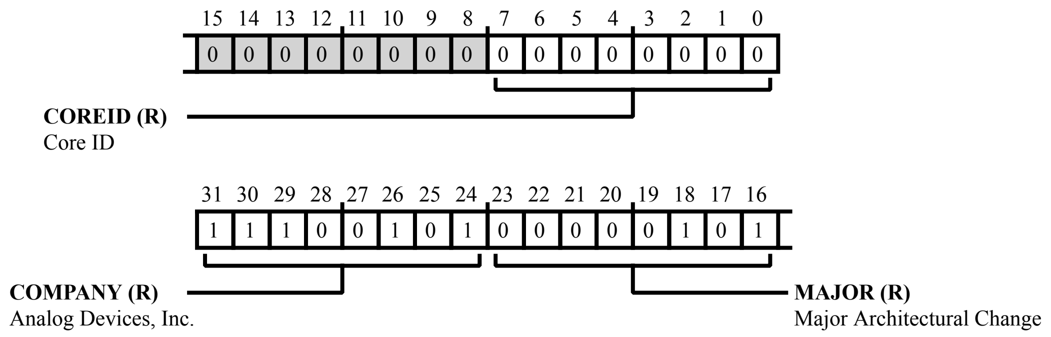


Figure 9-1: DSPID Register Diagram

Table 9-7: DSPID Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
31:24 (R/NW)	COMPANY	Analog Devices, Inc..	
		229	Analog Devices, Inc.
23:16 (R/NW)	MAJOR	Major Architectural Change.	
		4	Blackfin
		5	Blackfin+
7:0 (R/NW)	COREID	Core ID.	

Blackfin+ WP Register Descriptions

Watchpoint Unit (WP) contains the following registers.

Table 9-8: Blackfin+ WP Register List

Name	Description
`WPDACNTN[n]`	Watchpoint Data Address Count Register
`WPDACTL`	Watchpoint Data Address Control Register
`WPDAN[n]`	Watchpoint Data Address Register
`WPIACNTN[n]`	Watchpoint Instruction Address Count Register
`WPIACTL`	Watchpoint Instruction Address Control Register
`WPIAN[n]`	Watchpoint Instruction Address Register
`WPSTAT`	Watchpoint Status Register

Watchpoint Data Address Count Register

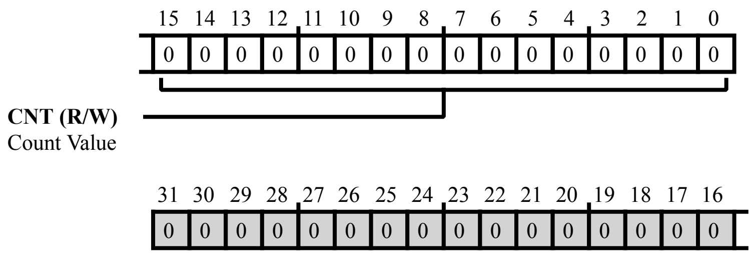


Figure 9-2: WPDACNTN[n] Register Diagram

Table 9-9: WPDACNTN[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Count Value.

Watchpoint Data Address Control Register

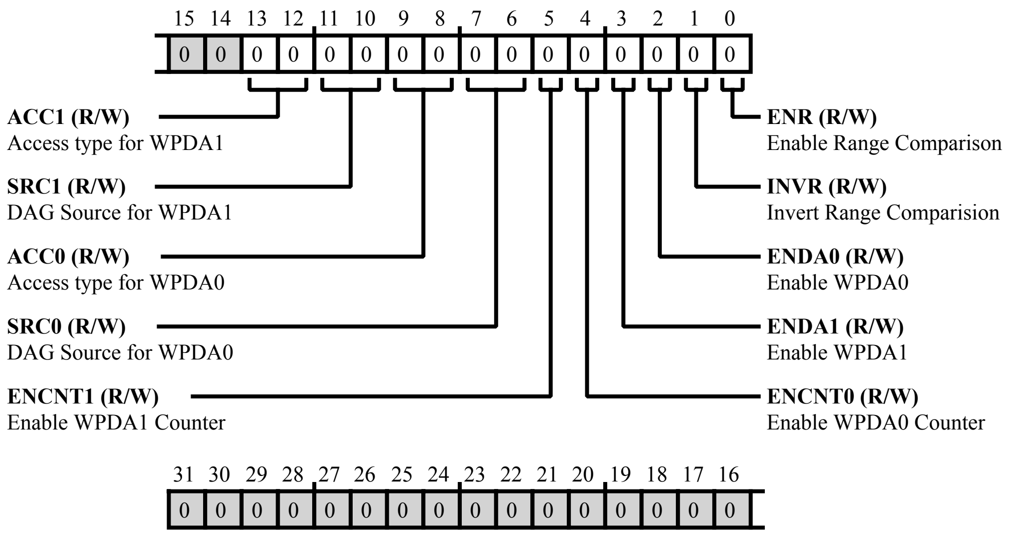


Figure 9-3: WPDACTL Register Diagram

Table 9-10: WPDACTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
13:12 (R/W)	ACC1	Access type for WPDA1.	
		0	Reserved
		1	Watch Writes only
		2	Watch Reads only
		3	Watch Reads and Writes
11:10 (R/W)	SRC1	DAG Source for WPDA1.	
		0	Reserved
		1	Watch DAG0
		2	Watch DAG1
		3	Watch Both DAGs
9:8 (R/W)	ACC0	Access type for WPDA0.	
		0	Reserved
		1	Watch Writes only
		2	Watch Reads only
		3	Watch Reads and Writes

Table 9-10: WPDACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
7:6 (R/W)	SRC0	DAG Source for WPDA0.	
		0	Reserved
		1	Watch DAG0
		2	Watch DAG1
		3	Watch Both DAGs
5 (R/W)	ENCNT1	Enable WPDA1 Counter.	
4 (R/W)	ENCNT0	Enable WPDA0 Counter.	
3 (R/W)	ENDA1	Enable WPDA1.	
2 (R/W)	ENDA0	Enable WPDA0.	
1 (R/W)	INVR	Invert Range Comparision.	
0 (R/W)	ENR	Enable Range Comparison. Address matches if it satisfies this equation WPDA0 < ADDRESS <= WPDA1	

Watchpoint Data Address Register

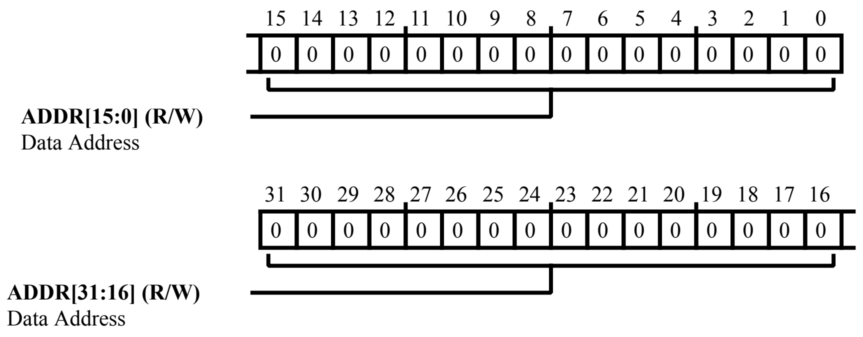


Figure 9-4: WPDAN[n] Register Diagram

Table 9-11: WPDAN[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Data Address.

