

The Motorola DSP56000 Digital Signal Processor

The DSP56000 brings 10.25-MIPS performance to digital signal processing and retains enough similarities to other Motorola microprocessors to make it easy to learn and program.

Kevin L. Kloker
Motorola, Inc.

The Motorola DSP56000 is a high-performance, user-programmable digital signal processor implemented in high-density, low-power CMOS technology. The first member of a new family of special-purpose microprocessors designed specifically for digital signal processing applications,^{1,2} it has a highly parallel architecture and can execute 10.25 million instructions per second. Its instruction set supports the real-time processing of 24-bit data with 56 bits of internal arithmetic precision. Combining a core processor, RAM, ROM, peripheral interfaces, and a memory expansion interface on a single, 88-pin chip, the DSP56000 represents the state of the art in digital signal processor design.

Though the DSP56000 has hardware and software features specifically added to support digital signal processing, it shares enough similarities with other Motorola microprocessors and single-chip microcomputers that it can be used as a very fast microprocessor in any application that requires high-speed calculations or real-time response. For example, it can be used as a controller in applications previously served by bit-slice processors. Providing both single-chip and expanded bus operation, it also fits many applications beyond the capabilities of 8- and 16-bit microcontrollers.

The digital signal processing environment

Digital signal processing is concerned with the real-time processing of digitized analog signals, which are discrete in

both amplitude and time.³ To illustrate the environment of a typical DSP system, I will consider a simple digital filtering application (Figure 1). The system shown is intended to filter an analog signal using digital means. The system starts with an analog input signal $x(t)$, which is converted to a sampled digital signal $x(n)$ by an analog-to-digital, or A/D, converter. As long as the system samples the analog input at a frequency f_s that is at least twice the information bandwidth of that input, all information present in the original analog signal is contained in the digital signal. No signal information is lost. However, the quality of the conversion to a discrete amplitude signal introduces quantization noise into the system. The sampled digital signal—that is, its signal-to-quantization-noise ratio, or SQNR—is a function of the A/D converter's accuracy. Choosing the resolution (number of bits) and linearity of the A/D converter is a trade-off between cost and the SQNR requirements of the system.⁴

Once the system has obtained a faithful digital representation of the analog signal, it can filter that representation in the digital signal processor. The digital signal processor stores the current A/D sample and $N-1$ previous samples in a sample shift register. The DSP56000, however, stores this data in a RAM and simulates the shift register function by modifying memory address pointers. The set of N filter coefficients $h(i)$, $i = 0, 1, \dots, N-1$, must also be stored in a RAM or ROM; these coefficients determine the impulse response and the filter characteristics. A larger N gives a longer impulse response and generally produces filters with sharper roll-off, greater stopband attenuation, and less fre-

quency ripple. It does so at the expense of more calculations per output sample and more storage for sample data and coefficients. Since the order N of the filter is finite and no feedback path exists to sustain a nonzero filter output given a zero input, the filter is called an N th order, finite impulse response, or FIR, digital filter. The FIR filtering operation requires N multiplies and $N-1$ additions to compute an output $y(n)$ each time the input signal is sampled. This operation is the kernel of digital signal processing algorithms. The DSP56000 contains a parallel hardware multiply/accumulate circuit that completes this operation in 97.5 ns. Repeated use of the multiply/accumulate operation produces a sum-of-products result, which is the FIR filter output $y(n)$. This output is converted to an analog signal $y(t)$ by a digital-to-analog, or D/A, converter. The analog output $y(t)$ is now a filtered version of the analog input $x(t)$.

The operation of the system shown in Figure 1 is straightforward. A sample is taken, a filter calculation is performed, and a filter output is sent to the D/A converter. The data samples are then time-shifted (delayed) by one sampling period and the operations are repeated in the next sampling period. A large amount of data must be stored, processed, and time-shifted during each sampling period. In practice, digital signal processing systems can be much more complex than this simple example. However, the example shows what characterizes every digital signal processing en-

vironment—multiply/accumulates, time shifts, and input/output in real time.

DSP advantages. The primary advantage of digital signal processing is that it transforms analog functions into digital hardware. It extends the advantages of digital circuitry—high density, precision, stability, and testability—to those parts of systems previously served by analog components. Digital signal processing also transforms analog functions in hardware into digital signal processing algorithms. Traditional analog functions such as filters are replaced by their digital equivalents in software form. Software is inherently flexible and thus is better suited to complex systems. Finally, digital signal processing implements applications that analog signal processing cannot. It makes possible digital speech transmission, storage, and synthesis for communications systems, for example.⁵

DSP limitations. DSP users cite the inability of existing devices to provide high processing speed and large memory sizes as one of the most frequent system limitations. Some DSP applications involve sampling rates of up to 100 MHz and can require hundreds of millions of multiply/accumulates per second. Another problem exists as well. Because DSP is a new discipline, engineers are not familiar with it and require training and experience in it. To apply digital signal processing techniques, they must rethink traditional product development approaches.

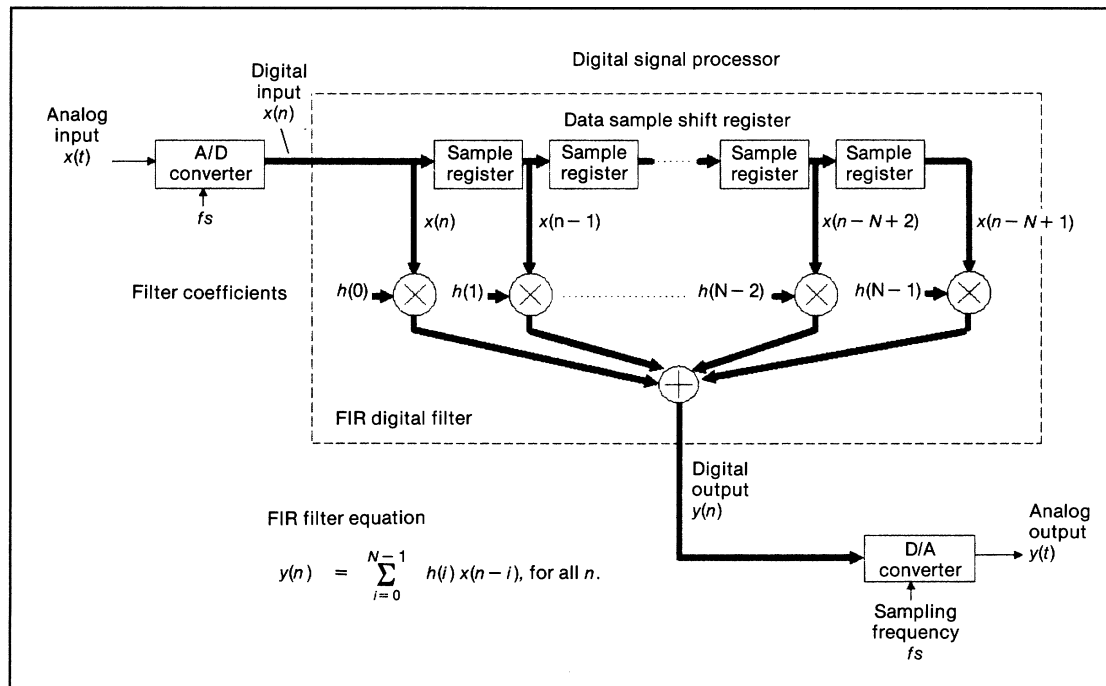


Figure 1. A simple digital filtering system.

DSP56000 features

The DSP56000 incorporates many new features designed to overcome the limitations of earlier DSP devices. It advances the state of the art in three areas—speed, precision, and integration.

Speed. Invariably, DSP users identify processor speed and system performance in a real-time, I/O-intensive environment as the most important factors in their designs. The speed with which a digital signal processor can execute DSP algorithms is critical to the realization of real-time operation. The DSP56000 achieves 10.25 MIPS and 10.25 million multiplies and accumulates per second, record highs for its class of device. It executes many DSP benchmarks two to four times faster than other recently announced DSP devices.⁶⁻⁸ To achieve this performance, the DSP56000's designers developed a unique multiplier/accumulator ALU and a multiple-bus architecture. They also designed the instruction set and interrupt structure to minimize software overhead in real-time systems.

Precision. The DSP56000 provides the greater data precision needed for advanced digital signal processing applications. Users often ask for more than 16-bit data, since they often employ double-precision arithmetic, data scaling, and overflow checks to maintain 16-bit precision. Yet many users do not need the dynamic range of 32-bit floating-point arithmetic, since the majority of data converters they use are 16 bits or less. Hence, the DSP56000's designers selected a 24-bit fixed-point data format, which provides 24-bit precision without sacrificing system speed or silicon. The 24-bit data word provides 144 dB of external dynamic range, sufficient for most real-world applications. The designers also chose 56-bit accumulators internal to the data ALU to provide 336 dB of internal dynamic range. With 56-bit accumulation, no precision is lost during intermediate calculations. Twenty-four-bit precision provides eight bits of margin against overflow, round-off, and truncation errors when 16-bit data are being processed and eliminates the need for extra processing steps to maintain maximum precision. The larger data size maximizes speed by allowing all calculations to be performed in single-precision fashion.

Integration. A typical DSP system such as that shown in Figure 1 contains two independent data memories (one for data samples and one for coefficients), a separate program memory, and a parallel or serial interface to peripheral devices such as A/D and D/A converters. Anticipating such system needs, the DSP56000 includes five on-chip memories, three on-chip peripherals, and a full-speed memory expansion port. Putting ample memory and peripheral resources on the chip maximizes speed and minimizes system chip count. By integrating the elements of a DSP system in a single CMOS chip, the DSP56000 provides a cost-effective solution to DSP users. Many DSP systems incorporate host microprocessors and additional DSP ICs, memory, and peripherals. By providing both parallel and serial input/output capability in an 88-pin package, the DSP56000 supports such expanded systems.

