SECTION 4
DATA ARITHMETIC LOGIC UNIT

This section describes the operation of the data arithmetic logic unit (ALU) registers and hardware. The data representation, rounding, and saturation arithmetic used within the data ALU are also presented. This section concludes with a discussion of the programming model.

4.1 OVERVIEW AND DATA ALU ARCHITECTURE

The DSP56000/DSP56001 central processor is composed of three execution units that operate in parallel. They are the data ALU, address generation unit (AGU), and the program controller (see Figure 4-1). These three units are register oriented rather than bus oriented and are designed to interface over the system buses with memory and memory-mapped I/O devices. The DSP56000/DSP56001 instruction set has been designed to allow flexible control of these parallel processing resources. Many instructions allow the programmer to keep each unit busy, thus enhancing performance. It was possible to make the programming model like that of conventional microprocessor units (MPUs), eliminating the

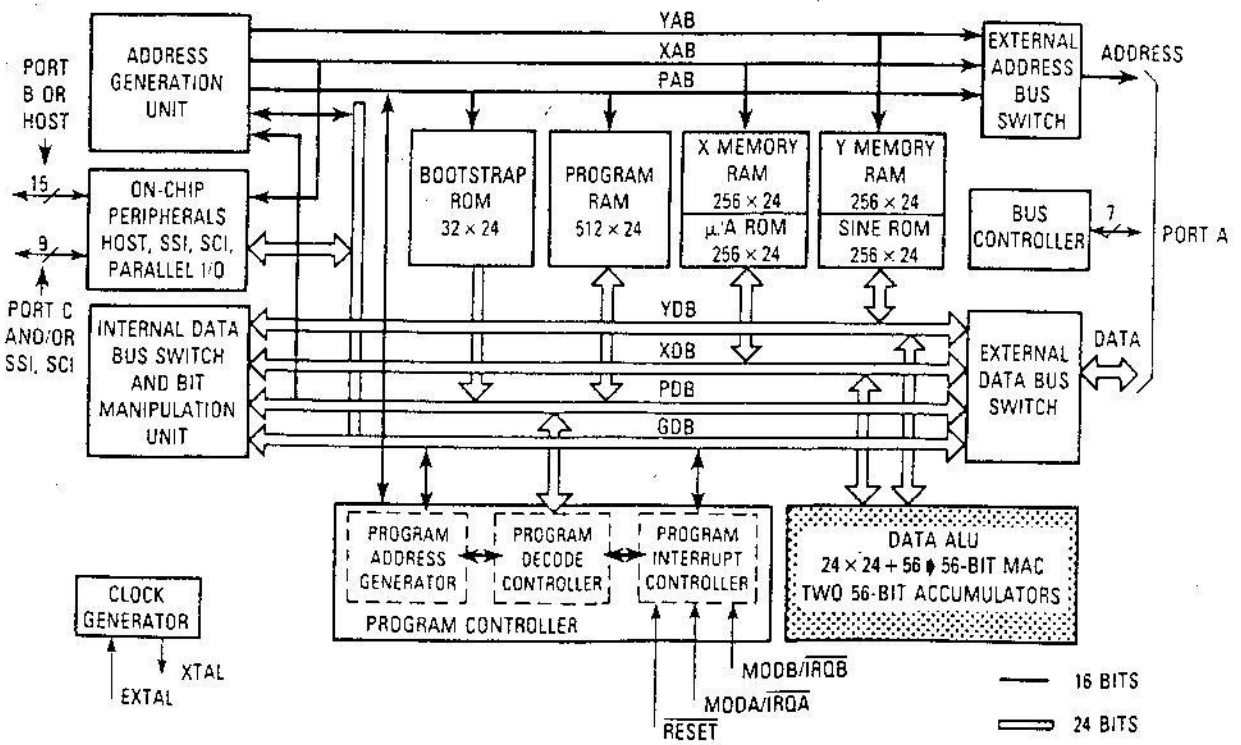
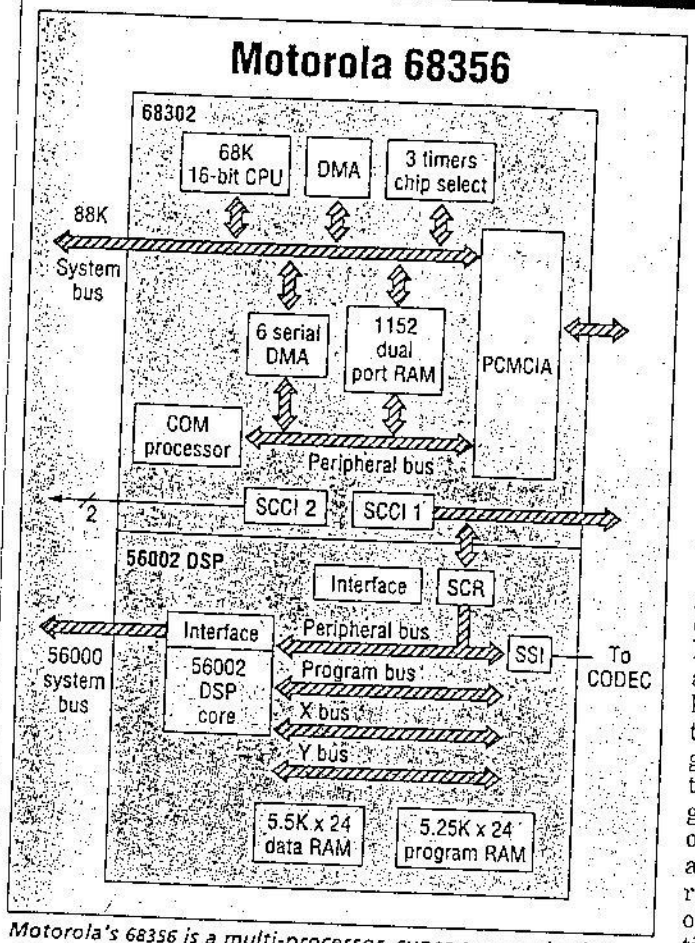


Figure 4-1. DSP56001 Block Diagram



Motorola's 68356 is a multi-processor, super communications chip. It combines a 68302 communications processor with a 24-bit 56002 DSP. The CPUs have different clocks and run separately, communicating vial shared memory or serial line

speech-recognition systems for Windows applications. "We design with Motorola 56001s, but use a Domain Technology emulator that substitutes a 56002 with OnCE emulation. This way we still use a 56001 but get 56002 emulation. It's a good system and gives me some of the capabilities of an old style ICE. For problems in the real-time code, I generally use the Motorola simulator to look at the code first."

