SECTION VIII

DSP HARDWARE

DSP HARDWARE

- RISC VS. CISC VS. DSP ARCHITECTURES
- RISC AND DSP APPLICATIONS
- DSP PROCESSOR REQUIREMENTS
  - Fast Arithmetic
  - Zero Overhead Looping
  - Extended Dynamic Range
  - Dual Operand Fetch
  - Circular Buffers
- ADSP-2101 MICROCOMPUTER GENERAL DESCRIPTION
- ADSP-2101 ARCHITECTURE OVERVIEW
  - Arithmetic Logic Unit (ALU)
  - Multiplier/Accumulator (MAC)
  - Shifter
  - Data Address Generators (DAGs)
  - Program Sequencer
  - Serial Ports
  - System Interface
- DEVELOPMENT SYSTEM

SECTION VIII

DSP HARDWARE

RISC VERSUS CISC ARCHITECTURES

As central processor (CPU) architectures developed, their instruction sets became more sophisticated. The complex-instruction-set (CISC) computer includes instructions for basic processor operations, plus single instructions that are highly sophisticated—for example, to evaluate a high-order polynomial. But CISC has a price: many of the instructions execute via microcode in the CPU and require numerous clock cycles—plus silicon real estate for code storage.

In contrast, the reduced-instruction-set computer (RISC) recognizes that, in many applications, basic instructions such as LOAD and STORE—with simple addressing schemes—are used much more frequently than the advanced instructions, and should not incur an execution penalty. These simpler instructions are hardwired in the CPU logic to execute in a single clock cycle, reducing execution time and CPU complexity.

RISC AND DSP APPLICATIONS

Although the RISC approach offers many advantages in general purpose computing, it is not well suited to DSP. For example, most RISCs do not support single-instruction multiplication, a very common and repetitive operation in DSP. The DSP is optimized to accomplish its task fast enough to maintain real-time operation in the context of the application, which requires single-cycle arithmetic operations and accumulations.

DSP algorithms have unique requirements not found in general purpose computing such as circular buffering, pointer updating and fast looping with zero overhead, bit reversing, barrel shifting, scaling, and data-dependent execution branching. Each of these should execute within the DSP instruction, and not as a separate time-consuming instruction cycle. The computational unit within the DSP must run efficiently, with data arriving from at least two separate data memories with no time penalty for data access. CISCs and RISCs support virtually none of these needs.

Similarly, software programming for RISCs and CISCs differs from that used in DSP. RISCs and CISCs are programmed in high-level languages to minimize software development and hide the assembly language from the programmer. For real-time DSP applications, however, code optimization (primarily of execution time, but also of memory usage) requires that the software engineer use assembly language to get satisfactory performance. If the initial results are not satisfactory after simulation, critical sections of the program are examined and recoded if necessary to reduce overall execution time.

ARCHITECTURES

- Complex Instruction Set Computer (CISC)
- Reduced Instruction Set Computer (RISC)
- DSP Processor

Figure 8.1

DSP PROCESSOR REQUIREMENTS

Many DSP algorithms (such as digital filters and FFTs) rely heavily on the efficient performance of the straight-forward sum-of-products mathematics. Examining the equation shown in Figure 8.2 reveals that there are some fundamental properties implicit in the mathematics.

THE DSP KERNEL EQUATION

$$\Sigma = h(0)x(0) + h(1)x(1) + h(2)x(2) + \ldots + h(N-1)x(N-1)$$

- The Equation is Based on Multiply-Accumulates (MACs)
- There are N MACs - One for Each product
- Each Product is Formed from 2 Values, One Value is a Signal, x(i), the Other a Stored Coefficient, h(i)

Figure 8.2

The three algorithmic properties place specific requirements on processor architectures aimed at digital signal processing. The fundamental properties of the DSP kernel function, $\Sigma h(i)x(i)$, and effects of finite word size arithmetic (i.e. quantization errors) combine to produce five DSP architectural requirements shown in Figure 8.3.

DSP ARCHITECTURE REQUIREMENTS

- Fast Arithmetic
- Zero Overhead Looping
- Extended Dynamic Range
- Dual Operand Fetch
- Circular Buffers

Figure 8.3

FAST ARITHMETIC

Fast arithmetic is the simplest of these requirements to understand. Since real-time DSP applications are driven by performance, the multiply-accumulate or MAC time is a central requirement; faster MACs mean potentially higher bandwidth. It is critical to remember that MAC time alone does not define DSP performance. This often forgotten fact leads to an inadequate measure of processor performance by simply examining its MIPS (million instructions per second) rating. Since most DSP and DSP-like architectures feature MACs that can execute an instruction every cycle, most processors are given a MIPS rating equal to its MAC throughput. This does not necessarily account for the other factors that can degrade a processors overall performance in real-world applications. The other four criteria can wipe out MAC gains if they are not satisfied.

In addition to the requirement for fast arithmetic, a DSP should be able to support other general purpose math functions and should therefore have an appropriate arithmetic logic unit (ALU) and a shifter function.

ZERO OVERHEAD LOOPING

Zero overhead looping is required by the repetitive nature of the kernel equation. The multiply-accumulate function and the data fetches needed are repeated N times every time the kernel function is calculated. Traditional microprocessors implement loops that have one instruction execution time or more of overhead associated with them. DSP architectures provide hardware support that eliminates the need for looping instructions within the loop body. For true DSP architectures, the difference of zero overhead body looping and programmed looping can easily exceed 20% in available bandwidth.