DSP56000 architecture

A block diagram of the DSP56000 architecture is shown in Figure 2. The architecture consists of a core processor that executes the DSP56000 instruction set and other on-chip resources such as memory and peripherals that provide storage and I/O capability to the core processor. On-chip memory and peripherals are not considered part of the core and may vary from one device family member to another. The chip's pins interface both the core processor and on-chip peripherals to external devices.

Core processor. The core processor consists of three separate execution units—the data ALU, the address ALU, and the program controller—connected by multiple buses. These units operate in a parallel rather than in a pipelined fashion; i.e., each execution unit works on the same instruction at the same time. This is in contrast to heavily pipelined processors, which work on a large number of different instructions at the same time.^{9,10} Working in parallel to minimize latency, the three execution units provide all the resources the DSP56000 needs to execute instructions in a single instruction cycle. Here, I define an instruction cycle as two clock cycles; thus, an instruction cycle is 97.5 ns with a 20.5-MHz processor clock. Each execution unit is itself single-cycle and nonpipelined. The architecture of each execution unit is different, being optimized to support its role in instruction execution. Each unit contains a set of registers, arithmetic elements, and executable operations. These operations are register-oriented rather than memory-oriented. Each execution unit operates on its own local registers—source operands are read from registers within the execution unit and modified by arithmetic element operations, and the results are stored in registers within the same execution unit. Data transfers between execution units or between execution units and memory occur in parallel with internal execution unit operations. The single-cycle, register-oriented execution units are key to DSP56000 performance because they do not impose pipelining latency restrictions on the user.

Programming model

The DSP56000 user programming model is shown in Figure 3. It provides numerous register resources that are partitioned into the three execution units of the processor. The instruction set is designed to allow flexible control of these parallel processing resources. Many instructions allow the programmer to keep each execution unit busy, thus enhancing program execution speed. The user can easily program the DSP56000 from the programming model without needing to refer to a complicated hardware block diagram. Providing this capability to the user is a significant achievement, since typical DSP IC programming resembles writing microcode for a bit-slice architecture. With a typical DSP IC, the user is often burdened with difficult programming resulting from the timing complexities of pipelined data paths. The DSP56000 takes a different approach. Because of its nonpipelined execution units, timing com-

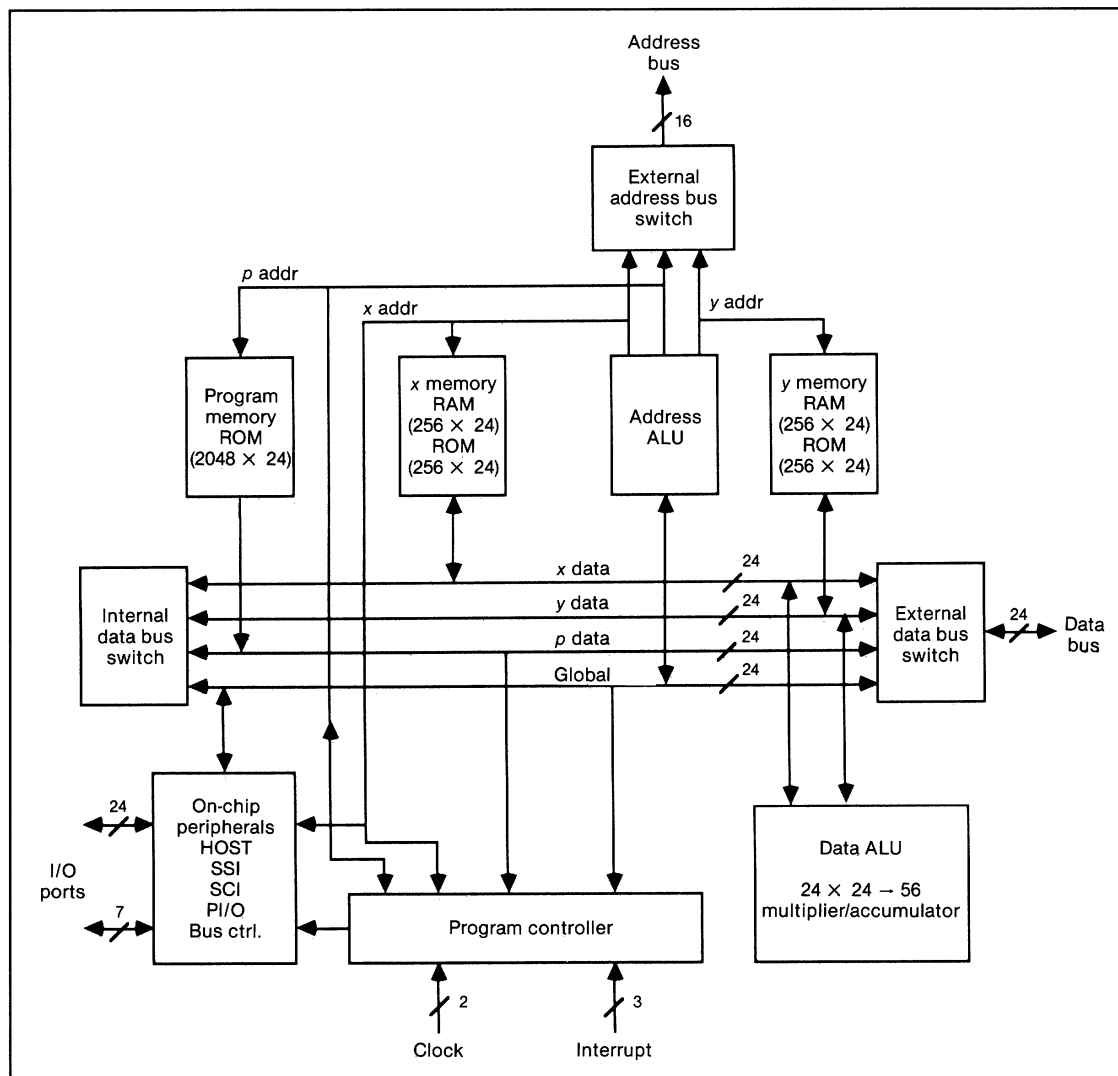


Figure 2. DSP56000 block diagram.

plexity is hidden from the user. No microcode is required and a conventional programming model, instruction set, and addressing mode definition can be used. Thus, DSP56000 programming resembles assembly language programming. The regularity of the programming model also makes possible high-level-language programming of the DSP56000.

The status register (SR) format is shown in Figure 4. The condition code register (CCR) portion indicates the results of operations on data operands. The mode register (MR) portion contains information about the system state of the processor.

The instruction set defines three separate memory spaces, *x* data, *y* data, and *p* program, which are each 65,536 locations by 24 bits wide. The total addressing capability is 196,608 24-bit words, or 589,824 bytes. All three memory spaces can be accessed in parallel in the same instruction cycle. The memory maps of these spaces are shown in Figure 5 for the normal expanded operating mode. (One obtains other memory maps by changing the operating mode register, OMR.) Noncore resources (memory and peripherals) are memory-mapped into these memory spaces to provide a clean hardware and software interface with the core processor.

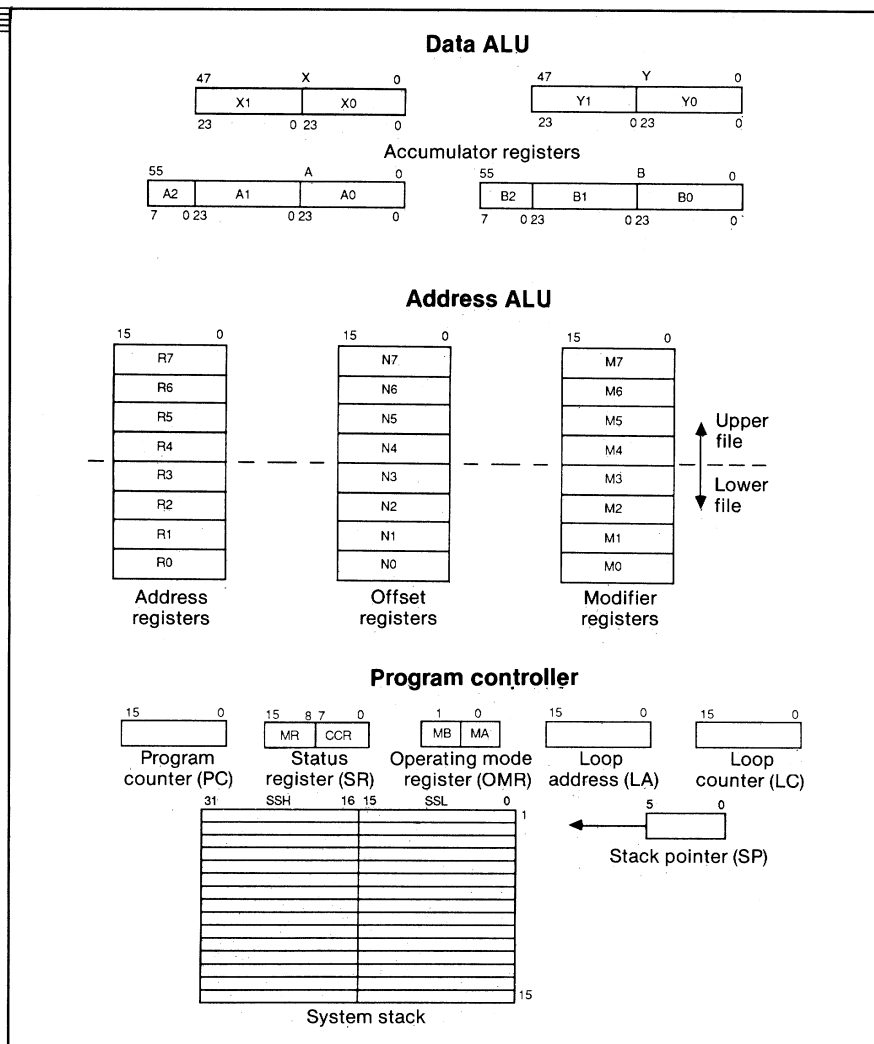
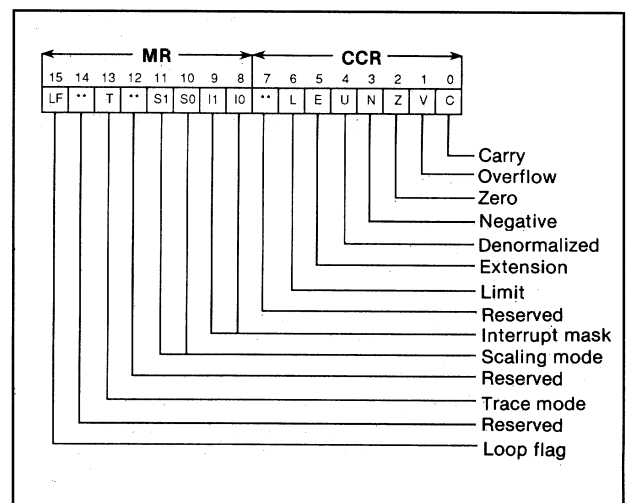


Figure 3. DSP56000 user programming model.

