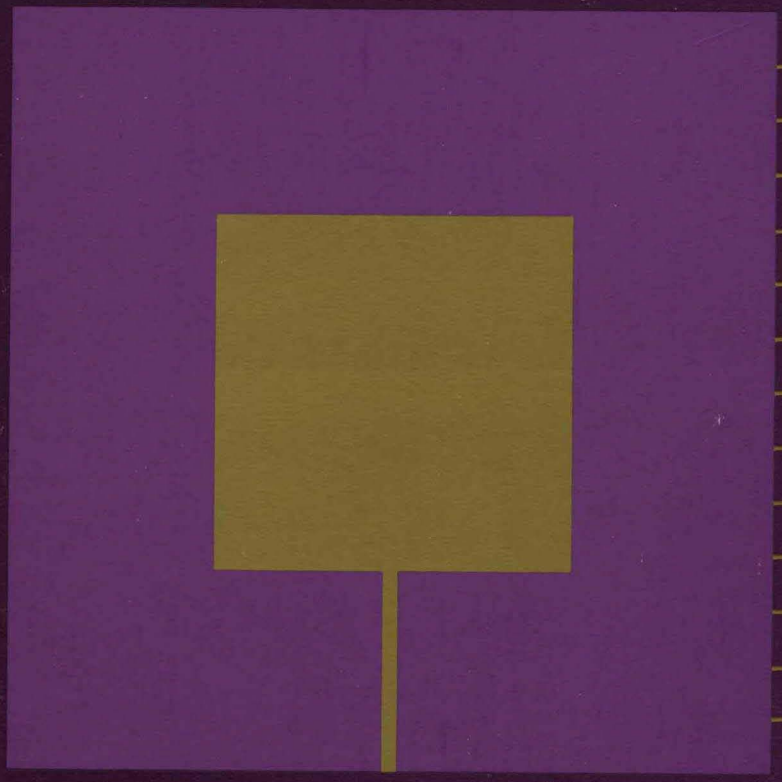
ADSP-2100 User's Manual

ANALOG DEVICES



ADSP-2100

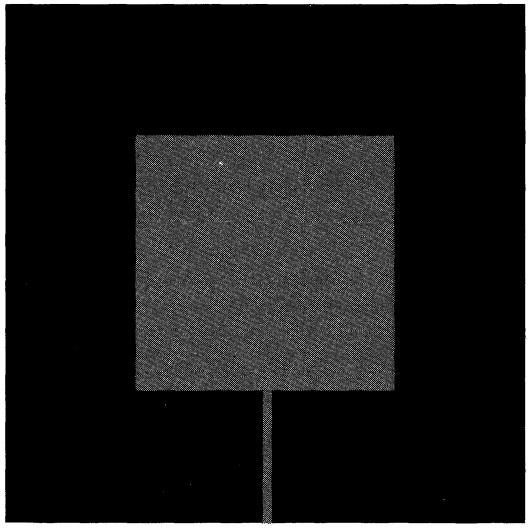
User's Manual

Architecture

ANALOG DEVICES

ADSP-2100

User's Manual



ANALOG DEVICES

You may contact the Digital Signal Processing Division in the following ways.

- By contacting your local Analog Devices Sales Representative
- For Marketing information, call (617) 461-3881 in Norwood, Massachusetts, USA
- For Applications Engineering information, call (617) 461-3672 in Norwood, Massachusetts, USA
- The Norwood office Fax number is (617) 461-3010
- The Norwood office may also be reached by
  Telex: 924491
  TWX: 710/394-6577
  Cables: ANALOG NORWOODMASS
- The DSP Division runs a Bulletin Board Service that can be reached at 300, 1200 or 2400 baud, no parity, 8 bits data, 1 stop bit by dialing:
  (617) 461-4258
- By writing to:
  Analog Devices
  DSP Division
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

ADSP-2100 User's Manual

PRINTED IN U.S.A. FOURTH EDITION

Literature

ADSP-2100 FAMILY MANUALS

ADSP-2100 User's Manual/Architecture
Complete description of architecture and system interface.

ADSP-2100 Cross-Software Manual
Complete programmer's reference including optional C compiler.

ADSP-2100 Emulator Manual
User's manual for the in-circuit Emulator.

ADSP-2100 Evaluation Board Manual
A guide to the Evaluation Board including schematics for prototyping.

ADSP-2101 User's Manual/Architecture (preliminary)
Complete description of architecture and system interface.

APPLICATIONS INFORMATION

ADSP-2100 Applications Handbook, Volume 1
Topics include arithmetic, filters, FFTs, LPC, modem algorithms.

ADSP-2100 Family Applications Handbook, Volume 2
Topics include graphics, pulse-code modulation, multirate filters, DTMF.

ADSP-2100 Family Applications Handbook, Volume 3
Topics include optimized and 2D FFTs, memory interface, multiprocessing, host interface, sonar beamforming.

SPECIFICATIONS INFORMATION

ADSP-2100A/ADSP-2100 Data Sheet

ADSP-2101 Data Sheet (preliminary)

Contents

Contents

Contents

CHAPTER 5 SYSTEM INTERFACE

Contents

Contents

FIGURES

Contents

TABLES

Introduction 1

1.1 GENERAL DESCRIPTION

The ADSP-2100 is a programmable single-chip microprocessor optimized for digital signal processing (DSP) and other high-speed numeric processing applications. The ADSP-2100 incorporates computational units, data address generators and a program sequencer in one device, utilizing external data and program memories.

The ADSP-2100 contains three full-function and independent computational units: an arithmetic/logic unit, a multiplier/accumulator and a barrel shifter. The computational units process 16-bit data directly and provide for multiprecision computation.

Two dedicated data address generators and a powerful program sequencer supply addresses. The sequencer supports single-cycle conditional branching and executes program loops with zero overhead. Dual address generators allow the processor to output simultaneous addresses for dual operand fetches. Together the sequencer and data address generators allow computational operations to execute with maximum efficiency. With the ability to store data in both program and data memory, the ADSP-2100 is capable of fetching two operands on the same instruction cycle.

Figure 1.1 is a simplified representation of the ADSP-2100 in a system context. The figure shows the two external memories used by the processor. Program memory stores instructions and is also used to store data. Data memory stores only data. The data memory address space may be shared with memory-mapped peripherals, if desired. Both memories may be accessed by external devices, such as a system host, if desired. Figure 1.1 also shows the processor control interface signals, ($\overline{\text{RESET}}$, $\overline{\text{HALT}}$ and $\overline{\text{TRAP}}$) the four interrupt request lines, the bus request and bus grant lines ($\overline{\text{BR}}$ and $\overline{\text{BG}}$) and the clock input (CLKIN) and output (CLKOUT). Complete interfacing information is presented in the chapter "System Interface."

The ADSP-2100 assembly language uses an algebraic syntax for ease of coding and readability. The sources and destinations of computations

1 Introduction

and data movements are written explicitly in each assembly statement, eliminating cryptic assembler mnemonics. Each assembly statement, however, corresponds to a single 24-bit instruction, executable in one cycle.

The ADSP-2100 architecture rivals the performance of a board level solution implemented with bit-slice building blocks, without the difficulty of microcode programming.

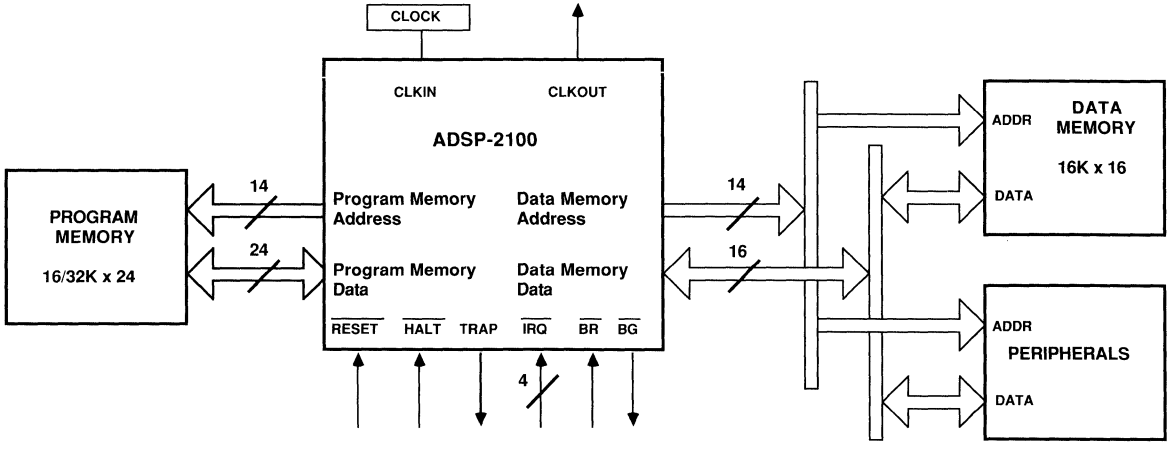


Figure 1.1 ADSP-2100 System

1.2 SUMMARY OF ADSP-2100 KEY FEATURES

- Separate Program and Data Buses, Extended Off-Chip
- Single-Cycle Direct Access to 16K x 16 of Data Memory
- Dual Purpose Program Memory for Both Instruction and Data Storage
- Single-Cycle Direct Access to 16K x 24 (Expandable to 32K) of Program Memory

Introduction 1

- Three Independent Computational Units:

  Arithmetic/Logic Unit (ALU)
  Multiplier/Accumulator (MAC)
  Barrel Shifter

- Two Independent Data Address Generators
- Powerful Program Sequencer
- Internal Instruction Cache
- Provisions for Multiprecision Computation and Saturation Logic
- Single-Cycle Instruction Execution
- Multifunction Instructions
- Four External Interrupts
- 50 MHz Clock Speed
- 80 ns Cycle Time
- Low Power Standby Mode
- 100-Pin Grid Array Package