EXTENDED DYNAMIC RANGE

Extended dynamic range is a requirement of finite word size arithmetic. The basic convolution requires repeated addition. Since each product can be a full range value, one accumulation of two products can cause overflow beyond the word size of the processor. Traditional microprocessors and microcomputers address overflow by providing an overflow flag. The program then has the added burden of testing the flag and conditionally adjusting the results. Architectures designed specifically for DSP reduce this problem by providing extended precision in the accumulator function of the MAC. Simply adding 8 bits extends performance by about 48dB. A further refinement of extended precision accumulation allows signed overflow and underflow so that the intermediate values of the MAC can track the real-world values that are applied with minimal chance of loss of accuracy.

DUAL OPERAND FETCH

Regardless of the nature of a processor, performance limitations are generally based on bus bandwidth. In the case of general purpose processors, code is dominated by single memory fetch instructions usually addressed as base plus offset value. This leads architects to embed fixed data into the instruction set so that this class of memory access is fast and memory efficient. DSP on the other hand is dominated by instructions

requiring two independent memory fetches. This is driven by the basic form of the convolution $\Sigma h(i)x(i)$. The goal of fast dual operand fetches is to keep the MAC fully loaded. We saw in the discussion on MACs that the performance of a DSP is first limited by MAC time. Assuming an adequate MAC cycle time, two data values need to be supplied at the same rate; reductions in operand fetch bandwidth will result in corresponding reductions in MAC bandwidth. Ideally, the operand fetches occur simultaneously with the MAC instruction so that the combination of the MAC and memory addressing occurs in one cycle.

Dual operand fetch is implemented in DSPs by providing separate buses for program memory data and data memory data. In addition, separate program memory address and data memory address buses are also provided. The MAC can therefore receive inputs from each data bus simultaneously. This architecture is often referred to as the Harvard Architecture.

CIRCULAR BUFFERS

If we examine the kernel equation more carefully, the advantages of circular buffering in DSP applications become apparent. A Finite Impulse Response (FIR) filter is used to demonstrate the point. First, coefficients or tap values for FIR filters are periodic in nature. Second, FIR filters use the newest real-world signal value and discard the oldest value.

In the series of FIR filter equations, the N coefficient locations are always accessed sequentially from h(0) to h(N-1). The associated data points circulate through the memory; new samples are added replacing the oldest data each time a filter output is computed. A fixed boundary RAM can be used to achieve this circulating buffer effect. The oldest data sample is replaced by the newest after each convolution. A "time history" of the N most recent samples is kept in RAM.

This delay line can be implemented in fixed boundary RAM in a DSP chip if new data values are written into memory, overwriting the oldest value. To facilitate memory addressing, old data values are read from memory starting with the value one location after the value that was just written. In a 4-tap FIR filter, for example, x(4) is written into memory location 0, and data values are then read from locations 1,2,3,and 0. This example can be expanded to accommodate any number of taps. By addressing data memory locations in this manner, the address generator need only supply sequential addresses regardless of whether the operation is a memory read or write. This data memory buffer is called *circular* because when the last location is reached, the memory pointer must be reset to the beginning of the buffer.

The coefficients are fetched simultaneously with the data. Due to the addressing scheme chosen, the oldest data sample is fetched first. Therefore, the last coefficient must be fetched first. The coefficients can be stored backwards in memory: h(N-1) is the first location, and h(0) is the last, with the address generator providing incremental addresses. Alternatively, coefficients can be stored in a normal manner with the accessing of coefficients starting at the end of the buffer, and the address generator being decremented.

This allows direct support of unit delay taps without software overhead. These data characteristics are DSP algorithm-specific and must be supported in hardware to achieve the best DSP performance. Implementing circular buffers in hardware allows buffer parameters (i.e. start, length, etc.) to be set up outside of the core instruction loop. This eliminates the need for extra instructions within the loop body. Lack of a hardware implementation for circular buffering can significantly impact MAC performance.

SUMMARY

Any processor can accomplish any software task, given enough time. However, DSPs are optimized for the unique computational requirements of real-time, real-world signal processing. Traditional computers are better suited for tasks that can be performed in non-real-time. In the following section, we will examine the architecture of a high-performance DSP Microcomputer, the ADSP-2101.

ADSP-2101 MICROCOMPUTER GENERAL DESCRIPTION

The ADSP-2101 is a single-chip microcomputer optimized for digital signal processing and other high-speed numeric processing applications. It combines the complete ADSP-2100 core architecture (three computational units, data address generators, and a program sequencer) with two serial ports, a programmable timer, extensive interrupt capabilities and on-board program and data memory RAM. The ADSP-2101 has 1K words of (16 bit) data memory RAM and 2K words of (24 bit) program RAM on chip.

The ADSP-2101's flexible architecture and comprehensive instruction set support a high degree of operational parallelism. In one cycle the ADSP-2101 can perform the functions shown in Figure 8.4.

ADSP-2101 SINGLE-INSTRUCTION CYCLE CAPABILITY

- Generate the Next Program Address
- Fetch the Next Instruction
- Perform One or Two Data Moves
- Update One or Two Data Address Pointers
- Perform a Computational Operation
- Receive and Transmit Data Via the Two Serial Ports
- Update Timer

Figure 8.4

ADSP-2101 ARCHITECTURE OVERVIEW

Figure 8.5 is an overall block diagram of the ADSP-2101. The processor contains three independent computational units: the ALU, the multiplier/accumulator (MAC) and the shifter. The computational units process 16-bit data directly and have provisions to support multiprecision computations. The ALU performs a standard set of arithmetic and logic operations; division primitives are also supported. The MAC performs single-cycle multiply, multiply/add, and multiply/subtract operations. The shifter performs logical and arithmetic shifts, normalization, denormalization, and derive exponent operations. The shifter can be used to efficiently implement numeric format control including multiword floating-point representations.