Watchpoint Instruction Address Count Register

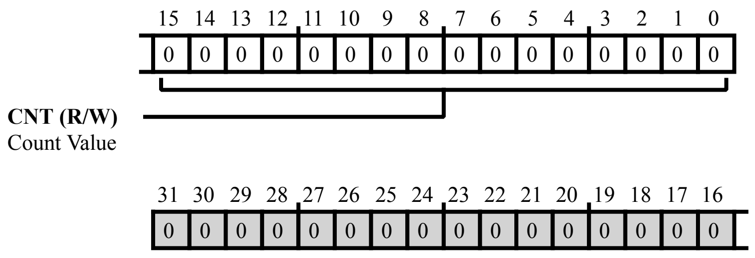


Figure 9-5: WPIACNTN[n] Register Diagram

Table 9-12: WPIACNTN[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
15:0 (R/W)	CNT	Count Value.

Watchpoint Instruction Address Control Register

The WPIACTL register configures up to six instruction watchpoints to monitor individual addresses (up to six), address ranges (up to three), or a combination of both.

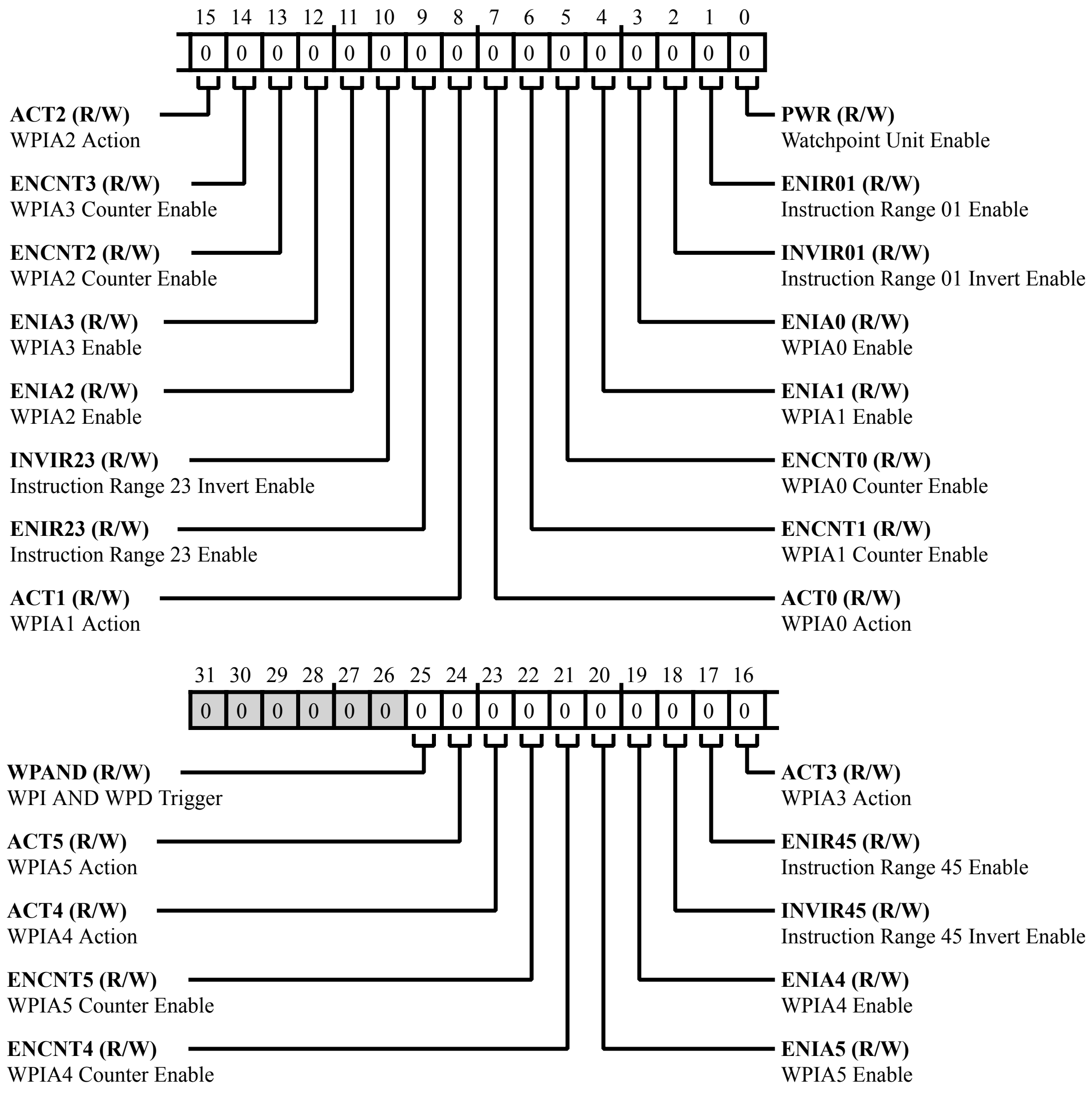


Figure 9-6: WPIACTL Register Diagram

Table 9-13: WPIACTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
25 (R/W)	WPAND	WPI AND WPD Trigger. The `WPIACTL.WPAND` bit determines whether a watchpoint hit has to occur in either or both of the instruction and data watchpoint units for an event to be triggered.	
		0	Trigger an event on every watchpoint match (instruction or data)
		1	Trigger event only if there is both an instruction watchpoint match and a data watchpoint match
24 (R/W)	ACT5	WPIA5 Action. The `WPIACTL.ACT5` bit determines whether an exception or emulation event occurs upon an instruction watchpoint 5 match.	
		0	Exception event
		1	Emulation event
23 (R/W)	ACT4	WPIA4 Action. The `WPIACTL.ACT4` bit determines whether an exception or emulation event occurs upon either an instruction watchpoint 4 match or an instruction watchpoint 4/5 range pair match.	
		0	Exception event
		1	Emulation event
22 (R/W)	ENCNT5	WPIA5 Counter Enable. The `WPIACTL.ENCNT5` bit determines whether the WPIA5 event is generated on every match or only after a specified number of matches occur (as set in the associated WPIACNT5 register).	
		0	Event on every match
		1	Event controlled by WPIACNT5
21 (R/W)	ENCNT4	WPIA4 Counter Enable. The `WPIACTL.ENCNT4` bit determines whether the WPIA4 event is generated on every match or only after a specified number of matches occur (as set in the associated WPIACNT4 register). When `WPIACTL.ENIR45` is set to enable the 4/5 range pair, this bit is used for the same function for the range.	
		0	Event on every match
		1	Event controlled by WPIACNT4