1 Introduction

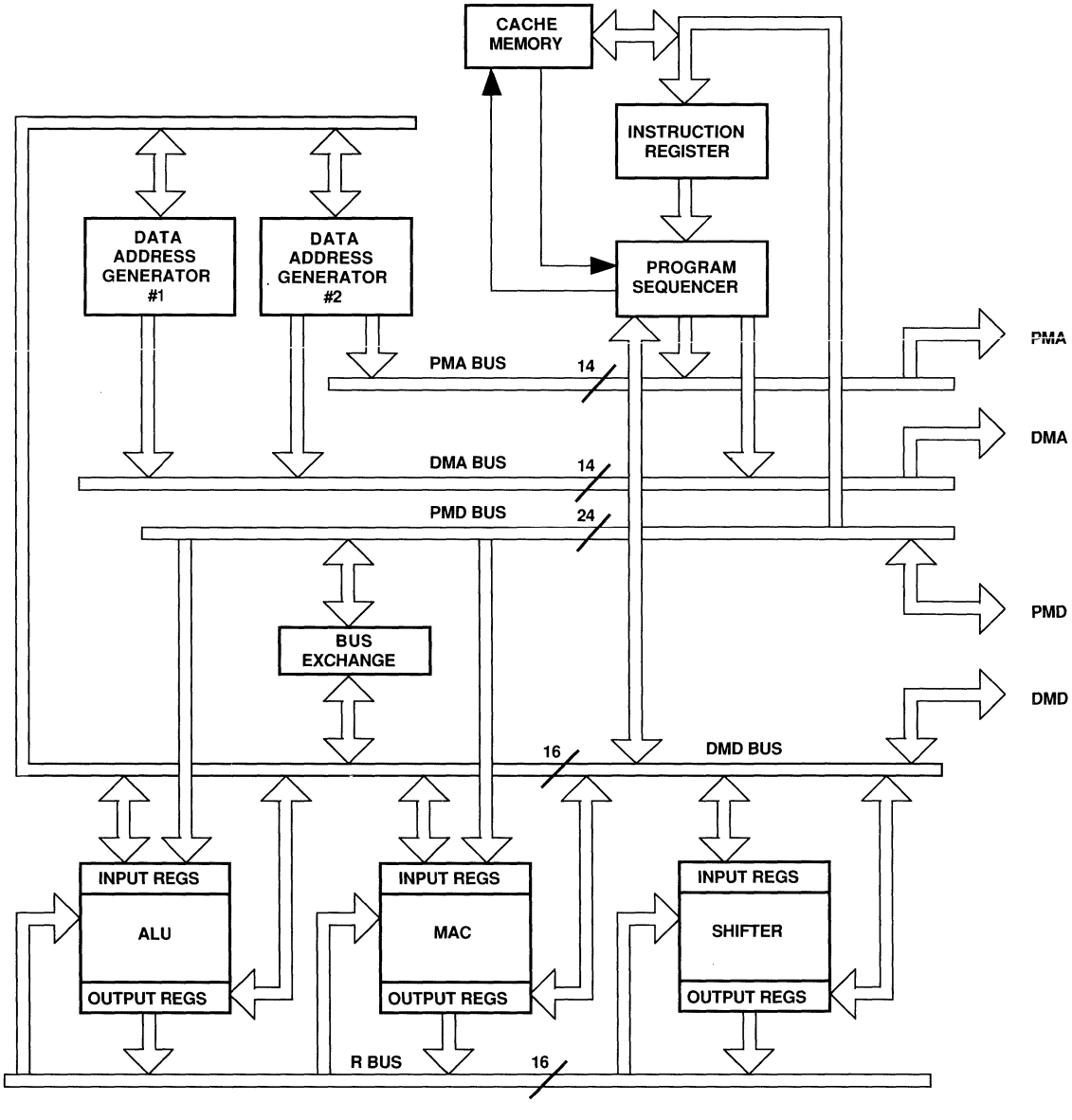


Figure 1.2 ADSP-2100 Internal Architecture

1.3 INTERNAL ARCHITECTURE

This section gives a broad overview of the ADSP-2100 internal architecture. The overview is based on Figure 1.2, on the facing page, which shows the architecture of the ADSP-2100. Each component is described in detail in the following chapters.

Component	*Chapter / Section*
• Arithmetic/logic unit	2.2
• Multiplier/accumulator	2.3
• Barrel shifter	2.4
• Two data address generators	3.2
• PMD-DMD bus exchange	3.3
• Program sequencer	4.2
• Status registers and stack	4.3
• Cache memory	4.4

These components are supported by five 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 (which interconnects the computational units)

The first four of these buses are extended off-chip for direct connection to external memories.

The program memory data (PMD) bus serves primarily to transfer instructions from off-chip memory to the internal instruction register. Instructions are fetched and loaded into the instruction register during one processor cycle and execute during the following cycle while the next instruction is being fetched. The instruction register introduces a single level of pipelining in the program flow. Instructions loaded into the instruction register are also written into the cache memory, described below.

The next instruction address is generated by the program sequencer depending on the current instruction and internal processor status. This address is output onto the program memory address (PMA) bus. The program sequencer minimizes program flow overhead with features such as conditional branching, loop counters and zero-overhead looping.

1 Introduction

The program memory address (PMA) bus is 14 bits wide allowing direct access of up to 16K words of instruction code and 16K words of data. The PMDA pin distinguishes between code and data access of program memory. The program memory data (PMD) bus is 24 bits wide to accommodate the 24-bit instruction width.

The data memory address (DMA) bus is 14 bits wide allowing direct access of up to 16K words of data. The data memory data (DMD) bus is 16 bits wide. The data memory data (DMD) bus provides a path for the contents of any register in the processor to be transferred to any other register or to any external data memory location in a single cycle. The data memory address comes from two sources: an absolute value specified in the instruction code (direct addressing) or the output of a data address generator (indirect addressing). Only indirect addressing is supported for data fetches via the program memory bus.

The program memory data (PMD) bus can also be used to transfer data to and from the computational units through direct paths or via the PMD-DMD bus exchange unit. The PMD-DMD bus exchange unit permits data to be passed from one bus to the other. It contains hardware to overcome the 8-bit width discrepancy between the two buses, if necessary.

The ADSP-2100 contains three computational blocks: an arithmetic/logic unit (ALU), a multiplier/accumulator (MAC) and a barrel shifter. Each unit functions independently of the others. All operate directly on 16-bit input data with provision for multiprecision operations. See the section "ADSP-2100 Arithmetic" in the next chapter.

All computational units contain a set of dedicated input and output registers. Computational operations generally take their operands from input registers and load the result into an output register. The registers act as a stopover point for data between the external memory and the computational circuitry, effectively introducing one pipeline level on input and one level on output. The computational units are arranged side by side instead of in a cascade fashion. To avoid excessive pipeline delays when a series of different operations are performed, the internal result (R) bus allows any of the output registers to be used directly as the input to another computation.

For a wide variety of calculations, it is desirable to fetch two operands at the same time; one from data memory and one from program memory. Fetching data from program memory, however, makes it impossible to fetch the next instruction on the same cycle. An additional cycle would be required to fetch the next instruction. To avoid this overhead, the ADSP-2100 incorporates an instruction cache which holds sixteen words. The benefit of the cache architecture is most apparent when executing a program loop totally contained in the cache memory. In this situation, the ADSP-2100 functions like a three bus system with an instruction fetch and two operand fetches taking place at the same time. Many algorithms can be coded in loops of sixteen instructions or less because of the efficiency and high-level syntax of the ADSP-2100 Assembly language. See the chapter "Instruction Set Overview."

Briefly, the cache functions in the following way. Every instruction loaded into the instruction register is also written into cache memory. As additional instructions are fetched, they overwrite the current contents of cache in a circular fashion. When the current instruction does a program memory data access, the cache automatically sources the instruction register if its contents are valid. Operation of the cache is completely transparent to user.

There are two independent data address generators (DAGs). Having two DAGs allows the simultaneous fetch of data stored in program and in data memory for executing dual-operand instructions in a single cycle. Data address generator one (DAG1) can supply addresses to the data memory only, but data address generator two (DAG2) can supply addresses to either the data memory or the program memory. Each DAG can handle linear addressing as well as modulo addressing for circular buffers.

With its multiple bus structure, the ADSP-2100 supports a high degree of operational parallelism. In a single cycle, the ADSP-2100 can fetch an instruction, compute the next instruction address, perform one or two data transfers, update one or two data address pointers and perform a computation. All instructions execute in a single cycle.

1 Introduction

1.4 ADSP-2100 DEVELOPMENT SYSTEM

The ADSP-2100 is supported with a complete set of software and hardware development tools. The ADSP-2100 Development System consists of the Cross-Software Development System to aid the software design and the real-time Emulator to facilitate the debug cycle. An Evaluation Board is available for evaluating the ADSP-2100. It is also suitable for limited prototyping of hardware interfacing.

The Cross-Software Development System includes:

- System Builder

This module allows the designer to specify the amount of RAM and ROM available, the allocation of program and data memory and any memory-mapped I/O ports for the target hardware environment. It uses high-level constructs to simplify this task. This specification is used by the other modules in the Cross-Software Development System.

- Assembler

This module assembles your source code and data modules. It supports the high-level syntax of the instruction set. To support modular code development, the Assembler provides flexible macro processing and include files. It provides a full range of diagnostics.

- Linker

The Linker links separately assembled modules. It maps the linked code and data output to the target system hardware, as specified by the System Builder output.

- Simulator

This module performs an instruction-level simulation. The user interface is both interactive and symbolic. It supports a full symbolic assembly and disassembly. The simulator fully simulates the hardware configuration described by the System Builder module. It flags illegal operations and provides several displays of the internal operations of the ADSP-2100.

- PROM Splitter

This module reads the Linker output and generates PROM burner compatible files.

- C Compiler

The C Compiler reads ANSI (Draft Standard) C source and outputs ADSP-2100 source code ready to be assembled. It also supports inline assembler code.

- In-Circuit Emulator

The Emulator provides stand-alone real-time in-circuit emulation, using the ADSP-2100 in a self-emulation mode. The Emulator design provides execution with little or no degradation in processor performance. In addition, there are interfaces to external instrumentation. The Emulator virtually duplicates the Simulator's interactive and symbolic user interface.

For complete information on the Development System, consult the *ADSP-2100 Cross-Software Manual* and the *ADSP-2100 Emulator Manual*.

1.5 MANUAL ORGANIZATION

The *ADSP-2100 User's Manual* provides the information necessary for an engineer to understand and evaluate the operation of the ADSP-2100. Together with the *ADSP-2100 Data Sheet*, this manual provides all the information required to design a hardware system with the ADSP-2100. You must consult the *ADSP-2100 Cross-Software Manual* for complete information on programming the chip. Additional applications information may be found in the *ADSP-2100 Applications Handbook, Volume 1* and *Volume 2*.

Chapter 2, "Computational Units," describes the internal architecture and function of the ADSP-2100 computational units.

Chapter 3, "Data Moves," describes the data address generators and the PMD-DMD bus exchange units.

1 Introduction

Chapter 4, "Program Control," describes the program sequencer, instruction cache and status words.

Chapter 5, "System Interface," describes the chip externally. It discusses all major interfaces to the ADSP-2100: the program memory (PM) interface, the data memory (DM) interface, the control interface and the interrupt lines. This chapter gives a functional description of the interfaces and their sequence of operations. For actual timing parameters, refer to the *ADSP-2100 Data Sheet*. A summary of the pin descriptions is given in this chapter.

Chapter 6, "Instruction Set Overview," is an overview of the ADSP-2100 instruction set. All instructions are grouped by major type. Detailed programmer's reference material is in the *ADSP-2100 Cross-Software Manual*; this chapter gives enough information for you to understand the capabilities and flexibility of the instruction set.

Appendix A, "Instruction Coding," shows the complete set of opcodes and gives the bit patterns for the choices for each field within the instruction word.

Appendix B, "Division Exceptions," discusses the details of signed and unsigned division.

This edition also includes an Index at the end of the book.

Computational Units 2

2.1 ARITHMETIC ON THE ADSP-2100

This chapter describes the architecture and function of the three computational units of the ADSP-2100: the arithmetic/logic unit, the multiplier/accumulator and the barrel shifter.

