Technical notes on using Analog Devices DSPs, processors and development tools
Contact our technical support at processor.support@analog.com and dsptools.support@analog.com
Or visit our on-line resources http://www.analog.com/ee-notes and http://www.analog.com/processors

Extended-Precision Fixed-Point Arithmetic on SIMD SHARC® Processors

Contributed by Brian M.

Rev 1 – July 7, 2005

Introduction

The SHARC® processor family was designed to efficiently perform 32-bit and 40-bit floating-point operations, and 32-bit fixed-point operations. There are times when it is desirable to perform arithmetic computations with greater precision.

Since the second and third generation SHARC processors have two 64-bit buses, and four 80-bit accumulators, the SHARC processors can be used to perform extended-precision 64-bit arithmetic.

This document describes how to implement extended-precision operations. It also discusses how the SHARC architecture performs long word (64-bit) accesses in SIMD mode. Lastly, it discusses the implementation of an extended-precision FIR and IIR filter. Source files for the example filters are provided in a separate ZIP file. Note that this application note is an adaptation of the discussion found in *Extended-Precision Fixed-Point Arithmetic on the Blackfin® Processor Platform (EE-186)* [4], and many of the same figures appear in this document.

Background

64-bit extended-precision arithmetic is a natural software extension of the native 32-bit fixed-point operations provided by SHARC processors. Since SHARC processors have 32-bit register files, two registers are used to represent one 63-bit or 64-bit fixed-point number. Second- and third-generation SHARC family have 64-bit buses, allowing the transfer of two whole 64-bit words per cycle. Before examining specific DSP algorithms, it is important to understand how to perform basic 64-bit (long-word) accesses, how long word accesses work in SIMD mode, how the VisualDSP++® tools work with 64-bit data, and how basic arithmetic operations can be implemented with extended precision.

64-bit Data in VisualDSP++ Tools

Before discussing the actual algorithms, it is helpful to understand how to initialize 64-bit data in the VisualDSP++ development tools. The VisualDSP++ linker provides two different section types to directly initialize 64-bit memory. Listing 1 shows both types of initialization. When a section is marked as `PM` or `DM` with a width of 64-bits (in the LDF file), the preprocessor packs two 32-bit words into a single 64-bit word. When a section is marked as `DATA64`, the preprocessor will take a single 64-bit word for each location to be initialized.

```
.section/DATA64 seg_data64;
// starting at 0x5C100
.var coeffs1[]= 0x123456789ABCDEF0;

.section/DM seg_dm64;
// starting at 0x60000
.var coeffs2[]= 0x55555555,0xAAAAAAAA;
```

Listing 1. Initializing 64-bit Memory

In VisualDSP++ 4.0 (and earlier), it is necessary to use only hexadecimal, octal, or binary

representation when initializing DATA64 memory, VisualDSP++ truncates any other format to 32 bits. In addition, all initializers must be zero-padded to contain exactly 16 hexadecimal numbers for reliable behavior.

Long-Word Accesses

Second-generation (and later) SHARC devices have two 64-bit buses, each of which can perform a single 64-bit transfer per cycle. First-generation SHARC DSPs (ADSP-2106x) do not have 64-bit buses. Since all of the registers in the register file are 32 bits wide, a single long-word read or write accesses two neighbor registers — the register defined explicitly in the instruction and the register implicitly defined by the relationships shown in Table 1.

PEx neighbor registers	PEy neighbor registers
r0 and r1	s0 and s1
r2 and r3	s2 and s3
r4 and r5	s4 and s5
r6 and r7	s6 and s7
r8 and r9	s8 and s9
r10 and r11	s10 and s11
r12 and r13	s12 and s13
r14 and r15	s14 and s15

Table 1. Long-Word Register Pairs

When the address of an access is to long-word space (as defined by the processor being used), the processor places the lower 32 bits of the long word into the named (explicit) register and the upper 32 bits into the neighbor (implicit) register.

```
r4=dm(coeffs1);  // neighbor r5 is accessed
r8=pm(coeffs2);  // neighbor r9 is accessed
```

