AN2208
Rev. 2, 12/2004

Implementing a Double-Precision (32-Bit) Complex FIR Filter on the MSC8101 Device

By Tina Redheendran

This document describes an optimized assembly code implementation of a double-precision complex FIR filter on the MSC8101 device. The MSC8101 SC140 core contains four multiply-accumulate (MAC) units, each capable of 16-bit by 16-bit multiplication. The SC140 core includes special instructions to implement multi-precision arithmetic if a higher precision than 16-bits is needed. This document shows how to use these instructions to perform double-precision or 32-bit multiplies. It also discusses complex multiplications and the register restrictions with the multi-register move and multi-precision instructions, showing how these restrictions can affect code development. Finally, the filter code is described and presented. It is assumed that you are familiar with filter algorithms. The code presented here is based on the code from the StarCore™ SC140 core library. This code is available on the MSC8101 product summary page at the Freescale Semiconductor web site listed on the back of this application note.

CONTENTS

1 Double-Precision Multiply

The MSC8101 uses five special instructions for double-precision multiplication:

mpysu/macsu	Fractional multiplication and multiply-accumulate with signed × unsigned operands.
mpyus/macus	Fractional multiplication and multiply-accumulate with unsigned × signed operands.
mpyuu/macuu	Fractional multiplication and multiply-accumulate with unsigned × unsigned operands.
dmacss	Fractional multiplication with signed × signed operands and 16-bit arithmetic right shift of the accumulator before accumulation.
dmacsu	Fractional multiplication with signed × unsigned operands and 16-bit arithmetic right shift of the accumulator before accumulation.

Figure 1 shows the use of these instructions. In this figure, two 32-bit operands, in D0 and D1, are to be multiplied. The **mpyuu** instruction multiplies the unsigned low portion of D0 with the unsigned low portion of D1 and places the result in D2. The **dmacsu** instruction multiplies the signed high portion of D0 with the unsigned low portion of D1, shifts the result to the right by 16-bits, and adds this result to the value in D2. The **macus** instruction multiplies the unsigned low portion of D0 with the signed high portion of D1 and adds this result to the value in D2. Finally, the **dmacss** instruction multiplies the signed high portion of D0 with the signed high portion of D1, shifts the result to the right by 16-bits, and adds this result to the value in D2. This gives the 32-bit result of the multiplication of the contents of registers D0 and D1 in register D2.