Figure 4. DSP56000 status register.

Data ALU

The data ALU execution unit performs arithmetic and logical operations on data operands. It consists of 10 local registers, two shifter/limiter blocks, an accumulator shifter, and a multifunction multiplier/accumulator (MAC) ALU that has two 56-bit inputs and one 56-bit output.¹¹ The two 56-bit inputs can be used for operations such as addition, subtraction, and comparison on 24-, 48-, or 56-bit numbers. During multiply and multiply/accumulate operations, one of the 56-bit inputs serves as the accumulator input while the other 56-bit input is reconfigured as a 24-bit multiplier and 24-bit multiplier input. The MAC ALU contains a 24×24 -bit parallel hardware multiplier/accumulator circuit that provides 56-bit accumulation. The MAC ALU is not pipelined and performs all operations in a single 97.5-ns instruction cycle. This is in contrast to common two-stage pipelined architectures employing a multiplier separated from an adder by a product pipeline register. The product pipeline register adds an extra cycle of delay before the



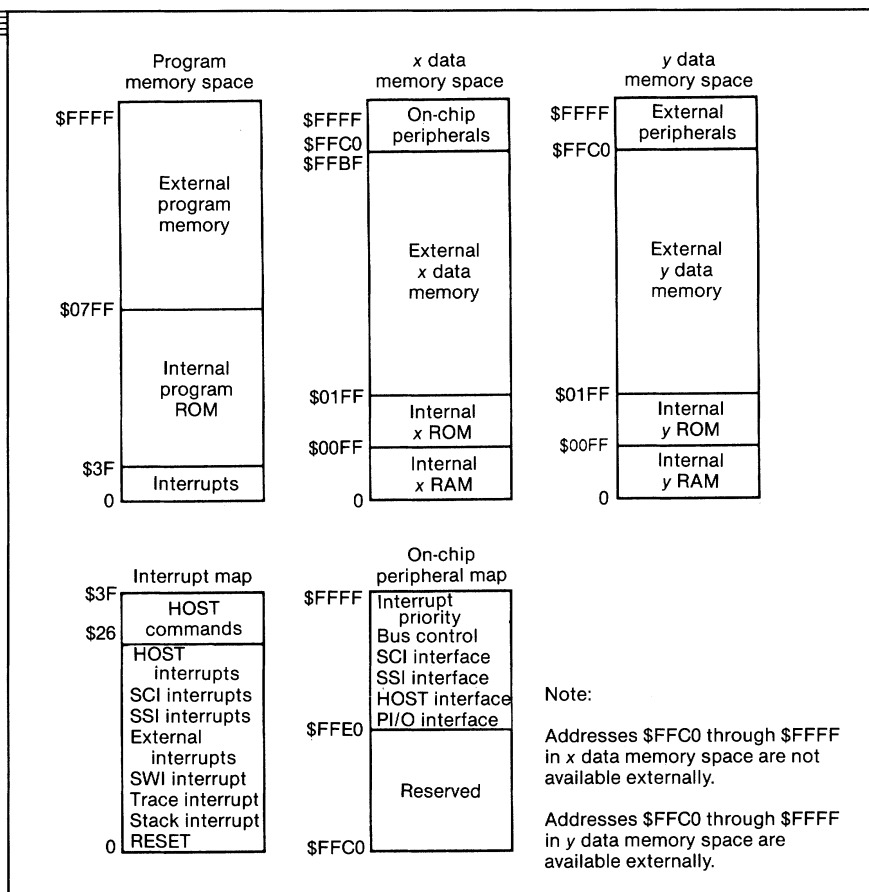


Figure 5. DSP56000 memory maps.

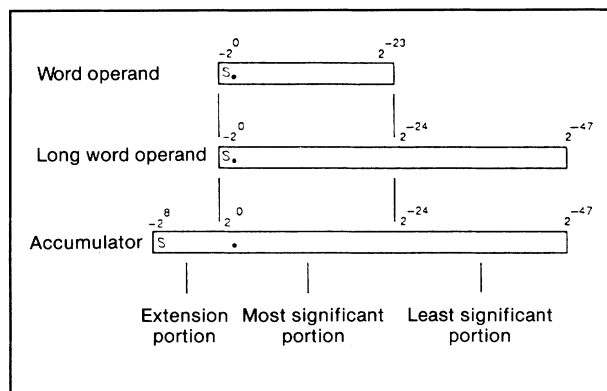


Figure 6. Bit weight and alignment of fractional data.

multiplier output is available. Since the MAC ALU has no product pipeline register, it exhibits no delay and executes many algorithms faster.

The data ALU has four 24-bit general-purpose input registers and six special output registers organized as two

56-bit (8 + 24 + 24-bit) accumulators. The input registers can be read or written as 24- or 48-bit data. The two 56-bit accumulator registers can be accessed in numerous ways as 24-, 48-, or 56-bit data. The MAC ALU output is always stored in an accumulator register. Data ALU operations support several different levels of precision, as indicated in Figure 6. Word operands are 24 bits long, long word operands are 48 bits long, and accumulator operands are 56 bits long. The MAC ALU supports the signed, two's-complement fractional data representation commonly used in digital signal processing algorithms. Fractional data, d , has a range of $-1.0 \leq d < +1.0$ and is identical to signed integer data except for the placement of the binary radix point. In practice, fractional and integer arithmetic are equivalent for all operations except multiply and divide. An integer multiplication product can be formed by a one-bit right shift of a fractional product, for example.

The data ALU provides the standard microprocessor set of logical and shifting operations to support a variety of algorithms. Logical operations are performed in the MAC ALU. A 56-bit accumulator shifter is included on one of the 56-bit MAC ALU inputs for one-bit left or right shifts. The data ALU does not contain a barrel shifter, but the MAC ALU can be used for multibit shifting operations. By multiplying the 24-bit data by a constant or variable, the

MAC ALU can perform left or right shifts of from 1 to 23 bits in a single instruction. The left-shifted result is available in the least significant portion of the accumulator and the right-shifted result is available in the most significant portion of the accumulator. This method is efficiently used by the instruction set to provide data normalization and denormalization as well as bit-field insertion and extraction.

Accumulator extension. The 56-bit accumulators store a complete 48-bit multiplication product plus eight bits of integer data called an accumulator extension. The extension portion of the 56-bit accumulator provides eight bits of protection against overflow during intermediate calculations and allows at least 256 repetitive multiply/accumulate operations before an overflow can occur. The extension bits eliminate the need to scale down the input data to avoid accumulator overflows arising from word growth in repetitive calculations. This is true because the fractional product, p , has a range of $-1.0 \leq p < +1.0$ and the 56-bit accumulator, a , has a range of $-256.0 \leq a < +256.0$.

When word or long word data are written to an accumulator register, the programmer may sign-extend its extension portion and zero its least significant portion to form a valid 56-bit signed number. When 56-bit accumulator data are read out of the data ALU, the programmer may post-process the data by enabling two special shifter/limiter circuits located between the accumulator registers and the data ALU outputs. In operation, a copy of the accumulator data is shifted left or right one bit if it has been enabled by the scaling mode bits in the status register. This allows block floating-point algorithms to be implemented for fast Fourier transforms and matrix manipulation. After shifting, the data are passed through an overflow protection circuit called a limiter. If the accumulator data can be stored in the destination without overflow, the limiter is disabled and the data are not modified. If the accumulator data cannot be stored in the destination without overflow (because the extension portion of the accumulator is in use), the limiter is enabled and substitutes the maximum data value (having the same sign) that the destination can store without overflow. This technique, called saturation arithmetic, minimizes the overflow error by avoiding the sign change usually associated with binary overflow.

Rounding. For many DSP algorithms, single-precision (24-bit) data ALU results are needed to minimize data storage requirements or to provide data for subsequent use as a multiplier input. Rounding the least significant portion of the accumulator into the most significant portion is better than truncation for maintaining maximum precision and avoiding introduction of a negative bias error. The DSP56000 provides "round to nearest even," or RN, rounding, which is performed during multiply and round (MPYR), multiply/accumulate and round (MACR), and round (RND) instructions.

Table 1.
DSP56000 addressing modes.

