Digital Signal Processing on the ColdFire™ Architecture

William Hohl, Joe Circello

Motorola, Inc.
High Performance Embedded Systems
6501 William Cannon Drive West
Austin, Texas 78735

Motorola, Inc.
Phoenix Design Center
432 North 44th St., Suite 200
Phoenix, Arizona 85008

ABSTRACT

The multiply-accumulate (MAC) unit is an addition to the ColdFire core that provides a new level of support for signal processing algorithms. This paper details the associated support functions within the instruction set. Signal processing features are discussed for potential applications in embedded systems, and examples are presented for implementing some useful algorithms.

1. INTRODUCTION

As consumer applications rely more heavily on embedded controllers, component costs and area constraints have forced manufacturers to consider integrating more logic into fewer components. The total cost of systems that use multiple-chip solutions, such as a single-chip microcontroller combined with a dedicated signal processor, can be reduced by employing an embedded processor that efficiently supports many digital signal processing routines.

Motorola's ColdFire™ architecture represents a design approach targeted specifically for the emerging class of advanced consumer electronics applications. Within the domain of such cost-driven embedded systems, several requirements shaped the overall processor design. First, the processor core is small enough to permit cost-effective integration of on-chip memories and other system modules and peripherals. Second, a high-density instruction set can minimize memory requirements. In many designs, the cost of the memory system exceeds the microprocessor cost, so this factor can significantly impact overall system cost. The ColdFire processor architecture addresses these requirements through a variable-length instruction set to maximize code density implemented in a RISC-based approach to provide a very efficient silicon design.

ColdFire's flexible system architecture and tools-driven implementation philosophy allows for different configurations with the core. The addition of a MAC unit to the processor is centered on the notion of providing a limited set of DSP operations that are currently being used in embedded code today, while supporting the current multiply instructions in the ColdFire architecture. The MAC unit provides functionality in three related areas: signed and unsigned integer multiplies, multiply-accumulate operations involving signed and unsigned operands, and miscellaneous register operations. Each of the three areas of support is addressed in detail in the succeeding sections.

The MAC unit is an extension of the basic multiplier structure found in almost all microprocessors on the market today, whether implemented in hardware or an iterative routine within the architecture itself. The idea behind this extension is to provide the ability to do operations native to signal processing algorithms as quickly as possible while considering the limits of the application. For example, small digital filters can certainly tolerate some variance in the execution time of the algorithm, while larger, more complicated algorithms such as orthogonal transforms (Fast Fourier, Discrete Cosine, Hartley, etc.) may have more demanding speed and memory band-

width requirements.

Obviously, the 68000 architecture was not designed for high-speed signal processing, and a large DSP engine might be excessive in an embedded environment. In striking a middle ground between speed, size, and functionality, the ColdFire MAC unit is optimized for a small set of operations that involve multiplication and cumulative additions and subtractions. Specifically, the multiplier array is optimized for single-cycle 16x16 multiplies producing a 32-bit result, with a possible accumulation cycle following. This approach was selected since it is common in a large portion of signal processing applications. In addition, the ColdFire core architecture has been extended to allow for an operand fetch in parallel with a MAC, resulting in an overall increase in performance for certain DSP operations.

2. ARCHITECTURE

The ColdFire processor implementation features two independent, decoupled pipeline structures to maximize performance while minimizing core size [1].

The Instruction Fetch Pipeline (IFP) is a 2-stage pipeline for prefetching instructions. The prefetched instruction stream is then gated into the 2-stage Operand Execution Pipeline (OEP), which decodes the instruction, fetches the required operands, and then executes the actual function. Since the IFP and OEP pipelines are decoupled by an instruction buffer which serves as a FIFO queue, the IFP can prefetch instructions in advance of their actual use by the OEP, thereby minimizing time stalled waiting for instructions.

The MAC unit is located within the Operand Execution Pipeline and effectively appears as another execute engine within this pipeline. The OEP is implemented as a two-stage pipeline featuring a traditional RISC datapath with a dual-read-ported register file feeding multiple execute engines. In this design, the pipeline stages have multiple functions. The Decode & Select stage and the Operand Cycle stage are both done in the DSOC cycle. Address Generation and the Execute Cycle are performed in the AGEX cycle.

The basic OEP contains two execute engines: an ALU and a barrel shifter. Each execute engine is a three-port device with two input operands producing an execute result.