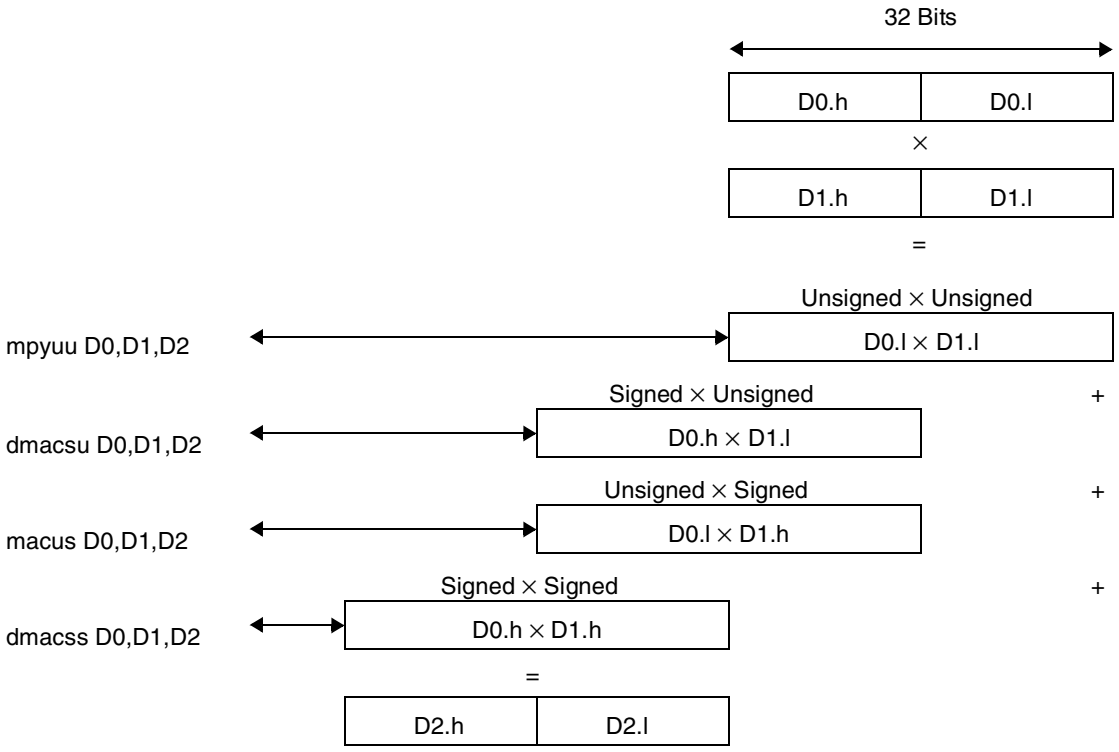

Figure 1. Double-Precision Multiplication

In order to implement a complex double-precision multiplication, four real double-precision multiplies, one 32-bit addition, and one 32-bit subtraction are needed, as shown here, where *Re(x)*, *Im(x)*, *Re(h)*, and *Im(x)* are all 32-bit values:

$$(Re(x)+jIm(x)) \times (Re(h)+jIm(h)) = Re(x)Re(h) - Im(x)Im(h) + j(Re(x)Im(h) + Im(x)Re(h)),$$

There are some register restrictions on the input registers of the double-precision instructions (the first two registers in the instruction). The input registers must be one of the following register pairs:

- D0,D1
- D2,D3
- D4,D5
- D6,D7
- D8,D9
- D10,D11
- D12,D13
- D14,D15

For example, the **mpyuu** D0,D2,D4 instruction is not allowed because D0,D2 is not a register pair. This restriction, along with the restrictions of the multi-register move instructions, affects the main loop of the filter code, as described in **Section 2.2**.

2 Filter Code Description

This section describes how the input arguments are passed to the code and how the code implements the filtering operation using the double-precision multiply described in **Section 1**.

2.1 Input Arguments

A calling routine passes the input arguments as follows to the code so that the code can identify where the data is located and determine how to process the data:

1. Write the input argument values into memory.
2. Set the r0 register to the beginning of the arguments.
3. Call the filter code.

Table 1 shows the input arguments for the code and their lengths.

Table 1. Input Arguments

Argument	Definition	Length (bits)
Output Address	The starting address in memory where the code stores the output values of the filter operation.	32
Input Address	The starting address in memory where the input samples to be processed are stored.	32
Coefficient Address	The starting address in memory where the filter coefficient values are stored.	32

Table 1. Input Arguments (Continued)

Argument	Definition	Length (bits)
Delay Base	The starting address in memory of the delay buffer. The delay buffer is where the code stores the data taps for the filter.	32
Delay Current	Points to the next address in the delay buffer where filter processing will occur. This allows the filter code to be called multiple times while continuing to process the same filter.	32
Number of Samples	The number of complex samples to be processed by the filter.	16
Number of Coefficients	Defines the size of the filter or the number of complex coefficients in the filter.	16

All data for the code—including the input values, output values, coefficient values, and the values in the delay buffer—are complex values with the real part stored first followed by the complex part. The first step for the code is to extract the input arguments needed to complete the filter. The arguments reside at the memory location to which the r0 register points. After the input arguments are read, the code begins the filter processing.

2.2 Filter Algorithm

After the code reads the input arguments from memory, it begins processing the input data based on the following equation:

$$y(n) = \sum_{i=0}^{n-1} [Re(x(n-i)) + jIm(x(n-i))][Re(h(i)) + jIm(h(i))]$$

Figure 2 shows a simplified block diagram of the program flow. The filter code takes the two values of input data, the real part and the imaginary part, from the input buffer and writes them to the delay buffer. The complex data from the delay buffer and coefficient buffer are combined based on the above equation to create the complex filter result. The real part and the imaginary part of the filter result are written to the output buffer.