To better understand the detailed discussion of these units you should first understand how the ADSP-2100 handles binary arithmetic. The ADSP-2100 is a 16-bit, fixed-point machine. Special features support multiword arithmetic and block floating point. Most operations assume a twos-complement number while others assume an unsigned-magnitude number or a simple binary string. This section discusses the arithmetic used by each computational unit or operation.

2.1.1 Binary String

This is the simplest form of binary notation. Sixteen bits are treated as a bit pattern. The best examples of computation using this format are the logical operations: NOT, AND, XOR. These ALU operations treat their operands as binary strings with no provision for sign bit or binary point placement.

2.1.2 Unsigned-Magnitude

Unsigned magnitude binary numbers have no sign bit. They are frequently thought of as positive, having nearly twice the magnitude of a signed number of the same bit length. The lower words of multiword numbers are treated as unsigned-magnitude numbers.

2.1.3 Signed Numbers: Twos-Complement

Twos-complement is one of the most common ways to represent signed binary numbers. It uses the MSB of a binary number as a sign bit. Twos-complement provides a unique representation for zero, where some other formats have both a positive and negative zero. In twos-complement the largest negative magnitude is one LSB greater than the largest positive magnitude.

2 Computational Units

In discussions of ADSP-2100 arithmetic "signed" refers to twos-complement. Most ADSP-2100 operations presume or support twos-complement arithmetic. The ADSP-2100 does not use signed-magnitude formats.

2.1.4 Fractional Representation: 1.15

A large number of DSP algorithms use sinusoidal and cosinusoidal values and coefficients. The ADSP-2100 is optimized for arithmetic values in a fractional binary format denoted by 1.15 ("one dot fifteen"). (Referred to in some contexts as 16.15.) This is a fixed-point format. Used with the MSB as a sign bit, the 1.15 means one sign bit and fifteen fractional bits representing values from –1 up to one LSB less than +1. In the ADSP-2100 the fractional notation and twos-complement always occur together.

2.1.5 ALU Arithmetic

All operations on the ALU treat operands and results as simple 16-bit binary strings, except the signed division primitive (DIVS). Various status bits treat the results as signed: the overflow (AV) condition code, and the zero (AZ) and negative (AN) flags.

The logic of the overflow bit (AV) is based on twos-complement. It is set if the MSB changes in a manner not predicted by the signs of the operands and the nature of the operation. For example, adding two positive numbers must generate a positive result; a change in the sign bit signifies an overflow and sets AV. Adding a negative and a positive may result in either a negative or positive result, but cannot overflow.

The logic of the carry bit (AC) is based on unsigned-magnitude. It is set if a carry is generated from bit 16 (the MSB). The (AC) bit is most useful for the lower word portions of a multiword operation.

2.1.6 MAC Arithmetic

The multiplier array itself produces results that are simple binary strings, but the inputs are "interpreted" according to the information given in the multiplication instruction itself (signed by signed, unsigned by unsigned, a mixture or round).

The number loaded into MR from the multiplier is assumed to be signed in that it is always sign-extended across the full 40 bit width of the MR register set.

There is a built-in shift left that occurs between the multiplier product (P) and the multiplier result register (MR). Figure 2.6, in the MAC section of this chapter, shows this graphically. This shift occurs because the ADSP-2100 assumes that the operands are in 1.15 format. Without the shift the 32-bit result would be in 2.30 format. If a 2.30 value is rounded to 16-bits, the result would be 2.14, which is incompatible with 1.15. For this reason, the multiplier result is always shifted one bit to the left, producing a 1.31 result, which can be rounded to 1.15.

Therefore, to multiply twos-complement integers (16.0 not 1.15 format), you must compensate for the shift that occurs. Typically, this would mean shifting the result down (or right) one bit to get the correct 32-bit, twos-complement value. Since the MAC output register set stores 40 bits, this result is not lost and can be retrieved with the Shifter.

2.1.7 Shifter Arithmetic

Many operations in the Shifter are explicitly geared to signed (twos-complement) or unsigned values: Logical Shifts assume unsigned-magnitude or binary string values and Arithmetic Shifts assume twos-complement.

The exponent logic assumes twos-complement numbers. The exponent logic supports block floating point, which is also based on twos-complement numbers.

2.1.8 Summary

In addition to the numeric types described in this section, the ADSP-2100 C Compiler supports a form of 32-bit floating-point in which one 16-bit word is the exponent and the other 16-bit word is the mantissa. See the discussion in the C Compiler chapter of the *ADSP-2100 Cross-Software Manual*.

2 Computational Units

The table below summarizes some of the arithmetic characteristics of the ADSP-2100 computational units and operations.

Operation	*Arithmetic Formats*	
	Operands	*Result*
ALU		
Addition	Signed or unsigned	Interpret flags
Subtraction	Signed or unsigned	Interpret flags
Logical Operations	Binary string	same
Division	Explicitly signed / unsigned	same
ALU Overflow	Signed	same
ALU Carry Bit	16-bit unsigned	same
ALU Saturation	Signed	same
MAC		
Multiplication (P)	1.15 Explicitly signed / unsigned	32-bits
Multiplication (MR)	1.15 Explicitly signed / unsigned	2.30 shifted to 1.31
Mult+Cum. Add	1.15 Explicitly signed / unsigned	2.30 shifted to 1.31
Mult+Cum. Subtract	1.15 Explicitly signed / unsigned	2.30 shifted to 1.31
MAC Overflow	Signed	same
MAC Saturation	Signed	same
Shifter		
Logical Shift	Unsigned / binary string	same
Arithmetic Shift	Signed	same
Exponent Detection	Signed	same

Table 2.1 Arithmetic Formats Used by the ADSP-2100

2.2 ARITHMETIC/LOGIC UNIT (ALU)

The Arithmetic/Logic Unit (ALU) provides a standard set of arithmetic and logical functions. The arithmetic functions are add, subtract, negate, increment, decrement and absolute value. These are supplemented by two division primitives with which multiple cycle division can be constructed. The logic functions are AND, OR, XOR (exclusive OR) and NOT.

2.2.1 ALU Block Diagram Discussion

Figure 2.1 shows a block diagram of the ALU.

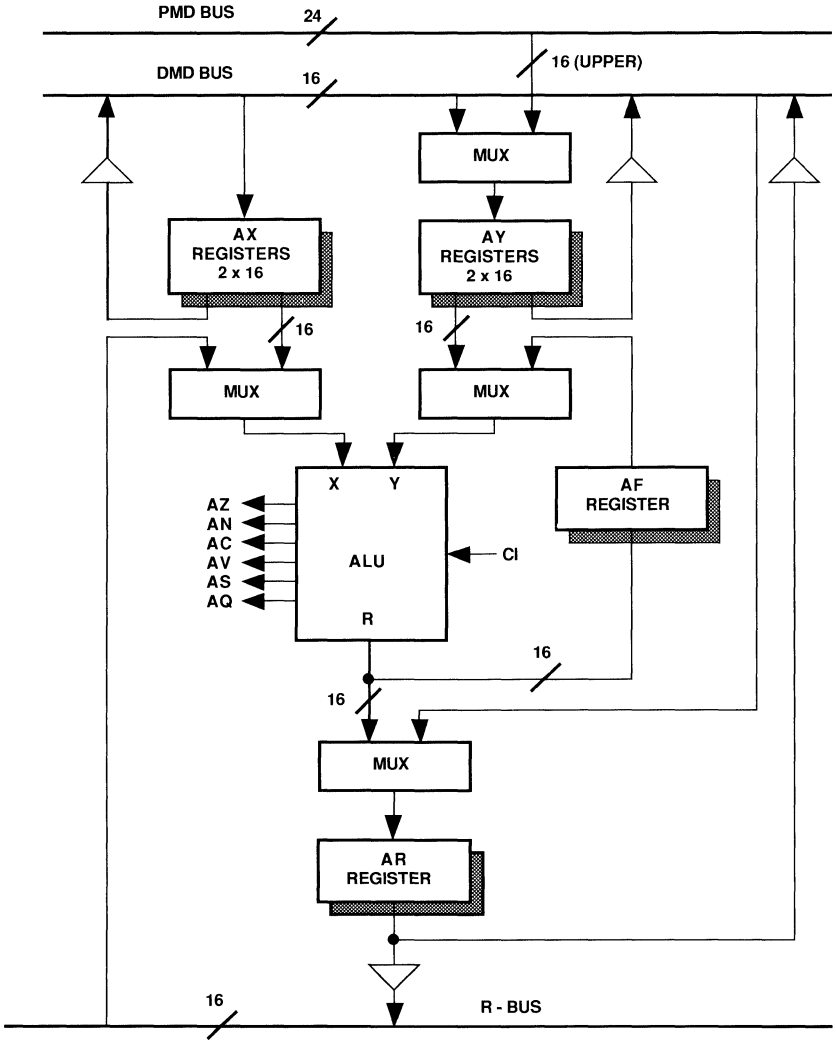


Figure 2.1 ALU Block Diagram

2 Computational Units

The ALU is 16 bits wide with two 16-bit input ports, X and Y, and one output port, R. The ALU accepts a carry-in signal (CI) which is the carry bit from the processor arithmetic status register (ASTAT). The ALU generates six status signals: the zero (AZ) status, the negative (AN) status, the carry (AC) status, the overflow (AV) status, the X-input sign (AS) status, and the quotient (AQ) status. All arithmetic status signals are latched into the arithmetic status register (ASTAT) at the end of the cycle.

The X input port of the ALU can accept data from two sources: the AX register file or the result (R) bus. The R bus connects the output registers of all the computational units, permitting them to be used as input operands directly. The AX register file is dedicated to the X input port and consists of two registers, AX0 and AX1. These AX registers are readable and writable from the DMD bus. The AX register file outputs are dual-ported so that one register can provide input to the ALU while either one simultaneously drives the DMD bus.

The Y input port of the ALU can also accept data from two sources: the AY register file and the ALU feedback (AF) register. The AY register file is dedicated to the Y input port and consists of two registers, AY0 and AY1. These registers are readable and writable from the DMD bus and writable from the PMD bus. The ADSP-2100 instruction set also provides for reading these registers over the PMD bus, but there is no direct connection; this operation uses the DMD-PMD bus exchange unit. The AY register file outputs are also dual-ported: one AY register can provide input to the ALU while either one simultaneously drives the DMD bus.

The output of the ALU is loaded into either the ALU feedback (AF) register or the ALU result (AR) register. The AF register is an ALU internal register which allows the ALU result to be used directly as the ALU Y input. The AR register can drive both the DMD bus and the R bus. It is also loadable directly from the DMD bus.

All the registers surrounding the ALU can be both read and written in the same cycle. Registers are read at the beginning of the cycle and written at the end of the cycle. All register reads, therefore, read values loaded at the end of a previous cycle. A new value written to a register cannot be read out until a subsequent cycle. This allows an input register to provide an operand to the ALU at the beginning of the cycle and be updated with the next operand from memory at the end of the same cycle. It also allows a

result register to be stored in memory and updated with a new result in the same cycle. See the discussion of "Multifunction Instructions" in the chapter "Instruction Set Overview" for an illustration of this same-cycle read and write.