Most engineers seem to be able to get their work done using background emulation. For those hard-to-find problems, many turn to logic analyzers as a last resort. And unlike most embedded systems engineers, DSP developers use simulation to get major code sets, usually math processing, to work.

"I really love TI's scan path emulator; its easy to use and works," says Chris Keiraly, vice president of engineering at Wintress (San Diego, CA), a system design house with designs that include golf and archery simulators driven by C30s. "It's a lot easier to use than standard ICEs, we had an 8-bit microcontroller ICE for a project that was just too hard to use. Most of our code is pretty clean. The 12-pin header is just great for hookups. If we run into a major hardware problem, we just rent a logic analyzer. But that doesn't happen often. We're pretty careful with our board designs."

But background mode emulation has its limits. Many DSP applications don't lend themselves to a strictly background-emulation-mode debugging style. For one thing, some applications require continuous control, particularly those with negative feedback paths. Stopping the DSP can cause the system to fail and even damage equipment.

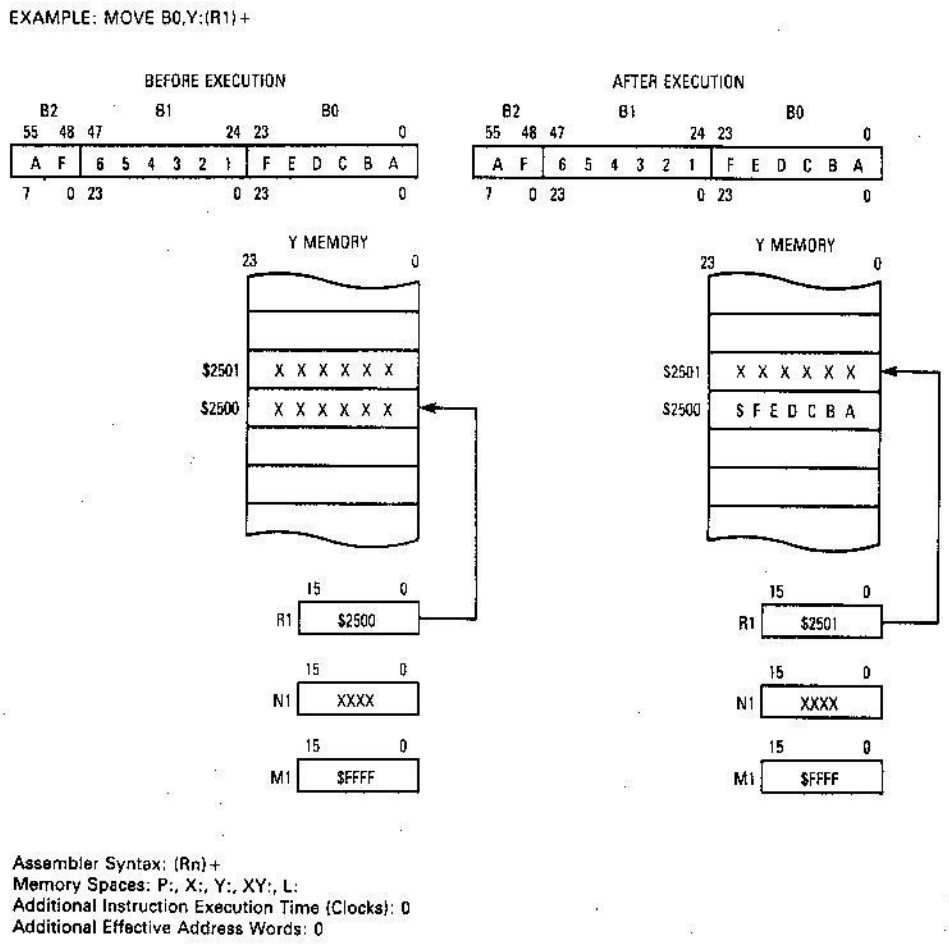
For many DSP chips, especially the first- and second-generation chips, you can get a full ICE for debugging. These ICEs let you to set complex breakpoints, single-step, and trace external bus activity in realtime. There are ICEs for TI C1X, C2X and C3X DSPs, and for Analog Devices' ADSP-21XX DSPs. TI and Analog Devices furnish their own ICEs. Companies such as Macrochip (Plano, TX) and Signum Systems (Thousand Oaks, CA) also furnish ICEs for TI's C25, C30 and C50 DSPs.

ICEs for DSP have some limitations, however. They monitor the external pins of a DSP and although some DSPs provide some access to their internal operation, the ICEs cannot monitor many internal operations. For DSPs, this can be problematical because so much processing is internal with little visible on the outside. Most DSPs, for example, have

continued on page 86

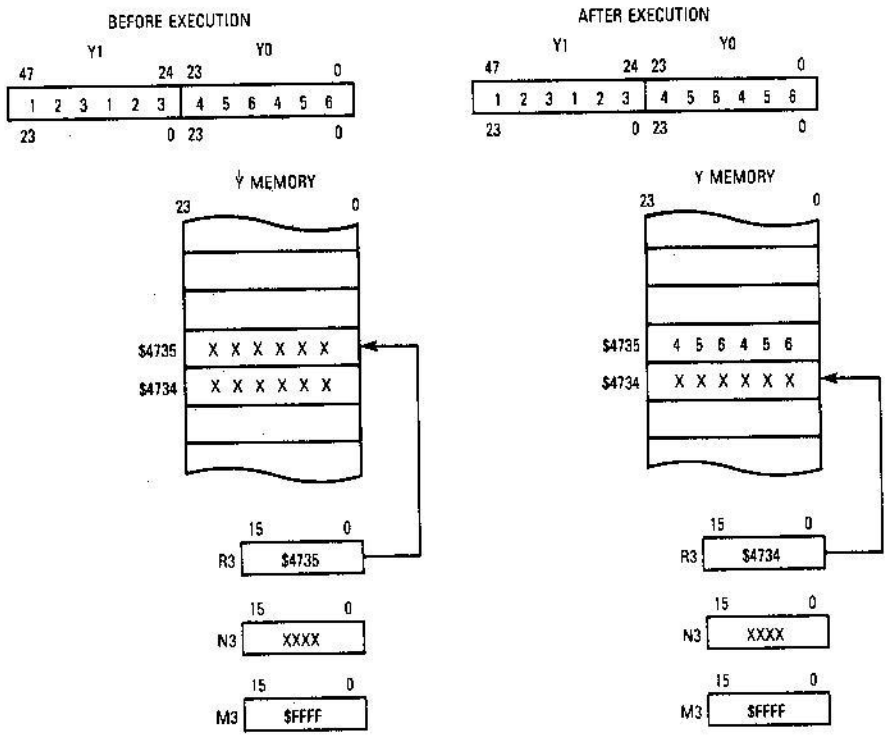
5.3.1.2 POSTINCREMENT BY 1. The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-5). After the operand address is used, it is incremented by 1 and stored in the same address register. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