Table 9-13: WPIACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
20 (R/W)	ENIA5	WPIA5 Enable. The `WPIACTL.ENIA5` bit determines whether or not the WPIA5 watchpoint is enabled to monitor an individual address. As such, this bit is only valid when `WPIACTL.ENIR45` is not set.	
		0	Disabled
		1	Enabled
19 (R/W)	ENIA4	WPIA4 Enable. The `WPIACTL.ENIA4` bit determines whether or not the WPIA4 watchpoint is enabled to monitor an individual address. As such, this bit is only valid when `WPIACTL.ENIR45` is not set.	
		0	Disabled
		1	Enabled
18 (R/W)	INVIR45	Instruction Range 45 Invert Enable. The `WPIACTL.INVIR45` bit determines whether the watchpoint event occurs when the instruction address is within or outside of the range defined by the WPIA4/WPIA5 register pair.	
		0	Event generated when WPIA4 < ADDRESS <= WPIA5
		1	Event generated when ADDRESS <= WPIA4 or ADDRESS > WPIA5
17 (R/W)	ENIR45	Instruction Range 45 Enable. When the `WPIACTL.ENIR45` bit is set, the WPIA4/WPIA5 instruction watchpoint pair define a range of addresses for comparisons, and the individual watchpoint enable bits `WPIACTL.ENIA4` and `WPIACTL.ENIA5` become invalid. When defined to be a range, the start address of the range is in WPIA4 and the end address is in WPIA5.	
		0	Disable Range
		1	Enable Range
16 (R/W)	ACT3	WPIA3 Action. The `WPIACTL.ACT3` bit determines whether an exception or emulation event occurs upon an instruction watchpoint 3 match.	
		0	Exception event
		1	Emulation event

Table 9-13: WPIACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
15 (R/W)	ACT2	WPIA2 Action. The `WPIACTL.ACT2` bit determines whether an exception or emulation event occurs upon either an instruction watchpoint 2 match or an instruction watchpoint 2/3 range pair match.	
		0	Exception event
		1	Emulation event
14 (R/W)	ENCNT3	WPIA3 Counter Enable. The `WPIACTL.ENCNT3` bit determines whether the WPIA3 event is generated on every match or only after a specified number of matches occur (as set in the associated WPIACNT3 register).	
		0	Event on every match
		1	Event controlled by WPIACNT3
13 (R/W)	ENCNT2	WPIA2 Counter Enable. The `WPIACTL.ENCNT2` bit determines whether the WPIA2 event is generated on every match or only after a specified number of matches occur (as set in the associated WPIACNT2 register).	
		0	Event on every match
		1	Event controlled by WPIACNT2
12 (R/W)	ENIA3	WPIA3 Enable. The `WPIACTL.ENIA3` bit determines whether or not the WPIA3 watchpoint is enabled to monitor an individual address. As such, this bit is only valid when `WPIACTL.ENIR23` is not set.	
		0	Diabled
		1	Enabled
11 (R/W)	ENIA2	WPIA2 Enable. The `WPIACTL.ENIA2` bit determines whether or not the WPIA2 watchpoint is enabled to monitor an individual address. As such, this bit is only valid when `WPIACTL.ENIR23` is not set.	
		0	Disabled
		1	Enabled

Table 9-13: WPIACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
10 (R/W)	INVIR23	Instruction Range 23 Invert Enable. The `WPIACTL.INVIR23` bit determines whether the watchpoint event occurs when the instruction address is within or outside of the range defined by the WPIA2/WPIA3 register pair.	
		0	Event generated when WPIA2 < ADDRESS <= WPIA3
		1	Event generated when ADDRESS <= WPIA3 or AD-DRESS > WPIA3
9 (R/W)	ENIR23	Instruction Range 23 Enable. When the `WPIACTL.ENIR23` bit is set, the WPIA2/WPIA3 instruction watchpoint pair define a range of addresses for comparisons, and the individual watchpoint enable bits `WPIACTL.ENIA2` and `WPIACTL.ENIA3` become invalid. When defined to be a range, the start address of the range is in WPIA2 and the end address is in WPIA3.	
		0	Disable Range
		1	Enable Range
8 (R/W)	ACT1	WPIA1 Action. The `WPIACTL.ACT1` bit determines whether an exception or emulation event occurs upon an instruction watchpoint 1 match.	
		0	Exception event
		1	Emulation event
7 (R/W)	ACT0	WPIA0 Action. The `WPIACTL.ACT0` bit determines whether an exception or emulation event occurs upon either an instruction watchpoint 0 match or an instruction watchpoint 0/1 range pair match.	
		0	Exception event
		1	Emulation event
6 (R/W)	ENCNT1	WPIA1 Counter Enable. The `WPIACTL.ENCNT1` bit determines whether the WPIA1 event is generated on every match or only after a specified number of matches occur (as set in the associated WPIACNT1 register).	
		0	Event on every match
		1	Event controlled by WPIACNT1

Table 9-13: WPIACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
5 (R/W)	ENCNT0	WPIA0 Counter Enable. The `WPIACTL.ENCNT0` bit determines whether the WPIA0 event is generated on every match or only after a specified number of matches occur (as set in the associated WPIACNT0 register).	
		0	Event on every match
		1	Event controlled by WPIACNT0
4 (R/W)	ENIA1	WPIA1 Enable. The `WPIACTL.ENIA1` bit determines whether or not the WPIA1 watchpoint is enabled to monitor an individual address. As such, this bit is only valid when `WPIACTL.ENIR01` is not set.	
		0	Disabled
		1	Enabled
3 (R/W)	ENIA0	WPIA0 Enable. The `WPIACTL.ENIA0` bit determines whether or not the WPIA0 watchpoint is enabled to monitor an individual address. As such, this bit is only valid when `WPIACTL.ENIR01` is not set.	
		0	Disabled
		1	Enabled
2 (R/W)	INVIR01	Instruction Range 01 Invert Enable. The `WPIACTL.INVIR01` bit determines whether the watchpoint event occurs when the instruction address is within or outside of the range defined by the WPIA0/WPIA1 register pair.	
		0	Event generated when WPIA0 < ADDRESS <= WPIA1
		1	Event generated when ADDRESS <= WPIA0 or ADDRESS > WPIA1
1 (R/W)	ENIR01	Instruction Range 01 Enable. When the `WPIACTL.ENIR01` bit is set, the WPIA0/WPIA1 instruction watchpoint pair define a range of addresses for comparisons, and the individual watchpoint enable bits `WPIACTL.ENIA0` and `WPIACTL.ENIA1` become invalid. When defined to be a range, the start address of the range is in WPIA0 and the end address is in WPIA1.	
		0	Disable Range
		1	Enable Range

Table 9-13: WPIACTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
0 (R/W)	PWR	Watchpoint Unit Enable. The `WPIACTL.PWR` bit determines whether or not the instruction watchpoint unit is enabled.	
		0	Disabled
		1	Enabled

Watchpoint Instruction Address Register

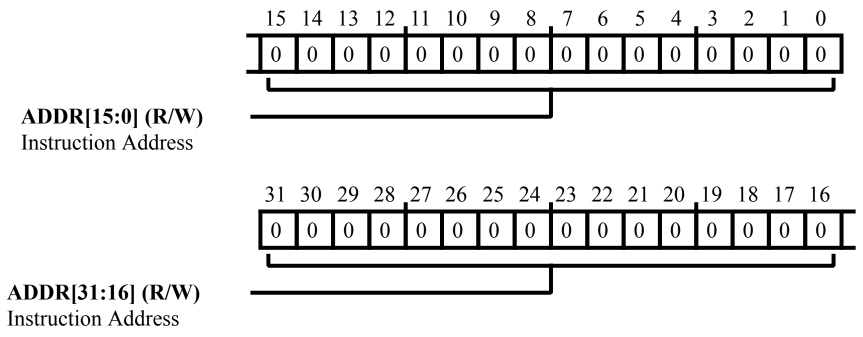


Figure 9-7: WPIAN[n] Register Diagram

Table 9-14: WPIAN[n] Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	ADDR	Instruction Address.