It is also possible to perform a long-word access to a normal-word (32-bit) address using the (LW) modifier in the instruction. For these accesses, the data at the even normal-word address corresponds to the explicitly defined register, and the odd normal-word address corresponds to the implicitly defined neighbor register. The following fetch performs the same operation as the first example above.

```
r4=dm(0xB8200)(LW);//implicit r5=dm(0xB8201)
r8=pm(0xC0000)(LW);//implicit r9=pm(0xC0001)
```

Dual-data accesses use both available buses to fetch data in a single cycle. Since both buses are 64-bits wide, we will take advantage of dual-data accesses when implementing filter code. However, for dual-data accesses, it is not possible to use the LW modifier. The only way to perform a long-word transfer is to use the DAGs to access a long-word address.

```
r4=dm(i0,m0),r8=pm(i8,m8);
// neighbors r5 and r9 are accessed as well
```

All example code in this application note will place the least significant 32-bits into the lower (even) neighbor register and the most significant 32-bits into the higher (odd) neighbor register.

SIMD Long-Word Accesses

In order to use both processing elements in the SIMD SHARC processors, it is necessary to fill both the explicitly referenced register file in PEx, and the implicitly defined register set in PEy. For normal 32-bit word accesses, the SIMD capabilities of the SHARC architecture automatically use the 64-bit bus to fill both register files. However, for long-word accesses, there is no space on the bus for the processor to automatically perform this access. Instead, the SHARC processor performs only the explicit transfer to the neighbor pair, as defined by the use of an Rn register for PEx and an Sn register for PEy.

```
r0=dm(coeffs1);  // neighbor r1 is accessed
s6=pm(coeffs2);  // neighbor s7 is accessed
```

Dual-data accesses take this one step further in SIMD mode. When both buses are being used for long-word accesses, the SIMD core transfers data on the DM bus to PEx, and data on the PM bus to PEy. The only exception to this is when broadcasting is enabled for I1 or I9 (BDCST1 or BDCST9 is set in MODE1). In this case, the same data is sent to both processing elements.

```
r4=dm(i0,m0),r8=pm(i8,m8);
```

- r4 and r5 written with address in i0
- s8 and s9 written with address in i8

In broadcast mode:

```
r4=dm(i1,m0),r8=pm(i9,m8);
```

- r4, r5, s4, s5 written with address in i1
- r8, r9, s8, s9 written with address in i8

Now that we have explored the way SHARC processor permit interaction between registers and long-word memory, we can explore implementing the extended-precision algorithms.

Addition/Subtraction

The SHARC processor instruction set does not contain a single-cycle 64-bit addition. This addition must be implemented in two steps, using four registers from the data file and the Carry-In from the addition of the lower 32 bits of the 64-bit words being added.

```
r4=r4+r8;
r5=r5+r9+CI;
```

Subtraction of 64-bit numbers is also implemented in two steps, using the Borrow from the lower 32-bits of the 64-bits being added.

```
r4=r4-r8;
r5=r5-r9+CI-1;
```

In these addition and subtraction instructions, note that any combination of data registers can be used. Refer to the discussion about long-word reads for an explanation of how to choose registers. To illustrate the high and low half of the 64-bit arithmetic shown above, the 64-bit values from the above sections would fall into the registers as follows:

```
r4=0x9ABCDEF0     r8=0x55555555
r5=0x12345678     r9=0xAAAAAAAA
```

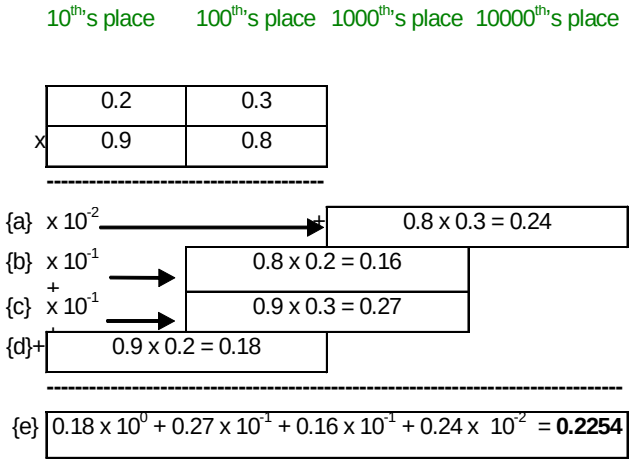

Figure 1. Decimal Multiplication in Detail

More information on the native SHARC operations can be found in the *ADSP-21160 SHARC DSP Instruction Set Reference*. [1]

Multiplication

In order to introduce the concept of extended-precision multiplication, it is useful to review the already familiar decimal multiplication.