5.3.1.3 POSTDECREMENT BY 1. The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-6). After the operand address is used, it is decremented by 1 and stored in the same address register. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.



Assembler Syntax: (Rn)+
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 5-5. Address Register Indirect — Postincrement



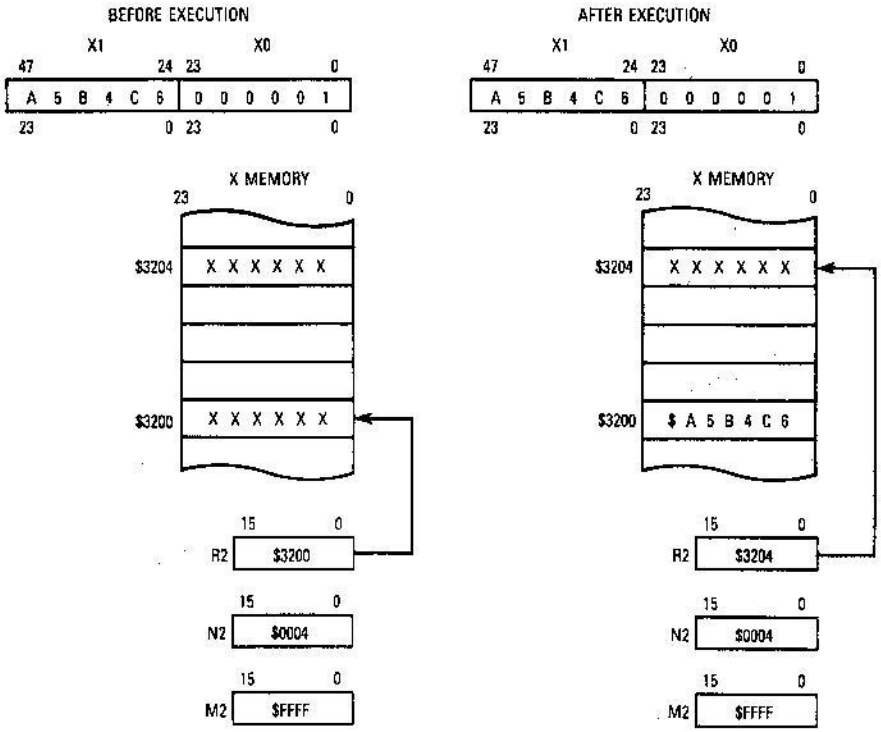
Assembler Syntax: (Rn)–
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 5-6. Address Register Indirect — Postdecrement

5.3.1.4 POSTINCREMENT BY OFFSET Nn. The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-7). After the operand address is used, it is incremented by the contents of the Nn register and stored in the same address register. The contents of the Nn register are unchanged. This mode can be used for making XY: memory references and for modifying the contents of Rn without an associated data move.

5.3.1.5 POSTDECREMENT BY OFFSET Nn. The address of the operand is in the address register, Rn (see Table 5-1 and Figure 5-8). After the operand address is used, it is decremented by the contents of the Nn register and stored in the same address register. The

EXAMPLE: MOVE X1,X:(R2)+N2



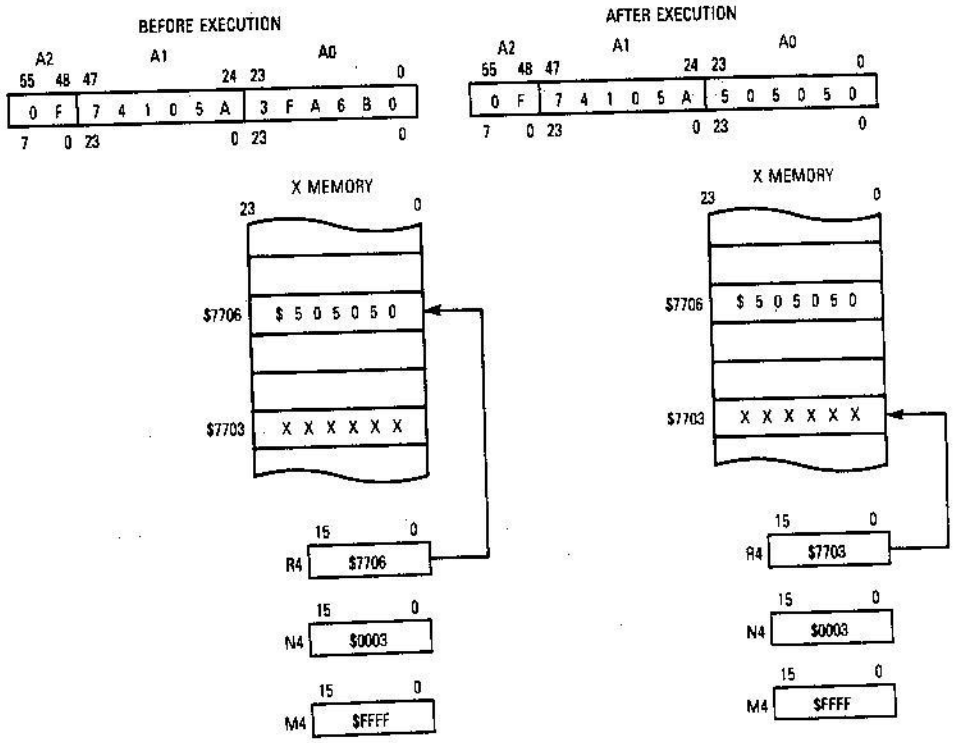
Assembler Syntax: (Rn)+Nn
Memory Spaces: P:, X:, Y:, XY:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

Figure 5-7. Address Register Indirect — Postincrement by Offset Nn

contents of the Nn register are unchanged. This mode cannot be used for making XY: memory references, but it can be used to modify the contents of Rn without an associated data move.