Watchpoint Status Register

The Watchpoint Status register (`WPSTAT`) monitors the status of all of the data and instruction watchpoints. It may be read and written in Supervisor or Emulator modes only. When a watchpoint or watchpoint range match occurs, the associated status bit is set and remains set until explicitly cleared by software.

Any write to the `WPSTAT` register will clear any and all set status bits.

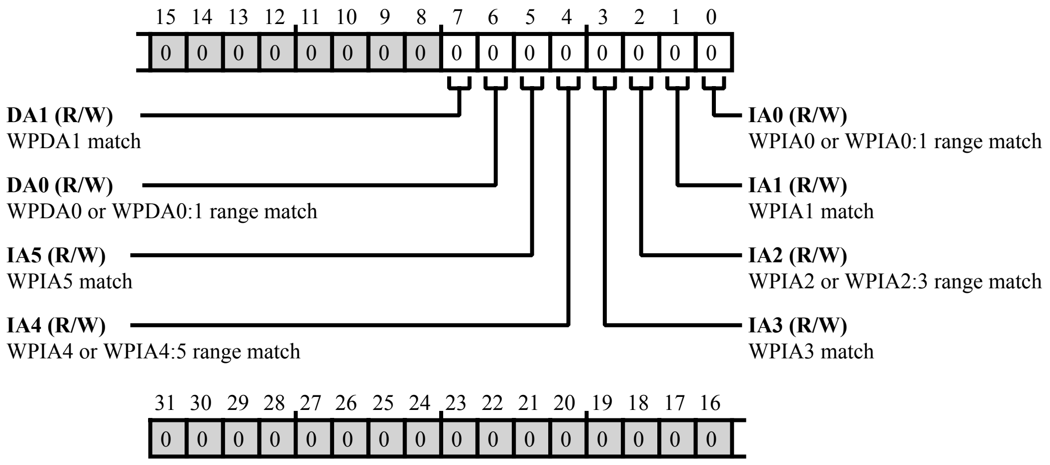


Figure 9-8: WPSTAT Register Diagram

Table 9-15: WPSTAT Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
7 (R/W)	DA1	WPDA1 match.
6 (R/W)	DA0	WPDA0 or WPDA0:1 range match.
5 (R/W)	IA5	WPIA5 match.
4 (R/W)	IA4	WPIA4 or WPIA4:5 range match.
3 (R/W)	IA3	WPIA3 match.
2 (R/W)	IA2	WPIA2 or WPIA2:3 range match.
1 (R/W)	IA1	WPIA1 match.

Table 9-15: WPSTAT Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration
0 (R/W)	IA0	WPIA0 or WPIA0:1 range match.

Blackfin+ PF Register Descriptions

Performance Monitor (PF) contains the following registers.

Table 9-16: Blackfin+ PF Register List

Name	Description
`PFCNTR0`	Counter 0 Register
`PFCNTR1`	Counter 1 Register
`PFCTL`	Control Register

Counter 0 Register

The PFCNTR0 register holds the count value for performance monitor counter 0. Depending on the configuration of the PFCTL register, this count decrements based on monitored occurrences of events or stall cycles related to events. When this count decrements to zero (expires), the PF issues an exception or an emulation event.

The PFCNTR0 counter retains its value even after the PF is disabled, so the counter must be cleared before it may be used again.

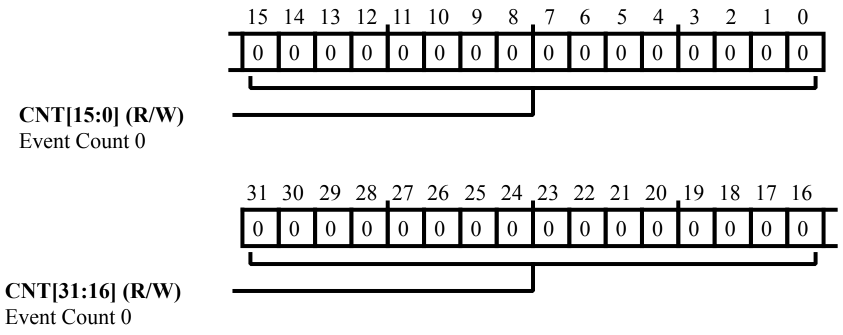


Figure 9-9: PFCNTR0 Register Diagram

Table 9-17: PFCNTR0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	Event Count 0. The PFCNTR0.CNT bits hold the count value for performance monitor counter 0.

Counter 1 Register

The PFCNTR1 register holds the count value for performance monitor counter 1. Depending on the configuration of the PFCTL register, this count decrements based on monitored occurrences of events or stall cycles related to events. When this count decrements to zero (expires), the PF issues an exception or an emulation event.

The PFCNTR1 counter retains its value even after the PF is disabled, so the counter must be cleared before it may be used again.

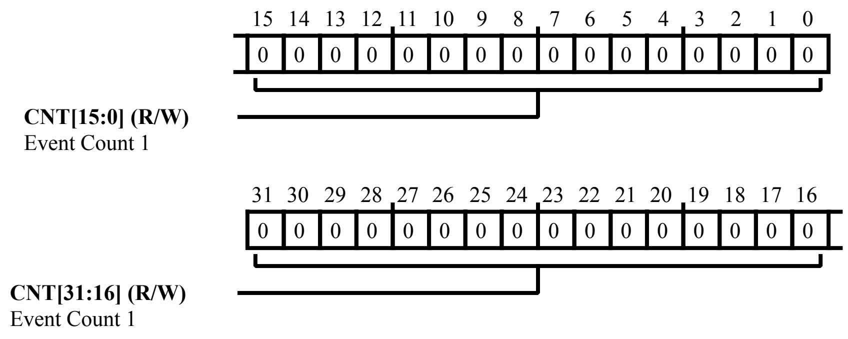


Figure 9-10: PFCNTR1 Register Diagram

Table 9-18: PFCNTR1 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	Event Count 1. The PFCNTR1.CNT bits hold the count value for performance monitor counter 1.

Control Register

The PFCTL register enables the performance monitor unit PF, selects whether event count expirations generate emulator or exception events, select the processor modes in which monitoring is enabled, and select the event type occurrences or stalls that the monitor counts.