The ALU section contains a duplicate bank of registers, shown in Figure 2.1 as a "shadow" behind the primary registers. There are actually two sets of AR, AF, AX, and AY register files. Only one bank is accessible at a time. The additional bank of registers can be activated during an interrupt service routine for extremely fast context switching. A new task, such as an interrupt service routine, can be executed without transferring current states to storage.

The selection of the primary or alternate bank of registers is controlled by a bit in the processor mode status register (MSTAT). Toggling this bit switches back and forth between the two register banks.

2.2.2 Standard Functions

The standard functions performed by the ALU are listed below with a brief comment.

R = X + Y	Add X and Y operands
R = X + Y + CI	Add X and Y operands and carry-in bit
R = X – Y	Subtract Y from X operand
R = X – Y + CI - 1	Subtract Y from X operand with "borrow"
R = Y – X	Subtract X from Y operand
R = Y – X + CI - 1	Subtract X from Y operand with "borrow"
R = – X	Negate X operand *(twos-complement)*
R = – Y	Negate Y operand *(twos-complement)*
R = Y + 1	Increment Y operand
R = Y – 1	Decrement Y operand
R = PASS X	Pass X operand to result unchanged
R = PASS Y	Pass Y operand to result unchanged
R = 0 *(PASS 0)*	Clear result to zero
R = ABS X	Absolute value of X operand
R = X AND Y	Logical AND of X and Y operands
R = X OR Y	Logical OR of X and Y operands
R = X XOR Y	Logical Exclusive OR of X and Y operands
R = NOT X	Logical NOT of X operand *(ones-complement)*
R = NOT Y	Logical NOT of Y operand *(ones-complement)*

2 Computational Units

2.2.3 ALU Input/Output Registers

The sources of ALU input and output registers are shown below.

Source for X input port	*Source for Y input port*	*Destination for R output port*
AX0, AX1	AY0, AY1	AR
AR	AF	AF
MR0, MR1, MR2		
SR0, SR1		

MR0, MR1 and MR2 are Multiplier/Accumulator result registers; SR0 and SR1 are Shifter result registers.

2.2.4 Multiprecision Capability

Multiprecision operations are supported in the ALU with the carry-in (CI) signal and ALU carry (AC) status bit. The carry-in signal is the AC status bit that was generated by a previous ALU operation. The "add with carry" (+CI) operation is intended for adding the upper portions of multiprecision numbers. The "subtract with borrow" (CI - 1 is effectively a "borrow") operation is intended for subtracting the upper portions of multiprecision numbers.

2.2.5 ALU Saturation Mode

The AR register has an optional saturation mode of operation which automatically sets it to plus or minus the maximum value if an ALU result overflows or underflows. This feature is a "mode" and is enabled by setting a bit in the processor mode status register (MSTAT). When enabled, the value loaded into AR during an ALU operation depends on the state of the overflow and carry status generated by the ALU on that cycle. The following table summarizes the loading of the AR when the saturation mode is enabled.

Overflow (AV)	*Carry (AC)*	*AR Contents*	
0	0	ALU Output	
0	1	ALU Output	
1	0	0111111111111111	*full-scale positive*
1	1	1000000000000000	*full-scale negative*

Table 2.2 Saturation Mode

The operation of the ALU saturation mode is in contrast to the Multiplier/Accumulator saturation ability, which is enabled only on an instruction by instruction basis. For the ALU, enabling saturation means that all subsequent operations are processed this way.

2.2.6 ALU Overflow Latch Mode

The ALU overflow latch mode, enabled by a bit in the processor mode status register (MSTAT), causes the AV bit to "stick" once it is set. In this mode, when an ALU overflow occurs, AV will be set and remain set, even if subsequent ALU operations do not generate overflows. In this mode, AV can only be cleared by writing a zero to it directly from the DMD bus.

2.2.7 Division

The ALU section supports division. The divide function is achieved with additional shift circuitry not shown in Figure 2.1, the block diagram. Division is accomplished with two special divide primitives. These are used to implement a non-restoring conditional add-subtract division algorithm. The division can be either signed or unsigned, however, the dividend and divisor must both be of the same type. Appendix B details various exceptions to the normal division operation as described in this section.

A single precision divide, with a 32-bit dividend (numerator) and a 16-bit divisor (denominator), yielding 16-bit quotient, executes in 16 cycles. Higher precision dividends can also be calculated. The divisor can be stored in AX0, AX1 or any of the R registers. The upper half of a signed dividend can be in either AY1 or AF. The upper half of an unsigned dividend must be in AF. The lower half of any dividend must be in AY0. At the end of the divide operation, the quotient will be in AY0.

The first of the two primitive instructions "divide-sign (DIVS)" is executed at the beginning of the division when dividing signed numbers. This operation computes the sign bit of the quotient by performing an exclusive-OR of the sign bits of the divisor and the dividend. The AY0 register is shifted one place so that the computed sign bit is moved into the LSB position. The computed sign bit is also loaded into the AQ bit of the arithmetic status register. The MSB of AY0 shifts into the LSB position of AF, and the upper 15 bits of AF are loaded with the lower 15 R bits from the ALU, which simply passes the Y input value straight through to the R output. The net effect is to left shift the AF-AY0 register pair and move the quotient sign bit into the LSB position. The operation of DIVS is illustrated in Figure 2.2 (on the following page).

When dividing unsigned numbers, the DIVS operation is not used. Instead, the AQ bit in the arithmetic status register (ASTAT) should be initialized to zero by manually clearing it. The AQ bit indicates to the following operations that the quotient should be assumed positive.

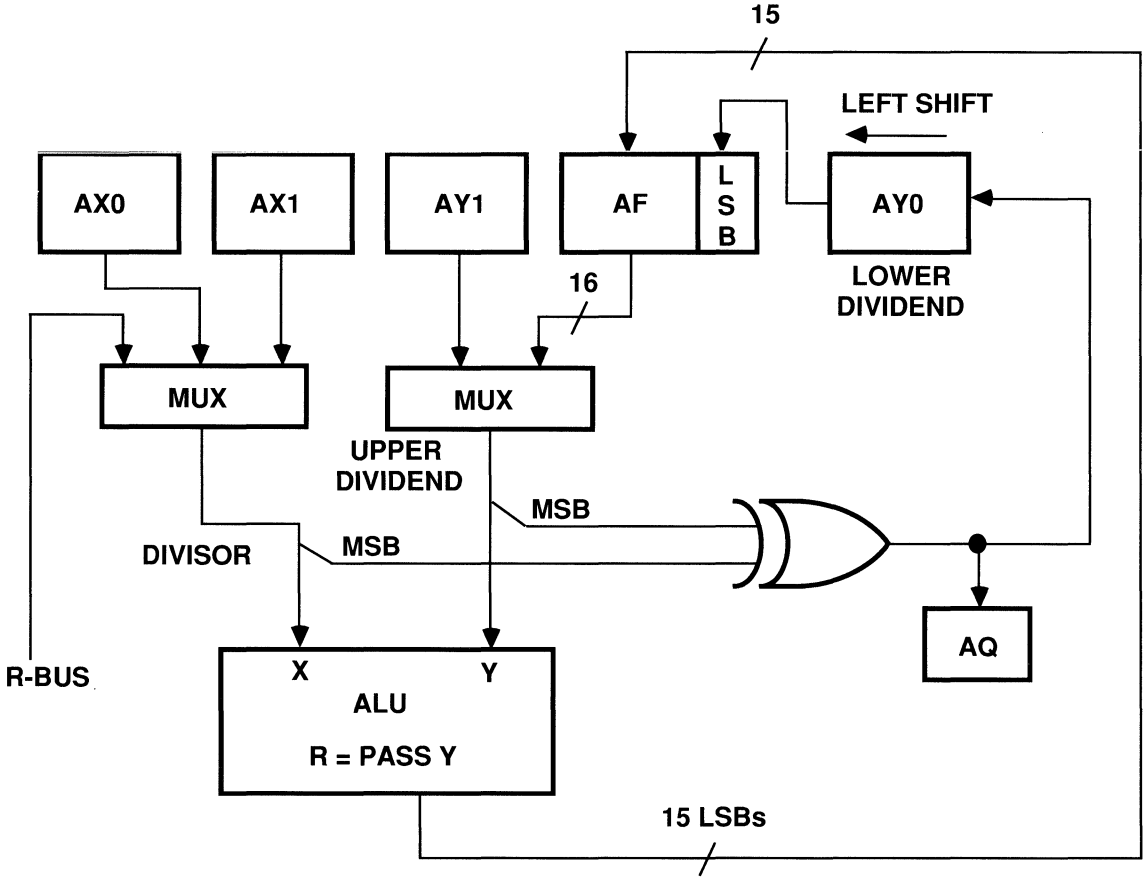


Figure 2.2 DIVS Operation

The second primitive instruction is the "divide-quotient (DIVQ)" operation which generates one bit of quotient at a time and is executed repeatedly to compute the remaining quotient bits. For unsigned single precision divides, the DIVQ instruction is executed 16 times to produce 16 quotient bits. For signed single precision divides, the DIVQ instruction is executed 15 times after the sign bit is computed by the DIVS operation. DIVQ instruction shifts the AY0 register left by one bit so that the new quotient bit can be moved into the LSB position. The status of the AQ bit generated from the previous operation determines the ALU operation to calculate the partial remainder. If AQ = 1, the ALU adds the divisor to the partial remainder in AF. If AQ = 0, the ALU subtracts the divisor from the partial remainder in AF. The ALU output R is offset loaded into AF just as with the DIVS operation. The AQ bit is computed as the exclusive-OR of the divisor MSB and the ALU output MSB, and the quotient bit is this value inverted. The quotient bit is loaded into the LSB of the AY0 register which is also shifted left by one bit. The DIVQ operation is illustrated in Figure 2.3.