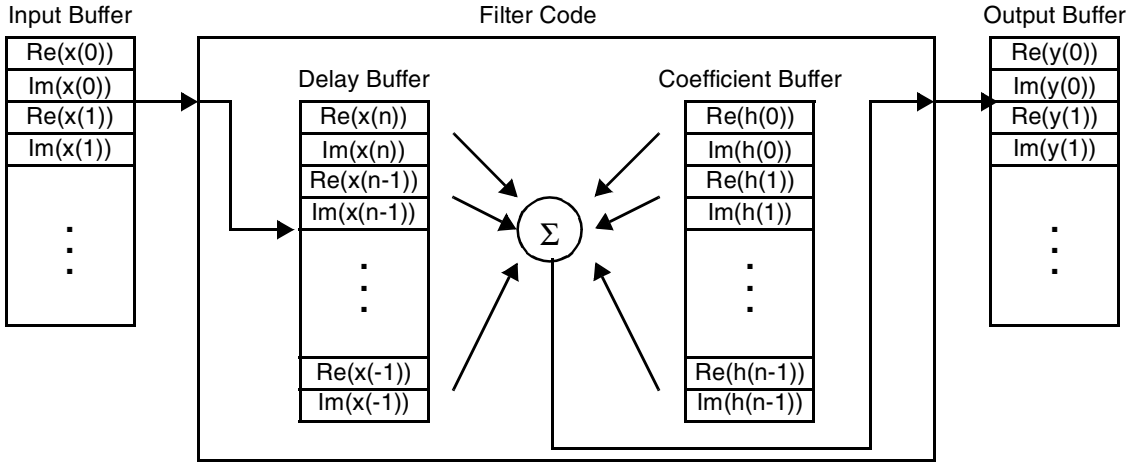

Figure 2. Filter Code Block Diagram

Following is a description of the actions performed by the code shown in **Section 3**:

1. Begin by reading the input arguments from memory.
2. Set up the modulo addressing registers for the delay and coefficient buffers.

3. Set up the looping parameters.
4. Read the first coefficient value from memory.

   Each data sample is read from memory with a **move.2l** instruction to read both the real and imaginary part of the 32-bit data. The **move.2l** instruction has the same restrictions as the double-precision instructions; the data registers must be a register pair (see **Section 1** for a list of the register pairs).

5. Begin looping for each sample, starting at COR_LOOP.

   The set-up for this loop includes clearing the accumulator registers (D4 for the real part of the result and D5 for the imaginary part), reading the input data values from memory, and writing them to the delay buffer. The coefficients and input data are read into different register pairs (data in register pair D0,D1 and coefficients in register pair D8,D9) because of the restrictions on the **move.2l** instruction. As discussed in **Section 1**, in order to multiply values together using the double-precision instructions, values must be in the same register pair. Therefore, the set-up code also transfers the coefficient and input data values from the move instruction registers to the appropriate registers for the double-precision instructions, as shown in **Figure 3**. For example, the Re(x) value is read from memory into the D0 register and the Im(x) value is read from memory into the D1 register. In order for Re(x) to be multiplied by Re(h), Re(h) must be transferred to D1. However, Im(x) must first be transferred out of D1. Six total transfers must be completed for each iteration of the main loop. This may seem like a lot of transfers, but the parallelism of the MSC8101 core allows many transfers to complete simultaneously with each other and other instructions.

6. The code begins looping for each coefficient starting at COR_S. This is the main loop of the code that includes the four double-precision multiplies needed for a complex double-precision multiply (see **Section 1** and **Figure 3** for details). The main loop code also includes accumulation of the real part of the result (in D4) and the imaginary part of the result (in D5), reading the new coefficients from memory and the data from the delay buffer. The last part of the main loop again transfers the new coefficients and data to the appropriate registers for the double-precision instructions.
7. After the main loop, save the real and imaginary parts of the result to the output buffer.
8. Updates the delay buffer pointer for the next cycle of the filter.
9. When the filter is complete for all input data samples, restore the modulo registers to linear accesses and write the delay buffer pointer to the memory location for the delay current argument.

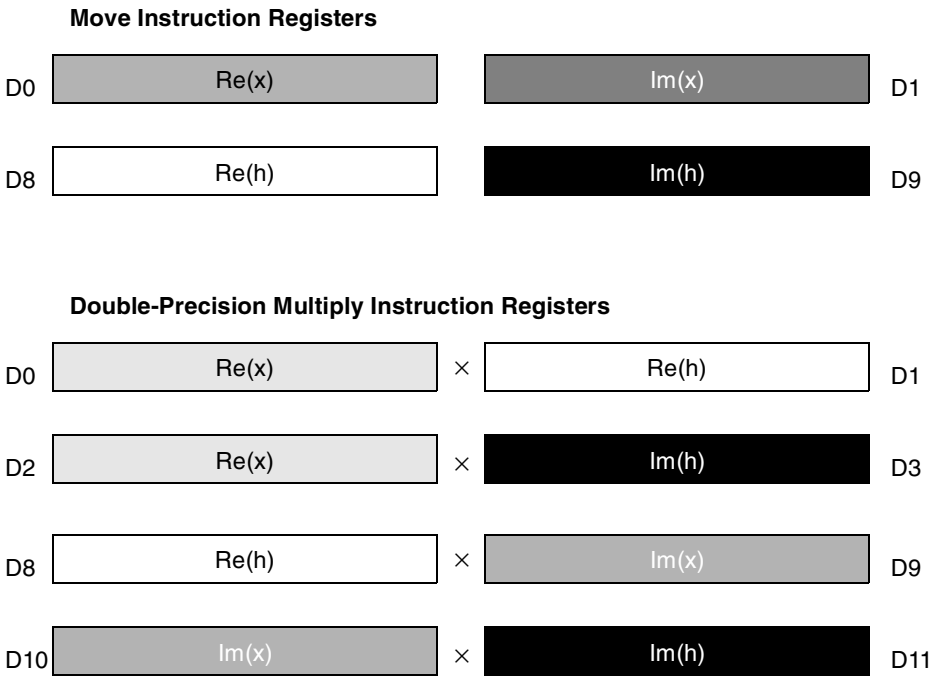

Figure 3. Instruction Register Restrictions

3 Filter Code

Example 1 shows the code that implements the double-precision complex FIR filter.