5.3.1.6 INDEXED BY OFFSET Nn. The address of the operand is the sum of the contents of the address register, Rn, and the contents of the address offset register, Nn (see Table 5-1 and Figure 5-9). The contents of the Rn and Nn registers are unchanged. This addressing mode, which requires an extra instruction cycle, cannot be used for making XY: memory references.

EXAMPLE: MOVE X:(R4) – N4,A0



Assembler Syntax: (Rn) – Nn
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 0
Additional Effective Address Words: 0

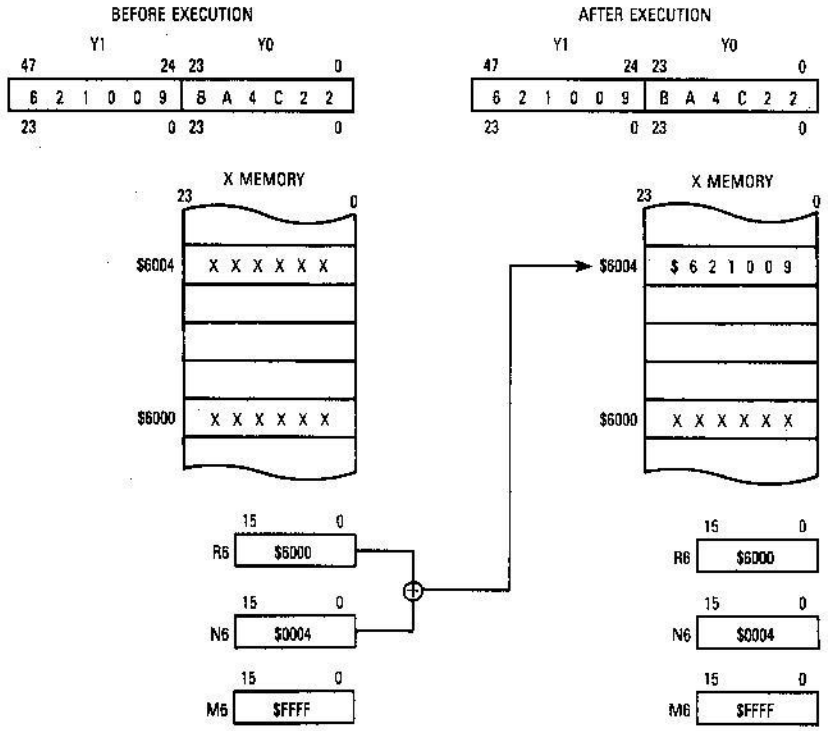
Figure 5-8. Address Register Indirect — Postdecrement by Offset Nn

5.3.1.7 PREDECREMENT BY 1. The address of the operand is the contents of the address register, Rn, decremented by 1 before the operand address is used (see Table 5-1 and Figure 5-10). The contents of Rn are decremented and stored in the same address register. This addressing mode requires an extra instruction cycle. This mode cannot be used for making XY: memory references, nor can it be used for modifying the contents of Rn without an associated data move.

5.3.2 Address Modifier Types

The DSP56000/DSP56001 address ALU supports linear, modulo, and reverse-carry arithmetic types for all address register indirect modes. These arithmetic types easily allow the

EXAMPLE: MOVE Y1,X:(R6+N6)



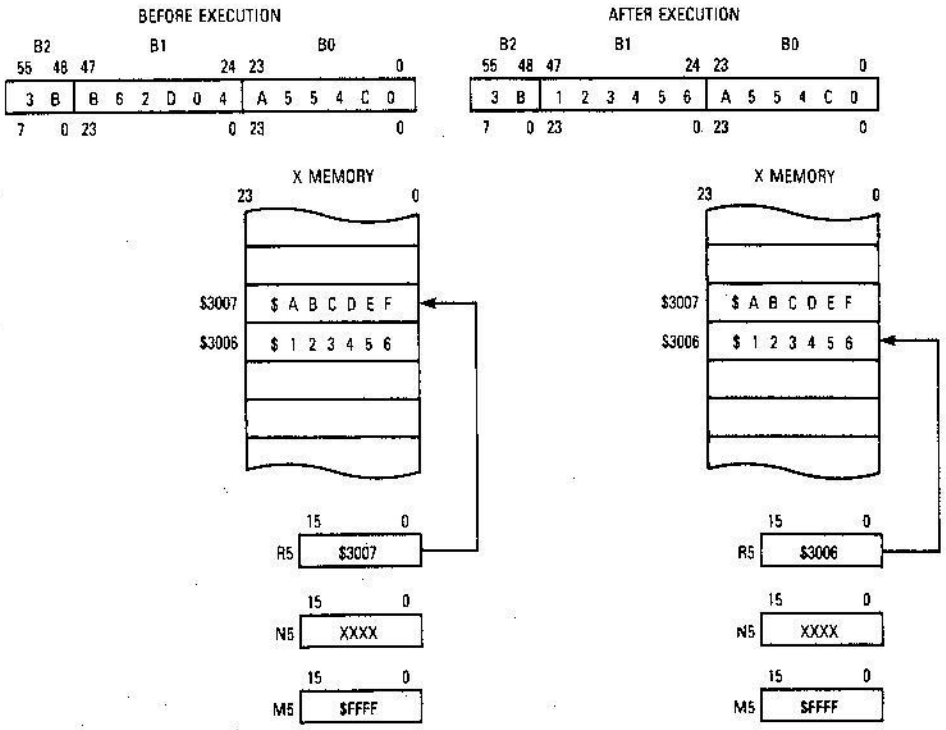
Assembler Syntax: (Rn+Nn)
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 0

Figure 5-9. Address Register Indirect — Indexed by Offset Nn

creation of data structures in memory for FIFOs (queues), delay lines, circular buffers, stacks, and bit-reversed FFT buffers. Data is manipulated by updating address registers (pointers) rather than moving large blocks of data. The contents of the address modifier register, Mn, define the type of arithmetic to be performed for addressing mode calculations; for modulo arithmetic, the contents of Mn also specify the modulus. All address register indirect modes can be used with any address modifier. Each address register, Rn, has its own modifier register, Mn, associated with it.

