ANALOG DEVICES Engineer To Engineer Note

EE-122

Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

CODING FOR PERFORMANCE ON THE ADSP-219x

To write a DSP algorithm for maximum performance, you must maximize the activity of the ADSP-219x's on-chip resources (such as buses and computation units) within the constraints of the register set size and the instruction set. Optimal coding for an algorithm uses the DM and PM buses, performs a useful computation, and performs one or two useful address operations in each cycle. Under these circumstances the processor is being used at it's highest efficiency. Realistically, it is impossible to achieve that efficiency level.

To help improve processor efficiency, many instructions can be executed in parallel with other instructions. Parallel operation means that more than one operation occurs in a single cycle. Instructions where multiple operations execute in a single cycle are called *multifunction instructions*. Multifunction instructions are critical for optimal performance and should be well understood by every assembly language programmer. There are two rules of thumb worth noting when using multifunction instructions:

1. The degree of register flexibility varies with each instruction type.

2. The more powerful the instruction, the less flexible it is.

This means that some multifunction instructions are very restrictive with permissible input and output registers, but those instructions also maximize use of the processor's resources.

Table 2-1 lists the multifunction operations, listed from most restrictive to least restrictive:

Table 2-1. Multifunction Operations

Instruction Type	Description
1	ALU/MAC with DM&PM dual read using DAGs 1&2 post-modify
4	Multifunction ALU/MAC with DM/PM read or write using DAG post-modify
8, 14	ALU/MAC/Shift & any DREG-to-DREG transfer
9, 16	Conditional ALU/MAC/Shift
9A	Unconditional Register File ALU/MAC

The Type 1 instruction is the least flexible, in terms of register usage, but the most powerful instruction in the instruction set. The computation operations at the top of this list are limited to a specific set of input and output registers. The limitation is a direct result of the fixed, 24-bit opcode. For specific descriptions of multifunction instructions, see the *Instruction Set Reference* chapter.

What Are the Register Restrictions for Multifunction Instructions?

Computation operations fall into two categories: those that accept any DREG as operands, and those that only recognize XOP and YOP. When the operand set is limited, it will generally require the X operand to come from a list of eight XOP registers, and the Y operand to come form a list of three YOP registers. The available registers are determined by the type of operation performed, as illustrated in Table 2-2.

Table 2-2. Available Registers for Multifunction Instructions

Operation	Available XOPs	Available YOPs
ALU	AX0, AX1 AR MR0, MR1, MR2 SR0, SR1	AY0, AY1 AF the value zero
MAC	MX0, MX1 **AR** MR0, MR1, MR2, SR0, SR1	MY0, MY1 **SR1** the value zero
Shifts	SI, SR0, SR1, SR2 AR, AX0, AX1, AY0, AY1, MX0, MX1, MY0, MY1 MR0, MR1, MR2	N/A

Table 2-3 lists the 16 DREGs. Notice that AF, SE, and SB *are not* DREGs.

Table 2-3. Data Registers (DREGs)

DREGs
AX0, AX1, AY0, AY1

MX0, MX1, MY0, MY1
AR, SI
MR2, MR1, MR0
SR2, SR1, SR0

In the ALU, a constant, or third operand type, is often available. These constants are listed in the *Instruction Set Reference* chapter and the *Visual DSP Programmer's Quick Reference for the ADSP-219x*.

Why Is the "Type 1" Instruction So Powerful?

The Type 1 instruction is the most powerful instruction in the instruction set because it performs a computation and uses both memory buses simultaneously. This is the instruction at the core of FIR filters, matrix multiplications, and many other algorithms.

You can combine any of the following operations into a Type 1 instruction:

- A DM read into an X input register (AX0, AX1, MX0, or MX1)
- A PM read into a Y input register (AY0, AY1, MY0, or MY1)
- An ALU or MAC operation (except DIVS, DIVQ, SAT SR/MR, or the bit operations TSTBIT, SETBIT CLRBIT, and TGLBIT)

Type 1 instructions also have the following restrictions:

- The computations must use XOPs and YOPs
- ALU operations must use AR, MAC operations must use MR

You can also perform a dual-memory read and omit the computation as long as you use the destination registers listed above.