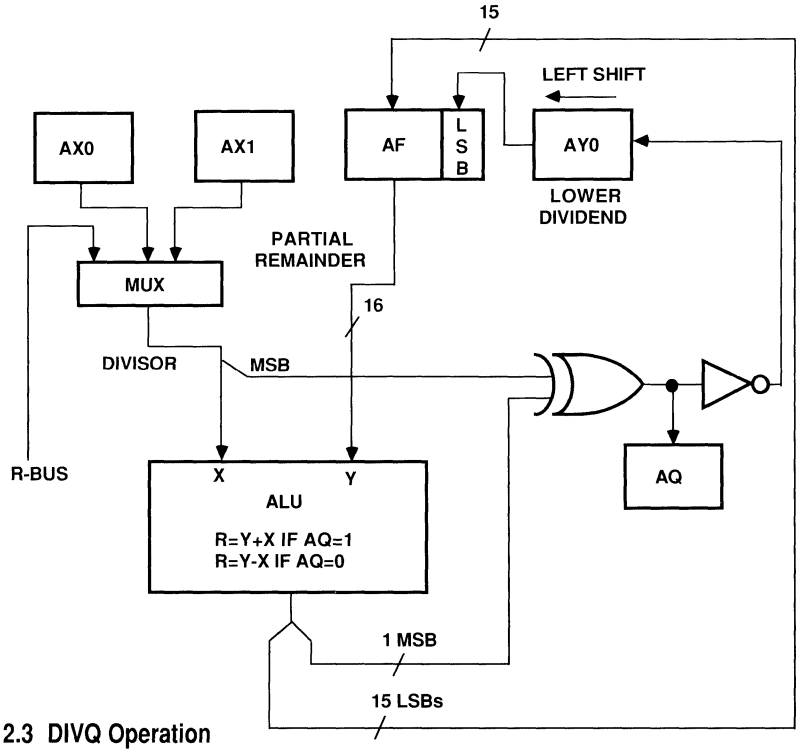


Figure 2.3 DIVQ Operation

2 Computational Units

The format of the quotient for any numeric representation can be determined by the format of the dividend and divisor. Let NL represent the number of bits to the left of the binary point, and NR represent the number of bits to the right of the binary point of the dividend; DL represent the number of bits to the left of the binary point, and DR represent the number of bits to the right of the binary point of the divisor; then the quotient has NL–DL+1 bits to the left of the binary point and NR–DR–1 bits to the right of the binary point.

Some format manipulation may be necessary to guarantee the validity of the quotient. For example, if both operands are signed and fully fractional (dividend in 1.31 format and divisor in 1.15 format) the result is fully fractional (in 1.15 format) and therefore the dividend must be smaller than the divisor for a valid result. To divide two integers (dividend in 32.0 format and divisor in 16.0 format) and produce an integer quotient (in 16.0 format), you must shift the dividend one bit to the left (into 31.1 format) before dividing. Additional discussion and code examples can be found in the *ADSP-2100 Applications Handbook, Volume 1.*

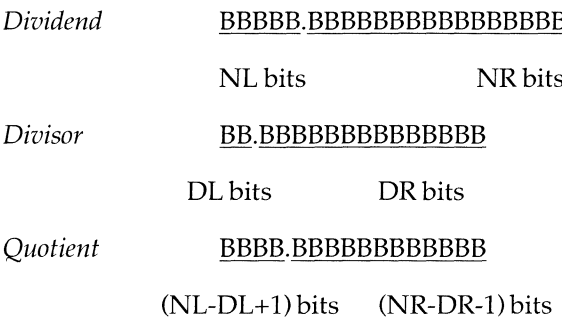


Figure 2.4 Quotient Format

The algorithm overflows if the result cannot be represented in the format of the quotient as calculated above or when the divisor is zero or less than the dividend.

2.2.8 ALU Status

The ALU status bits in the ASTAT register are defined below. Complete information about the ASTAT register and specific bit mnemonics and positions is provided in Chapter 4, "Program Control."

Flag	*Name*	*Definition*
AZ	Zero	Logical NOR of all the bits in the ALU result register. True if ALU output equals zero.
AN	Negative	Sign bit of the ALU result. True if the ALU output is negative.
AV	Overflow	Exclusive-OR of the carry outputs of the two most significant adder stages. True if the ALU overflows.
AC	Carry	Carry output from the most significant adder stage.
AS	Sign	Sign bit of the ALU X input port. Affected only by the ABS instruction.
AQ	Quotient	Quotient bit generated only by the DIVS and DIVQ instructions.

2.3 MULTIPLIER/ACCUMULATOR (MAC)

The Multiplier/Accumulator (MAC) provides high-speed multiplication, multiplication with cumulative addition, multiplication with cumulative subtraction and clear-to-zero functions. A feedback function allows part of the accumulator output to be directly used as one of the multiplicands on the next cycle.

2.3.1 MAC Block Diagram Discussion

Figure 2.5, on the following page, shows a block diagram of the multiplier/accumulator section.

The multiplier has two 16-bit input ports X and Y, and a 32-bit product output port P. The 32-bit product is passed to a 40-bit adder/subtractor which adds or subtracts the new product from the content of the multiplier result (MR) register. The MR register is 40-bits wide. In this manual, we refer to the entire register as MR. The register actually consists of three smaller registers: MR0 and MR1 which are 16 bits wide and MR2 which is 8 bits wide. The adder/subtractor is greater than 32 bits to allow for intermediate overflow in a series of multiply/accumulate operations. The multiply overflow (MV) status bit is set when the accumulator has overflowed beyond the 32-bit boundary, that is, when there are significant (non-sign) bits in the top nine bits of the MR register (based on twos-complement arithmetic).

2 Computational Units

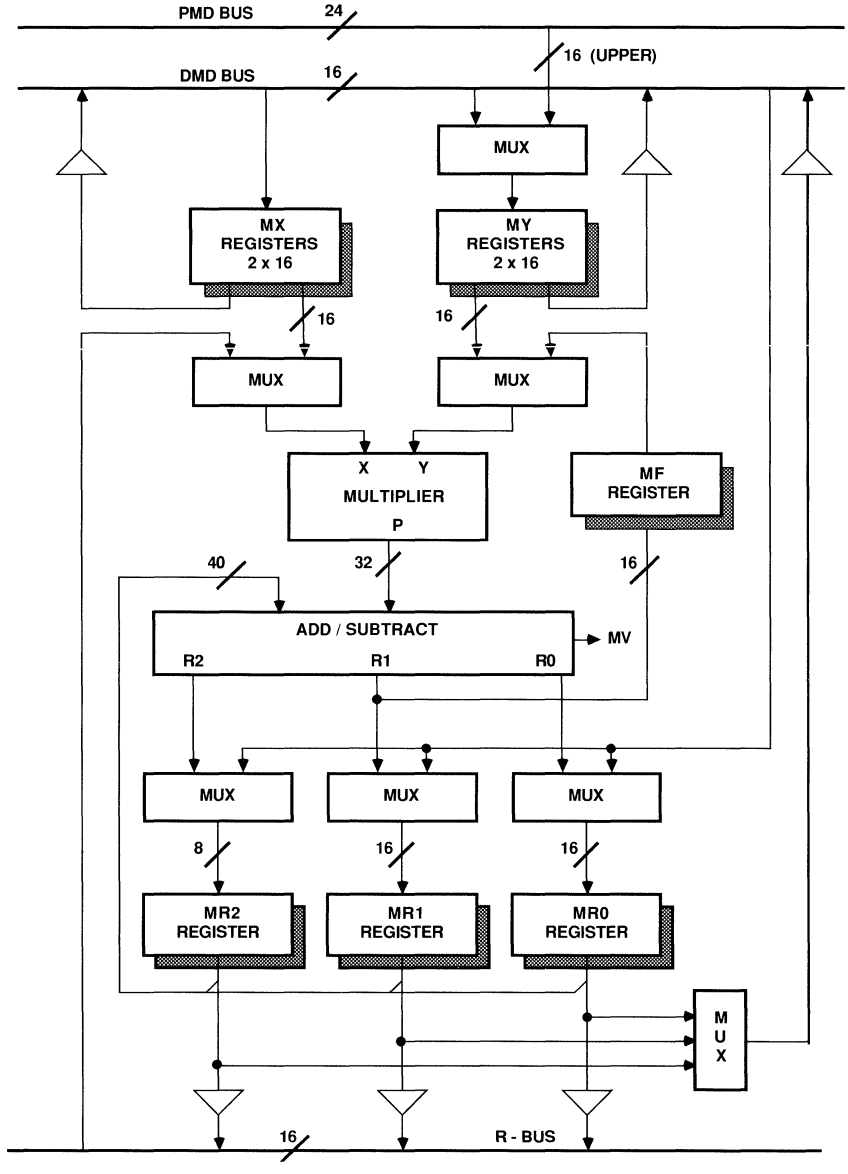


Figure 2.5 Multiplier/Accumulator Block Diagram

The input/output registers of the MAC section are similar to the ALU.

The X input port can accept data from either the MX register file or from any register on the result (R) bus. The R bus connects the output registers of all the computational units, permitting them to be used as input operands directly. There are two registers in the MX register file, MX0 and MX1. These registers can be read and written from the DMD bus. The MX register file outputs are dual-ported so that a single register can drive the DMD bus at the same time it supplies operands to the multiplier.

The Y input port can accept data from either the MY register file or the MF register. The MY register file has two registers, MY0 and MY1; these registers can be read and written from the DMD bus and written from the PMD bus. The ADSP-2100 instruction set also provides for reading these registers over the PMD bus, but there is no direct connection; this operation uses the DMD-PMD bus exchange unit. The MY register file outputs are also dual-ported so that a single register can drive the DMD bus at the same time it supplies operands to the multiplier.

The output of the adder/subtractor goes to either the MF register or the MR register. The MF register is a feedback register which allows bits 16–31 of the result to be used directly as the multiplier Y input on a subsequent cycle. The 40-bit adder/subtractor register (MR) is divided into three sections: MR2, MR1, and MR0. Each of these registers can be preloaded directly from the DMD bus and output to either the DMD bus or the R bus.

All the registers surrounding the MAC can be both read and written in the same cycle. Registers are read at the beginning of the cycle and written at the end of the cycle. All register reads, therefore, read values loaded at the end of a previous cycle. A new value written to a register cannot be read out until a subsequent cycle. This allows an input register to provide an operand to the MAC at the beginning of the cycle and be updated with the next operand from memory at the end of the same cycle. It also allows a result register to be stored in memory and updated with a new result in the same cycle. See the discussion of "Multifunction Instructions" in the chapter "Instruction Set Overview" for an illustration of this same-cycle read and write.

The MAC section contains a duplicate bank of registers, shown in Figure 2.5 as a "shadow" behind the primary registers. There are actually two sets of MR, MF, MX, and MY register files. Only one bank is accessible at a

time. The additional bank of registers can be activated during an interrupt service routine for extremely fast context switching. A new task, such as an interrupt service routine, can be executed without transferring current states to storage.

The selection of the primary or alternate bank of registers is controlled by a bit in the processor mode status register (MSTAT). Toggling this bit switches back and forth between the two register banks.

2.3.2 MAC Operations

This section explains the functions of the MAC, its input formats and its handling of overflow and saturation.

2.3.2.1 Standard Functions

The functions performed by the MAC are:

X*Y	Multiply X and Y operands
MR+X*Y	Multiply X and Y operands and add result to MR register
MR-X*Y	Multiply X and Y operands and subtract result from MR register
0	Clear result (MR) to zero

In performing a multiply/accumulate, the multiplier output P is fed into a 40-bit adder/subtractor which adds or subtracts the new product with the current contents of the MR register to form the final 40-bit result R. The 32-bit P output is format adjusted, that is, sign-extended and shifted one bit to the left before being added to MR. Bit 31 of P lines up with bit 32 of MR (which is bit 0 of MR2) and bit 0 of P lines up with bit 1 of MR (which is bit 1 of MR0). The LSB is zero-filled. The multiplier result format is shown in Figure 2.6.