5.3.2.1 LINEAR MODIFIER (Mn=$FFFF). Address modification is performed using normal 16-bit linear (modulo 65,536) arithmetic (see Table 5-2). A 16-bit offset, Nn, and +1 or −1

EXAMPLE: MOVE X:-(R5),B1



Assembler Syntax: -(Rn)
Memory Spaces: P:, X:, Y:, L:
Additional Instruction Execution Time (Clocks): 2
Additional Effective Address Words: 0

Figure 5-10. Address Register Indirect — Predecrement

can be used in the address calculations. The range of values can be considered as signed (Nn from −32,768 to +32,767) or unsigned (Nn from 0 to +65,535) since there is no arithmetic difference between these two data representations. Addresses are normally considered unsigned, and data is normally considered signed.

5.3.2.2 MODULO MODIFIER (Mn = MODULUS − 1). The address modification is performed modulo M, where M ranges from 2 to +32,768 (see Table 5-3). Modulo M arithmetic causes the address register value to remain within an address range of size M, defined by a lower and upper address boundary (see Figure 5-11). The value m = M − 1 is stored in the modifier register, Mn. The lower boundary (base address) value must have zeros in the k LSBs, where $2^k \geq M$, and therefore must be a multiple of 2^k. The upper boundary is

Table 5-2. Linear Address Modifiers

Modifier Mn Value	Addressing Mode Arithmetic
0	Reverse Carry (Bit Reverse)
1	Modulo 2
2	Modulo 3
:	:
:	Modulo (Mn + 1)
:	:
32766	Modulo 32767
32767	Modulo 32768
:	Reserved
65535	*Linear (Modulo 65536)*

the lower boundary plus the modulo size minus one (base address plus M − 1). Since $M \leq 2^k$, once M is chosen, a sequential series of memory blocks (each of length 2^k) is created where these circular buffers can be located. If $M < 2^k$, there will be a space between sequential circular buffers of $(2^k) - M$. For example, to create a circular buffer of 21 stages, M is 21, and the lower address boundary must have its five LSBs equal to zero ($2^k \geq 21$, thus $k \geq 5$). The Mn register is loaded with the value 20. The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. The upper boundary of the buffer is then the lower boundary plus 21. There will be an unused space of 11 memory locations between the upper address and next usable lower address. The address pointer is not required to start at the lower address boundary or to end on the upper address boundary; it can initially point anywhere within the defined modulo address range. Neither the lower nor the upper boundary of the modulo region is stored; only the size of the modulo region is stored in Mn. The boundaries are determined by the contents of Rn. Assuming the (Rn)+ indirect addressing mode, if the address register pointer increments past the upper boundary of the buffer

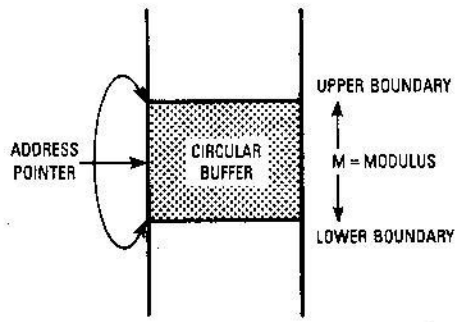


Figure 5-11. Circular Buffer

base address plus M – 1), it will wrap around through the base address (lower boundary). Alternatively, assuming the (Rn) – indirect addressing mode, if the address decrements past the lower boundary (base address), it will wrap around through the base address plus M – 1 (upper boundary).

If an offset, Nn, is used in the address calculations, the 16-bit absolute value, |Nn|, must be less than or equal to M for proper modulo addressing. If Nn>M, the result is data dependent and unpredictable, except for the special case where $Nn = P \times 2^k$, a multiple of the block size where P is a positive integer. For this special case, when using the (Rn) + Nn addressing mode, the pointer, Rn, will jump linearly to the same relative address in a new buffer, which is P blocks forward in memory (see Figure 5-12). Similarly, for (Rn) – Nn, the pointer will jump P blocks backward in memory. This technique is useful in sequentially processing multiple tables or N-dimensional arrays. The range of values for Nn is –32,768 to +32,767. The modulo arithmetic unit will automatically wrap around the address pointer by the required amount. This type address modification is useful for creating circular buffers for FIFOs (queues), delay lines, and sample buffers up to 32,768 words long as well as for decimation, interpolation, and waveform generation. The special case of (Rn) ± Nn mod M with $Nn = P \times 2^k$ is useful for performing the same algorithm on multiple blocks of data in memory — e.g., parallel infinite impulse response (IIR) filtering.