Addressing mode	Address modifier	Assembler syntax
Register direct	No	Any register name
Address register indirect		
No update	Yes	(Rn)
Postincrement by 1	Yes	(Rn) +
Postdecrement by 1	Yes	(Rn) -
Postincrement by offset Nn	Yes	(Rn) + Nn
Postdecrement by offset Nn	Yes	(Rn) - Nn
Predecrement by 1	Yes	-(Rn)
Index by offset Nn (Rn and Nn are unchanged)	Yes	(Rn + Nn)
Special		
Immediate data (24-bit)	No	#expr
Absolute address (16-bit)	No	expr
Immediate short data (8-, 12-bit)	No	#expr
Short jump address (12-bit)	No	expr
Absolute short address (6-bit)	No	expr
I/O short address (6-bit)	No	expr

n = register number 0 to 7
 expr = any valid assembler expression

Address ALU

The address ALU execution unit calculates addresses to locate data operands in memory. Two multiply/accumulate input operands are typically required by the data ALU at each instruction cycle. Data ALU results must be stored at a less frequent rate. The address ALU provides a flexible addressing capability through a large address register set, 14 addressing modes, and three types of address arithmetic. It consists of twenty-four 16-bit address registers, two 16-bit address arithmetic units, and an address output multiplexer. The address ALU can provide two independent memory addresses at each instruction cycle and update them with two address register indirect addressing modes. A summary of the addressing modes and their assembler syntax is given in Table 1.

The 24 address registers are organized into three sets of eight registers. The eight address registers Rn ($n = 0$ to 7) are used as address pointers to locate data operands in memory. The eight offset registers Nn are used as optional offset values to update the address registers. The offset registers may contain signed or unsigned data. The eight

Table 2.
Address modifiers.

Modifier register Mn value	Address update arithmetic for address register Rn
0	Reverse-carry (bit-reversed)
1	Modulo 2
2	Modulo 3
.	.
.	Modulo $(Mn + 1)$
.	.
32766	Modulo 32767
32767	Modulo 32768
.	.
.	Reserved
.	.
65535	Linear (modulo 65536)

modifier registers Mn select the type of address arithmetic to be performed when an address register Rn is to be updated. As shown in Table 2, the contents of the modifier registers are encoded to select the type of address arithmetic—linear, modulo, or reverse carry. The type of address arithmetic defines the type of data structure (array, sample shift register, or queue) being accessed in memory. Each address register Rn is assigned an offset register Nn and a modifier register Mn having the same register number for use in address calculations. For example, the address calculation $(R0) + N0$ postincrements the contents of address register $R0$ by the contents of offset register $N0$ using the type of address arithmetic specified by the contents of modifier register $M0$.

The address ALU output multiplexer allows any address register Rn to be used as a pointer to any memory space. This must be done so that pointers will not be duplicated in multiple registers when multiple memory spaces are accessed. For example, complex data pairs are typically stored in two data memory spaces (real part in x memory and imaginary part in y memory) at the same address. Efficient access to complex data pairs requires each pointer to be able to access both the x and y data memory spaces.

Address modifiers. During an address calculation, a set of three registers Rn , Nn , and Mn are accessed by the appropriate address arithmetic unit. The contents of the selected modifier register are decoded by the address arithmetic unit so it can determine the type of address arithmetic it should perform on the selected address pointer Rn . To understand the role of the address modifiers in creating data structures in memory, consider the examples of eight-bit address arithmetic shown in Figure 7.

The linear address modifier example is identical to conventional microprocessor address calculations, in which address updating is performed by a conventional adder. The example shows the postincrement by the offset Nn addressing mode, where the offset register contains the value 5. Linear addressing is most useful for addressing arrays of data.

The reverse-carry address modifier example is performed through the propagation of the adder carry in the reverse direction, i.e., from the most significant bit to the least significant bit of the adder. A characteristic of typical fast Fourier transform algorithms is that the data and coefficients may be stored in a nonsequential order called bit-reversed order. Reverse-carry addressing can calculate bit-reversed addresses for sequential access of FFT data and coefficients. For a 2^k -point FFT, a postincrement by 2^{k-1} using reverse-carry address arithmetic will generate the bit-reversed address sequence. The example shown generates the bit-reversed address sequence for a 16-point FFT buffer starting at address 64. Reverse-carry addressing is equivalent to doing bit-reversed addressing with simpler hardware.

The modulo address modifier example creates a circular (modulo) address region in memory with a lower boundary and an upper boundary. Modulo arithmetic keeps the address register pointing to a location within the modulo region by automatic wraparound if the pointer increments or decrements out of the modulo region. The modulo size—i.e., the length of the modulo region—is specified by the contents of the modifier register plus one. (The size is 20 in the example.) The modulo size can be any number from 2 to 32768. The lower-boundary address and upper-boundary address of the modulo region need not be directly specified, since the modulo size implicitly defines all possible modulo region boundaries. The lower-boundary address must have as many least significant zeroes as the modifier value has significant bits (five bits for $Mn = 19$). For example, a modulo region of size 20 ($Mn = 19$) can have a lower-boundary address at any integer multiple of 32 (0, 32, 64, 96, 128, 160, and so on). The upper-boundary address is the lower-boundary address plus the modulo size minus one. One of the possible modulo regions is selected implicitly by loading the address register Rn pointing to a location within a valid modulo region. ($R0 = 75$ selects the modulo region from 64 to 83 in the example.) Modulo address arithmetic simulates a shift register in memory by simply updating an address pointer and eliminates the need to move data to perform time-shift functions. Modulo address modifiers can create FIFO queues, delay lines, and sample shift registers in memory. They are also useful for interpolating and decimating filters and for generating periodic waveforms.

Program controller

The program controller execution unit performs instruction flow control, instruction decoding, and exception processing. It consists of a program address generator, or PAG, an instruction decoder, an interrupt controller, and a bus controller. The PAG is nonpipelined and calculates a new

Linear address modifier

$M0 = 255 = 1111\ 1111$ for linear addressing with R0.

Original registers: $N0 = 5$, $R0 = 75 = 0100\ 1011$

Postincrement by offset N0: $R0 = 80 = 0101\ 0000$

Postincrement by offset N0: $R0 = 85 = 0101\ 0101$

Postincrement by offset N0: $R0 = 90 = 0101\ 1010$

Reverse-carry address modifier

$M0 = 0 = 0000\ 0000$ for reverse-carry addressing with R0.

Original registers: $N0 = 8$, $R0 = 64 = 0100\ 0000$

Postincrement by offset N0: $R0 = 72 = 0100\ 1000$

Postincrement by offset N0: $R0 = 68 = 0100\ 0100$

Postincrement by offset N0: $R0 = 76 = 0100\ 1100$

Modulo address modifier

$M0 = 19 = 0001\ 0011$ for modulo 20 addressing with R0.

Original registers: $N0 = 5$, $R0 = 75 = 0100\ 1011$

Postincrement by offset N0: $R0 = 80 = 0101\ 0000$

Postincrement by offset N0: $R0 = 65 = 0100\ 0001$

Postincrement by offset N0: $R0 = 70 = 0100\ 0110$

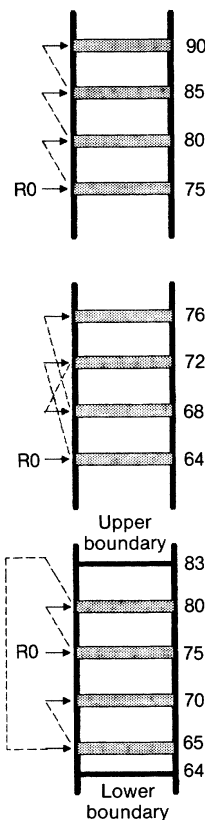


Figure 7. DSP56000 address arithmetic (8-bit examples).

instruction address every instruction cycle. Instruction prefetching is used to form a two-stage instruction pipeline having the basic timing shown in Table 3. One instruction cycle is used for the fetch operation, one for the decode operation, and one for the execute operation. During normal instruction execution, the PAG is responsible for fetching the instruction word two locations ahead of the currently executing instruction. The instruction fetch and decode operations overlap with instruction execution and take no execution time, with the exception of change-of-flow instructions and possibly PAG bus accesses. Change-of-flow instructions must fetch two instruction words to refill the instruction pipe. The PAG may not have immediate access to on-chip program memory if the currently executing instruction is accessing a data operand in on-chip program memory. The PAG may also introduce wait states during external instruction fetches because of slow off-chip program memory or because the currently executing instruction is using the memory expansion port. During a resource conflict, instruction fetches have lower priority than data accesses, since the currently executing instruction has the highest priority for use of chip resources.

Hardware DO loops. Many DSP programs spend 90 percent of the time executing in 10 percent of the program code. Digital filtering routines usually consist of a small code kernel that is executed many times. The DSP56000 provides hardware DO loop control to replace the software "decrement counter and branch" instruction normally

associated with DO loops. Straight-line coding is not needed to maximize speed since compact, looped code runs at the same speed as straight-line code. When executing DO loops, the DSP56000 eliminates the usual change-of-flow overhead by modifying the PAG instruction fetch sequence. It initiates a hardware DO loop by executing the DO instruction. The hardware DO loop mechanism stores the loop count (LC), loop starting address, and loop ending address (LA) in special registers and processes them in parallel with the executing program. Inside the DO loop, the instruction fetch address is compared to the loop ending address. When the end of the loop is detected, the loop count is tested for one. If the loop count is not one, it is decremented and the instruction word at the loop starting address is fetched. If the loop count is one, the DO loop execution is complete and normal sequential instruction fetches resume. The saving and restoring of the previous hardware DO loop registers on a stack makes it possible to nest hardware DO loops with minimal overhead. A separate system stack is used to minimize overhead during nested hardware DO loops and multilevel interrupts. This hardware stack is 32 bits wide and 15 locations deep. The double width allows two registers to be transferred to/from the system stack every instruction cycle. One can extend the system stack to any depth by moving the stack data to/from memory using software stacking techniques.

Exception processing. Exception processing in a DSP environment is primarily associated with the transfer of data

Table 3.
DSP56000 instruction timing.