The internal result (R) bus directly connects the computational units so that the output of any unit may be the input of any unit on the next cycle.

A powerful program sequencer and two dedicated data address generators ensure efficient use of these computational units. The sequencer supports conditional jumps, subroutine calls and returns in a single cycle. With internal loop counters and loop stacks, the ADSP-2101 executes looped code with zero overhead; no explicit jump instructions are required to maintain the loop.

The data address generators (DAGs) handle address pointer updates. Each DAG keeps track of four address pointers. Whenever the pointer is used to access data (indirect addressing), it is post-modified by the value of a specified modify register. A length value may be associated with each pointer to implement automatic modulo addressing for circular buffers. With two independent DAGs, the processor can generate two addresses simultaneously for dual operand fetches. The circular buffering feature is also used by the serial ports for automatic data transfers.

ADSP-2101 BLOCK DIAGRAM

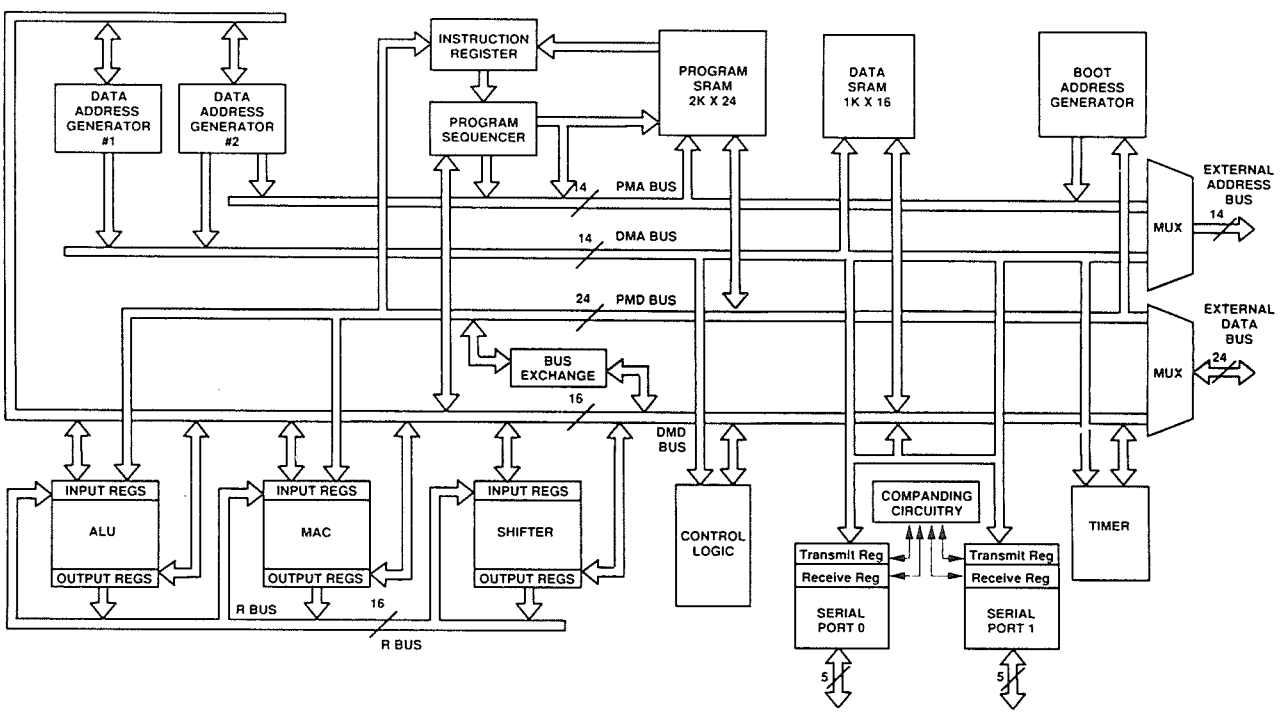

Figure 8.5

COMMON FEATURES OF ADSP-2100 FAMILY

- Arithmetic Logic Unit
- Multiplier/Accumulator (With 40-Bit Accumulator)
- Barrel Shifter
- Two Data Address Generators
- Program Sequencer

Figure 8.6

Efficient data transfer is achieved with the use of five internal buses. The two address buses (PMA and DMA) share a single external address bus, and the two data buses (PMD and DMD) share a single external data bus. The $\overline{BMS}$, $\overline{DMS}$, and $\overline{PMS}$ signals indicate which memory space the external buses are being used for.

ADSP-2101 INTERNAL BUSES

- Program Memory Address (PMA) Bus
- Program Memory Data (PMD) Bus
- Data Memory Address (DMA) Bus
- Data Memory Data (DMD) Bus
- Result (R) Bus

Figure 8.7

Program memory can store both instructions and data, permitting the ADSP-2101 to fetch two data operands in a single cycle, one from program memory and one from data memory as well as an instruction from program memory. Because the on-board program memory is so fast, the ADSP-2101 can fetch an operand from program memory and the next instruction in the same cycle.

The memory interface supports slow memories and memory-mapped peripherals with programmable wait state generation. External devices can gain control of buses with bus request/grant signals ($\overline{BR}$ and $\overline{BG}$). One execution mode allows the ADSP-2101 to continue running while the buses are granted to another master as long as an external memory operation is not required. The other execution mode requires the processor to halt while the buses are granted.

The two serial ports provide a complete serial interface with companding in hardware and a wide variety of framed and frameless data transmit and receive modes of operation. Each port can generate an internal programmable serial clock or accept an external serial clock.