How Flexible Is an ALU/MAC Multifunction Instruction?

ALU/MAC instructions have four levels of flexibility for using operands. The level of flexibility depends on the multifunction context, see Table 2-4.

Table 2-4. Operand Flexibility for ALU/MAC Multifunction Instructions

Multifunction Context	Operands	Results

Dual memory reads	XOP and YOP only	AR and MR only
One memory read/write or one register-to-register move	XOP and YOP only	AR/AF or MR/SR
Conditional	XOP, YOP, or constants	AR/AF or MR/SR
Unconditional ALU/MAC operation	Any DREG or constants	AR/AF or MR/SR

All computations are available in all contexts except as follows:

- Bit operations (TSTBIT, SETBIT, CLRBIT, and TGLBIT) are only available as conditional or unconditional instructions; they cannot be executed in parallel with memory or register moves.
- The specialized operations DIVS and DIVQ must stand alone and are unconditional.
- The specialized instructions `IF MV SAT MR` and `IF SV SAT SR` must stand alone and cannot be used with other instructions.

How Flexible Is a Shifter Multifunction Instruction?

Shifter instructions also have four levels of flexibility for using operands. The level of flexibility depends on the multifunction context, see Table 2-5.

Table 2-5. Operand Flexibility for Shifter Multifunction Instructions

Multifunction Context	Operands	Results
Dual memory reads	Shifts not allowed	N/A
One memory read/write	XOP only	SE/SB only
One register-to-register move or conditional	Any DREG	SE/SB only
Unconditional shift	Any DREG	SE/SB or constant

The selection of SE or SB depends on the shifter instruction performed, see the *Instruction Set Reference* chapter for details.

How Do I Use the AF (ALU Feedback) Register?

AF is a special register provided primarily for ALU feedback operations. In these operations, the ALU results are used again in the ALU, as you would, for example, in calculating a running sum. The AF register is complimentary to AR, which is used for results that leave the ALU.

The AF register is the only computation register that is inaccessible to register transfer operations. To read the AF register, the processor must first execute an operation like the following PASS computation:

```
   AF = <some operation>;
   AR = PASS AF;
```

The AF register is useful if you need to reuse an ALU result in a multiplication operation where you are constrained to YOPs. AF is a legal YOP, as shown in the following example:

```
      DO loop UNTIL CE;
   loop:    AF = ax0 + AF, ax0 = dm(i0, m0);
```

How Do I Use the SR1 (MAC Feedback) Register?

SR1 is the MAC feedback register, analogous to AF, but it is available to all register accesses. SR1 is the result register you should use for repeated multiplications or multiply-accumulates, as in the following example, when computing a long product:

```
      SR1 = dm(i0, m1);                         /*first value*/
      mx0 = dm(i0, m1);                         /*second value*/
      DO loop UNTIL CE;
   loop:    SR = mx0 * SR1, mx0 = dm(i0, m1);   /*others*/
```

How Can I Optimize Code with Multifunction Instructions?

The key to optimizing algorithms is to inspect the memory traffic. The first time you execute your algorithm, you must handle every operand and result with a memory transfer. Memory transfers slow the algorithm's execution.

For example, let's say you had code running on an infinitely flexible architecture that could execute any computation in a single cycle with all operands and results in memory, as shown below:

```
   dm(addr) = dm(addr) + dm(addr);
   dm(addr) = dm(addr) + dm(addr);
   dm(addr) = dm(addr) + dm(addr);
   dm(addr) = dm(addr) LSHIFT dm(addr);
```

Each computation is split into three parts: read operands, compute, and write results. These operations are pipelined across three instructions. Code cannot run at full speed because it is memory bus limited.

On approximately alternating cycles, the processor reads two operands, then writes back a result. This bandwidth limitation is the result of available memory buses, however, and not the result of available computation results registers. The resulting code might resemble the following example:

```
/*compute          DM               PM */
               ax0 = dm(i, m),ay0 = pm(i, m);
ar = ax0 + ay0,ax1 = dm(i, m),ay1 = pm(i, m);
               dm(i, m) = ar;        /*bus not available*/
ar = ax1 - ay1,mx0 = dm(i, m),my0 = pm(i, m);
               dm(i, m) = ar;        /*bus not available*/
mr = mx0 * my0,si = dm(i, m);
```

```
                dm(i, m) = mr1;          /*bus not available*/
                se = dm(i, m);
sr = LSHIFT si;
                dm( 1, m) = sr1;         /*bus not available*/
```

For the first three computations, the code computes in every other cycle (cycle 2N), reading its operands in the even cycles (2N - 2) and writing its results in odd cycles (2N + 1).