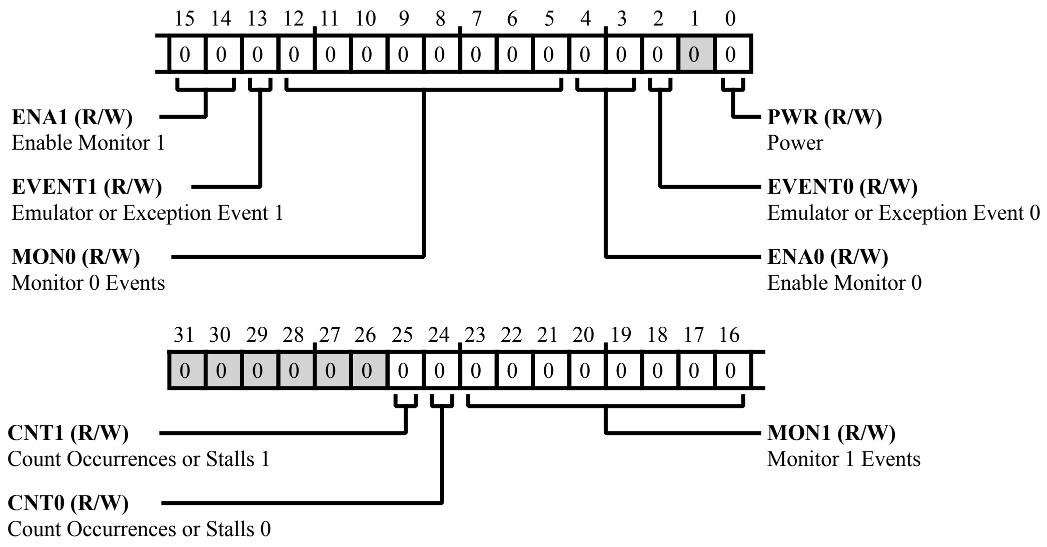


Figure 9-11: PFCTL Register Diagram

Table 9-19: PFCTL Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration	
25 (R/W)	CNT1	Count Occurrences or Stalls 1. The PFCTL.CNT1 bit selects whether monitor 1 counts the number of event type occurrences or event type related stall cycles for the event type selected with the PFCTL.MON1 bits.	
		0	Count stall cycles due to event type 1
		1	Count occurrences of event type 1
24 (R/W)	CNT0	Count Occurrences or Stalls 0. The PFCTL.CNT0 bit selects whether monitor 1 counts the number of event type occurrences or event type related stall cycles for the event type selected with the PFCTL.MON0 bits.	
		0	Count stall cycles due to event type 0
		1	Count occurrences of event type 0
23:16 (R/W)	MON1	Monitor 1 Events. The PFCTL.MON1 bits select the event type that is monitored, causing the PFCNTR1 count to decrement. The PFCTL.CNT1 bit selects whether it is occurrences of this type of event or stall cycles related to this type of event that affect the count. For	

Table 9-19: PFCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		information about event type values for the `PFCTL.MON1` bit field, see the functional description of the PF unit.	
		0	Loop 0 iterations: count each time we iterate for loop 0
		1	Loop 1 iterations: count each time we iterate for loop 1
		2	Loop buffer 0 not optimized: count once each time there is a stall when the loop is not initialized efficiently
		3	Loop buffer 1 not optimized: count once each time there is a stall when the loop is not initialized efficiently
		4	PC invariant branches: Count number of PC invariant branches
		5	Reserved
		6	Conditional branches: Count number of conditional branches (not number of stalls)
		7	Reserved
		8	Reserved
		9	Total branches including calls, returns, branches, but not interrupts
		10	Stalls due to CSYNC, SSYNC
		11	EXCPT instructions: Count number EXCPT instructions that are executed
		12	CSYNC, SSYNC instructions: 1 count for each that counts number of each instruction that are executed
		13	Committed instructions: count total number of committed instructions
		14	Interrupts taken: Count total number of interrupts that are taken. Optionally count total interrupts and interrupts at a given IVG level
		15	Misaligned address violation exceptions
		16	DAG register read-after-write stall
		17	Reserved
		18	Reserved
		19	Stall cycles due to compute register read-after-write hazards
		20-31	Reserved

Table 9-19: PFCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		32	BRCC Static learn request
		33	BRCC Dynamic learn request
		34	BRCC Static or Dynamic learn request
		35	BRCC Mispredict (Static or Dynamic)
		36	BRCC Taken correctly
		37	BRCC Prediction commited
		38	BRCC learn request written to table
		39	BRCC Prediction from Branch Predictor
		40	JUMP learn request written to table
		41	JUMP Prediction from Branch Predictor
		42	RTS learn request written to table
		43	RTS Prediction from Branch Predictor
		44	JDI learn request written to table (does not exist)
		45	JDI Prediction from Branch Predictor (does not exist)
		46	CALL learn request written to table
		47	CALL Prediction from Branch Predictor
		48	BRCC Update written to table
		49	Update written to table
		50	Prediction from Branch Predictor
		51	Prediction per entry
		52	Prediction Taken per entry
		53	Pending Store Buffer Overwritten
		54	Branch Predictor timeouts
		55-127	Reserved
		128	Stall due to DAG to DAG Bank Collision
		129	Cache Hit
		130	Cache Miss
		131	Stall due to Fill Buffer Unavailable
		132-143	Reserved
		144	Stall due to Write Buffer unavailable

Table 9-19: PFCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		145	Stall due to Fill Buffer Unavailable
		146	Stall due to DAG to DAG Bank Collision
		147	Stall due to DAG to DMA Bank Collision
		148	Stall due to MMU Stall to Core
		149	Reserved
		150	Reserved
		151	Reserved
		152	Cache Fill Completed
		153	Cache Line Replacement
		154	Cache Hit (counted in pairs)
		155	Cache Miss (counted in pairs)
		156	DMA Read
		157	DMA Write
		158-255	Reserved
15:14 (R/W)	ENA1	Enable Monitor 1. The `PFCTL.ENA1` bits select in which processor modes (user, supervisor, or both) the performance monitor 1 is enabled.	
		0	Disable monitor 1
		1	Enable monitor 1 in user mode only
		2	Enable monitor 1 in supervisor mode only
		3	Enable monitor 1 in user and supervisor mode
13 (R/W)	EVENT1	Emulator or Exception Event 1. The `PFCTL.EVENT1` bit selects whether expiration of the `PFCNTR1` count down causes an emulation event or an exception.	
		0	Exception on expired count 1
		1	Emulation event on expired count 1
12:5 (R/W)	MON0	Monitor 0 Events. The `PFCTL.MON0` bits select the event type that is monitored, causing the `PFCNTR0` count to decrement. The `PFCTL.CNT0` bit selects whether it is occurrences of this type of event or stall cycles related to this type of event that affect the count. For information about event type values for the `PFCTL.MON0` bit field, see the functional description of the PF unit.	
		0	Loop 0 iterations: count each time we iterate for loop 0

Table 9-19: PFCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		1	Loop 1 iterations: count each time we iterate for loop 1
		2	Loop buffer 0 not optimized: count once each time there is a stall when the loop is not initialized efficiently
		3	Loop buffer 1 not optimized: count once each time there is a stall when the loop is not initialized efficiently
		4	PC invariant branches: Count number of PC invariant branches
		5	Reserved
		6	Conditional branches: Count number of conditional branches (not number of stalls)
		7	Reserved
		8	Reserved
		9	Total branches including calls, returns, branches, but not interrupts
		10	Stalls due to CSYNC, SSYNC
		11	EXCPT instructions: Count number EXCPT instructions that are executed
		12	CSYNC, SSYNC instructions: 1 count for each that counts number of each instruction that are executed
		13	Committed instructions: count total number of committed instructions
		14	Interrupts taken: Count total number of interrupts that are taken. Optionally count total interrupts and interrupts at a given IVG level
		15	Misaligned address violation exceptions
		16	DAG register read-after-write stall
		17	Reserved
		18	Reserved
		19	Stall cycles due to compute register read-after-write hazards
		20-31	Reserved
		32	BRCC Static learn request
		33	BRCC Dynamic learn request
		34	BRCC Static or Dynamic learn request