This is usually a more convenient format because it eliminates the redundant sign bit from the lower 32 bits of the result when multiplying signed numbers. This justification is maintained even if one or both of the inputs are in unsigned format.

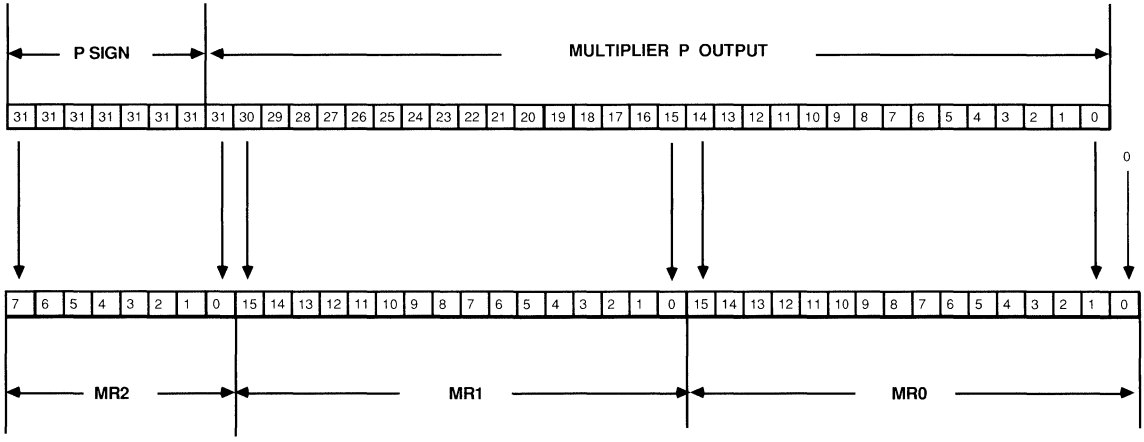


Figure 2.6 Multiplier Result Format

2.3.2.2 Input Formats

To facilitate multiprecision multiplications, the multiplier accepts X and Y inputs represented in any combination of signed twos-complement format and unsigned format.

X input		*Y input*
signed	x	signed
unsigned	x	signed
signed	x	unsigned
unsigned	x	unsigned

The input formats are specified as part of the instruction. These are dynamically selectable each time the multiplier is used.

The (signed x signed) mode is used when multiplying two signed single precision numbers or the two upper portions of two signed multiprecision numbers.

The (unsigned x signed) and (signed x unsigned) modes are used when multiplying the upper portion of a signed multiprecision number with the lower portion of another or when multiplying a signed single precision number by an unsigned single precision number.

The (unsigned x unsigned) mode is used when multiplying unsigned single precision numbers or the non-upper portions of two signed multiprecision numbers.

2.3.2.3 MAC Input/Output Registers

The sources of MAC input and output are:

Source for X input port	*Source for Y input port*	*Destination for R output port*
MX0, MX1	MY0, MY1	MR (MR2, MR1, MR0)
AR	MF	MF
MR0, MR1, MR2		
SR0, SR1		

2.3.2.4 MR Register Operation

As described, and shown on the block diagram, the MR register is divided into three sections: MR0 (bits 0-15), MR1 (bits 16-31), and MR2 (bits 32-39). Each of these registers can be preloaded from the DMD bus and output to the R bus or the DMD bus.

The 8-bit MR2 register is tied to the lower 8 bits of these buses. When MR2 is output onto the DMD bus or the R bus, it is sign extended to form a 16-bit value. MR2 also has an automatic sign extend capability. When MR1 is preloaded from the DMD bus, every bit in MR2 will be set equal to the sign bit (MSB) of MR1, so that MR2 appears as an extension of MR1. To preload the MR2 register with a value other than MR1's sign extension, you must load MR2 after MR1 has been loaded.

2.3.2.5 MAC Overflow and Saturation

The adder/subtractor generates an overflow status signal (MV) which is loaded into the processor arithmetic status (ASTAT) every time a MAC operation is executed. The MV bit is set when the accumulator result, interpreted as a twos-complement number, crosses the 32-bit boundary. That is, MV is set if the accumulator result crosses the MR1/MR2 boundary. Another way of stating this is that the MV bit is set if the upper nine bits of the result register MR are not all ones or all zeros.

The MR register has a saturation capability which sets MR to plus or minus the maximum value if an overflow or underflow has occurred. The saturation operation depends on the overflow status bit (MV) in the processor arithmetic status (ASTAT) and the MSB of the MR2 register. The following table summarizes the MR saturation operation.

MV bit	*MSB of MR2*	*MR content after saturation*
0	0	no change
0	1	no change
1	0	00000000 0111111111111111 1111111111111111 *full-scale positive*
1	1	11111111 1000000000000000 0000000000000000 *full-scale negative*

Table 2.3 MAC Saturation Content

Saturation in the MAC is an instruction rather than a mode as in the ALU. The saturation instruction is intended to be used at the completion of a string of multiply/accumulates so that intermediate overflows do not cause the accumulator to saturate.

Overflowing beyond the MSB of MR2 should never be allowed. The true sign bit of the result is then irretrievably lost and saturation may not produce a correct value. It takes more than 255 overflows (MV type) to reach this state, however.

2.3.2.6 Rounding Mode

The accumulator has the capability for rounding the 40-bit result R at the boundary between bit 15 and bit 16. Rounding can be specified as part of the instruction code. The rounded output is directed to either MR or MF. When rounding is invoked with MF as the output register, register contents in MF represent the rounded 16-bit result. Similarly, when MR is selected as the output, MR1 contains the rounded 16-bit result; the rounding effect in MR1 affects MR2 as well and MR2 and MR1 represent the rounded 24-bit result.

The accumulator uses an unbiased rounding scheme. The conventional method of biased rounding is to add a 1 into bit position 15 of the adder chain. This method causes a slight positive net bias since the midway value is always rounded upward. This problem is eliminated by detecting this midway point and rounding half of the midway values upward and

half of them downward, yielding a zero net bias over a large number of values. When the midway point is detected, bit 16 in the result output is forced to zero. This is also known as round to even.

For example, using x to represent any bit pattern (not all zeros), here are two examples of how this rounding scheme operates.

Example 1	*MR2*	*MR1*	*MR0*
Unrounded value:	xxxxxxxx	xxxxxxxx00100101	1xxxxxxxxxxxxxxx
Bit 15 = 1 Add 1 to bit 15 and carry			1
Rounded value:	xxxxxxxx	xxxxxxxx00100110	0xxxxxxxxxxxxxxx

The first example illustrates the typical rounding operation. The compensation to avoid net bias becomes visible when the lower 15 bits are all zero and bit 15 is one, i.e. the midpoint value. This is shown below.

Example 2	*MR2*	*MR1*	*MR0*
Unrounded value:	xxxxxxxx	xxxxxxxx01100110	1000000000000000
Bit 15 = 1 and bits 0-14 = 0 Add 1 to bit 15 and carry			1
Rounded value:	xxxxxxxx	xxxxxxxx01100111	0000000000000000
Since bit 16 = 1, force it to 0	xxxxxxxx	xxxxxxxx01100110	0000000000000000

In this last case, bit 16 is forced to zero. This algorithm is employed on every rounding operation, but is only evident when the bit patterns shown in the last example are present.

2.4 BARREL SHIFTER

The shifter unit provides a complete set of shifting functions for 16-bit inputs, yielding a 16-bit or 32-bit output. These include arithmetic shift, logical shift, normalization, derivation of exponent and derivation of

common exponent for an entire block of numbers. These basic functions can be combined to efficiently implement any degree of numerical format control, including full floating point representation.

2.4.1 Shifter Block Diagram Discussion

Figure 2.7 (on the following page) shows a block diagram of the shifter section. The shifter section can be divided into the following components: the shifter array, the OR/PASS logic, the exponent detector, and the exponent compare logic.

The shifter array is a 16x32 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, in a single cycle. This gives 49 possible placements within the 32-bit field. The placement of the 16 input bits is determined by a control code (C) and a HI/LO reference signal.

The shifter array and its associated logic are surrounded by a set of registers. The shifter input (SI) register provides input to the shifter array and the exponent detector. The SI register is 16 bits wide and is readable and writable from the DMD bus. The shifter array and the exponent detector can also take any result registers from the R bus as inputs. The shifter result (SR) register is 32 bits wide and is divided into two 16-bit sections, SR0 and SR1. The SR0 and SR1 registers can be preloaded from the DMD bus and output to either the DMD bus or the R bus. The SR register is also fed back to the OR/PASS logic to allow double-precision shift operations.

The SE register ("shifter exponent") drives the shifting operation itself. SE is 8 bits wide and holds the exponent during the normalize and denormalize operations. The SE register is loadable and readable from the lower 8 bits of the DMD bus. It is a twos-complement, 8.0 value.

The SB register ("shifter block") is important in block floating-point operations where it holds the block exponent shift value, that is, the value by which the block values must be shifted to conform to the actual exponent. SB is 5 bits wide and holds the most recent block exponent value. The SB register is loadable and readable from the lower 5 bits of the DMD bus. It is a twos-complement, 5.0 value.

Whenever the SE or SB registers are output onto the DMD bus, they are sign-extended to form a 16-bit value.