Normal instruction timing										
Instruction cycle	1	2	3	4	5	6	7	8	9	10
Fetch	n3	n4	n5	n6	n7	n8	n9	n10	n11	n12
Decode	n2	n3	n4	n5	n6	n7	n8	n9	n10	n11
Execute	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10
n = normal instruction word										
Fast interrupt instruction timing										
Instruction cycle	1	2	3	4	5	6	7	8	9	10
Fetch	n3	n4	iv1	iv2	n5	n6	n7	n8	n9	n10
Decode	n2	n3	n4	iv1	iv2	n5	n6	n7	n8	n9
Execute	n1	n2	n3	n4	iv1	iv2	n5	n6	n7	n8
n = normal instruction word iv = interrupt vector instruction word (no change of flow)										
Long interrupt instruction timing										
Instruction cycle	1	2	3	4	5	6	7	8	9	10
Fetch	n3	n4	iv1	iv2	i1	i2	i3	n5	n6	n7
Decode	n2	n3	n4	JSR	—	i1	RTI	—	n5	n6
Execute	n1	n2	n3	n4	JSR	—	i1	RTI	—	n5
n = normal instruction word iv = interrupt vector instruction word (change of flow) JSR = jump to interrupt service routine i = interrupt service routine instruction word RTI = return from interrupt										

between processor memory or registers and a peripheral device. When an interrupt occurs, a limited context switch must be performed with minimum overhead. Saving all of the machine state is too time consuming and usually not necessary. The DSP56000 provides a sophisticated interrupt structure which reduces the timing overhead associated with servicing interrupts. Each peripheral device and external interrupt pin may be programmed to one of three interrupt priority levels (IPLs) so that time-critical interrupts are always serviced first. When more than one interrupt is pending within the same IPL, a fixed priority table deter-

mines the secondary priority level within that IPL. Most on-chip peripherals have separate interrupt vectors for each interrupting condition so the cause of the interrupt will be known before the interrupt service routine is entered. There is no need to poll devices or test status bits to determine the interrupting condition. An interrupt vector may be error-free or not error-free. If it is error-free, no error conditions are associated with the interrupt request, and a data transfer can service the request immediately without checking error flags. If the interrupt vector is not error-free, error conditions are associated with the interrupt request, and the inter-

rupt service routine must first check the error flags. As can be seen from Table 4, 18 out of 32 possible interrupt vectors are used. The two external interrupt pins, IRQA and IRQB, may be programmed as level-sensitive or negative-edge triggered.

The timing of the interrupt controller is shown in Table 3. First, the interrupt controller synchronizes and prioritizes pending interrupts to determine the highest-priority, unmasked interrupt request. Second, the instruction fetch stream is temporarily redirected to fetch two interrupt instruction words (iv1 and iv2) at the interrupt vector addresses. The two interrupt instruction words at the interrupt vector addresses are fetched into the instruction pipeline without waiting for the current instruction to finish execution. The program counter is held constant since the interrupt controller provides the two interrupt vector addresses. Finally, normal instruction fetches (*n5*, *n6*, and so on, according to the program counter's contents) resume immediately after the two interrupt instruction words have been fetched. The usual interrupt vector change-of-flow is avoided and the extra cycles required to empty the instruction pipeline and refill it are eliminated. However, if instruction word *n4* is the first word of a two-word instruction, the execution of *n4* is aborted and *n4* is refetched in place of *n5*. Since most instructions are one word long, this occurs infrequently. The two interrupt instruction words iv1 and iv2 are decoded and executed. They may be two single-word instructions or one two-word instruction.

Fast interrupts. If execution of the two interrupt instruction words does not cause a change of flow, the interrupt routine is called a fast interrupt. In a fast interrupt, normal instruction execution continues without delay following the execution of the two interrupt instruction words iv1 and iv2. Fast interrupts do not save the machine state, so instructions that modify the machine state should not be used. Fast interrupts do not require a return from interrupt (RTI) instruction, since no context switch is performed. Although any non-change-of-flow instruction can be used, a special move peripheral (MOVEP) instruction has been provided to support fast interrupts with a memory-to-memory data transfer between memory-mapped peripheral devices and any memory space. Fast interrupts can process up to 1.7 million interrupts (or 10.25 million bytes) per second, yet they consume only 33 percent of the total execution time. This performance level rivals that of dedicated direct memory access hardware. Fast interrupts are a software alternative to DMA that provides several advantages. Unlike DMA, fast interrupts can service peripheral devices with the flexibility that only software can offer. Fast interrupts can use any addressing mode with linear, modulo, or reverse-carry address arithmetic. They can service on-chip as well as off-chip peripherals and memory using minimal hardware. Fast interrupts can also handle peripheral errors by vectoring to different interrupt service routines and can support termination conditions other than the traditional word count.

Table 4.
Interrupt sources.

Interrupt vector starting address	Error-free status	Interrupting condition
\$0000	—	Hardware RESET
\$0002	No	Stack error
\$0004	Yes	Trace
\$0006	Yes	Software interrupt (SWI)
\$0008	Yes	External interrupt IRQA
\$000A	Yes	External interrupt IRQB
\$000C	Yes	SSI receive data
\$000E	No	SSI receive data with exception
\$0010	Yes	SSI transmit data
\$0012	No	SSI transmit data with exception
\$0014	Yes	SCI receive data
\$0016	No	SCI receive data with exception
\$0018	Yes	SCI transmit data
\$001A	Yes	SCI idle line
\$001C	Yes	SCI timer
\$001E	—	Reserved for hardware development
\$0020	Yes	HOST receive data
\$0022	Yes	HOST transmit data
\$0024	Yes	HOST command (default)
\$0026	Yes	Available for HOST command (AHC)
\$0028	Yes	AHC
...	...	...
\$003E	Yes	AHC

Long interrupts. If either of the two interrupt instruction words is a jump to subroutine (JSR) instruction, the interrupt routine is called a long interrupt. Programming a JSR instruction at the interrupt vector addresses iv1 or iv2 causes a long interrupt routine. When the JSR instruction is decoded, the DSP56000 performs a context switch by saving the current program counter and status register on the stack and updating the interrupt mask in the status register. The program counter is loaded with the JSR destination address and the long interrupt routine (i1, i2, and so on) begins execution. If he desires it, the programmer can save more of the machine state by using software stacking operations. The long interrupt routine is terminated by the usual RTI instruction (located at i2 in this example). As shown in Table 3, this mechanism allows long interrupts to be vectored via the JSR destination address with a minimum of timing overhead. This change of flow is as fast as conventional vectored interrupts and is a natural extension of the fast interrupt mechanism.

Multiple-bus architecture

The internal multiple-bus architecture of the DSP56000 consists of four data buses and three address buses connecting the various resources on the chip. Support of the fast multiplier/accumulator requires two multiplier input operands to be transferred between memory and the data

Table 5.
DSP56000 instruction set.

Arithmetic instructions		Program control instructions	
ABS	Absolute value	Jcc	Jump conditionally
ADC	Add with carry	JMP	Jump
ADD	Add	JScC	Jump to subroutine conditionally
ADDL	Shift left then add	JSR	Jump to subroutine
ADDR	Shift right then add	NOP	No operation
ASL	Arithmetic shift left	REP	Repeat next instruction
ASR	Arithmetic shift right	RESET	Reset on-chip peripheral devices
CLR	Clear	RTI	Return from interrupt
CMP	Compare	RTS	Return from subroutine
CMPM	Compare magnitude	STOP	Stop processing
DIV	Divide iteration	SWI	Software interrupt
MAC	Multiply/accumulate	WAIT	Wait for interrupt
MACR	Multiply/accumulate and round	Loop instructions	
MPY	Multiply	DO	Start hardware loop
MPYR	Multiply and round	ENDDO	Exit from hardware loop
NEG	Negate	Move instructions	
NORM	Normalize iteration	LUA	Load updated address
RND	Round	MOVE	Move data
SBC	Subtract with carry	MOVEC	Move control register
SUB	Subtract	MOVEM	Move program memory
SUBL	Shift left then subtract	MOVEP	Move peripheral data
SUBR	Shift right then subtract		
Tcc	Transfer conditionally		
TFR	Transfer		
TST	Test		
Logical instructions			
AND	Logical AND		
ANDI	AND immediate control register		
EOR	Logical exclusive OR		
LSL	Logical shift left		
LSR	Logical shift right		
NOT	Complement		
OR	Logical inclusive OR		
ORI	OR immediate control register		
ROL	Rotate left		
ROR	Rotate right		
Bit manipulation instructions			
BCLR	Bit test and clear		
BSET	Bit test and set		
BCHG	Bit test and change		
BTST	Bit test on memory		
JCLR	Jump if bit clear		
JSET	Jump if bit set		
JSCLR	Jump to subroutine if bit clear		
JSSET	Jump to subroutine if bit set		

ALU upon each instruction cycle. Instruction fetches must also be done at the same rate. This translates to three data transfers (two data and one instruction) every instruction cycle. The bus structure supports general register-to-register, register-to-memory, and memory-to-register data movement and can transfer up to three 24-bit words at the same time. The resources connected to a bus define its primary data transport function (see Figure 2 again). One address and data bus is associated with each of the three memory spaces (x data, y data, and p program), whereas a fourth data bus, called the global data bus, is shared by all three memory spaces. The role of the global data bus is to physically extend the x , y , and p data buses so they can be connected to remote chip resources while keeping these buses as short as possible. Keeping the buses local enhances the speed of the device.

Internal bus switch. Communication between the buses in the multiple-bus structure takes place in an internal bus switch. The internal bus switch is similar to a switch matrix and can connect any two internal buses without adding any pipeline delays. This flexibility allows a general data move capability for easier programming. Since the internal data bus switch can access each memory space, the bit manipulation unit is physically located in this block.

Table 6.
Parallel move operations.