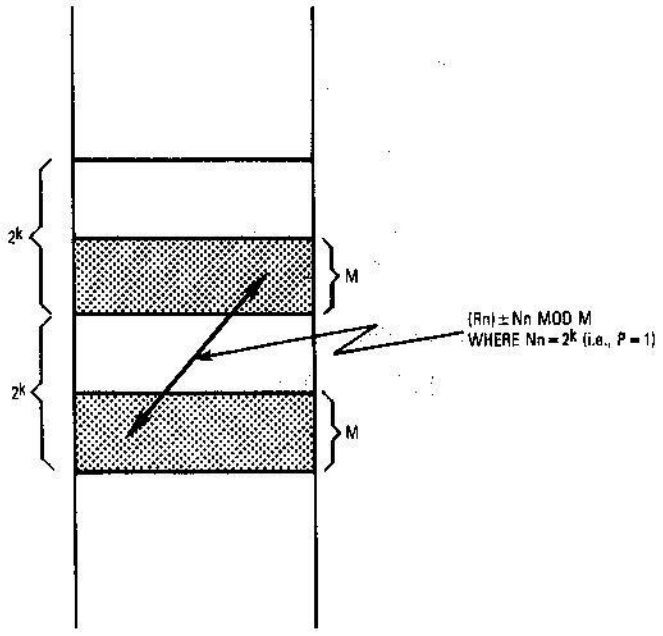


Figure 5-12. Linear Addressing with a Modulo Modifier

An example of address register indirect modulo addressing is shown in Figure 5-13. Starting at location 64, a circular buffer of 21 stages is created. The addresses generated are offset by 15 locations. The lower boundary = $L \times (2^k)$ where $2^k \geq 21$; therefore, k = 5 and the lower address boundary must be a multiple of 32. The lower boundary may be chosen as 0, 32, 64, 96, 128, 160, etc. For this example, L is arbitrarily chosen to be 2, making the lower boundary 64. The upper boundary of the buffer is then 84 (the lower boundary plus 20 (M – 1)). The Mn register is loaded with the value 20 (M – 1). The offset register is arbitrarily chosen to be 15 (Nn≤M). The address pointer is not required to start at the lower address boundary and can begin anywhere within the defined modulo address range — i.e., within the lower boundary + (2^k) address region. The address pointer, Rn, is arbitrarily chosen to be 75 in this example. When R2 is postincremented by the offset by the MOVE instruction, instead of pointing to 90 (as it would in the linear mode) it wraps around to 69. If the address register pointer increments past the upper boundary of the buffer (base address plus M – 1), it will wrap around to the base address. If the address decrements past the lower boundary (base address), it will wrap around to the base address plus M – 1.

If Rn is outside the valid modulo buffer range and an operation occurs that causes Rn to be updated, the contents of Rn will be updated according to modulo arithmetic rules. For example, a MOVE B0,X:(R0) + N0 instruction (where R0 = 6, M0 = 5, and N0 = 0) would apparently leave R0 unchanged since N0 = 0. However, since R0 is above the upper boundary, the AGU calculates R0 + N0 – M0 – 1 for the new contents of R0 and sets R0 = 0.

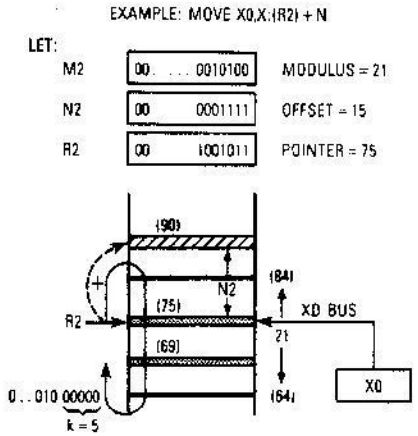


Figure 5-13. Modulo Modifier Example

Table 5-6. Address-Modifier-Type Encoding Summary

Modifier Mn	Rn Update Arithmetic
0	Reverse-Carry (Bit-Reverse) Addressing
1	Modulo 2
2	Modulo 3
:	:
:	Modulo (Mn + 1) Addressing
:	:
32767	Modulo 32768
:	Reserved
65535	Linear Addressing (Modulo 65536)

Bit-reverse addressing is useful for 2^k-point FFT addressing. Modulo addressing is useful for creating circular buffers for FIFOs (queues), delay lines, and sample buffers up to 32,768 words long. The linear addressing is useful for general-purpose addressing. There is a reserved set of modifier values (from 32,768 to 65,534) that should not be used.

Figure 5-15 gives examples of the three addressing modifiers using 8-bit registers for simplification (all AGU registers in the DSP56000/DSP56001 are 16 bit). The addressing mode used in the example, postincrement by offset Nn, adds the contents of the offset register to the contents of the address register after the address register is accessed. The results of the three examples are as follows:

The linear address modifier addresses every fifth location since the offset register contains $5.

Using the bit-reverse address modifier causes the postincrement by offset Nn addressing mode to use the address register, bit reverse the four LSBs, increment by 1, and bit reverse the four LSBs again.

The modulo address modifier has a lower boundary at a predetermined location, and the modulo number plus the lower boundary establishes the upper boundary. This boundary creates a circular buffer so that, if the address register is pointing within the boundaries, addressing past a boundary causes a circular wraparound to the other boundary.

Table B-2. 27-MHz Benchmark Results for the DSP56001R27

Benchmark Program	Sample Rate (Hz) or Execution Time	Memory Size (Words)	Number of Clock Cycles
20-Tap FIR Filter	500.0 kHz	50	54
64-Tap FIR Filter	190.1 kHz	138	142
67-Tap FIR Filter	182.4 kHz	144	148
8-Pole Cascaded Canonic Biquad IIR Filter (4×)	540.0 kHz	40	50
8-Pole Cascaded Canonic Biquad IIR Filter (5×)	465.5 kHz	45	58
8-Pole Cascaded Transpose Biquad IIR Filter	385.7 kHz	48	70
Dot Product	444.4 ns	10	12
Matrix Multiply 2×2 times 2×2	1.556 µs	33	42
Matrix Multiply 3×3 times 3×1	1.259 µs	29	34
M-to-M FFT 64 Point	98.33 µs	489	2655
M-to-M FFT 256 Point	489.8 µs	1641	13255
M-to-M FFT 1024 Point	2.453 ms	6793	66240
P-to-M FFT 64 Point	92.56 µs	704	2499
P-to-M FFT 256 Point	347.9 µs	2048	9394
P-to-M FFT 1024 Point	1.489 ms	7424	40144

B

```
        page 132,66,0,6
        opt   rc
; ***************************************
;Motorola Austin DSP Operation     June 30, 1988
; ***************************************
;DSP56000/1
;20-tap FIR filter
;File name:  1-56.asm
; ********************************************************************************
;       Maximum sample rate: 379.6 kHz at 20.5 MHz/500.0 kHz at 27.0 MHz
;       Memory Size: Prog: 4+6 words; Data: 2×20 words
;       Number of clock cycles: 54 (27 instruction cycles)
;       Clock Frequency: 20.5 MHz/27.0 MHz
;       Instruction cycle time: 97.6 ns/74.1 ns
; ********************************************************************************
;       This FIR filter reads the input sample
;       from the memory location Y:input
;       and writes the filtered output sample
;       to the memory location Y:output
;
;       The samples are stored in the X memory
;       The coefficients are stored in the Y memory
; ********************************************************************************
```