Two-Digit Decimal Multiplication

Let's start by recalling how any decimal multiplication can be performed by knowing how to multiply single-digit numbers. As an example, consider this two-digit by two-digit decimal multiplication:

```
0.23 x 0.98 = 0.2254
```

Figure 1 illustrates how this particular operation can be broken down into smaller operations. This is basically multiplication "by hand". To compute the final result, the following operations are necessary:

- Four single-digit multiplications (lines {a}, {b}, {c}, and {d} in Figure 1)

  ```
  0.8 x 0.3 = 0.24, 0.8 x 0.2 = 0.16,
  0.9 x 0.3 = 0.27, 0.9 x 0.2 = 0.18
  ```

- Three operations to shift the sub-products into the correct digit-significant slot (lines {b}, {c}, and {d} in Figure 1)

  0.24×10^{-2}, 0.27×10^{-1}, 0.16×10^{-1}

- Three additions (line {e} in Figure 1)

$0.18x10^{0} + 0.27x10^{-1}$,
$0.16x10^{-1} + 0.24x10^{-2}$,
$(0.18x10^{0} + 0.27x10^{-1}) + (0.16x10^{-1} + 0.24x10^{-2})$

Two-Digit Hexadecimal Multiplication

Hexadecimal multiplication is not much different from its decimal counterpart. Let's consider a multiplication of two 64-bit fractional numbers, where the operands are stored in the 32-bit data registers R4, R5, R8, and R9.

64-Bit Accuracy with 32-Bit Multiplications

One can use elementary arithmetic to achieve a 64-bit multiplication result with single-cycle 32-bit multiplications.

Each of the two 64-bit operands are split into two 32-bit halves (R4 and R5 for the first operand, R8 and R9 for the second), as shown in Figure 2.

From this figure, it is easy to see the operations required to emulate the 64-bit multiplication with a combination of instructions using 32-bit multiplies:

- Four 32-bit multiplications to yield four 64-bit results (lines {a}, {b}, {c}, and {d} in Figure 2)

```
R8 x R4, R8 x R5, R9 x R4, R9 x R5
```

- Three operations to shift the sub-products into the correct digit-significant slot (lines {a}, {b}, and {c} in Figure 2). Since we are performing fractional arithmetic, the result is 1.127 (1.63 x 1.63 = 2.126 with a redundant sign bit). Most of the time, the result can be truncated to 1.63 to fit a native 64-bit data register. Therefore, the result of the multiplication should be in reference to the sign bit (or the most significant bit). This way, the rightmost least significant bits can be safely discarded by truncation.

```
(R8 x R4)>>64, (R8 x R5)>>32,
(R9 x R4)>>32
```

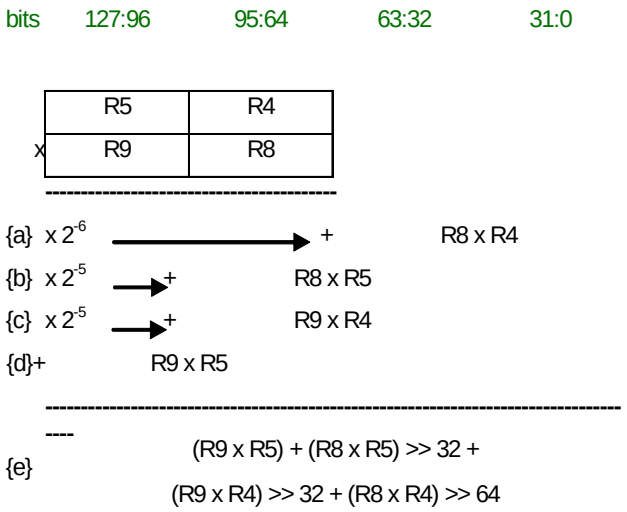

Figure 2. Hexadecimal Multiplication in Detail

- Three operations to preserve bit place in the final answer (line {e} in Figure 2):

```
(R8 x R4)>>64 + (R8 x R5)>>32,
(R9 x R4)>>32 + R9 x R5,
((R8 x R4)>>64 + (R8 x R5)>>32) +
((R9 x R4)>>32 + R9 x R5)
```