Table 9-19: PFCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		35	BRCC Mispredict (Static or Dynamic)
		36	BRCC Taken correctly
		37	BRCC Prediction commited
		38	BRCC learn request written to table
		39	BRCC Prediction from Branch Predictor
		40	JUMP learn request written to table
		41	JUMP Prediction from Branch Predictor
		42	RTS learn request written to table
		43	RTS Prediction from Branch Predictor
		44	JDI learn request written to table (does not exist)
		45	JDI Prediction from Branch Predictor (does not exist)
		46	CALL learn request written to table
		47	CALL Prediction from Branch Predictor
		48	BRCC Update written to table
		49	Update written to table
		50	Prediction from Branch Predictor
		51	Prediction per entry
		52	Prediction Taken per entry
		53	Pending Store Buffer Overwritten
		54	Branch Predictor timeouts
		55-127	Reserved
		128	Stall due to DAG to DAG Bank Collision
		129	Cache Hit
		130	Cache Miss
		131	Stall due to Fill Buffer Unavailable
		132-143	Reserved
		144	Stall due to Write Buffer unavailable
		145	Stall due to Fill Buffer Unavailable
		146	Stall due to DAG to DAG Bank Collision
		147	Stall due to DAG to DMA Bank Collision

Table 9-19: PFCTL Register Fields (Continued)

Bit No. (Access)	Bit Name	Description/Enumeration	
		148	Stall due to MMU Stall to Core
		149	Reserved
		150	Reserved
		151	Reserved
		152	Cache Fill Completed
		153	Cache Line Replacement
		154	Cache Hit (counted in pairs)
		155	Cache Miss (counted in pairs)
		156	DMA Read
		157	DMA Write
		158-255	Reserved
4:3 (R/W)	ENA0	Enable Monitor 0. The PFCTL.ENA0 bits select in which processor modes (user, supervisor, or both) the performance monitor 0 is enabled.	
		0	Disable monitor 0
		1	Enable monitor 0 in user mode only
		2	Enable monitor 0 in supervisor mode only
		3	Enable monitor 0 in user and supervisor mode
2 (R/W)	EVENT0	Emulator or Exception Event 0. The PFCTL.EVENT0 bit selects whether expiration of the PFCNTR0 count down causes an emulation event or an exception.	
		0	Exception on expired count 0
		1	Emulation event on expired count 0
0 (R/W)	PWR	Power. The PFCTL.PWR bit enables the PF.	
		0	Disable
		1	Enable

ADSP-BF70x Debug-Related (REGFILE) Register Descriptions

The Debug-Related Register File (REGFILE) contains the following registers.

Table 9-20: ADSP-BF70x Debug-Related (REGFILE) Register List

Name	Description
CYCLES	Cycle Counter Register
CYCLES2	Cycle Counter 2 Register

Cycle Count (32 LSBs) Register

The CYCLES register holds the least significant 32 bits of the 64-bit cycle count. The counter is enabled by setting the SYSCFG.CCEN bit. Once enabled, the CYCLES register increments once per core clock (CCLK) cycle, including wait states. As the CYCLES register is in an unknown state after reset, it should be set to 0 prior to every use.

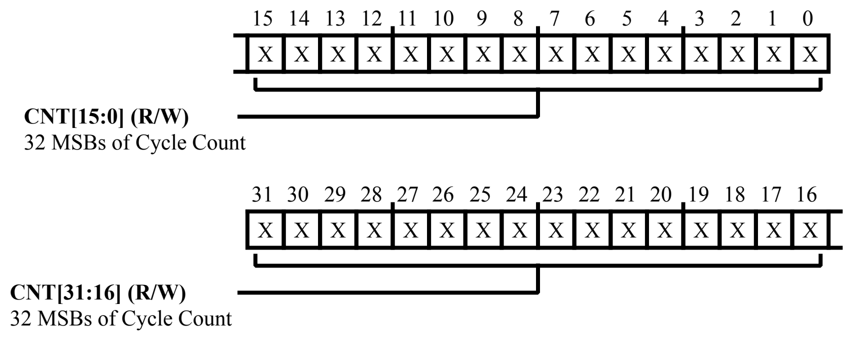


Figure 9-12: CYCLES Register Diagram

Table 9-21: CYCLES Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	32 MSBs of Cycle Count. The CYCLES.CNT bits hold the most significant 32 bits of the 64-bit cycle count value.

Cycle Count (32 MSBs) Register

The CYCLES2 register holds the most significant 32 bits of the 64-bit cycle count. The counter is enabled by setting the SYSCFG.CCEN bit. Once enabled, CYCLES2 increments every time the 32-bit CYCLES register wraps back to zero. As the CYCLES2 register is in an unknown state after reset, it should be set to 0 prior to each use.

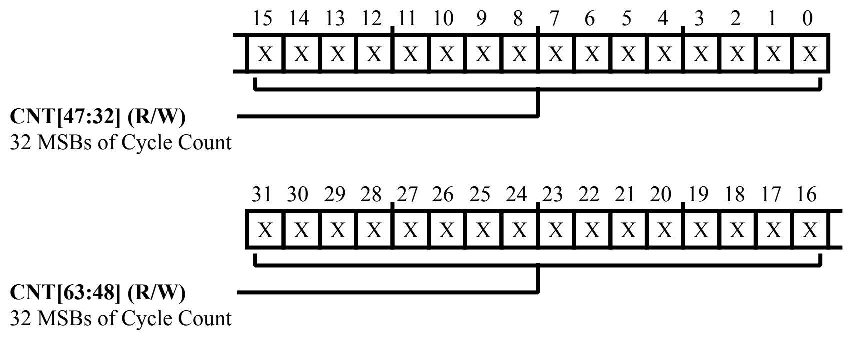


Figure 9-13: CYCLES2 Register Diagram

Table 9-22: CYCLES2 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
31:0 (R/W)	CNT	32 MSBs of Cycle Count. The CYCLES2.CNT bits hold the most significant 32 bits of the 64-bit cycle count value.

Blackfin+ OPT Register Descriptions

Optional Core Features (OPT) contains the following registers.

Table 9-23: Blackfin+ OPT Register List

Name	Description
OPT_FEATURE0	Feature Core 0 Register

Feature Core 0 Register

The OPT_FEATURE0 register is a status register that indicates the presence of optional core features and resources. The status reported in this register only indicates the presence of the feature or resource. The information in this register does NOT indicate whether or not the feature or resource is enabled (if applicable) or has been configured (if applicable).