Boot circuitry provides for loading on-chip program memory automatically from byte-wide external memory. After reset three wait states are automatically generated. This allows, for example, an 80ns ADSP-2101 to use an external 250ns EPROM as boot memory. Multiple programs can be selected and loaded from the EPROM with no additional hardware.

ARITHMETIC LOGIC UNIT (ALU)

The ALU is shown in Figure 8.8. The ALU provides a standard set of arithmetic and logic functions: add, subtract, negate, increment, decrement, absolute value, AND, OR, Exclusive OR and NOT. Two divide primitives are also provided. The ALU takes two 16-bit inputs, X and Y, and generates one 16-bit output, R. The carry-in feature enables multiword computations. Six arithmetic status bits are generated: AZ (zero), AN (negative), AV (overflow), AC (carry), AS (sign), and AQ (quotient).

The X input port can be fed by either the AX register set or any result register via the R-bus (AR, MR0, MR1, MR2, SR0, or SR1). The AX register set contains two registers, AX0 and AX1. The AX registers can be loaded from the DMD bus. The Y input port can be fed by either the AY register set or the ALU feedback (AF) register. The AY register set contains two registers, AY0 and AY1. The AY registers can be loaded from either the DMD bus or the PMD bus.

The register outputs are dual-ported so that one register can provide input to the ALU while either one simultaneously drives the DMD bus. The ALU output can be loaded into either the AR register or the AF register.

The AR register has a saturation capability; it can be automatically set to plus or minus the maximum value if an overflow or underflow occurs. The AR register can drive both the R bus and the DMD bus and can be loaded from the DMD bus.

The ALU contains a duplicate bank of registers shown in Figure 8.8 behind the primary registers. The secondary set contains all the registers described above (AX0, AX1, AY0, AY1, AF, AR). Only one set is accessible at a time. The two sets of registers allow fast context switching, such as for interrupt servicing.

ALU BLOCK DIAGRAM

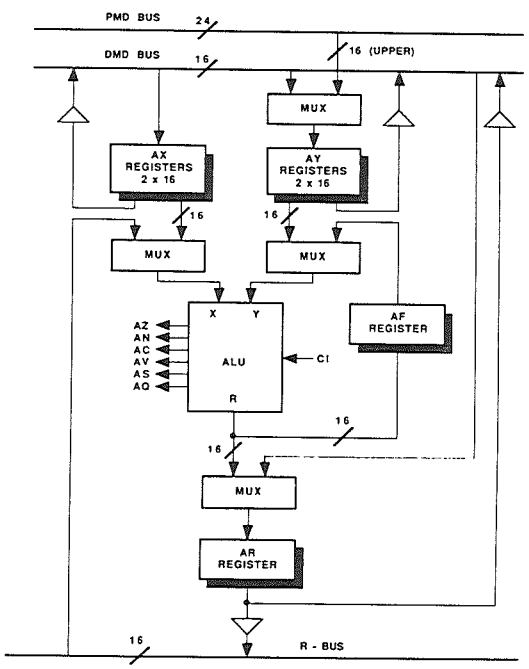

Figure 8.8

ALU FEATURES

- Feedback Paths
- Six Status Flags
- Saturation
- Implements Divide Primitives
- Provisions for Double Precision
- Complete Set of Background Registers

Figure 8.9

EXAMPLE ALU INSTRUCTIONS

- AR = AX0 + AY0
- AF = MR1 XOR AY1
- AR = AX0 + AF

Figure 8.10

MULTIPLIER/ACCUMULATOR (MAC)

The multiplier/accumulator (MAC) implements high-speed multiply, multiply/add, and multiply/subtract operations. A block diagram of the MAC section is shown in Figure 8.11.

The multiplier takes two 16-bit inputs, X and Y, and generates one 32-bit output, P. The 32-bit output is routed to a 40-bit accumulator which can add or subtract the P output from the value in MR. MR is a 40-bit register which is divided into three sections: MR0 (Bits 0-15), MR1 (Bits 16-31), and MR2 (Bits 21-29). The result of the accumulator is either loaded into the MR register or into the 16-bit MAC feedback (MF) register. The multiplier accepts the X and Y inputs in either signed or unsigned formats.

In default operation the result is shifted one bit to the left to remove the redundant sign bit for fractional justification; an optional mode on the ADSP-2101 inhibits this shift for integer operations. The accumulator generates one status bit, MV, which is set when the accumulator result overflows the 32-bit boundary. A saturate instruction is available to change the contents of the MR register to the maximum or minimum 32-bit value if MV is set. The accumulator also has the capability for rounding the 40-bit result at the boundary between bit 15 and bit 16.

The MAC and ALU registers are similar. The X input port can be fed by either the MX register set (MX0, MX1) or any result register via the R-bus (AR, MR0, MR1, MR2, SR0, or SR1). The MX register set is readable and loadable from the DMD bus and has dual-ported outputs.