The MAC unit implements a three-stage arithmetic pipeline containing the multiply array followed by adder logic. The first stage of the MAC pipeline corresponds to the AGEX cycle of the OEP and the third stage represents the final execute cycle, EX3.

Consider a simple register-to-register MAC operation. During the DSOC stage of the OEP, the instruction is decoded and the source operands are fetched from the register file. During the AGEX cycle, the source operands are sent to the MAC unit along with information defining the exact operation to be performed. The formation of the partial products is completed in the AGEX stage and

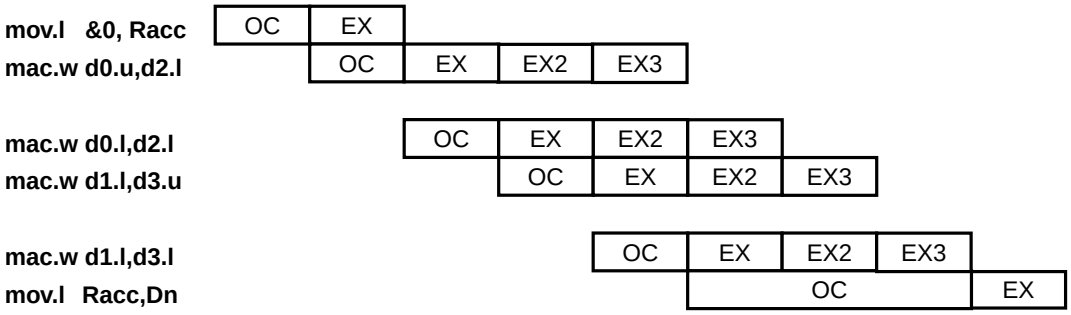

Fig. 1 Pipeline timing example with MAC instructions

the summation of these partial products started. During the final EX3 cycle, the formation of the product is completed and final summation involving the accumulator is performed. At the conclusion of the EX3 cycle, the final result is loaded into the accumulator. A timing diagram is shown in Figure 1.

In addition to performing the MAC operations, the integer multiply opcodes (MULS, MULU) are executed in the MAC unit. By executing these opcodes in the MAC unit, the instruction execution times are minimized and deterministic compared to the 2-bit/cycle algorithm with early termination normally used by the OEP.

3. GENERAL OPERATION

The MAC unit is designed to support the integer multiply instructions and provide additional functionality for multiply-accumulate operations. The MAC module has been optimized for 16-bit multiplications in order to keep the area consumption low. Two 16-bit operands produce a 32-bit result, and both signed and unsigned operations are supported. Long operations are performed by reusing the multiplier array at the expense of a small amount of extra control logic. Again, the product of two 32-bit operands is a 32-bit result.

The new instructions to the ColdFire ISA are shown in Table 1. They provide for the multiplication of two numbers, followed by the addition/subtraction of this number to/from the value contained in the accumulator. The product may be optionally shifted left or right one bit before the addition or subtraction takes place. Hardware support for saturation arithmetic can be enabled by setting the Overflow/Saturation Mode Control bit in the status register. This can be used to minimize software overhead when dealing with potential overflow conditions using signed operands.

Since the multiplier array is implemented in a 3-stage pipeline, the MAC instructions have an effective issue rate of one clock for word operations and three clocks for long operations. All arith-

Table 1: MAC Instruction Summary

Instruction Summary	Command Mnemonic	Description
Multiply Signed	MULS <ea>y,Dx	Multiplies two signed operands yielding a signed result
Multiply Unsigned	MULU <ea>y,Dx	Multiplies two unsigned operands yielding an unsigned result
Multiply Accumulate	MAC Ry,RxSF MSAC Ry,RxSF	Multiplies two unsigned/signed operands, then adds/subtracts the product to/from the accumulator
Multiply Accumulate with Load	MAC Ry,RxSF,<ea>,Rw MSAC Ry,RxSF,<ea>,Rw	Multiplies two unsigned/signed operands, then adds the product to the accumulator while loading a register with the memory operand
Load Accumulator	MOV.L {Ry,#imm},Racc	Loads the accumulator with 32-bit operand
Store Accumulator	MOV.L Racc,Rx	Writes the contents of the accumulator to a register
Load MAC Status Register	MOV.L {Ry,#imm},MACSR	Writes a value to the MAC status register
Store MAC Status Register	MOV.L MACSR,Rx	Writes the contents of the MAC status register to a register
Move MACSR to CCR	MOV.L MACSR,CCR	Writes the contents of the MAC status register to processor's CCR
Load Mask Register	MOV.L {Ry,#imm},Rmask	Loads mask register with lower 16-bits of operand
Store Mask Register	MOV.L Rmask,Rx	Writes mask register to a register