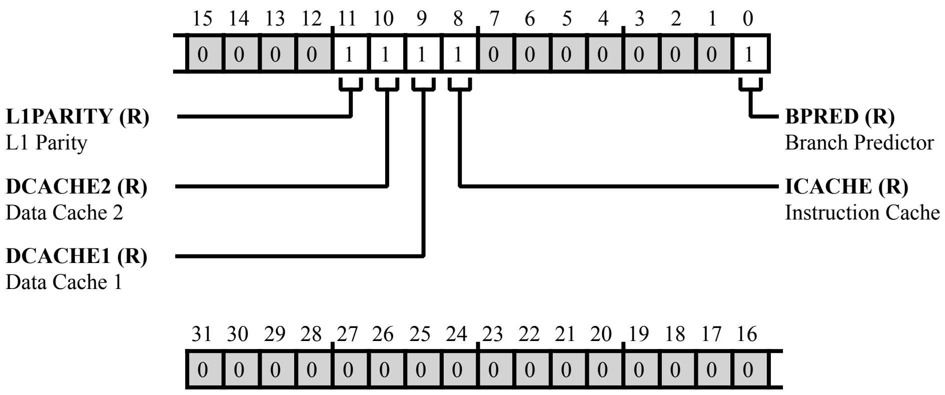


Figure 9-14: OPT_FEATURE0 Register Diagram

Table 9-24: OPT_FEATURE0 Register Fields

Bit No. (Access)	Bit Name	Description/Enumeration
11 (R/NW)	L1PARITY	L1 Parity. The OPT_FEATURE0.L1PARITY bit indicates whether or not the processor core has L1 parity error detection. If =1, the core has this feature.
10 (R/NW)	DCACHE2	Data Cache 2. The OPT_FEATURE0.DCACHE2 bit indicates whether or not the processor core has at least two data caches. If =1, the core has this feature.
9 (R/NW)	DCACHE1	Data Cache 1. The OPT_FEATURE0.DCACHE1 bit indicates whether or not the processor core has at least one data cache. If =1, the core has this feature.
8 (R/NW)	ICACHE	Instruction Cache. The OPT_FEATURE0.ICACHE bit indicates whether or not the processor core has branch predictor. If =1, the core has this feature.
0 (R/NW)	BPRED	Branch Predictor. The OPT_FEATURE0.BPRED bit indicates whether or not the processor core has branch predictor. If =1, the core has this feature.

10 Program Trace Macrocell (PTM)

The processor core implements Program Trace Macrocell (PTM) which implements a subset of Coresight Program Flow Trace Architecture (CSPFT) specification by ARM and provides instruction trace capability. For Cortex A5 trace unit features refer to the *Embedded Trace Macrocell (ETM)* chapter the hardware reference manual.

Features

The trace module has the following features

- Address comparators and Context ID comparators for filtering trace data and use as event resources.
- External inputs and outputs for use as event resources.
- Events can be created using address comparators, context ID comparators and external inputs.
- Counters to count events occurrences.

Functional Description

The following section describes the features available in the trace module.

Address Comparators

The trace module provides 4 address comparators. Program the Address Comparator Value register with the address to be matched and the corresponding Address Comparator Access Type register with additional information about the required comparison shown in the following list.

- Include or exclude range
- Linking the address comparison with Context ID comparator

Address comparators can be used

- Individually, as single address comparators (SACs)
- In pairs, as address range comparators (ARCs), in which case two adjacent address comparators form an ARC.

Context ID Comparators

The trace module provides 1 Context ID comparator.

Context ID comparator consists of a Context ID Comparator Value Register which can hold a Context ID value, for comparison with the current Context ID and a Context ID Comparator Mask Register which can hold a mask value, which is used to mask all Context ID comparisons. If Context ID Comparator Mask Register is programmed to zero then no mask is applied to the Context ID comparisons.

Events

The trace module includes a number of event resources, address comparators, context ID comparators and external inputs.

Event resources can be used to define events. Event register can be programmed to define the corresponding event as the result of a logical operation involving one or two event resources.

Each event resource is either active or inactive, active event resource generates a logical TRUE signal and an inactive event resource generates a logic FALSE signal. An event is logical combinational of event resources, therefore at any given time each event is either TRUE or FALSE.

Counters

The trace module provides 2 counters that are controlled using events. Each 16-bit counter can count from 0 to 65535. Counter behavior is controlled by the following registers.

Counter Enable Event Register

Enables the counter and counts down while the counter enable event is TRUE.

Counter Reload Event Register

Reloades the counter from the Counter Reload Value Register when a counter reload event occurs.

Counter Reload Value Register

Holds the value that is loaded into the counter when the counter reload event is TRUE.

Counter Value Register

Finds the current value of the counter at any time through a read and writes a new value into the counter when programming the trace module.

Trace Security

The trace module supports that is controlled by the Debug Enable input signal. It controls whether the trace module is allowed to trace instructions. If this signal is de-asserted, all tracing will stop, all internal resources are disabled and trace module's state is held.

Programming Model

The trace module registers are memory-mapped in a 4KB region as per CoreSight programmers model

References

- CoreSight™ Program Flow Trace™ Architecture Specification - ARM IHI 0035B – Available at http://infocenter.arm.com
- CoreSight™ Architecture Specification - ARM IHI 0029B – Available at http://infocenter.arm.com

11 Numeric Formats

The Blackfin+ family processors support 8-, 16-, 32-, and 40-bit fixed-point data in hardware. Special features in the computation units allow support of other formats in software. This appendix describes various aspects of these data formats. It also describes how to implement a block floating-point format in software.

Unsigned or Signed: Two's-complement Format

Unsigned integer numbers are positive, and no sign information is contained in the bits. Therefore, the value of an unsigned integer is interpreted in the usual binary sense. The least significant words of multiple-precision numbers are treated as unsigned numbers.

Signed numbers supported by the Blackfin+ family are in two's-complement format. Signed-magnitude, one's-complement, binary-coded decimal (BCD) or excess-n formats are not supported.

Integer or Fractional Data Formats

The Blackfin+ family supports both fractional and integer data formats. In an integer, the radix point is assumed to lie to the right of the least significant bit (LSB), so that all magnitude bits have a weight of 1 or greater. This format is shown in the ***Integer Format*** figure. Note in two's-complement format, the sign bit has a negative weight.

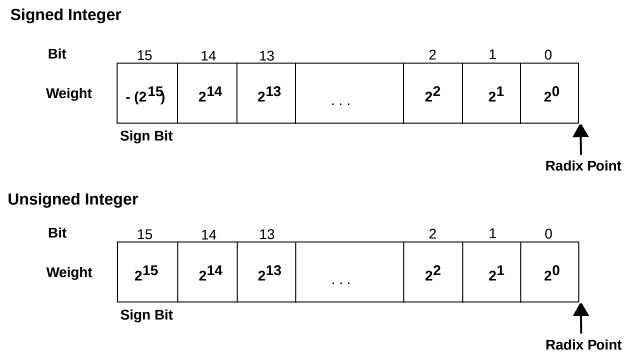


Figure 11-1: Integer Format

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in the ***Example of Fractional Format*** figure, the assumed radix point lies to the left of the three LSBs, and the bits have the weights indicated.

The native formats for the Blackfin processor family are a signed fractional 1.M format and an unsigned fractional 0.N format, where N is the number of bits in the data word and M = N - 1.

The notation used to describe a format consists of two numbers separated by a period (.); the first number is the number of bits to the left of the radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in the ***Example of Fractional Format*** figure is 13.3.