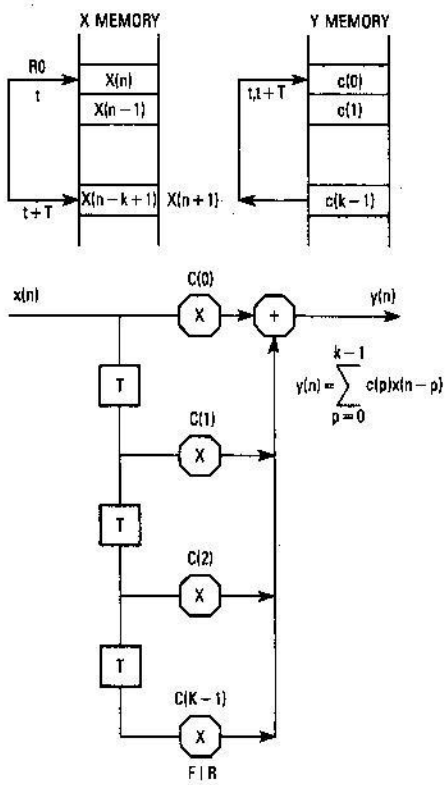


Figure B-1. 20-Tap FIR Filter Example (Sheet 1 of 2)

```
;
;
;
;*****************************************************************************
;
;             initialization
; ****************************
n           equ         20
start       equ         $40
wddr        equ         $0
cddr        equ         $0
input       equ         $ffe0
output      equ         $ffe1
;
            org         p:start
            move        #wddr,r0                ;r0 ⇒ samples
            move        #cddr,r4                ;r1 ⇒ coefficients
            move        #n-1,m0                 ;set modulo arithmetic
            move        m0,m4                   ;for the 2 circular buffers
;
            opt         cc
;           filter loop :8+(n-1) cycles
;*****************************************************************************
            movep       y:input,x: (r0)         ;input sample in memory
            clr         a           x:(r0)+,x0  y: (r4)+,y0

            rep         #n-1
            mac         x0,y0,a     x:(r0)+,x0  y: (r4)+,y0
            macr        x0,x0,a     (r0)-

            movep       a,y:output              ;output filtered sample
;*****************************************************************************
        end
```

Figure B-1. 20-Tap FIR Filter Example (Sheet 2 of 2)

B

```
;
;This program originally available on the Motorola DSP bulletin board.
;It is provided under a DISCLAIMER OF WARRANTY available from
;Motorola DSP Operation, 6501 William Cannon Drive, Austin, Tx., 78735.
;
;Radix-2, In-Place, Decimation-In-Time FFT (smallest code size).
;
;Last Update 30 Sep 86     Version 1.1
;
fftr2a          macro           points,data,coef
fftr2a          ident           1,1
;
;Radix-2 Decimation-In-Time In-Place FFT Routine
;
;               Complex input and output data
;                   Real data in X memory
;                   Imaginary data in Y memory
;               Normally ordered input data
;               Bit reversed output data
;                   Coefficient lookup table
;                   -Cosine values in X memory
;                   -Sine vaues in Y memory
;
;Macro Call — fftr2a     points,data,coef
;
;                   points                  number of points (2-32768, power of 2)
;                   data                    start of data buffer
;                   coef                    start of sine/cosine table
;
;Alters Data ALU Registers
;                   x1          x0          y1          y0
;                   a2          a1          a0          a
;                   b2          b1          b0          b
;
;Alters Address Registers
;                   r0          n0          m0
;                   r1          n1          m1
;                               n2
;
;                   r4          n4          m4
;                   r5          n5          m5
;                   r6          n6          m6
;
;Alters Program Control Registers
;                   pc          sr
;
;Uses 6 locations on System Stack
;
```

Figure B-2. Radix 2, In-Place, Decimation-In-Time FFT (Sheet 1 of 2)

```
;Latest Revision — September 30, 1986
;
                move    #points/2,n0            ;initialize butterflies per group
                move    #1,n2                   ;initialize groups per pass
                move    #points/4,n6            ;initialize C pointer offset
                move    #-1,m0                  ;initialize A and B address modifiers
                move    m0,m1                   ;for linear addressing
                move    m0,m4
                move    m0,m5
                move    #0,m6                   ;initialize C address modifier for
                                                ;reverse carry (bit-reversed) addressing
;
;Perform all FFT passes with triple nested DO loop
;
                do      #@cvi (@log(points)/@log(2)+0.5),_end_pass
                move    #data,r0                ;initialize A input pointer
                move    r0,r4                   ;initialize A output pointer
                lua     (r0)+n0,r1              ;initialize B input pointer
                move    #coef,r6                ;initialize C input pointer
                lua     (r1)-,r5                ;initialize B output pointer
                move    n0,n1                   ;initialize pointer offsets
                move    n0,n4
                move    n0,n5

                do      n2,_end_grp
                move    x:(r1),X1       y:(r6),y0       ;lookup -sine and
                                                        ; -cosine values
                move    x:(r5),a        y:(r0),b        ;preload data
                move    x:(r6)+n6,x0                    ;update C pointer

                do      n0,_end_bfy
                mac     x1,y0,b         y:(r1)+,y1      ;Radx 2 DIT
                                                        ;butterfly kernel
                macr    -x0,y1,b        a,x:(r5)+       y:(r0),a
                subl    b,a             x:(r0),b        b,y:(r4)
                mac     -x1,x0,b        x:(r0)+,a       a,y:(r5)
                macr    -y1,y0,b        x:(r1),x1
                subl    b,a             b,x:(r4)+       y:(r0),b
_end_bfy
                move    a,x:(r5)+n5     y:(r1)+n1,y1    ;update A and B pointers
                move    x:(r0)+n0,x1    y:(r4)+n4,y1
_end_grp
                move    n0,b1                   ;divide butterflies per group by two
                lsr     b       n2,a1           ;multiply groups per pass by two
                lsl     a       b1,n0
                move    a1,n2
_end_pass
                endm
```

Figure B-2. Radix 2, In-Place, Decimation-In-Time FFT (Sheet 2 of 2)

B

```
     page 132,66,0,6
     opt   rc
;*******************************************
;Motorola Austin DSP Operation June 30, 1988
;*******************************************
;DSP56000/1
;8-pole 4-multiply cascaded canonic IIR filter
;File name: 4-56.asm
;**************************************************************************
;     Maximum sample rate: 410.0 kHz at 20.5 MHz/540.0 kHz at 27.0 MHz
;     Memory Size: Prog: 6+10 words; Data: 4(2+4) words
;     Number of clock cycles: 50 (25 instruction cycles)
;     Clock Frequency: 20.5 MHz/27.0 MHz
;     Cycle time:            97.5 ns/74.1 ns
;**************************************************************************
;     This IIR filter reads the input sample
;     from the memory location Y:input
;     and writes the filtered output sample
;     to the memory location Y:output
;
;     The samples are stored in the X memory
;     The coefficients are stored in the Y memory
;
;
;     The equations of the filter are:
;         w(n)=        x(n)-ai1*w(n-1)-ai2*w(n-2)
;         y(n)=        w(n)+bi1*w(n-1)+bi2*w(n-2)
;
```