The final expression for a 32-bit multiplication is:

```
((R8 x R4)>>64 + (R8 x R5)>>32)
+ ((R9 x R4)>>32 + R9 x R5)
```

63-Bit Accuracy with 32-Bit Multiplication

From Figure 2, it is easy to see that the multiplication of the least significant half-word (R8 x R4) does not contribute much to the final result. In fact, if the final result is ultimately truncated to 1.63, this multiplication can affect only the least significant bit of the 1.63 result. For many applications, the loss of accuracy due losing to this bit is offset by the performance increase over the 64-bit multiplication. Three operations (one 32-bit multiplication, one shift, and one addition) can be eliminated if 63-bit accuracy is acceptable in the final design:

```
((R8 x R4)>>64 + (R8 x R5)>>32)
+ ((R9 x R4)>>32 + R9 x R5)
```

The remaining instructions necessary to obtain a 63-bit-accurate 1.63 answer are three 32-bit multiplications, two additions, and a shift:

```
((R8 x R5)>>32) + ((R9 x R4)>>32 + R9 x R5)
```

Further rearrangement of terms yields the final form of 63-bit-accurate multiplication:

```
((R8 x R5) + R9 x R4)>>32 + (R9 x R5)
```

Filter Implementation

The following example filters apply the 64-bit precision multiplications. For ease of reading, only the kernels of the SISD examples are shown. However, more efficient SIMD implementations are included in the attached file and are briefly discussed below.

Double-Precision FIR Filter Implementation

If we consider `R0/R1` to be the data value and `R2/R3` to be a coefficient value, then each multiplication in the FIR will be of the form described earlier:

```
((R0 x R2)>>64 + (R0 x R3)>>32)
+ ((R1 x R2)>>32 + R1 x R3)
```

The kernel for a 64-bit-accurate FIR implementation is shown in Listing 2. The number of cycles needed to execute the implementation is `N*(4*(T-1)+20)`, where `N` is the size of the input buffer and `T` is the number of filter taps.

SIMD FIR Filter Implementation

Since there are two sets of multipliers available in SHARC processors, it is worth the effort to take advantage of both sets in an efficient implementation.

There are two ways to take advantage of the second processing element. The most obvious is to operate on two sets of data at the same time and have each processing element deal exclusively with a single channel. However, for an FIR filter, it is possible to split the input into two halves and treat the halves as two channels of data that use the same filter taps (i.e., broadcast mode).

For an extended-precision implementation, the limitations (e.g., placement of data in different memory blocks, and interleaving the input data and coefficients) imposed by normal-precision (32-bit) SIMD operations do not exist, since the long-word reads fill all available bandwidth. In fact, the only restriction on the placement of the data in these examples is that the coefficients mustto be in a different block of memory than the delay line and input data; this is necessary to avoid block conflicts when fetching and writing back data.

This allows the algorithm to access two different points in the input without sacrificing performance or rearranging the data. The example SIMD code is set up for the split-input method, but it can be changed easily to handle two separate channels of data by modifying the input pointer and increasing the loop count to cover an entire block rather than half.

The 64-bit buses will allow the transfer of only one data value and one coefficient value per cycle. In order to use SIMD, it is necessary to fetch at least two data values and one coefficient per multiply. Therefore, it is necessary to perform two sets of reads per MAC instruction (i.e., a 64-bit MAC consists of two instructions).

```
mrb=mrb+r3*r0(suf), r0=dm(i0,m7),
    r2=pm(i9,m14);

r2=dm(i1,m6),r0=pm(i8,m15);
```

Since SIMD long-word dual-data accesses are split between the processing elements, the second read in the SIMD MAC uses the same registers accessed by the opposite bus, as can be seen in the example above. This sequence of instructions is found frequently in both the SIMD FIR and SIMD IIR code that is attached.