The Y input port can be fed by either the MY register set (MY0, MY1) or the MF register. The MY register set it readable from the DMD bus and readable and loadable from both the DMD and the PMD bus. Its outputs are also dual-ported. The accumulator output can be loaded into either the MR register or the MF register. The MR register is connected to both the R-bus and

MAC BLOCK DIAGRAM

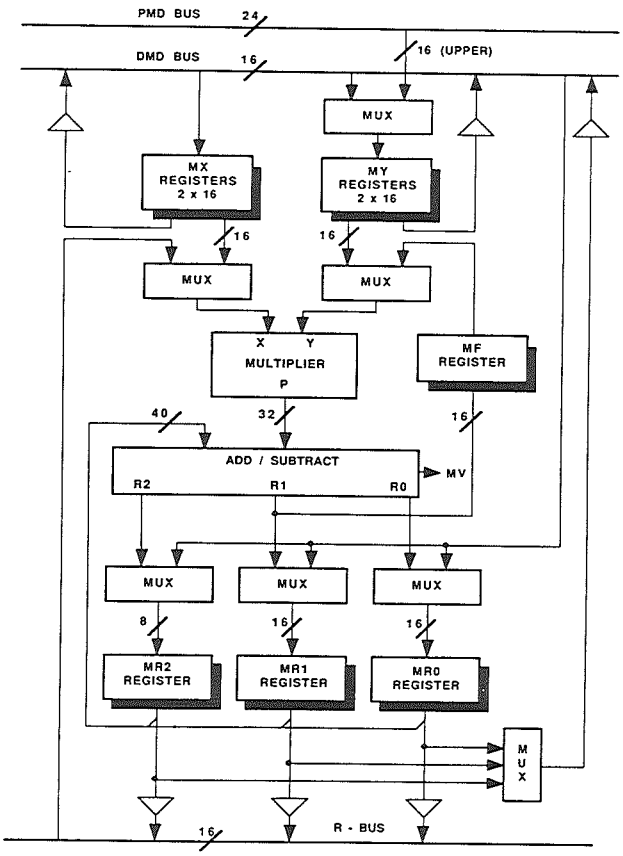

Figure 8.11

the DMD-bus. Like the ALU section, the MAC section contains two complete banks of registers (MX0, MX1, MY0, MY1, MF, MR0, MR1, MR2) to allow fast context switching.

MAC FEATURES

- Feedback Paths
- 40-Bit Accumulator
- Saturation
- Mixed Mode Input Operands
- Provisions for Multiprecision
- Complete Set of Background Registers

Figure 8.12

EXAMPLE MAC INSTRUCTIONS

- MR = MXO * MY0
- MR = 0
- MF = AR * MF
- MR = MX0 * MF
- MR = MR + MX1 * MY0

Figure 8.13

SHIFTER

The shifter gives the ADSP-2101 its unique capability to handle data formatting and numeric scaling. Figure 8.14 shows a block diagram of the shifter.

The shifter can be divided into the following components: the shifter array, the OR/PASS logic, the exponent detector and the exponent compare logic. These components give the shifter its six basic functions: arithmetic shift, logical shift, normalization, denormalization, derive exponent and derive block exponent.

The shifter array is a 16 x 32 barrel shifter. It accepts a 16-bit input and can place it anywhere in the 32-bit output field, from off-scale right to off-scale left. The shifter can perform arithmetic shifts (shifter output is sign-extended to the left) or logical shifts (shifter output is zero-filled to the left). The placement of the 16-bit input is determined by the control code (C) and the HI/LO reference signal.

SHIFTER BLOCK DIAGRAM

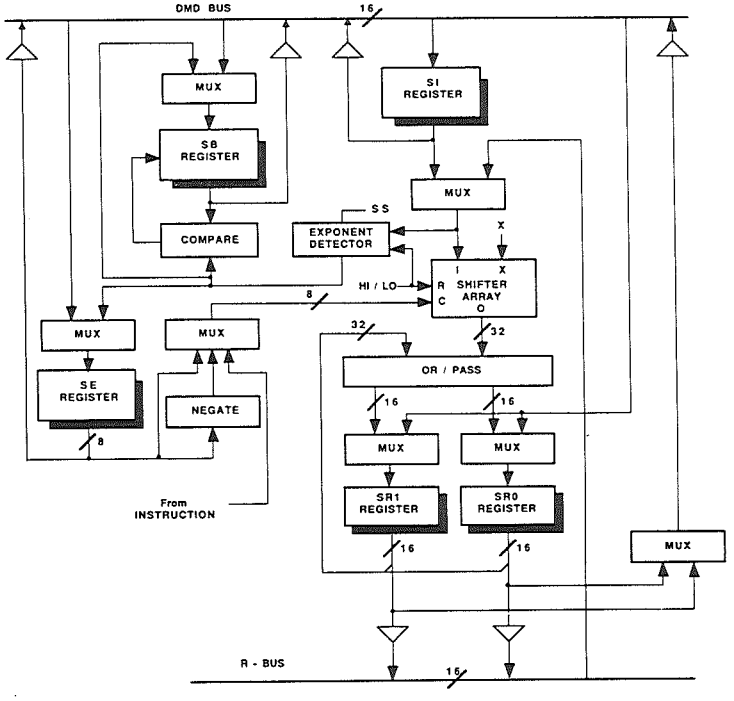

Figure 8.14

SHIFTER FEATURES

- Arithmetic and Logical Shifts
- Left and Right Shifts
- True Block Floating Point
- Direct Support for Double Precision
- Complete Set of Background Registers

Figure 8.15

SHIFTER OPERATIONS

- Normalize
- Denormalize
- Shift Immediate
- Derive Exponent
- Derive Block Exponent

Figure 8.16

EXAMPLE SHIFTER INSTRUCTIONS

- SR = ASHIFT SI BY -6
- SR = LSHIFT SR BY 3
- SR = NORM MR1

Figure 8.17

DATA ADDRESS GENERATORS (DAGs)

A block diagram of a data address generator is shown in Figure 8.18. The data address generators (DAGs) provide indirect addressing for data stored in the program and data memory spaces. The processor contains two independent DAGs so that two data operands (one in program memory and one in data memory) can be addressed simultaneously. The two data address generators are identical except that DAG1 has a bit reversal option on the output (used for FFTs) and can only generate data memory addresses, while DAG2 can generate both program and data memory addresses but has no bit reversal capability. Both DAGs can also be used for serial port autobuffering.