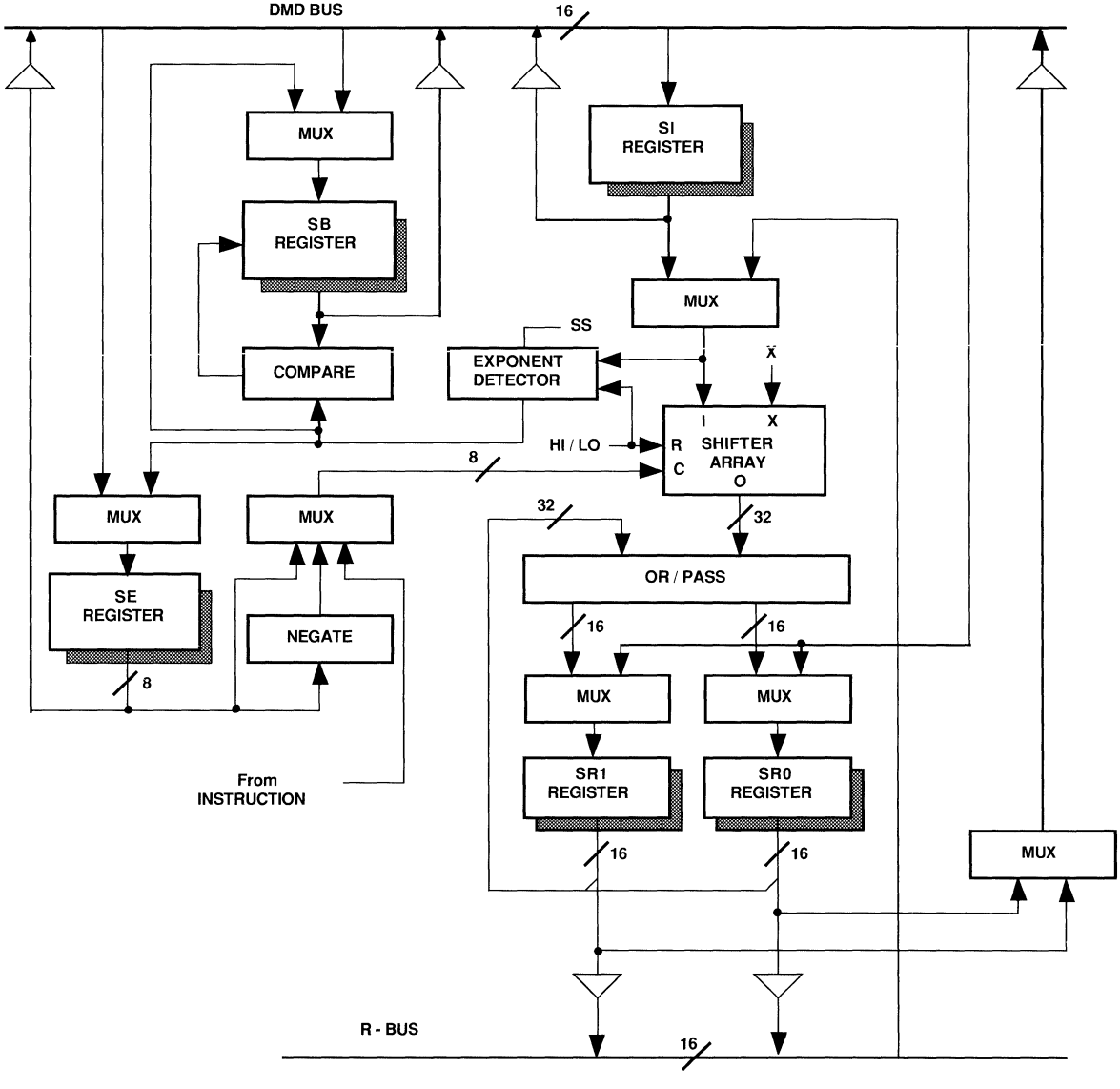


Figure 2.7 Shifter Block Diagram

The SI, SE and SR registers can be read and written in the same cycle. Registers are read at the beginning of the cycle and written at the end of the cycle. All register reads, therefore, read values loaded at the end of a previous cycle. A new value written to a register cannot be read out until a subsequent cycle. This allows an input register to provide an operand to the Shifter at the beginning of the cycle and be updated with the next operand from memory at the end of the same cycle. It also allows a result register to be stored in memory and updated with a new result in the same cycle. See the discussion of "Multifunction Instructions" in the chapter "Instruction Set Overview" for an illustration of this same-cycle read and write.

The shifter section contains a duplicate bank of registers, shown in Figure 2.7 as a "shadow" behind the primary registers. There are actually two sets of SE, SB, SI, SR1, and SR0 registers. Only one bank is accessible at a time. The additional bank of registers can be activated during an interrupt service routine for extremely fast context switching. A new task, such as an interrupt service routine, can be executed without transferring current states to storage.

The selection of the primary or alternate bank of registers is controlled by a bit in the processor mode status register (MSTAT). Toggling this bit switches back and forth between the two register banks.

The shifting of the input is determined by a control code (C) and a HI/LO reference signal. The control code is an 8-bit signed value which indicates the direction and number of places the input is to be shifted. Positive codes indicate a left shift (upshift) and negative codes indicate a right shift (downshift). The control code can come from three sources: the content of the shifter exponent (SE) register, the negated content of the SE register or an immediate value from the instruction.

The HI/LO signal determines the reference point for the shifting. In the HI state, all shifts are referenced to SR1 (the upper half of the output field), and in the LO state, all shifts are referenced to SR0 (the lower half of the output field). The HI/LO reference feature is useful when shifting 32-bit values since it allows both halves of the number to be shifted with the same control code. HI/LO reference signal is dynamically selectable each time the shifter is used.

The shifter fills any bits to the right of the input value in the output field with zeros, and bits to the left are filled with the extension bit (X). The extension bit can be fed by three possible sources depending on the instruction being performed. The three sources are the MSB of the input, the AC bit from the arithmetic status register (ASTAT) or a zero. The AC bit is used when constructing 32-bit results from successive multiword operations in the ALU.

Table 2.4 gives a listing of shifter array output as a function of the control code and HI/LO signal.

The OR/PASS logic allows the shifted sections of a multiprecision number to be combined into a single quantity. When PASS is selected, the shifter array output is passed through and loaded into the shifter result (SR) register unmodified. When OR is selected, the shifter array is bitwise ORed with the current contents of the SR register before being loaded there.

The exponent detector derives an exponent for the shifter input value. The exponent detector operates in one of three ways which determine how the input value is interpreted. In the HI state, the input is interpreted as a single precision number or the upper half of a double precision number. The exponent detector determines the number of leading sign bits and produces a code which indicates how many places the input must be up-shifted to eliminate all but one of the sign bits. The code is negative so that it can become the effective exponent for the mantissa formed by removing the redundant sign bits.

In the HI-extend state (HIX), the input is interpreted as the result of an add or subtract performed in the ALU section which may have overflowed. Therefore the exponent detector takes the arithmetic overflow (AV) status into consideration. If AV is set, then a +1 exponent is output to indicate an extra bit (the ALU Carry bit); if AV is not set, then HI-extend functions exactly like the HI state. When performing a derive exponent function in HI or HI-extend modes, the exponent detector also outputs a shifter sign (SS) bit which is loaded into the arithmetic status register (ASTAT). The sign bit is the same as the MSB of the shifter input except when AV is set; when AV is set in HI-extend state, the MSB is inverted to restore the sign bit of the overflowed value.

ABCDEFGHIJKLMNPR represents the 16-bit input pattern
X stands for the extension bit

Control Code		Shifter Array Output			
HI reference	LO Reference				
+16 to +127	+32 to +127	00000000	00000000	00000000	00000000
+15	+31	R0000000	00000000	00000000	00000000
+14	+30	PR000000	00000000	00000000	00000000
+13	+29	NPR00000	00000000	00000000	00000000
+12	+28	MNPR0000	00000000	00000000	00000000
+11	+27	LMNPR000	00000000	00000000	00000000
+10	+26	KLMNPR00	00000000	00000000	00000000
+9	+25	JKLMNPR0	00000000	00000000	00000000
+8	+24	IJKLMNPR	00000000	00000000	00000000
+7	+23	HIJKLMNP	R0000000	00000000	00000000
+6	+22	GHIJKLMN	PR000000	00000000	00000000
+5	+21	FGHIJKLM	NPR00000	00000000	00000000
+4	+20	EFGHIJKL	MNPR0000	00000000	00000000
+3	+19	DEFGHIJK	LMNPR000	00000000	00000000
+2	+18	CDEFGHIJ	KLMNPR00	00000000	00000000
+1	+17	BCDEFGHI	JKLMNPR0	00000000	00000000
0	+16	ABCDEFGH	IJKLMNPR	00000000	00000000
-1	+15	XABCDEFG	HIJKLMNP	R0000000	00000000
-2	+14	XXABCDEF	GHIJKLMN	PR000000	00000000
-3	+13	XXXABCDE	FGHIJKLM	NPR00000	00000000
-4	+12	XXXXABCD	EFGHIJKL	MNPR0000	00000000
-5	+11	XXXXXABC	DEFGHIJK	LMNPR000	00000000
-6	+10	XXXXXXAB	CDEFGHIJ	KLMNPR00	00000000
-7	+9	XXXXXXXA	BCDEFGHI	JKLMNPR0	00000000
-8	+8	XXXXXXXX	ABCDEFGH	IJKLMNPR	00000000
-9	+7	XXXXXXXX	XABCDEFG	HIJKLMNP	R0000000
-10	+6	XXXXXXXX	XXABCDEF	GHIJKLMN	PR000000
-11	+5	XXXXXXXX	XXXABCDE	FGHIJKLM	NPR00000
-12	+4	XXXXXXXX	XXXXABCD	EFGHIJKL	MNPR0000
-13	+3	XXXXXXXX	XXXXXABC	DEFGHIJK	LMNPR000
-14	+2	XXXXXXXX	XXXXXXAB	CDEFGHIJ	KLMNPR00
-15	+1	XXXXXXXX	XXXXXXXA	BCDEFGHI	JKLMNPR0
-16	0	XXXXXXXX	XXXXXXXX	ABCDEFGH	IJKLMNPR
-17	-1	XXXXXXXX	XXXXXXXX	XABCDEFG	HIJKLMNP
-18	-2	XXXXXXXX	XXXXXXXX	XXABCDEF	GHIJKLMN
-19	-3	XXXXXXXX	XXXXXXXX	XXXABCDE	FGHIJKLM
-20	-4	XXXXXXXX	XXXXXXXX	XXXXABCD	EFGHIJKL
-21	-5	XXXXXXXX	XXXXXXXX	XXXXXABC	DEFGHIJK
-22	-6	XXXXXXXX	XXXXXXXX	XXXXXXAB	CDEFGHIJ
-23	-7	XXXXXXXX	XXXXXXXX	XXXXXXXA	BCDEFGHI
-24	-8	XXXXXXXX	XXXXXXXX	XXXXXXXX	ABCDEFGH
-25	-9	XXXXXXXX	XXXXXXXX	XXXXXXXX	XABCDEFG
-26	-10	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXABCDEF
-27	-11	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXABCDE
-28	-12	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXABCD
-29	-13	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXABC
-30	-14	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXAB
-31	-15	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXA
-32 to -128	-16 to -128	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

Table 2.4 Shifter Array Characteristic

In the LO state, the input is interpreted as the lower half of a double precision number. In the LO state, the exponent detector interprets the SS bit in the arithmetic status register (ASTAT) as the sign bit of the number. The SE register is loaded with the output of the exponent detector only if SE contains –15. This occurs only when the upper half–which must be processed first–contained all sign bits. The exponent detector output is also offset by –16 to account for the fact that the input is actually the lower half of a 32-bit value. Table 2.5 gives the exponent detector characteristics for all three modes.

The exponent compare logic is used to find the largest exponent value in an array of shifter input values. The exponent compare logic in conjunction with the exponent detector derives a block exponent. The comparator compares the exponent value derived by the exponent detector with the value stored in the shifter block exponent (SB) register and updates the SB register only when the derived exponent value is larger than the value in SB register. See the examples below.