metic operations use register-based input operands, and summed values are stored internally in the accumulator. Thus, an additional move instruction is necessary to transfer the data to a general-purpose register. One feature added to the 16x16 MAC instructions is the ability to choose the upper or lower word of a register as the input operand. As an example, in the case of a filtering operation, this is useful if one data register is loaded with thirty-two bits of input data and another register is loaded with thirty-two bits of coefficient data. Two 16-bit MACs can be completed without having to fetch additional operands between instructions by alternating the word choice during the calculations.

One obstacle in obtaining high throughput rates in DSP engines is moving large amounts of data quickly. Large blocks of data are most efficiently moved using the MOVEM opcode. Since this instruction automatically generates line-sized burst references, it is ideal for loading registers quickly with input data or filter coefficients or for storing output data. A new instruction combines the ability to load an operand from memory into a register at the same time that a MAC operation is being performed. This makes certain DSP operations much more manageable, especially digital filtering and convolution.

Three program-visible registers have been added with the MAC unit: the 32-bit accumulator (Racc), the status register (MACSR), and a mask register (Rmask). The status register contains the overflow/saturation mode control bits, the negative, zero, and overflow flags, as well as signed/unsigned operation control bits. The mask register allows an operand address to be effectively constrained within a certain range defined by this 16-bit value. This feature minimizes the addressing support required for filtering, convolution, or any routine that uses circular buffers or queues.

The MAC unit is capable of shifting a product before the result is added or subtracted to or from the accumulator. Since there is the possibility of overflowing a 32-bit product, the following guidelines are followed when performing MAC instructions. For all left shifts, a zero is inserted next to the least significant bit of the product. For unsigned operations, both word and long, a zero is shifted into the product on right shifts. For signed, word-length operations, the sign bit is shifted into the product on right shifts, unless the product itself is zero. For signed, long operations, the sign bit is shifted into the product unless an overflow occurs or the product itself is zero, in which case a zero is shifted in.

4. APPLICATIONS

This section details two examples of algorithms that occur frequently in signal processing: a transform and a simple digital filter. Additionally, measured performance results from a disk servo application are presented.

A. Discrete Cosine Transform

The first example demonstrates both matrix multiplication and the use of the multiply-accumulate instruction with a two-dimensional 8x8 Discrete Cosine Transform (DCT). The DCT has become an attractive alternative to the Karhunen-Loève transform for image compression because it compares closely with respect to rate distortion criteria, and because there are several efficient ways to implement it.

A number of algorithms have been recognized over the years, most notably those of Hou [2] and Lee [3], which create higher-order DCT's from lower-order ones. In fact, it can be shown that the algorithm for computing the Discrete Cosine Transform resembles that of a Fast Fourier Transform (with some postprocessing of the data). However, for the purpose of illustration, a much more direct method is used here, namely to use the matrix formulation to multiply the input data by the coefficients. Although this method will work, it tends to be fairly slow, and in practice, a faster implementation can be achieved by using the signal flow graph [2] since the number of multiplies is greatly reduced.

The two-dimensional DCT is given by:

$$\theta(k,l) = \frac{2}{N}\alpha(k)\alpha(l)\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n)\cos\left(\frac{\pi k(2m+1)}{2N}\right)\cos\left(\frac{\pi l(2n+1)}{2N}\right)$$

where *x(m,n)* is an *N*x*N* field, and $\alpha(k) = 1/(\sqrt{2})$ for $k = 0$, unity otherwise. If the DCT is carried out with a matrix formulation, the routine is based on the following equation:

$$\begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\\ \theta_3\\ \theta_4\\ \theta_5\\ \theta_6\\ \theta_7\end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\ \lambda & \gamma & \mu & \nu & -\nu & -\mu & -\gamma & -\lambda\\ \beta & \delta & -\delta & -\beta & -\beta & -\delta & \delta & \beta\\ \gamma & -\nu & -\lambda & -\mu & \mu & \lambda & \nu & -\gamma\\ \alpha & -\alpha & -\alpha & \alpha & \alpha & -\alpha & -\alpha & \alpha\\ \mu & -\lambda & \nu & \gamma & -\gamma & -\nu & \lambda & -\mu\\ \delta & -\beta & \beta & -\delta & -\delta & \beta & -\beta & \delta\\ \nu & -\mu & \gamma & -\lambda & \lambda & -\gamma & \mu & -\nu \end{bmatrix}\begin{bmatrix}x_0\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\end{bmatrix}$$