Complete source code for 64-bit-accurate FIR and IIR filters is contained in the accompanying compressed package.

fir64

```
fir64_start:
    //Multiply least significant halves with taps in MRF
    mrf=r0*r2 (uuf), r0=dm(i0,m7), r2=pm(i9,m14);
    lcntr=NUM_COEFFS-1, do m_st until lce;
    m_st: mrf=mrf+r0*r2 (uuf), r0=dm(i0,m7), r2=pm(i9,m14);

    //Copy from most significant 32-bits of the low half multiplication
    //Store Output from last pass through the loop
    r6=mr1f, dm(i5,m6)=r4;
    mr0b=r6;
    r7=0;
    mr1b=r7;

    //Cross multiply upper and lower halves in MRB
    //Multiply upper halves in MRF
    mrf=r1*r3(ssf);
    mrb=mrb+r1*r2(suf);

    lcntr=NUM_COEFFS-1, do MAC until lce;
        mrb=mrb+r3*r0(suf),r0=dm(i0,m7), r2=pm(i9,m14);
        mrb=mrb+r1*r2(suf);
    MAC:    mrf=mrf+r1*r3(ssf);
    mrb=mrb+r3*r0(suf),r2=dm(i0,m6);

    //Collect MRB and MRF into registers for final addition
    //Read next input
    r4=mr0f,r0=dm(i4,m6);
    r5=mr1f;
    r6=mr0b;
    r7=mr1b;

    //Shift result of mixed multiplies by 31 bits (don't need sign bit)
    r6=lshift r6 by -31;
    r6=r6 OR lshift r7 by 1;
    r7=lshift r7 by -31;

    //Final 64-bit addition of MRB and MRF
    //Store input to DL, Read first tap
    r4=r4+r6,dm(i0,m7)=r0, r2=pm(i9,m14);
fir64_end:  r5=r5+r7+CI;
```

Listing 2. 64-bit FIR

64-Bit-Accurate IIR Filter

Again, we consider `R0/R1` to be the data value and `R4/R5` to be a coefficient value; each multiplication in the IIR will be of the form described earlier:

```
((R0 x R4)>>64 + (R0 x R5)>>32)
+ ((R1 x R4)>>32 + R1 x R5)
```

The kernel for a 64-bit-accurate IIR implementation (a single biquad) is shown in Listing 3. The number of cycles needed to execute the implementation is `N*33`, where `N` is the size of the input buffer.

The SIMD considerations for this algorithm are similar to those mentioned for the FIR implementation. For this algorithm, however, it is not possible to split the input into two halves. Therefore, the included SIMD code can only be used to filter two channels of data simultaneously. Again, the only restriction on the placement of the data in this example is that the

coefficients need to be in a different block of memory than the delay-line and input data.