There are three register files in each DAG: the modify (M) register file, the index (I) register file, and the length (L) register file. Each of these register files contains four 14-bit registers which are readable and loadable from the DMD bus. The I registers hold the

DATA ADDRESS GENERATOR BLOCK DIAGRAM

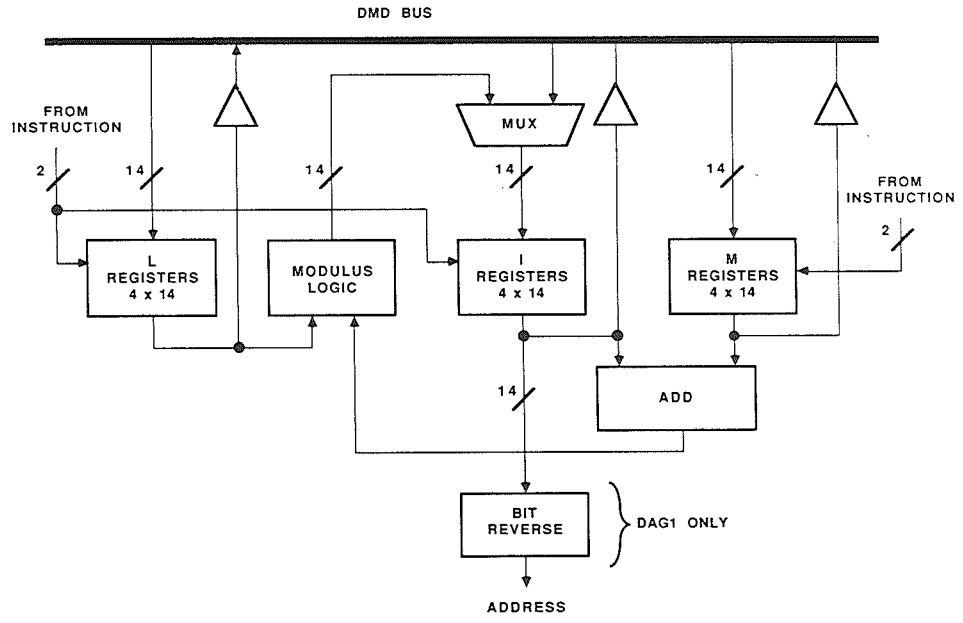

Figure 8.18

actual addresses used to access external memory. When using the indirect addressing mode, the selected I register content is driven onto either the PMA or DMA bus. This value is post-modified by adding the (signed) contents of the selected M register. The modified address is passed through the modulus logic.

Associated with each I register is an L register which contains the length of the buffer addressed by the I register. The L register and the modulus logic together enable circular buffer addressing with automatic wraparound at the buffer boundary. Automatic wraparound is also used by the serial ports to generate the serial port interrupt when operating in autobuffering mode. The modulus logic is disabled by setting the L register to zero.

8

ADDRESS GENERATOR FEATURES

- **Automatic Modulo Addressing**
- **Simultaneous Address Update**
- **Bit-Reverser (DAG # 1**

Figure 8.19

EXAMPLE ADDRESSING INSTRUCTIONS

- AX0 = DM (I0, M3)
- MODIFY (I1, M2)
- MR = MR+MX0*MY0, MX0=DM(I0,M1), MY0=PM(I4,M4)

Figure 8.20

PROGRAM SEQUENCER

The program sequencer incorporates powerful and flexible mechanisms for program flow control such as zero-overhead looping, single-cycle branching (both conditional and unconditional), and automatic interrupt processing. Figure 8.21 shows a block diagram of the program sequencer. The sequencing logic controls the flow of the program execution. It outputs a program memory address onto the PMA bus from one of four sources: the PC incrementer, PC stack, instruction register, or interrupt controller. The next address source selector controls which of these four sources are selected based on the current instruction word and the processor status. A fifth possible source for the next program memory address is provided by DAG2 when a register indirect jump is executed.

The program counter (PC) is a 14-bit register which contains the address of the currently executing instruction. The PC output goes to the incrementer. The incremented output is selected as the next program memory address if program flow is sequential. The PC value is pushed into the 16 x 14 PC stack when a CALL instruction is executed or when an interrupt is processed. The PC stack is popped when the return from a subroutine or interrupt is executed. The PC stack is also used in zero-overhead looping.

The program sequencer section contains six status registers. These are the Arithmetic Status register (ASTAT), the Stack Status register (SSTAT), the Mode Status register (MSTAT), the Interrupt Control register (ICNTL), the Interrupt Mask register (IMASK) and the Interrupt Force and Clear register (IFC).

The interrupt controller allows the processor to respond to the six possible interrupts with a minimum of overhead. Individual interrupt requests are logically ANDed with the bits in IMASK; the highest priority unmasked interrupt is then selected.

The interrupt control register, ICNTL, allows each interrupt to be set as either edge- or level-sensitive. Depending on Bit 4 in ICNTL, interrupt routines can either be nested with higher priority interrupts taking precedence or processed sequentially with only one interrupt service active at a time.