where $\beta = \sin\left(\frac{\pi}{8}\right)$, $\delta = \cos\left(\frac{\pi}{8}\right)$, $\lambda = \cos\left(\frac{\pi}{16}\right)$, $\gamma = \cos\left(\frac{3\pi}{16}\right)$, $\mu = \sin\left(\frac{3\pi}{16}\right)$, and $\nu = \sin\left(\frac{\pi}{16}\right)$.

The DCT kernel is separable (like the FFT) so the transform can be performed in two passes, first along the rows and then along the columns. The inverse DCT can be created by replacing the coefficients with those for an inverse transform. In this implementation, a separate matrix transpose routine is not necessary before the second pass, since the operands are stored in memory in their transposed position during the first pass. Note that the transform here is not unitary, so the scaling factor of $\frac{2}{N}$ only applies to the forward transform.

The bulk of the DCT routine is comprised of eight small loops, one of which is shown in Figure 2. Each loop calculates one vector of output data. In this implementation, all coefficients and the data samples are word-length operands. Instead of fetching two operands to multiply and then accumulating the result, a **movm** instruction loads four general-purpose registers with all eight data samples, and another **movm** loads four more registers with all eight coefficients. The MAC instructions are done in series, effectively one per clock. Using the upper/lower word select bit, two MAC operations are performed using the same registers. The accumulator is then transferred to a general-purpose register and moved out to memory.

```
        movm.l  (%a0),&0x0078     # load d3-d6
loop_one:
        movm.l  (%a1),&0x7800     # load a3-a6
        mov.l   &0x3000,%acc      # rounding value
        mac.w   %d3:u,%a3:u
        mac.w   %d3:l,%a3:l
        mac.w   %d4:u,%a4:u
        mac.w   %d4:l,%a4:l
        mac.w   %d5:u,%a5:u
        mac.w   %d5:l,%a5:l
        mac.w   %d6:u,%a6:u
        mac.w   %d6:l,%a6:l
        mov.l   %acc,%d7
        swap    %d7
        mov.w   %d7,(%a2)
        add.l   doffset,%a1
        lea     16(%a2),%a2
        subq.l  &1,%d2            # decrement loop
        bne.b   loop_one
```

Fig. 2 Assembly code that computes one vector of data

A method similar to the one by Ford [4] and Srinivasan et al. [5] was used to account for roundoff errors and truncation. The coefficients themselves are prescaled, and only the upper 16 bits of the accumulator are saved (only the integer portion of the intermediate products are kept).

To measure performance, the code was executed on a cycle-accurate ColdFire simulation model containing 512 bytes of instruction cache, 512 bytes of onboard RAM, and the ColdFire core with the MAC unit. Both source and coefficient data were stored in external memory, while intermediate data was stored in the RAM. Depending on the ColdFire part that is actually used, these storage areas could easily be changed to improve performance. For the configuration used here, the DCT routine executed in approximately 6,300 cycles, or 189 microseconds at a frequency of 33 MHz.

B. Filters

Digital filters are found in a host of embedded applications. Examples include speech and image processing, and servo controllers in hard disk drives. In general, they can be described by

$$y(i) = \sum_{k=1}^{N-1} a(k)y(i-k) + \sum_{k=0}^{N-1} b(k)x(i-k)$$

where the output *y(i)* is determined by past output

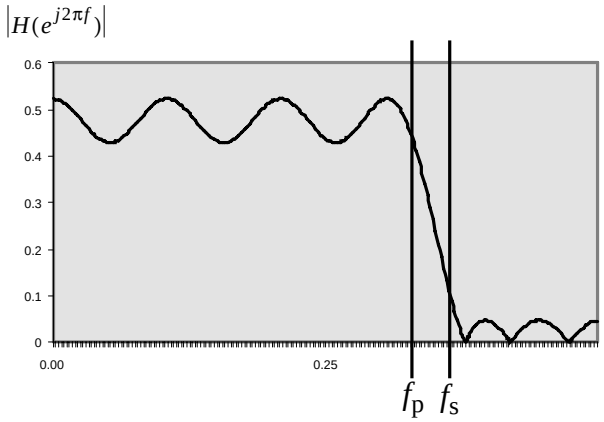

Fig. 3 Magnitude response of a 21-tap FIR filter

and/or input values *x(i)*. This is the general form of an infinite impulse response (IIR) filter. A finite impulse response (FIR) filter can be obtained by setting the coefficients *a(k)* to zero. In either case, the operations involved in computing such a filter are multiplies and a summing of products.