Parallel move operation	Example of assembler syntax
No parallel move	ADD X0,A
Register → register	ADD X0,A Y1,R0
Address register update	ADD X0,A (R1) – N1
Immediate short data → reg.	ADD X0,A #183,R4
Immediate data → register	ADD X0,A #\$F97B4A,B
Immediate data → register plus register → register	ADD X0,A #\$123456,X1 B,Y1
Absolute short address ↔ reg.	ADD X0,A Y:\$3A,R4
Absolute address ↔ register	ADD X0,A A,X:\$FFE3
Absolute address ↔ register plus register → register	ADD X0,A A,X1 Y0,Y:\$3F80
x memory ↔ register	ADD X0,A X0,X:(R5) +
x memory ↔ register plus register → register	ADD X0,A X:(R0) – ,X0 A,Y0
y memory ↔ register	ADD X0,A Y:(R0) + ,R7
y memory ↔ register plus register → register	ADD X0,A B,X0 A,Y:(R0) + N0
x memory ↔ register plus y memory ↔ register	ADD X0,A X1,X:(R3) + Y:(R6) – ,B
L long memory ↔ register	ADD X0,A AB,L:(R2) +

Here, “source,destination” assembler format is used and “ADD X0,A” is a sample opcode and operand.

DSP56000 instruction set

The DSP56000 has an easy to learn, microprocessor-style instruction set that is efficient for many different algorithms. Its instruction set has some characteristics of a reduced-instruction-set computer because of its register-based (load/store) orientation and because most of the instructions are executed in a single cycle. However, enough complex-instruction-set computer functionality is built into each instruction that the DSP56000 achieves the high performance needed for small code loops typically consisting of only two to five instructions. Lack of data pipelining contributes to the ease of programming and eliminates any architectural bias against an algorithm. During development of the DSP56000, its designers used a set of 15 common DSP benchmarks to test the speed and coding efficiency of the instruction set whenever they made changes to it. These benchmarks included digital filters and fast Fourier transforms for real and complex data. (The benchmarks are listed in an appendix to the DSP56000 manual.¹²) The designers used other benchmarks to measure performance for two-dimensional problems such as matrix manipulation and image processing. Many algorithms achieve high performance on the general-purpose DSP56000. A list of DSP56000 opcodes is shown in Table 5.

In addition to the usual set of microprocessor opcodes, the DSP56000 instruction set provides a powerful set of multiply and multiply/accumulate instructions with options for rounding and positive or negative product accumula-

tion. Other opcode additions include absolute value, shift left or right then add or subtract, compare magnitude, normalize, round, and transfer conditional instructions. When used after a compare or compare magnitude instruction, the transfer conditional (Tcc) instruction can perform maximum value, minimum value, maximum absolute value, minimum absolute value, and other functions.

Parallel move operations. Most arithmetic and logical instructions can specify up to two data transfers in the same instruction. These data transfers are called parallel move operations and are executed in one instruction cycle in parallel with the instruction opcode. This allows two or three conventional instructions to be combined into one parallel instruction, with corresponding gains in speed and coding efficiency. Parallel move operations allow the register-based execution units to be kept busy by concurrently preloading new input operands and storing previous results. They also provide concurrent communications between execution units. Used with the local registers in each execution unit, the parallel move operations provide “software pipelining” controlled by the user. Thus, the user can adapt the DSP56000’s parallel architecture to his application. The parallel move operations are shown in Table 6. The assembler syntax samples illustrate the different parallel move operations that can be specified with the same ADD X0,A instruction. Note that the same register may be a source operand more than once in the same instruction.

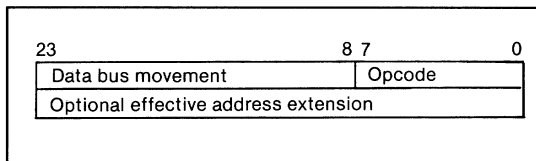


Figure 8. DSP56000 instruction encoding format.

However, no register may be specified as a destination operand more than once in the same instruction.

The general instruction encoding format is shown in Figure 8. All instructions are one or two 24-bit words in length. The first word generally contains an 8-bit opcode field and a 16-bit data bus movement field. The opcode field includes the instruction opcode with its source and destination register operands. The data bus movement field specifies source and destination operands for parallel move operations over the x data and y data bus. For address register indirect addressing modes, the data bus movement field specifies up to two address registers and associated addressing modes. As shown earlier in Table 1, the DSP56000 provides a set of 14 addressing modes to minimize address generation overhead. Both register direct and address register indirect (pointer) addressing modes are available as one-word instructions. Absolute short, I/O short, and short jump addresses and immediate short data addressing

modes are also provided, being encoded into the data bus movement field for fast, one-word instructions. Absolute address and immediate data addressing modes are encoded in the second optional effective address extension word to provide full 16-bit addresses and 24-bit data.

On-chip resources

The DSP56000 provides a large set of on-chip memory and I/O peripheral resources to support the core processor. The on-chip memories include two 256×24 -bit RAMs, two 256×24 -bit data ROMs and one 2048×24 -bit program ROM. Data or program code can be moved from any memory space to another, whether on-chip or off-chip. Microcomputer-style I/O capability is provided by three on-chip peripherals—the parallel host MPU/DMA interface (host), the asynchronous serial communications interface (SCI), and the synchronous serial interface (SSI)—on 24 programmable, general-purpose port pins. Noncore resources are supported by standard move and bit manipulation instructions. This avoids irregular I/O instructions and will allow Motorola to easily change noncore resources in future DSP56000 family members without affecting the instruction set.

Memory expansion port. The DSP56000 chip pinout is shown in Figure 9. Memory expansion off-chip is provided by a memory expansion interface called Port A. External

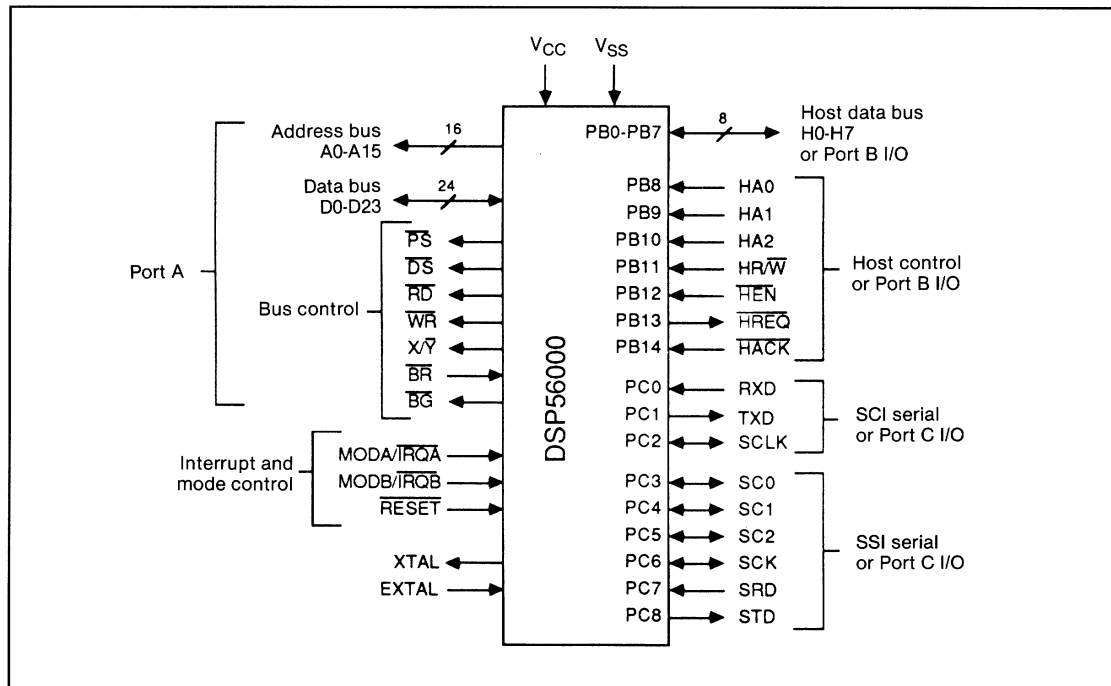


Figure 9. DSP56000 pinout.

peripherals and slave processors (microprocessors or another DSP IC) may also be accessed through this port. Separate 16-bit address and 24-bit data buses are used to multiplex the three internal address buses and four internal data buses off-chip, respectively. Off-chip (external) memory spaces are a logical extension of on-chip (internal) memory spaces. The bus controller determines, from the value of the address, whether a memory access is external or internal and schedules the bus activity to optimally use the memory expansion bus. If only one external memory access is requested per instruction cycle, the request is granted immediately and no extra clock cycles are required. If two or three external memory accesses are requested in a given instruction cycle, a minimum of one or two extra instruction cycles, respectively, are required to complete the instruction. The seven bus control signals create a synchronous bus that can perform 10.25 million accesses per second. Full-speed operation with no wait states requires a memory access time of 55 ns. The bus controller can be programmed to insert 0 to 15 wait states for four types of external memory access. Each wait state is one clock cycle (or 48.75 ns) long with a 20.5-MHz processor clock. This allows fast and slow external devices to be mixed on the memory expansion bus. An external device can also gain control of the memory expansion bus by asserting the bus request (BR) control input. In response to a BR, the DSP56000 releases control of the bus and asserts the bus grant (BG) control output at the end of the current bus access.

General-purpose I/O pins. On the DSP56000, 24 pins can be programmed to be general-purpose I/O pins; these pins are called Port B and Port C (see Figure 9 again). When they are programmed in this way, they can be used as I/O flags for synchronization and control purposes. Individual pin control, data direction, and data transfer functions are provided by six internal memory-mapped registers. The user can change or test these registers using standard bit manipulation and jump-on-bit-condition instructions. Each port pin can also be programmed to serve as a dedicated pin for one of the three on-chip peripherals.