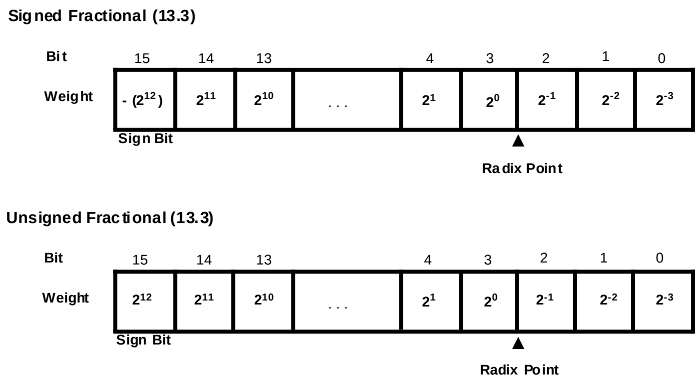


Figure 11-2: Example of Fractional Format

The ***Fractional Formats and Their Ranges*** table shows the ranges of signed numbers representable in the fractional formats that are possible with 16 bits.

Table 11-1: Fractional Formats and Their Ranges

Format	# of Integer Bits	# of Fractional Bits	Max Positive Value (0x7FFF) In Decimal	Max Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	1	15	0.999969482421875	-1.0	0.000030517578125
2.14	2	14	1.999938964843750	-2.0	0.000061035156250
3.13	3	13	3.999877929687500	-4.0	0.000122070312500
4.12	4	12	7.999755859375000	-8.0	0.000244140625000
5.11	5	11	15.999511718750000	-16.0	0.000488281250000
6.10	6	10	31.999023437500000	-32.0	0.000976562500000
7.9	7	9	63.998046875000000	-64.0	0.001953125000000
8.8	8	8	127.996093750000000	-128.0	0.003906250000000
9.7	9	7	255.992187500000000	-256.0	0.007812500000000
10.6	10	6	511.984375000000000	-512.0	0.015625000000000
11.5	11	5	1023.968750000000000	-1024.0	0.031250000000000
12.4	12	4	2047.937500000000000	-2048.0	0.062500000000000
13.3	13	3	4095.875000000000000	-4096.0	0.125000000000000
14.2	14	2	8191.750000000000000	-8192.0	0.250000000000000

Table 11-1: Fractional Formats and Their Ranges (Continued)

Format	# of Integer Bits	# of Fractional Bits	Max Positive Value (0x7FFF) In Decimal	Max Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
15.1	15	1	16383.500000000000000	-16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	-32768.0	1.000000000000000

Binary Multiplication

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location), and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The Blackfin+ family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is shown in the ***Format of Multiplier Result*** figure. The product of two 16-bit numbers is a 32-bit number. If the inputs' formats are M.N and P.Q, the product has the format (M + P).(N + Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

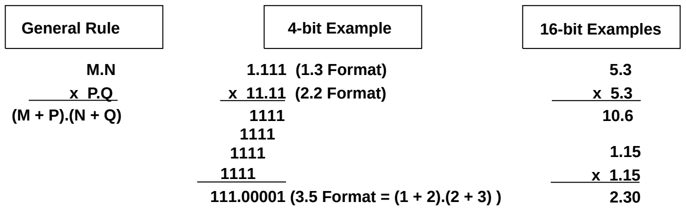


Figure 11-3: Format of Multiplier Result

Fractional Mode And Integer Mode

A product of 2 two's-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.

The Blackfin+ family provides a means (a signed fractional mode) by which the multiplier result is always shifted left one bit before being written to the result register. This left shift eliminates the extra sign bit when both operands are signed, yielding a result that is correctly formatted.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a signed fractional data format, it is most convenient to use the 1.15 format.

Block Floating-Point Format

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. However, some additional programming is required to maintain a block floating-point format.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. A block of fixed-point values can be converted to block floating-point format by shifting each value left by the same amount and storing the shift value as the block exponent.

Typically, block floating-point format allows you to shift out non-significant MSBs (most significant bits), increasing the precision available in each value. Block floating-point format can also be used to eliminate the possibility of a data value overflowing. See the ***Data With Guard Bits*** figure. Each of the three data samples shown has at least two non-significant, redundant sign bits. Each data value can grow by these two bits (two orders of magnitude) before overflowing. These bits are called guard bits.

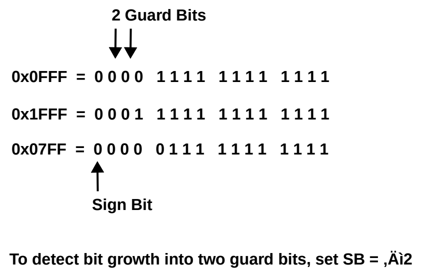

2 Guard Bits
0x0FFF = 0000 1111 1111 1111
0x1FFF = 0001 1111 1111 1111
0x07FF = 0000 0111 1111 1111
Sign Bit
To detect bit growth into two guard bits, set SB = ,Äì2

Figure 11-4: Data With Guard Bits

If it is known that a process will not cause any value to grow by more than the two guard bits, then the process can be run without loss of data. Later, however, the block must be adjusted to replace the guard bits before the next process.

The ***Block Floating-point Adjustment*** figure shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows.

- Assume the output of the `SIGNBITS` instruction is `SB` and `SB` is used as an argument in the `EXPADJ` instruction. Initially, the value of `SB` is +2, corresponding to the two guard bits. During processing, each resulting data value is inspected by the `EXPADJ` instruction, which counts the number of redundant sign bits and adjusts `SB` if the number of redundant sign bits is less than two. In this example, `SB` = +1 after processing, indicating the block of data must be shifted right one bit to maintain the two guard bits.
- If `SB` were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.

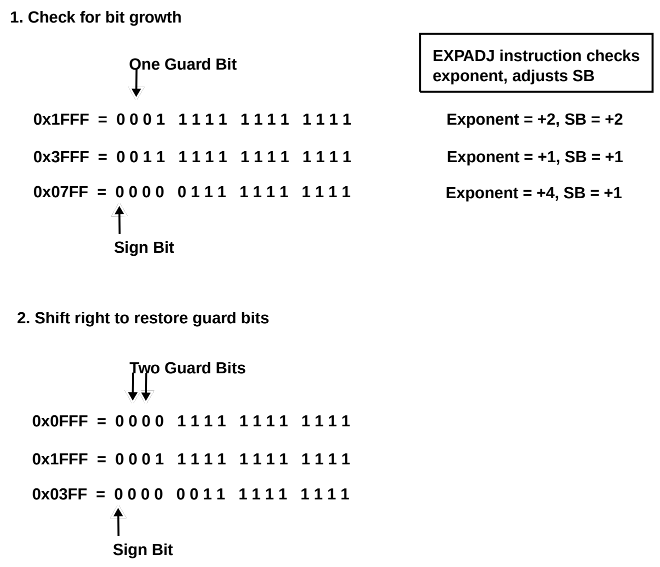

1. Check for bit growth
One Guard Bit
EXPADJ instruction checks exponent, adjusts SB
0x1FFF = 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0x3FFF = 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0x07FF = 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
Exponent = +2, SB = +2
Exponent = +1, SB = +1
Exponent = +4, SB = +1
Sign Bit
2. Shift right to restore guard bits
Two Guard Bits
0x0FFF = 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
0x1FFF = 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0x03FF = 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
Sign Bit

Figure 11-5: Block Floating-point Adjustment

Index