The design of a digital filter is based on the desired frequency response, the type of filter structure used, and the type of processor or range of precision used to implement it. Suppose that a low-pass FIR filter is specified with a magnitude response like the one shown in Figure 3. A large pass band (f_p) is required, from a normalized frequency of 0 to 0.33, and the stop band (f_s) is specified to begin at 0.37.

Using these criteria, the coefficients *h(k)* are found using the Parks-McClellan algorithm [6], which is one of several ways in which to determine FIR filter coefficients. These coefficients are quantized to 16 bits, which for our example, are considered to be adequate. In general, though, finite word-length effects need to be carefully evaluated for shorter filters. There are a number of readable texts that deal with unit-pulse response coefficient calculation, including Williams[7], Oppenheim and Schafer[8], and Ifeachor and Jervis[9].

Scaling and overflow must be accounted for in fixed-point operations. Since the output of the filter is given by

Table 2: l_1 Scaled Coefficients

Coefficient	Decimal	Hex
h_0, h_{20}	0.008697	011D
h_1, h_{19}	0.026267	035D
h_2, h_{18}	-0.019490	FD82
h_3, h_{17}	0.007113	00E9
h_4, h_{16}	0.013134	01AE
h_5, h_{15}	-0.028302	FC62
h_6, h_{14}	0.021353	02BC
h_7, h_{13}	0.015199	01F2
h_8, h_{12}	-0.071330	F6DF
h_9, h_{11}	0.122376	0FAA
h_{10}	0.333457	2AAF

$$y(k) = \sum_{m=0}^{N-1} h(m)x(k-m)$$

and the input *x(n)* is assumed to have a magnitude of at most unity, the coefficients can be scaled by the l_1 norm of h,

$$\|h\|_1 = \sum_n |h(n)|$$

to avoid overflow completely. Table 2 shows the scaled unit-pulse response samples for the filter.

If a less conservative scaling strategy is used (such as scaling by the l_2 norm), then saturation operations can be used to minimize problems caused by overflow. The MAC unit also provides a saturation mode to handle overflow conditions. If the OMC bit is set in the MAC status register, the accumulator is set to the most positive or the most negative value on any operation which overflows the 32-bit accumulator.

The next decision to make is the structure of the filter. This decision is based on factors such as ease of programming, sensitivity to coefficient

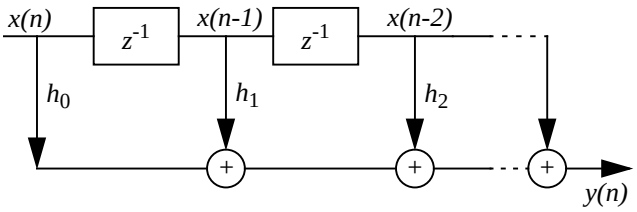

Fig. 4 Direct nonrecursive structure of an FIR filter

errors, and regularity of the design. For this example, a direct nonrecursive structure is implemented, which would look something like the filter in Figure 4. The boxes labeled z^{-1} represent unit delays, which in an assembly language program merely represent storage variables.

A short example of code is provided in the Appendix for the FIR filter. The routine begins by loading the mask register with the complement of the sum of the address of the top of the queue and the value 0x3F. In general, this type of filtering operation is optimally performed using a circular buffer, where the buffer length is rounded up to the nearest power of two. In this example, the 21-tap filter would be implemented using a 32-entry buffer. By aligning the base of the circular buffer on a 0-modulo-128 address (i.e., 0-modulo-4N, where N is the number of 16-bit entries in the queue), the mask register can easily be used to constrain the address register pointing to the data samples in the desired range with minimum overhead. See Figure 5.