ADSP-2101 PROGRAM SEQUENCER

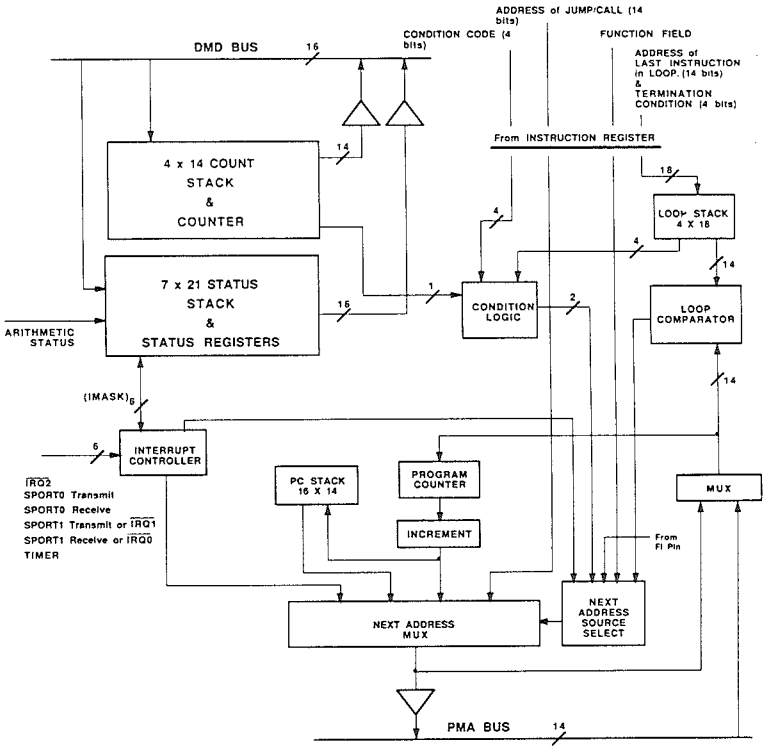

Figure 8.21

PROGRAM SEQUENCER FEATURES

- Automatic Operation, Transparent to User
- Full Interrupt Capabilities
- Four Stacks
- Single-Cycle Conditional Branch
- Zero-Overhead Looping

Figure 8.22

SERIAL PORTS

The ADSP-2101 incorporates two complete serial ports (SPORT0 and SPORT1) for serial communications and multiprocessor coordination. A block diagram of one of the serial ports is shown in Figure 8.23.

Each serial port has a 5-pin interface consisting of the signals shown in Figure 8.24.

SERIAL PORT BLOCK DIAGRAM

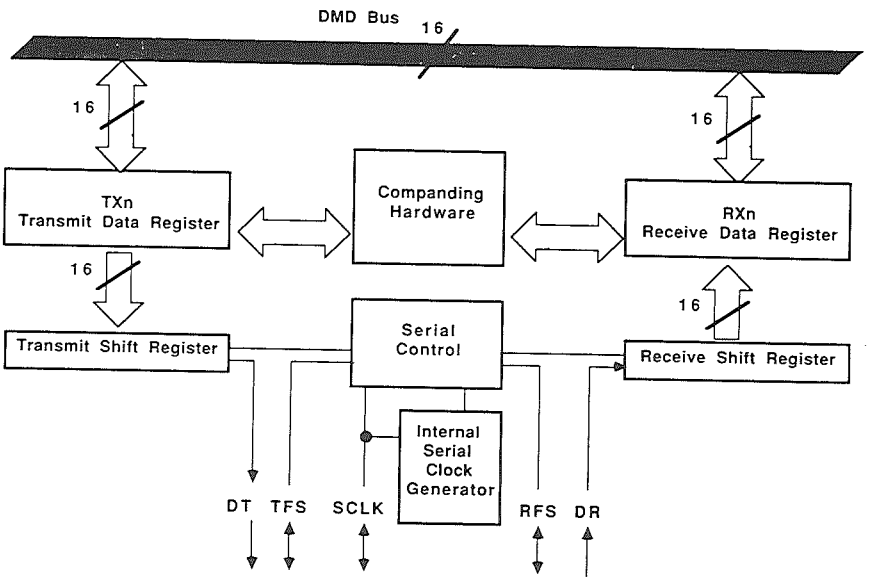

Figure 8.23

SERIAL PORT INTERFACE LINES

- SCLK — Serial Clock I/O
- RFS — Receive Frame Synch I/O
- TFS — Transmit Frame Synch I/O
- DR — Serial Data Receive
- DT — Serial Data Transmit

Figure 8.24

Each SPORT has a receive and a transmit register. Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding data to reduce the number of bits that must be sent. The ADSP-2101 supports both of the widely used algorithms for companding: A-law and μ-law. The type of companding can be independently selected for each SPORT.

SERIAL PORT FEATURES

- Dual Purpose Function of Serial Port 1
- Optional μ-Law and A-Law Companding
- Automatic Data Memory Buffering
- Programmable Word Length
- Multichannel Capabilities

Figure 8.25

SYSTEM INTERFACE

Figure 8.26 shows a basic system configuration with the ADSP-2101, two serial codecs, a boot EPROM and optional external Program and Data memories. Up to 15K words of data memory and 16K words of program memory can be supported. Programmable wait state generation allows the processor to interface easily to slow memories.

The ADSP-2101 also provides one external interrupt and two serial ports or three external interrupts and one serial port.