Example 1. Double-Precision Complex FIR Filter Code

```
section .text

       GLOBAL          _dp_c_fir
       ALIGN           16

_dp_c_fir
       push d6                 push d7
       move.l (r0)+,r3                                 ;r3 = output address
       move.l (r0)+,r1                                 ;r1 = input address
       move.l (r0)+,r2                                 ;r2 = coeffs address
       move.l (r0)+,d0                                 ;d0 = delay start address
       move.l (r0)+,r9         tfra r0,r4              ;r9 = delay pointer, r4 = delay arg address
       move.w (r0)+,d11        move.l d0,b0            ;d11 = #samp, b0 = modulo base for r0 = delay start
       tfra r2,r10                                     ;r10 = b2 = modulo base for r2 = coeffs address
       move.w (r0)+,d12        move.l #$0808,mctl      ;d12 = #coeffs, set modulo to m0 for r0 and r2
       dosetup2 COR_LOOP       doen2 d11               tfr d12,d13      asll #3,d12
                                                       ;outer loop (COR_LOOP) for #samp, d12 = #coeffs/8
       tfra r9,r0              move.l d12,m0           ;r0 = delay pointer, m0 = d12 = modulo buff length
       dosetup3 COR_S                                  ;inner loop (COR_S)
       move.2l (r2)+,d8:d9                             ;d8=Re(h) & d9=Im(h)

       falign
COR_LOOP
       loopstart2

       clr d4          clr d5              tfr d9,d3           tfr d9,d11    move.2l (r1)+,d0:d1
                  ;clear accumulators d4 & d5, d3=d11=Im(h), d0=Re(x) & d1=Im(x)
       tfr d0,d2       tfr d1,d9           tfr d1,d10          tfr d8,d1     move.2l d0:d1,(r0)-doen3 d13
                  ;d0=d2=Re(x), d9=d10=Im(x), d1=d8=Re(h), write input to delay buff, inner loop for #coeffs
```

```
       falign
COR_S
       loopstart3
                  mpyuu d0,d1,d6          mpyuu d2,d3,d7              mpyuu d8,d9,d14         mpyuu
d10,d11,d15
                  dmacsu d0,d1,d6         dmacsu d2,d3,d7             dmacsu d8,d9,d14        dmacsu
d10,d11,d15
                  macus d0,d1,d6          macus d2,d3,d7              macus d8,d9,d14         macus
d10,d11,d15
                  [dmacss d0,d1,d6        dmacss d2,d3,d7             dmacss d8,d9,d14        dmacss
d10,d11,d15
                                          move.2l (r2)+,d8:d9         move.2l (r0)-,d0:d1]
                       ;d6=Re(x)Re(h), d7=Re(x)Im(h), d14=Re(h)Im(x), d15=Im(x)Im(h)
                       ;d8=new Re(h), d9=new Im(h), d0=new Re(x), d1=new Im(x)
                  [add d6,d4,d4           add d7,d5,d5                tfr d9,d3               tfr d9,d11
                                          move.l d1,d10               move.l d0,d2]
                       ;d4=d4+Re(x)Re(h), d5=d5+Re(x)Im(h), d3=d11=new Im(h), d10=new Im(x), d0=d2=new Re(x)
                  sub d15,d4,d4           add d14,d5,d5               tfr d8,d1               tfr d1,d9
                       ;d4=d4-Im(x)Im(h)=Re(y), d5=d5+Re(h)Im(x)=Im(y), d1=new Re(h), d9=new Im(x)
       loopend3

       tfr d3,d9       move.2l d4:d5,(r3)+          adda #16,r0     ;write result, update delay pointer
       loopend2
       move.l #0,mctl  move.l r0,(r4)                               ;restore mctl, write delay pointer to arg
       move.l #-1,m0                                                ;restore m0
       rtsd                                                         ;return from subroutine with delay
_dp_c_fir_end
       pop d6          pop d7

END
   ENDSEC
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN2208
Rev. 2
12/2004

freescale™
semiconductor