Once the starting addresses of the sample data and the coefficients are stored in registers, all of the coefficients are loaded into registers d0 through d5 via the **movm** instruction. Since there are only 21 coefficients (all but one of which are doubled), this is possible; larger filters will have to load samples and coefficients as memory permits. The routine then calls a subroutine which retrieves the newest input sample. This subroutine is also responsible for maintaining and checking the address pointer that indicates the top of the queue. After each pass through the filter, the top of the queue is decremented until it reaches the lower

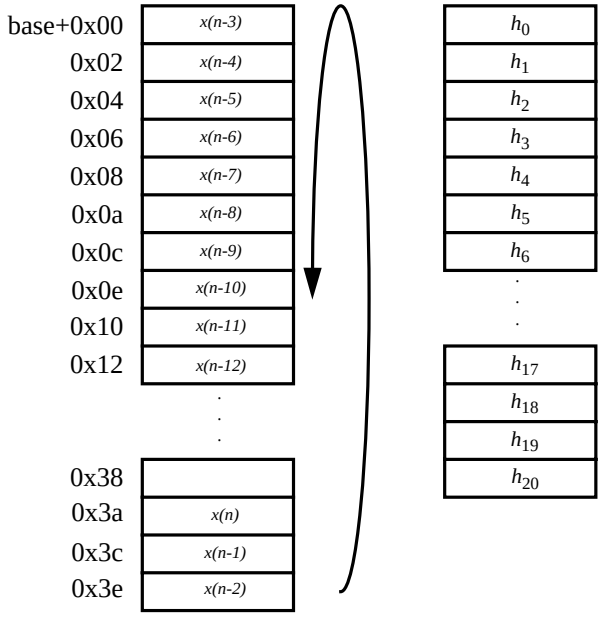

Fig. 5 Arrays of input data samples and coefficients

address boundary. Once there, it is reset to the bottom of the stack on the next pass.

Once the pointer is set to the top of the array, pairs of samples and coefficients are multiplied (similar to the DCT routine), where several multiplications are performed with the same registers by alternating the upper and lower word select bits. By using a MAC instruction with a parallel move and the post-increment addressing mode, the effective address of each operation is AND'ed with the contents of the mask register *after* being incremented. The contents of the accumulator are stored to memory and the routine branches to the beginning of the code.

C. Disk Servo Code

In another application, the performance of a disk servo controller was analyzed. These types of applications are generally characterized by significant amounts of control functions (testing variables and conditional branches) along with signal processing (low-pass filtering) and estimation calculations for linear-state feedback control functions.

In the initial version of this C language applica-

Table 3: Measured Disk Servo Performance

Application	Relative Performance
Compiled C, no MAC opcodes	1.00x
Compiled C, MAC macros	1.45x
Compiled C, MAC + Handtuning	1.69x

tion, approximately 11% of the total executed instructions were multiply operations. Next, the source code was modified to reference assembly language macros which defined the MAC opcodes. For this optimization, the compiler output was executed and the resulting performance measured. Finally, the functions which performed the filtering operation were optimized in assembly language, and the performance was measured again.

5. INSTRUCTION EXECUTION TIMES

The execution times for MAC operations are provided in Table 4. These numbers are presented in terms of **C(r/w)**, where **C** is the processor core clock cycles, **r** is the number of operand memory reads, and **w** is the number of operand memory writes. The timing assumptions are the same as those for the ColdFire Family instruction set architecture and can be found in [10].

6. CONCLUSIONS

As described in the previous sections, the MAC unit extends the reach of the ColdFire family of processors into areas traditionally reserved for dedicated signal processors. The optimized performance coupled with an approximate 10,000 gate implementation size make the MAC unit a cost-effective module for those embedded applications requiring digital signal processing capabilities.

The MAC unit is included as part of the ColdFire processor core available in the FlexCore program in the latter half of 1996. Additionally, the MAC unit will be included in several new standard ColdFire microprocessor products in the future.