iir64

```
iir64_start:
    //Multiply lower halves in MRF
    //Read X-2, B2
    mrf=mrf + r0*r4 (uuf), r0=dm(i0,m6), r4=pm(i9,m14);
    //Read A1, Adjust DL pointer
    mrf=mrf + r0*r4 (uuf), r0=dm(i0,m6), r4=pm(i9,m14);
    //Read Y-1, A2
    mrf=mrf - r12*r4 (uuf), r0=dm(i0,m7), r4=pm(i9,m14);
    //Read Y-2, B0
    mrf=mrf - r0*r4 (uuf), r0=dm(i0,m7), r4=pm(i9,m14);

    //Copy from most significant 32-bits of the low half multiplication
    r7=0;
    r6=mr1f;
    mr1b=r7;
    mr0b=r6;

    //Cross multiply upper and lower halves in MRB
    //Multiply upper halves in MRF
    //Read Y-1
    mrf=r3*r5 (ssf);
    mrb=mrb+r3*r4 (suf),r0=dm(i0,m7);
    //Read X-1, B1
    mrb=mrb+r5*r2 (suf),r0=dm(i0,m5),r4=pm(i9,m14);
    //Update X-1
    mrf=mrf+r1*r5 (ssf);
    mrb=mrb+r1*r4 (suf), dm(i0,m6)=r2;
    //Read X-2, B2
    mrb=mrb+r5*r0 (suf), r2=dm(i0,m5),r4=pm(i9,m14);
    //Update X-2
    mrf=mrf+r3*r5 (ssf);
    mrb=mrb+r3*r4 (suf), dm(i0,m6)=r0;
    //Read Y-1, A1
    mrb=mrb+r5*r2 (suf), r0=dm(i0,m6),r4=pm(i9,m14);
    //Store output
    mrf=mrf-r1*r5 (ssf), dm(i5,m6)=r12;
    //Read Y-2
    mrb=mrb-r1*r4 (suf), r2=dm(i0,m5);
    //Update Y-2, read A2
    mrb=mrb-r5*r0 (suf), dm(i0,m7)=r0, r4=pm(i9,m14);
    //Modify DL Pointer
    mrf=mrf-r3*r5 (ssf), modify(i0,m7);
    //Modify DL Pointer
    mrb=mrb-r3*r4 (suf), modify(i0,m7);
    //Read X-1
    mrb=mrb-r5*r2 (suf), r0=dm(i0,m6);

    //Collect MRB and MRF into registers for final addition
    //Modify DL Pointer
    r12=mr0f,modify(i0,m6);
    r13=mr1f;
    r6=mr0b;
    r7=mr1b;
```

```
    //Shift result of mixed multiplies by 31 bits
    r6=lshift r6 by -31;
    r6=r6 OR lshift r7 by 1;
    r7=lshift r7 by -31;

    //Final 64-bit addition of MRB and MRF
    //Read next input, B0
    r12=r12+r6,r2=dm(i4,m6), r4=pm(i9,m14);
    r13=r13+r7+CI;

    //Update Y-1, read B1
iir64_end: mrf=r2*r4 (uuf), dm(i0,m7)=r12, r4=pm(i9,m14);
```

Listing 3. 64-bit Biquad (IIR)

Summary

This application note describes an effective method for implementing extended-precision arithmetic on SHARC processors. The arithmetic itself is not the only issue that must be considered when implementing these algorithms. In addition to the arithmetic computation, it is necessary to consider how long-word accesses and SIMD operations work on the extended data. After introducing extended-precision addition, subtraction, and multiplication, this document discussed the use of these operations to implement FIR and IIR filters. Table 2 summarizes the performance of the FIR and IIR filters found in the compressed package supplied with this document.

Specifically, this note applies to any of the SHARC devices in the ADSP-2116x, ADSP-2126x, and ADSP-2136x families. While the projects and cycle counts provided are specific to the ADSP-2136x family, the filter code does not need to be changed to be ported to another SIMD SHARC processor. The only required changes are to the .LDF file, since all of these processors are code compatible and have identical internal memory bus structures.

	64-bit Accuracy	**SIMD 64-bit Accuracy**
FIR	`N*(4*(T-1)+20) cycles°`	`(N/2 + T/2)*(6*(T-1)+22) cycles°` `N*(6*(T-1)+22) cycles°°`
IIR	`33*N cycles°`	`43*N cycles°°`
`°Cycles for one N-point channel of data` `°°Cycles for two N-point channels of data`		

Table 2. Computation Time for 64-bit Filter Implementations

References

[1] *ADSP-21160 SHARC DSP Instruction Set Reference.* Rev 2.0, November 2003. Analog Devices, Inc.

[2] *ADSP-2136x SHARC Processor Programming Reference.* Rev 0.3, February 2005. Analog Devices, Inc.

[3] *ADSP-21161 SHARC Processor Hardware Reference.* Rev 4.0, February 2005. Analog Devices, Inc.

[4] *Extended-Precision Fixed-Point Arithmetic on the Blackfin Processor Platform (EE-186).* Rev 3. May 2003. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – July 07, 2005 *by Brian M.*	Initial Release