ADSP-2101 BASIC SYSTEM CONFIGURATION

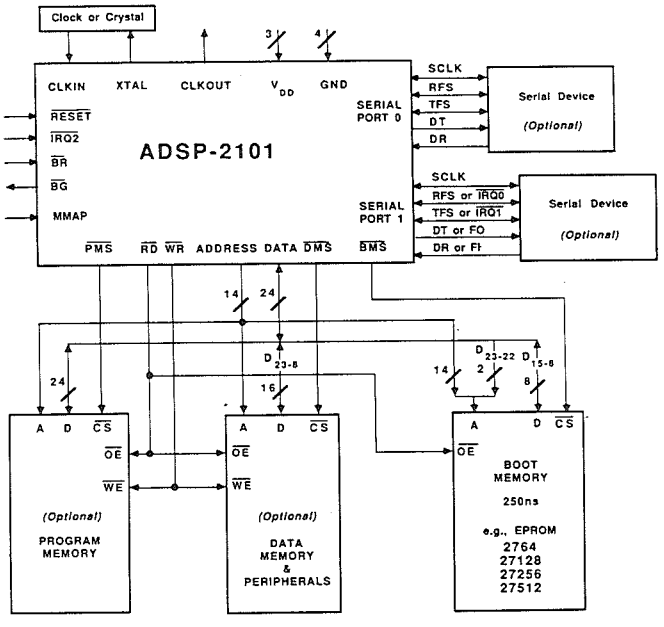

Figure 8.26

DEVELOPMENT SYSTEM

The ADSP-2101 is supported by a complete set of tools for software and hardware system development. The System Builder provides a high-level method for defining the architecture of systems under development. The Assembler produces the object code and the Linker combines object modules and library calls into an executable file. The Simulator provides an interactive instruction-level simulation with a reconfigurable user interface. A PROM Splitter generates PROM burner compatible files. The C Compiler generates ADSP-2101 assembly source code. An Emulator aids in the hardware debugging of ADSP-2101 systems.

HARDWARE AND SOFTWARE DEVELOPMENT TOOLS

- System Builder
- C Compiler
- Assembler
- Linker
- Simulator
- Prom Splitter
- Evaluation Board
- In-Circuit Emulator

Figure 8.27

ANALOG DEVICES DSP PROCESSOR PORTFOLIO

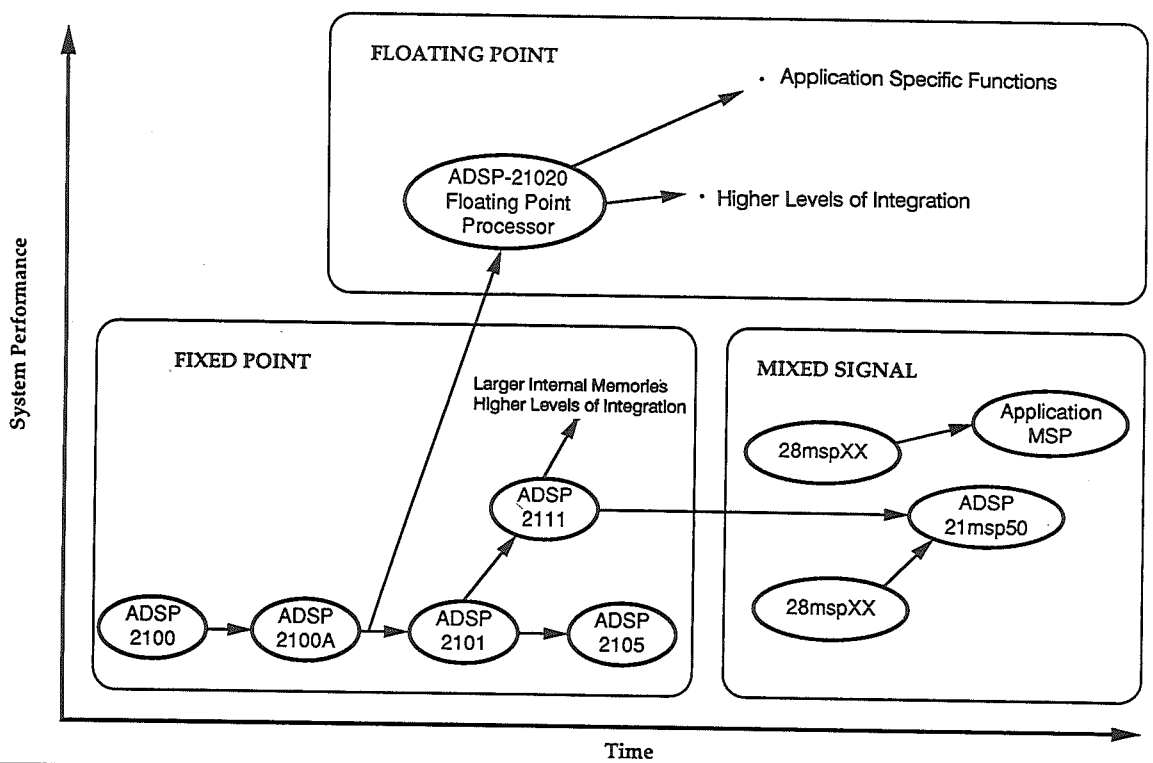

Figure 8.28

REFERENCES
(AVAILABLE FROM ANALOG DEVICES)

1. ADSP-2100/ADSP-2100A Digital Signal Processor, Data Sheet
2. ADSP-2101 DSP Microcomputer, Data Sheet
3. ADSP-2105 DSP Microcomputer, Data Sheet
4. ADSP-2111 DSP Microcomputer with Host Port, Data Sheet
5. ADDS-21XX DSP Software Development Tools, Data Sheet
6. ADDS-21XX DSP Hardware Development Tools, Data Sheet
7. ADDS-2101-SW DSP Software Development Tools, Data Sheet
8. ADSP-2101 Emulator, Data Sheet
9. ADDS-2101-EZ Tools, Data Sheet
10. ADSP-2101 User's Manual
11. ADSP-2101 Cross-Software Manual
12. ADSP-2101 Emulator Manual
13. ADSP-2101 EZ-ICE Manual
14. ADSP-2101 EZ-LAB Manual
15. ADSP-2111 User's Manual
16. *Digital Signal Processing Applications Using the ADSP-2100 Family* (Applications Handbook, Volumes 1, 2, and 3)
17. ADSP-2100 Family Applications Handbook, Volume 4