7. REFERENCES

[1] J. Circello, “ColdFire: A Hot Processor Architecture,” *Byte*, Vol. 20, no. 5, pp. 173-174, May 1995.
[2] H.S. Hou, “A Fast Recursive Algorithm for Computing the Discrete Cosine Transform,” *IEEE Transactions on ASSP*, vol. ASSP-35, No. 10, Oct. 1987, pp. 1455-1461.
[3] B.G. Lee, “FCT - Fast Cosine Transform,” *Proceedings of 1984 Conference on ASSP*, March 1984, pp. 28.A.3.1-28.A.3.4.
[4] W. Hohl, “8x8 Discrete Cosine Transform Implementation on the TMS320C25 or the TMS320C30,” *Digital Signal Processing Applications with the TMS320 Family*, vol. 3, pp. 169-190, 1990.
[5] S. Srinivasan et al., “Cosine Transform Block Codec for Images Using the TMS32010,” *Proceedings of IEEE ISCAS ‘86*, vol. 1, pp. 299-302.
[6] T.W. Parks and C.S. Burrus, *Digital Filter Design*, New York: John Wiley & Sons, 1987.
[7] C.S. Williams, *Designing Digital Filters*, New Jersey: Prentice-Hall, 1986.
[8] A.W. Oppenheim and R.W. Schafer, *Digital Signal Processing*, New Jersey: Prentice-Hall, 1975.
[9] E.C. Ifeachor and B.W. Jervis, *Digital Signal Processing, A Practical Approach*, Addison-Wesley, 1993.
[10] *MCF5200 Family Programmer’s Reference Manual*, Motorola Incorporated, 1996.

William Hohl is a systems architect with Motorola’s High Performance Embedded Systems Division. He received the BSEE and MSEE degrees from Texas A&M University, and his interests include digital signal processing and computer architecture. He previously designed the debug unit for the ColdFire Family, and most recently developed the MAC architecture.

Joe Circello is a microprocessor architect for Mo-

Table 4: Instruction Execution Timings

Opcode	<ea>	Effective Address							
		Rn	(An)	(An)+	-(An)	(d16,An) (d16,PC)	(d8,An,Xn*SF) (d8,PC,Xn*SF)	xxx.wl	#xxx
MULS.W	<ea>y,Dx	4(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	7(1/0)	6(1/0)	4(0/0)
MULU.W	<ea>y,Dx	4(0/0)	6(1/0)	6(1/0)	6(1/0)	6(1/0)	7(1/0)	6(1/0)	4(0/0)
MULS.L	<ea>y,Dx	6(0/0)	8(1/0)	8(1/0)	8(1/0)	8(1/0)	-	-	-
MULU.L	<ea>y,Dx	6(0/0)	8(1/0)	8(1/0)	8(1/0)	8(1/0)	-	-	-
MAC.W	<ea>y,Rx	1(0/0)	-	-	-	-	-	-	-
MAC.L	<ea>y,Rx	3(0/0)	-	-	-	-	-	-	-
MSAC.W	<ea>y,Rx	1(0/0)	-	-	-	-	-	-	-
MSAC.L	<ea>y,Rx	3(0/0)	-	-	-	-	-	-	-
MAC.W	Ry,Rx-SF,<ea>,Rw	-	2(1/0)	2(1/0)	2(1/0)	2(1/0)*	-	-	-
MAC.L	Ry,Rx-SF,<ea>,Rw	-	4(1/0)	4(1/0)	4(1/0)	4(1/0)*	-	-	-
MSAC.W	Ry,Rx-SF,<ea>,Rw	-	2(1/0)	2(1/0)	2(1/0)	2(1/0)*	-	-	-
MSAC.L	Ry,Rx-SF,<ea>,Rw	-	4(1/0)	4(1/0)	4(1/0)	4(1/0)*	-	-	-
MOV.L	<ea>,ACC	1(0/0)	-	-	-	-	-	-	1(0/0)
MOV.L	<ea>,MACSR	1(0/0)	-	-	-	-	-	-	1(0/0)
MOV.L	<ea>,MASK	1(0/0)	-	-	-	-	-	-	1(0/0)
MOV.L	ACC,<ea>	3(0/0)	-	-	-	-	-	-	-
MOV.L	MACSR,<ea>	3(0/0)	-	-	-	-	-	-	-
MOV.L	MASK,<ea>	3(0/0)	-	-	-	-	-	-	-

torola's High Performance Embedded Systems Division. With 21 years of experience in mainframes to microprocessors, he is a veteran designer specializing in pipeline organization and performance analysis. While at Motorola, he was pipeline architect for the 68060 and has developed the ColdFire architecture.