Why not read SE and SI when it's time to LSHIFT? As mentioned earlier, the dual-read multifunction instruction (Type 1) cannot provide total flexibility in the choice of the two registers read from memory. Since SE and SI aren't on the list of permissible registers, they must be read one at a time. Also SE isn't one of the DREGs and data transfers with SE cannot be executed in parallel with a computation. (This isn't usually a problem since in an inner loop, the shift distance contained in SE is often a constant number of bits.)

How Can I Optimize Code By Reducing Memory Traffic?

The statistics for the previous example are:

Executed	Out of
4 computations	10 cycles
12 memory accesses	20 possible memory accesses

which is equal to -16/30, or 55% utilization of these three resources.

A better measure of the processor's efficiency is to count memory writes as two accesses. Since writes cannot be performed in parallel, both buses are "used," so to speak, in any memory write. By this measure, the above code used 20/30, or 66% of its resources.

To improve the efficiency, examine the data more closely for ways to reduce the memory traffic on each pass through the loop. Often, when you use an operand more than once, you can avoid multiple reads of the same operand. You might also save operand reads by reordering the loop, running it backwards, or executing the additions for iterations [i] alongside the subtractions for iteration [i + n/2], or something similar.

To further optimize your code, look for results that are fed into subsequent operations. Keeping these results in registers has an advantage if you can eliminate a read and a write.

Let's say that the inspection of the above example revealed the following data relationships:

Relationship	# of Accesses Saved
Shift exponent SE is a constant	1
Shift result of D at index [i] is one input to C at index [i + 1]	3
Additions at index	

[i] use the same X input as the subtractions at index [N - 1]	1

Count writes as two "access slots," so saving a read and a write, as in the second relationship, saves three accesses. As a guess, we've reduced the memory traffic from 16 to 11, so the minimum achievable cycle count is now 11/2 = 5.5, or 6 cycles.

Here's the results from taking advantage of the first two relationships:

```
/*compute                 DM                       PM */
                    ax0 = dm(i, m),          ay0 = pm(i, m);
ar = ax0 + ay0,     ax1 = dm(i, m),          ay1 = pm(i, m);
                    dm(i, m) = ar;           /*bus not available*/
ar = ax1 - ay1,     mx0 = dm(i, m),          my1 = dm(i, m);
                    dm(i, m) = ar;           /*bus not available*/
mr = mx0 * sr1,     dm(i, m) = mr1;          /*bus not available*/
sr = LSHIFT my1;
```

We've implemented the following changes:

- Don't read SE, set it up ahead of time.
- Instead of SI, use AY1 as the shift input operand, since we can load AY1 in a dual read, but not SI. Now the load of the shift operand can be moved up to the fourth instruction.
- Don't write the shifter result SR1 at the seventh instruction. Instead, leave it in place to be used in the next loop at instruction 6.

We're one instruction away from the optimal six instructions. Can we reduce the number of instructions by taking advantage of the third relationship? We'll make the final optimization in the code in two steps. First, eliminate the read of AX1, since we'll share that with the read of AX0. (The following example doesn't show the changes we'd make to the associated index registers to make the computations occur in a different order.) Here's the results so far:

```
/*compute                 DM                       PM */
                    ax0 = dm(i, m),          ay0 = pm(i, m);
ar = ax0 + ay0,                              ay1 = pm(i, m);
                    dm(i, m) = ar;           /*bus not available*/
ar = ax0 - ay1,     mx0 = dm(i, m),          ay1 = dm(i, m);
                    dm(i, m) = ar;           /*bus not available*/
mr = mx0 * sr1,     dm(i, m) = mr1;          /*bus not available*/
sr = LSHIFT ay1;
```

Now it's apparent that the last instruction has no memory accesses. If we could move another memory access down from the second instruction, for example, then we could combine the first two instructions into a multifunction instruction. Unfortunately, there is a problem: the shifter operand in the seventh instruction is MY1, and that operand is not available in a shifter instruction (it's not on the XOP list).