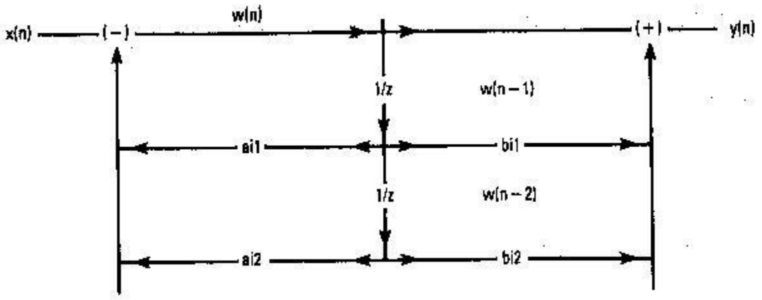


Figure B-3. 8-Pole 4-Multiply Cascaded Canonic IIR Filter (Sheet 1 of 2)


```
;
;****
;
;     i
;****
nsec
start
data
coef
input
output
igain
;
;*****
;
end_cell
;*******
```

```
;   All coefficients are divided by 2:
;   w(n)/2 = x(n)/2 - ai1/2*w(n-1) - ai2/2*w(n-2)
;   y(n)/2 = w(n)/2 + bi1/2*w(n-1) + bi2/2*w(n-2)
;
;       X Memory Organization                 Y Memory Organization
;                                                 b1N/2      Coef.+4*nsec-1
;                                                 b2N/2
;                                                 a1N/2
;                                                 a2N/2
;                 wN(n-1)     Data+2*nsec-1       .
;                 wN(n-2)                         .
;                   .                             b11/2
;                   .                             b21/2
;                 w1(n-1)                         a11/2
;           R0 -> w1(n-2)     Data          R4 -> a21/2      Coef.
;
;*********************************************************************************
;
;    initialization
;*****************************
nsec        equ     4
start       equ     $40
data        equ     0
coef        equ     0
input       equ     $ffe0
output      equ     $ffe1
igain       equ     0.5
            ori     #$08,mr             ;set scaling mode
            move    #data,r0            ;point to filter states
            move    #coef,r4            ;point to filter coefficients
            move    #2*nsec-1,m0
            move    #4*nsec-1,m4
            move    #igain,y1           ;y1 = initial gain

            opt     cc
;                 filter loop: 4*nsec+9
;*****************************************************************
            movep       y:input,y0                          ;get sample
            mpy     y0,y1,a     x:(r0)+,x0  y:(r4)+,y0      ;x0 = 1st section w(n-2),y0 = a12/2
;
            do          #nsec,end_cell                      ;do each section
            mac     -x0,y0,a    x:(r0)-,x1  y:(r4)+,y0      ;x1 = w(n-1),y0 = ai1/2
            macr    -x1,y0,a    x1,x:(r0)+  y:(r4)+,y0      ;push w(n-1) to w(n-2),y0 = bi2/2
            mac     x0,y0,a     a,x:(r0)+   y:(r4)+,y0      ;push w(n) to w(n-1),y0 = bi1/2
            mac     x1,y0,a     x:(r0)+,x0  y:(r4)+,y0      ;next iter:x0 = w(n-2),yo = ai2/2
end_cell
            rnd     a                                       ;round result
            movep   a,y:output                              ;output sample
;*****************************************************************
            end
```

Figure B-3. 8-Pole 4-Multiply Cascaded Canonic IIR Filter (Sheet 2 of 2)

```
   page  132,60,1,1
;newlms2n.asm
;     New Implementation of the delayed LMS on the DSP56000 Revision C
;Memory map:
;      Initial X                                    H
;  x(n)  x(n-1)  x(n-2)  x(n-3)  x(n-4)      hx    h0    h1    h2    h3
;    ]                                        ]     ]
;    r0                                       r5    r4
;hx is an unused value to make the calculations faster.
;
   opt        cc
ntaps         equ        4
input         equ        $FFC0
output        equ        $FFC1

   org        x:$0
state         ds         5
   org        y:$0
coef          ds         5
;
   org        p:$40
   move       #state,r0              ;start of X
   move       #2,n0
   move       #ntaps,m0              ;mod 5
   move       #coef+1,r4             ;coefficients
   move       #ntaps,m4              ;mod 5
   move       #coef,r5               ;coefficients
   move       m4,m5                  ;mod 5
_smploop                             ;                                    Prog   Icyc
   movep                  y:input,a  ;get input sample                    word
   move       a,x:(r0)               ;save input sample                     1      1
;error signal is in y1
;FIR sum in a=a+h(k) old*x(n-k)
;h(k)new in b=h(k)old+error*x(n-k-1)

   clr        a       x:(r0)+,x0                ;x0=x(n)                    1      1
   move               x:(r0)+,x1  y:(r4)+,y0    ;x1=x(n-1),y0=h(0)          1      1
   do         #taps/2,_lms                      ;                           2      3
   mac        x0,y0,a y0,b        b,y:(r5)+     ;a=h(0)*x(n),b=h(0)         1      1
   macr       x1,y1,b x:(r0)+,x0  y:(r4)+,y0    ;b=h(0)+e*x(n-1)=h(0)new    1      1
                                                ;x0=x(n-2) y0=h(1)
   mac        x1,y0,a y0,b        b,y:(r5)+     ;a=a+h(1)*x(n-1) b=h(1)     1      1
   macr       x0,y1,b x:(r0)+,x1  y:(r4)+,y0    ;b=h(1)+e*x(n-2)            1      1
;                                               ;x1=x(n-3) y0=H(2)
_lms
   move                           b,y:(r5)+     ;save last new c( )         1      1
   move       (r0)-n0                           ;pointer update             1      1

;(Get d(n), subtract fir output (reg a), multiply by "u", put
;the result in y1. This section is application dependent.)
   movep      a,y:output                        ;output fir if desired
   jmp        _smploop
   end                                          ;
;                                                          Totals:   11   2N+8
```

Figure B-4. LMS FIR Adaptive Filter