Host processor interface. Although the DSP56000 can operate as a stand-alone processor, in many systems it is accompanied by a host microprocessor. The host processor functions as the system controller and user interface, while the DSP56000 handles the real-time digital signal processing. To support its use in such multiple-processor systems, the DSP56000 includes an on-chip host processor interface. The host interface is a byte-wide, full-duplex parallel port that can be connected directly to the data bus of a host processor. The host processor may be any of a number of industry-standard microcomputers or microprocessors, another DSP IC, or DMA hardware. The host interface appears as a memory-mapped peripheral occupying eight bytes in the host processor address space. It gives the host processor an eight-bit bidirectional data bus and seven control lines to use to control data transfers. It provides 14 internal registers to support double-buffered data transfers between

the host processor and the core processor by means of asynchronous polling or interrupts. In DMA mode, the host interface allows an external DMA controller to perform DMA transfers between an external memory or device and the host interface's registers using the host request (HREQ) and host acknowledge (HACK) handshake lines. From the DSP56000's perspective, the DMA data can be transferred between the host interface registers and any DSP56000 register or memory location (internal or external) by means of the standard instruction set. The fast interrupt mechanism can be used to minimize the data transfer overhead.

*A major accomplishment
of the DSP56000 design was
low power consumption in both
active and standby modes.*

The host interface provides DMA initialization commands that are used to set up the host interface DMA channel. It also has a special host command feature that allows the host processor to issue a vectored interrupt request to a DSP56000 program. The host processor may select one of 32 DSP56000 host command interrupts by writing a vector address register. Host commands are useful for debugging, performing on-line diagnostics, implementing control protocols, and setting up DMA.

Serial communications interface. The serial communications interface, or SCI, provides full-duplex serial communications with a variety of serial devices—including microprocessors, other DSP ICs, terminals, and modems—either directly or via RS-232 lines. The SCI supports industry-standard asynchronous character modes and allows parity and multidrop options. The multidrop option includes wake-up-on-idle-line and wake-up-on-address-bit capabilities. A synchronous shift register mode allows I/O expansion and high-speed, synchronous data transmission. The SCI consists of separate transmit and receive sections and a programmable baud-rate generator. Seven internal registers provide double-buffered data transfer and control functions. Three I/O pins are used for transmit data, receive data, and baud-rate clock functions. The baud rate can be internally or externally generated for asynchronous rates of up to 320K bits per second and synchronous rates of up to 2.5M bits per second. The internal baud-rate generator can function as a periodic interrupt timer when it is not being used by the transmit and receive sections.

Synchronous serial interface. The synchronous serial interface, or SSI, provides a full-duplex, double-buffered

Table 7.
DSP56000 benchmark summary.

Benchmark	Performance
<i>N</i> -tap real FIR filter with data shift	97.5 ns per tap
<i>N</i> -tap real, LMS adaptive FIR filter with data shift	292.5 ns per tap
<i>N</i> real, cascaded IIR biquad filters (four coefficients)	390 ns per filter
<i>N</i> -tap complex FIR filter with data shift	390 ns per tap
256-point complex FFT (radix 2, looped)	0.706 ms
1024-point complex FFT (radix 2, looped)	4.994 ms
Two-dimensional convolution (3 × 3 coefficient mask)	975 ns per output
Finding of maximum absolute value and index in array	195 ns per point

serial port that allows the DSP56000 to communicate with a variety of serial devices. These include one or more industry-standard codec A/D and D/A converters, other DSP ICs, microprocessors, and serial peripheral devices. The SSI consists of separate transmit and receive sections and an SSI clock generator. The clock generator defines the serial bit rate, the serial word size, and the number of serial words per frame. The data in each serial frame are controlled by software, allowing any user protocol to be implemented. Several clock and frame sync timing options provide flexible, synchronous, serial communications at rates of up to 5M bits per second. The SSI uses three to six I/O pins, depending on the operating mode; it has eight internal registers.

Three SSI operating modes support the requirements of different serial devices. The normal operating mode is used for periodic devices that transmit or receive one data word with each serial frame. One time slot is defined for data transmission at the start of each serial frame. A codec A/D or D/A converter is an example of such a periodic device. The on-demand operating mode is for nonperiodic communications such as those from one DSP56000 to another. No time slots are defined for transmission and data are transmitted as soon as they are available. The network operating mode defines from 2 to 32 time slots per serial frame which can be used for creating a network of communicating serial devices. Each device can transmit or receive during one or more assigned time slots. The time slot assignments for each serial device are determined by the user's software. In network mode, multiple DSP56000s can communicate without needing glue chips to do so. In network mode, the DSP56000 can also be interfaced directly to the time-division-multiplexed, serial I/O channels used in telecommunications applications.

DSP56000 implementation

The DSP56000 is implemented in a 1.5- μ m, double-level metal, N-well, high-density CMOS process. It is an 88-pin integrated circuit available in pin-grid-array or surface-mount packaging. Its maximum clock rate is currently 20.5 MHz and is provided by an on-chip crystal oscillator or an external clock. One of the major design accomplishments was low power consumption in both active and standby modes. Because of the DSP56000's full CMOS design, its power consumption scales down linearly with clock frequency, allowing the user to reduce processor speed to save power. Since the DSP56000 contains substantial on-chip memory, the user can locate program and data sections with high dynamic frequency in on-chip memory to avoid driving the memory expansion port. The DSP56000's bus controller does not toggle the external address and data bus pins unless an external access is made. The DSP56000 deselects all external devices to their low-power standby modes when they are not being accessed. It also saves power by using STOP and WAIT low-power standby modes. Execution of a STOP or WAIT instruction puts the DSP56000 into a

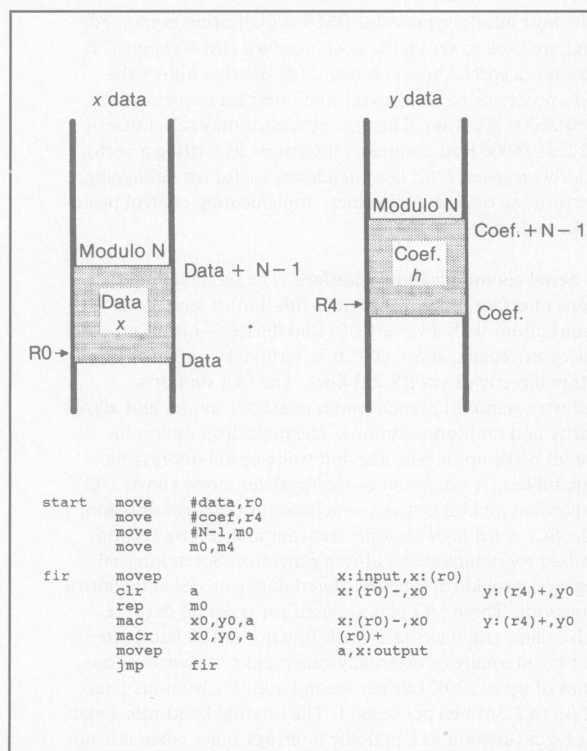


Figure 10. DSP56000 finite impulse response (FIR) filter—data memory organization (top) and program (bottom).

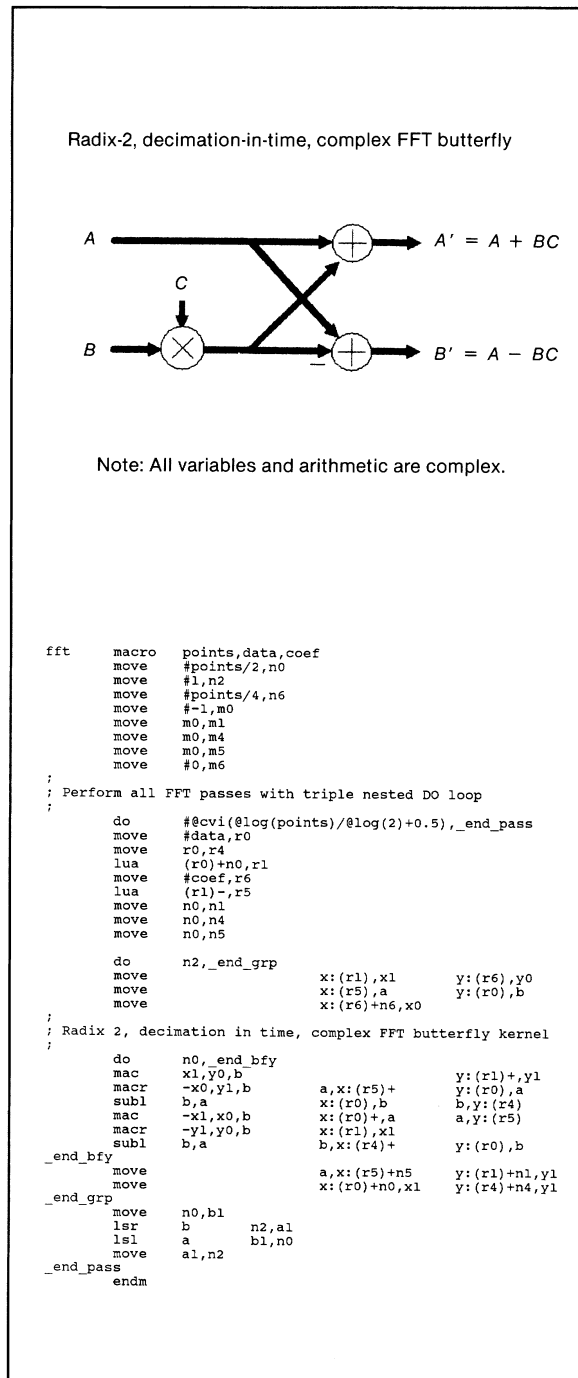


Figure 12. DSP56000 fast Fourier transform (FFT)—diagram of the “butterfly” kernel (top) and program (bottom).

organization, and the DSP56000 assembler source program needed to implement N cascaded biquad digital filters. The memory is organized to store the filter storage $w(n-1)$ and $w(n-2)$ in x data memory and the filter coefficients $a1$, $a2$, $b1$, and $b2$ in y data memory. In this example, the time-shift function is performed by parallel move operations instead of by modulo addressing. The two instructions at START set up data pointer R0 and coefficient pointer R4 for linear arithmetic. The program loop at IIR loads a digital “input,” performs N cascaded biquad filters, and stores the filter “output.” The actual biquad filter program consists of only four multiply/accumulate instructions located within a hardware DO loop. The DO instruction initiates the hardware DO loop with the instruction following the DO instruction and ends it with the instruction before the label END_IIR. The digital output is rounded (RND) to single precision after the hardware DO loop has executed. The four-coefficient, biquad filter kernel executes in only four instructions, i.e., in 390 ns per biquad filter. Similar five-coefficient biquad filter kernels execute in five instructions, or 487.5 ns. For typical voiceband processing with an 8-kHz sampling rate, the DSP56000 can implement over 300 biquad digital filters in real time.