At this point, to execute a shift in parallel with another instruction, we should determine what operands are available in multifunction instructions. Fortunately, the XOP list for the shifter includes AR. But AR is not on the list of registers that can be read from memory in a dual-read Type 1 instruction. How do we move the data from AY1 to AR?

The answer is the PASS computation. The PASS is the same as a register-to-register transfer that can, in this example, be used in parallel, as shown below:

```
/*compute                   DM                       PM */
ar = ax0 + ay0,     ax0 = dm(i, m),         ay0 = pm(i, m);
                    dm(i, m) = ar;          /*bus not available*/
ar = ay0 - ay1,     mx0 = dm(i, m),         ay1 = dm(i, m);
ar = PASS ay1,      dm(i, m) = ar;          /*bus not available*/
mr = mx0 * sr1,     dm(i, m) = mr1;         /*bus not available*/
sr = LSHIFT ar;                             ay1 = pm(i, m);
```

Now the only cleanup that remains is to correct the indexing to compensate for memory accesses that were moved forward or backward in the loop. In the above example, the registers in **bold type** are loaded for the next loop iteration. The index registers in those accesses must be initialized with a value incremented by 1, and the associated data registers must be loaded before the top of the loop with the previous loop's data.

How Can I Optimize my Code By Taking Advantage of the 6-Stage Pipeline?

The ADSP-219x has a six stage pipeline comprising the Look-ahead, Prefetch, Fetch, Address generation, Decode, and Execute Stages. The depth of the pipeline allows an increase in operating speed for the processor core.
The ADSP-219x programming model now gives a programmer the option (which didn't exist on the ADSP-218x) to use delayed branches and function calls. Using delayed branches two instructions can now be executed in the pipeline while pending the branch/call. Similar to delayed branches, during RTI or RTS, the number of non-productive cycles can be reduced from 4 to 2 with the use of a delayed return. Please refer to *EE-123: An Overview of the ADSP-219x Pipeline* for the restrictions on using delay slots.

Do-until loops create an interesting case for the pipeline depending on the number of iterations inside the loop, as well as the type of instructions that occur inside the loop. Short loops, less than 5 instructions deep and assuming none of these instructions is of Type 1, incur a one-time overhead of 4 cycles during loop entry. Do-until loops that are longer than 5 instructions deep have no overhead, assuming they also do not contain instructions of Type-1.

The effects of the pipelined instructions become more interesting for short Do-until loops that contain Type 1 instructions, as shown in the following example.

```
.
.
CNTR.=20;
DO sop UNTIL CE;
sop: MR=MR+MX0*MY0 (SS),MX0=DM(I0,M1), MY0=PM(I4,M5);
.
.
```

In this case, pipeline conflicts occur because the instruction also uses the PMA and PMD buses, and the instruction cache comes into the picture, resulting in more cycles. There are also latencies depending on the number of loop iterations.

Loop unrolling is one way to optimize the short loop. Using the previous short loop example, shown in the following code segment is an example of loop unrolling.

```
.
.
CNTR.=4;
DO sop UNTIL CE;
     MR=MR+MX0*MY0 (SS),MX0=DM(I0,M1), MY0=PM(I4,M5);
     MR=MR+MX0*MY0 (SS),MX0=DM(I0,M1), MY0=PM(I4,M5);
     MR=MR+MX0*MY0 (SS),MX0=DM(I0,M1), MY0=PM(I4,M5);
     MR=MR+MX0*MY0 (SS),MX0=DM(I0,M1), MY0=PM(I4,M5);
sop: MR=MR+MX0*MY0 (SS),MX0=DM(I0,M1), MY0=PM(I4,M5);
.
.
```

Unrolling loops results is more instructions inside your loop, with less loop iterations. As a result, code space is increased and depending on available memory, this may not be a desirable alternative.

For a more detailed description of Pipeline usage for Jumps/Calls, RTI/RTS, and Do-until Loops, please refer to *EE-123: An Overview of the ADSP-219x Pipeline*.