Appendix

Example source code for the FIR filter

```
main:
   clr.l       %d0                    # zero the data register
   clr.l       %d1                    # zero the data register
   clr.l       %d2                    # zero the data register
   clr.l       %d3                    # zero the data register
   clr.l       %d4                    # zero the data register
   clr.l       %d5                    # zero the data register
   clr.l       %d6                    # zero the data register
   clr.l       %d7                    # zero the data register
   lea         queue,%a0              # pointer to sample queue
   movm.l      #0x00ff,(%a0)          # initialize the sample queue
   lea         -2(%a0),%a0            # adjust queue pointer
   movm.l      #0x00ff,(%a0)          # finish initializing the sample queue
   lea         queue+0x3e,%a0         # address of last entry in queue
   mov.l       %a0,%d0
   not.l       %d0                    # create value for mask register
   mov.l       %d0,%mask              # initialize MAC mask register
   lea         coeff,%a1              # base of coefficient table
   movm.l      (%a1),&0x00fc          # load coefficients into d2-d7
L%1:
   bsr.b       do_filter              # perform 21-tap FIR filter
   bra.b       L%1                    # loop continously

do_filter:
   mov.l       #0,%acc                # initialize accumulator
   bsr.b       get_sample             # retrieve sample x(n)
                                      # get_sample argument:
                                      # queue pointer in %a0
                                      # get_sample returns:
                                      # x(n), x(n-1) in %d0
                                      # pointer to x(n-2) in %a0
   mac.w       %d2:u,%d0.u,(%a0)+&,%d1
               # h( 0) * x(n) {x(n- 2), x(n- 3)} -> d1
   mac.w       %d2:l,%d0.l            # h( 1) * x(n- 1)
   mac.w       %d3:u,%d1.u,(%a0)+&,%d0
               # h( 2) * x(n- 2) {x(n- 4), x(n- 5)} -> d0
   mac.w       %d3:l,%d1.l            # h( 3) * x(n- 3)
   mac.w       %d4:u,%d0.u,(%a0)+&,%d1
               # h( 4) * x(n- 4) {x(n- 6), x(n- 7)} -> d1
   mac.w       %d4:l,%d0.l            # h( 5) * x(n- 5)
   mac.w       %d5:u,%d1.u,(%a0)+&,%d0
               # h( 6) * x(n- 6) {x(n- 8), x(n- 9)} -> d0
   mac.w       %d5:l,%d1.l            # h( 7) * x(n- 7)
   mac.w       %d6:u,%d0.u,(%a0)+&,%d1
               # h( 8) * x(n- 8) {x(n-10), x(n-11)} -> d1
   mac.w       %d6:l,%d0.l            # h( 9) * x(n- 9)
   mac.w       %d7:u,%d1.u,(%a0)+&,%d0
               # h(10) * x(n-10) {x(n-12), x(n-13)} -> d0
   mac.w       %d6:l,%d1.l            # h(11) * x(n-11)
   mac.w       %d6:u,%d0.u,(%a0)+&,%d1
               # h(12) * x(n-12) {x(n-14), x(n-15)} -> d1
   mac.w       %d5:l,%d0.l             # h(13) * x(n-13)
   mac.w       %d5:u,%d1.u,(%a0)+&,%d0
               # h(14) * x(n-14) {x(n-16), x(n-17)} -> d0
   mac.w       %d4:l,%d1.l            # h(15) * x(n-15)
   mac.w       %d4:u,%d0.u,(%a0)+&,%d1
               # h(16) * x(n-16) {x(n-18), x(n-19)} -> d1
   mac.w       %d3:l,%d0.l            # h(17) * x(n-17)
   mac.w       %d3:u,%d1.u,(%a0)+&,%d0
               # h(18) * x(n-18) {x(n-20)} -> d0
   mac.w       %d2:l,%d1.l            # h(19) * x(n-19)
   mac.w       %d2:u,%d0.u            # h(20) * x(n-20)
   mov.l       %acc,%d0               # move accumulator to general register
   swap        %d0                    # align most significant 16 bits of acc
   mov.w       %d0,<ea>               # store y(n)
   rts

get_sample:
      .
      .
      .
   rts

   data
   align       104
coeff:
   long        0x011d035d             # h( 0)/h(20), h( 1)/h(19)
   long        0xfd8200e9             # h( 2)/h(18), h( 3)/h(17)
   long        0x01aefc62             # h( 4)/h(16), h( 5)/h(15)
   long        0x02bc01f2             # h( 6)/h(14), h( 7)/h(13)
   long        0xf6df0faa             # h( 8)/h(12), h( 9)/h(11)
   long        0x2aaf2aaf             # h(10)

# queue must be aligned on 0-mod-128 address
queue:         space                  64
```