Fast Fourier transform. The fast Fourier transform, or FFT, is widely used in signal processing to perform spectral analysis of time-domain data. Most FFTs operate on complex data having real and imaginary components and hence must use complex arithmetic. The DSP56000 is very efficient at complex arithmetic. Complex arithmetic generally requires at least four multiplier input registers to store two complex inputs, two accumulators to store the real and imaginary parts of the complex result, and address pointers that can access complex data pairs. A large number of FFT algorithms exist, but here we will demonstrate only the simple, radix-2, decimation-in-time, in-place complex FFT. The heart of the radix-2 FFT is a complex “butterfly” kernel, which is shown in Figure 12. Note that all data paths and calculations are done in complex arithmetic. This butterfly kernel is executed many times to calculate a complete FFT. The complete DSP56000 macro program for a radix-2, decimation-in-time, in-place complex FFT is also shown in Figure 12. This macro may be called to perform an FFT of any size from 2 to 32,768 points. The example uses three nested hardware DO loops to compact the complete FFT program into only 40 words of program memory. The butterfly kernel routine inside the innermost DO loop requires only six instructions, or 585 ns, to perform one butterfly calculation on two complex data points A and B. The single complex multiplication ($B \times C$) is implemented as four real multiplies. The inner, middle, and outer DO loops build on the kernel routine to process one butterfly group, one butterfly pass, and the complete FFT transform, respectively. This radix-2 FFT using looped code executes a 256-point complex FFT in 814 μ s and a 1024-point complex FFT in 6.41 ms. Other variations achieve even higher performance. For instance, modified versions of the code in the example execute a 1024-point complex FFT in less than 5 ms.

Since the FFT data are stored in 24-bit words, little or no scaling is required with typical 12-to-16-bit input data. The DSP56000 scaling modes can be used to scale the data of each FFT pass one bit left or right with minimum overhead. This can add 20 more bits of dynamic range to a 1024-point FFT while still maintaining full FFT speed.

DSP56000 design tools

The DSP56000 is supported by a software development package consisting of a full-featured macro cross assembler, ASM56000, a software simulator, SIM56000, and associated documentation. The package is available for the IBM PC/MS-DOS, VAX/VMS, and Unix environments. ASM56000 offers the usual complement of features found in modern assemblers, such as extensive error reporting, conditional assembly, file inclusion, nested macros with macro library support, local labels, sections, and external

an instruction or clock cycle basis, single stepping or tracing with multilevel conditional or unconditional breakpoints. It provides instruction and clock cycle counts and generates histograms of those counts to support the analysis of program execution time. It displays the enabled set of registers and memory, highlights the write operations, and provides options for displaying upon a read, write, or other access. Of particular note are SIM56000's input and output commands, which assign the simulator's I/O, with a device pin, memory location, or on-chip peripheral, to a terminal or a disk file. Its I/O data format can be untimed or "time stamped," and nested repeat directives can be used to generate arbitrary input data sequences. Besides being an accurate simulator of the DSP56000, SIM56000 is very easy to use. Help files describe each simulator command and a help line on the display indicates the command line syntax as commands are entered. A symbolic calculator assists the programmer with hexadecimal, decimal, and binary calcula-

Digital signal processors are transforming analog circuits into software in the 1980's in the same way microprocessors transformed digital control logic into software in the 1970's.

definition/reference directives. It also provides arbitrary expression evaluation with Boolean operators and built-in functions for data type conversion, string comparison, and common transcendental functions such as sine, cosine, logarithm, exponent, and square root. These functions allow constants and lookup tables for DSP algorithms to be parameterized by macro arguments and dynamically generated at assembly time. ASM56000 also provides assembler output listings with instruction cycle counts, cross reference tables, and memory utilization reports. The memory utilization reports provide a global view of the allocated and free space in the DSP56000 memory map, which can become fragmented by inefficient placement of modulo and reverse-carry (bit-reversed) storage regions. A report identifies the free space available for additional storage in each memory map.

The SIM56000 software simulator emulates, on a clock cycle basis, the functions of the DSP56000, including on-chip peripheral activity and external I/O pin activity. SIM56000 enables the software developer to execute DSP56000 object code generated from ASM56000 or from SIM56000's own single-line assembler. Throughout the DSP56000 chip design, SIM56000 was used to compare against the chip data base simulations by running the same DSP56000 programs through both simulations. Both the chip design and the software products were debugged rapidly through this comparison testing. SIM56000 performs, on

tions. The programmer may define, store, and execute simulator command macros. He can also do program patching by using SIM56000's single-line assembler/disassembler. He can log all simulator activity to disk files for future analysis, and he can save the simulator state so he can resume the simulation later. SIM56000 uses ASCII disk files for all of its I/O (but not for saving the simulator state), enhancing access by other programs and utilities.

Motorola is developing a DSP56000 evaluation module, or EVM, to serve as a low-cost design tool. The EVM consists of an evaluation board, or EVB, an interface card for the IBM PC bus, and a user interface software package, EVM56000. The EVB contains a full-speed 20.5-MHz DSP56000, 8192 $\times$ 24 bits of external program/data RAM, an expansion connector for adding prototype hardware, and a monitor ROM. It is controlled by the EVM56000 software running in the PC environment, which presents a user interface similar to that of the SIM56000 software simulator. EVM56000 retains many of SIM56000's features and adds commands to support up to eight EVBs on the same host with foreground/background access to the user interface. The EVB can be used as a hardware accelerator for speeding up simulations or as a prototype board for developing target applications.

Additional software support includes software source libraries, application notes, and a high-level-language compiler. Moreover, Motorola supports the user with training seminars, video classes, and an electronic bulletin board.

The DSP56000 is a state-of-the-art, high-performance digital signal processor. The similarities between it and other Motorola microprocessors make it easy to learn and easy to program, yet the differences open up new user applications. The future for programmable digital signal processors looks extremely bright. Digital signal processors are transforming analog circuits into software in the

1980's in the same way microprocessors transformed digital control logic into software in the 1970's. By the 1990's, most communications, computer, and control equipment will use digital signal processing ICs. DSP ICs will evolve as microprocessors have. Work continues on developing an entire family of faster and more capable DSP devices to serve the needs of DSP users. ㊦

References

1. "Motorola's Sizzling New Signal Processor," *Electronics*, March 10, 1986, pp. 30-33.
2. J. Bates, "Motorola's DSP56000: A Fourth Generation Digital Signal Processor," *Professional Program Session Record, 1986 Mini/Micro Northeast Computer Conf.*, No. 18, Paper 5.
3. A.V. Oppenheim and R.W. Schaffer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., 1975.
4. R.C. Jaeger, "Tutorial: Analog Data Acquisition Technology, Parts 1-4," *IEEE Micro*, May 1982, pp. 20-37, Aug. 1982, pp. 46-56, Nov. 1982, pp. 20-35, and Feb. 1983, pp. 52-61.
5. L.R. Rabiner and R.W. Schaffer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, N.J., 1978.
6. S. Abiko et al., "Architecture and Applications of a 100-ns CMOS VLSI Digital Signal Processor," *Proc. IEEE ICASSP 86 Conf.*, pp. 393-396.
7. T. Nishitani et al., "Advanced Single-Chip Signal Processor," *Proc. IEEE ICASSP 86 Conf.*, pp. 409-412.
8. J.R. Boddie et al., "The Architecture, Instruction Set and Development Support for the WE DSP32 Digital Signal Processor," *Proc. IEEE ICASSP 86 Conf.*, pp. 421-424.
9. K.A. El-Ayat and R.K. Agarwal, "The Intel 80386—Architecture and Implementation," *IEEE Micro*, Dec. 1985, pp. 4-22.
10. D. Phillips, "The Z80000 Microprocessor," *IEEE Micro*, Dec. 1985, pp. 23-36.
11. K.L. Kloker, "Architectural Features of the Motorola DSP56000 Digital Signal Processor," *VLSI Signal Processing, II* (Proc. 1986 IEEE Workshop on VLSI Signal Processing), IEEE Press, New York, 1986.
12. *DSP56000 Digital Signal Processor User's Manual*, Pub. no. DSP56000UM/AD, Motorola, Inc., Austin, Texas, 1986.



Kevin L. Kloker, a principal staff engineer in Motorola's Corporate Research Laboratories, was the principal architect of the Motorola DSP56000 programmable digital signal processor. His areas of interest include VLSI processor architectures, computer arithmetic, software languages, and digital signal processing. He received a BSEE from Bradley University and a MSEE from the Illinois Institute of Technology in 1976 and 1980, respectively, and joined Motorola in 1976. He is a member of the Motorola Science Advisory Board Associates, the IEEE, Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.

Questions about this article can be directed to Kloker at Motorola, Inc., 1301 East Algonquin Road, Schaumburg, IL 60196.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 177 Medium 178 Low 179