2.4.2 Shifter Operations

The shifter performs the following functions (instruction mnemonics shown in parenthesis):

- Arithmetic Shift (ASHIFT)
- Logical Shift (LSHIFT)
- Normalize (NORM)
- Derive Exponent (EXP)
- Block Exponent Adjust (EXPADJ)

These basic shifter instructions can be used in a variety of ways, depending on the underlying arithmetic requirements. The following sections present single and multiple precision examples for these functions:

- Derivation of a Block Exponent
- Immediate Shifts
- Denormalization
- Normalization

The shift functions (arithmetic shift, logical shift, and normalize) can be optionally specified with PASS/OR and HI/LO modes so as to facilitate multiprecision operations. PASS passes the value through to SR directly. OR logically ORs the shift result with the current contents of SR. OR is

S = Sign bit
N = Non-sign bit
D = Don't care bit

HI Mode

Shifter Array Input	Output
SNDDDDDD DDDDDDDD	0
SSNDDDDD DDDDDDDD	-1
SSSNDDDD DDDDDDDD	-2
SSSSNDDD DDDDDDDD	-3
SSSSSNDD DDDDDDDD	-4
SSSSSSND DDDDDDDD	-5
SSSSSSSN DDDDDDDD	-6
SSSSSSSS NDDDDDDD	-7
SSSSSSSS SNDDDDDD	-8
SSSSSSSS SSNDDDDD	-9
SSSSSSSS SSSNDDDD	-10
SSSSSSSS SSSSNDDD	-11
SSSSSSSS SSSSSNDD	-12
SSSSSSSS SSSSSSND	-13
SSSSSSSS SSSSSSSN	-14
SSSSSSSS SSSSSSSS	-15

HIX Mode

AV	Shifter Array Input	Output
1	DDDDDDDD DDDDDDDD	+1
0	SNDDDDDD DDDDDDDD	0
0	SSNDDDDD DDDDDDDD	-1
0	SSSNDDDD DDDDDDDD	-2
0	SSSSNDDD DDDDDDDD	-3
0	SSSSSNDD DDDDDDDD	-4
0	SSSSSSND DDDDDDDD	-5
0	SSSSSSSN DDDDDDDD	-6
0	SSSSSSSS NDDDDDDD	-7
0	SSSSSSSS SNDDDDDD	-8
0	SSSSSSSS SSNDDDDD	-9
0	SSSSSSSS SSSNDDDD	-10
0	SSSSSSSS SSSSNDDD	-11
0	SSSSSSSS SSSSSNDD	-12
0	SSSSSSSS SSSSSSND	-13
0	SSSSSSSS SSSSSSSN	-14
0	SSSSSSSS SSSSSSSS	-15

LO Mode

SS	Shifter Array Input	Output
S	NDDDDDDD DDDDDDDD	-15
S	SNDDDDDD DDDDDDDD	-16
S	SSNDDDDD DDDDDDDD	-17
S	SSSNDDDD DDDDDDDD	-18
S	SSSSNDDD DDDDDDDD	-19
S	SSSSSNDD DDDDDDDD	-20
S	SSSSSSND DDDDDDDD	-21
S	SSSSSSSN DDDDDDDD	-22
S	SSSSSSSS NDDDDDDD	-23
S	SSSSSSSS SNDDDDDD	-24
S	SSSSSSSS SSNDDDDD	-25
S	SSSSSSSS SSSNDDDD	-26
S	SSSSSSSS SSSSNDDD	-27
S	SSSSSSSS SSSSSNDD	-28
S	SSSSSSSS SSSSSSND	-29
S	SSSSSSSS SSSSSSSN	-30
S	SSSSSSSS SSSSSSSS	-31

Table 2.5
Exponent Detector Characteristic

used to join two 16-bit quantities into a 32-bit value in SR. The HI and LO modifiers reference the shift to the upper or lower half of the 32-bit SR register. These shift functions take inputs from either the SI register or any other result register and load the 32-bit shifted result into the SR register.

2.4.2.1 Shifter Input/Output Registers

The sources of shifter input and output are:

Source for Shifter input	*Destination for Shifter output*
SI	SR (SR0, SR1)
AR	
MR0, MR1, MR2	
SR0, SR1	

2.4.2.2 Derive Block Exponent

This function detects the exponent of the number largest in magnitude in an array of numbers. The EXPADJ instruction performs this function. The sequence of steps for a typical example is shown below.

A. Load SB with –16

The SB register is used to contain the exponent for the entire block. The possible values at the conclusion of a series of EXPADJ operations range from -15 to 0. The exponent compare logic updates the SB register if the new value is greater than the current value. Loading the register with –16 initializes it to a value certain to be less than any actual exponents detected.

B. Process the first array element:

Array(1) = `11110101 10110001`

Exponent = `–3`

– 3 > SB (–16)

SB gets `–3`

C. Process next array element:

Array(2)= 00000001 01110110

Exponent = −6

−6 < −3

SB remains −3

D. Continue processing array elements.

When and if an array element is found whose exponent is greater than SB, that value is loaded into SB. When all array elements have been processed, the SB register contains the exponent of the largest number in the entire block. No normalization is performed. EXPADJ is purely an inspection operation. The value in SB could be transferred to SE and used to normalize the block to maximum precision on the next pass through the Shifter. Or it could be simply associated with that data for subsequent interpretation.

2.4.2.3 Immediate Shifts

An immediate shift simply shifts the input bit pattern to the right (downshift) or left (upshift) by a given number of bits. Immediate shift instructions use the data value in the instruction itself to control the amount and direction of the shifting operation. (See the chapter "Instruction Set Overview" for an example of this instruction.) The data value controlling the shift is an 8-bit signed number. The SE register is not used or changed by an immediate shift.

The following example shows the input value downshifted relative to the upper half of SR (SR1). This is the (HI) version of the shift.

Input 10110110 10100011

Shift value −5

SR XXXXX**101 10110101 00011**XXX XXXXXXXX

Here is the same input value shifted in the other direction, referenced to the lower half (LO) of SR.

Input `10110110 10100011`

Shift value `+5`

SR `XXXXXXXX XXX`**`10110 11010100 011`**`XXXXX`

In addition to the direction of the shifting operation, the shift may be either arithmetic (ASHIFT) or logical (LSHIFT). For example, the following shows a logical shift, relative to the upper half of SR (HI).

Input `10110110 10100011`

Shift value `-5`

SR `00000`**`101 10110101 00011`**`000 00000000`

This example shows an arithmetic shift of the same input and shift code.

Input `10110110 10100011`

Shift value `-5`

SR `11111`**`101 10110101 00011`**`000 00000000`

2.4.2.4 Denormalize

Denormalizing refers to shifting a number according to a predefined exponent. The operation is effectively a block floating point to fixed point conversion.

Denormalizing requires a sequence of operations. First, the SE register must contain the exponent value. This value may be explicitly loaded or may be the result of some previous operation. Next the shift itself is performed, taking its shift value from the SE register, not from an immediate data value.

There are two examples of denormalizing a double-precision number below. The first shows a denormalization in which the upper half of the number is shifted first, followed by the lower half. Since computations may produce output in either order, the second example shows the same operation in the other order, i.e. lower half first.

Always select the arithmetic shift for the higher half (HI)of the twos-complement input (or logical for unsigned-magnitude). Likewise, the first half processed uses the PASS modifier.

Modifiers = HI, PASS Shift operation = Arithmetic, SE = –3

First Input `10110110 10100011` (upper half of desired result)

SR `111`**`10110 11010100 011`**`00000 00000000`

Now the lower half is processed. Always select a logical shift for the lower half of the input. Likewise, the second half processed must use the OR modifier to avoid overwriting the previous half of the output value.

Modifiers = LO, OR Shift operation = Logical, SE = –3

Second Input `01110110 01011101` (lower half of desired result)

SR `11110110 11010100 011`**`01110 11001011`**

Here is the same input processed in the reverse order. The higher half is always arithmetically shifted and the lower half is logically shifted. The first input is PASSed through to SR, but the second half is ORed to create one double-precision value in SR.

Modifiers = LO, PASS Shift operation = Logical, SE = –3

First Input `01110110 01011101` (lower half of desired result)

SR `00000000 00000000 000`**`01110 11001011`**

Modifiers = HI, OR Shift operation = Arithmetic, SE = –3

Second Input `10110110 10100011` (upper half of desired result)

SR `111`**`10110 11010100 011`**`01110 11001011`

2.4.2.5 Normalize

Normalizing a number is the process of shifting a twos-complement number within a field so that the rightmost sign bit lines up with the MSB position of the field and recording how many places the number was shifted. The operation can be thought of as a fixed to floating point conversion, generating an exponent and a mantissa.

Normalizing is a two stage process. The first stage derives the exponent. The second stage does the actual shifting. The first stage uses the EXP instruction which detects the exponent value and loads in into the SE register. This instruction (EXP) recognizes a (HI) and (LO) modifier. The second stage uses the NORM instruction. NORM recognizes (HI) and (LO) and the PASS and OR modifiers as well. NORM uses the negated value of the SE register as its shift control code. The negated value is used so that the shift is made in the correct direction, producing a mantissa corresponding to the exponent value in SE.

Here is a normalization example for a single precision input.

Detect Exponent Modifier = HI

Input `11110110 11010100`

SE set to -3

Normalize, with modifier = HI Shift driven by value in SE

Input `11110110 11010100`

SR **`10110110 10100`**`000 00000000 00000000`

For a single precision input, the normalize operation can use either the (HI) or (LO) modifier, depending on whether you want the result in SR1 or SR0, respectively.

Double precision values follow the same general scheme. The first stage detects the exponent and the second stage normalizes the two halves of the input. For double precision, however, there are two operations in each stage.

For the first stage, the upper half of the input must be operated on first. This first exponent derivation loads the exponent value into SE. The second exponent derivation, operating on the lower half of the number will not alter the SE register unless SE = –15. This happens only when the first half contained all sign bits. In this case, the second operation will load a value into SE. (See Table 2.5) This value is used to control both parts of the normalization that follows.

For the second stage, now that SE contains the correct exponent value, the order of operations is immaterial. The first half (whether HI or LO) is normalized with the PASS modifier and the second half with the OR modifier to create one double precision value in SR. The (HI) and (LO) modifiers identify which half is being processed.

Here is a complete example of a typical double precision normalization.

1. *Detect Exponent, Modifier = HI*

   First Input `11110110 11010100` (Must be upper half)

   SE set to `-3`

2. *Detect Exponent, Modifier = LO*

   Second Input `01101110 11001011`

   SE unchanged, still `-3`

3. *Normalize, Modifiers = HI, PASS, SE = –3 (negated)*

   First Input `11110110 11010100`

   SR **`10110110 10100`**`000 00000000 00000000`

4. *Normalize , Modifiers = LO, OR, SE = –3 (negated)*

   Second Input `01101110 11001011`

   SR `10110110 10100`**`011 01110110 01011`**`000`

If the upper half of the input contains all sign bits, the SE register value is determined by the second derive exponent operation as shown below.

1. *Detect Exponent, Modifier = HI*

   First Input `11111111 11111111` (Must be upper half)

   SE set to `-15`

2. *Detect Exponent, Modifier = LO*

   Second Input `11110110 11010100`

   SE now set to `-19`

3. *Normalize, Modifiers = HI, PASS, SE = –19 (negated)*

   First Input `11111111 11111111`

   SR `00000000 00000000 00000000 00000000`

All values of SE less than -15 (resulting in a shift of +16 or more) upshift the input completely off scale.

4. *Normalize , Modifiers = LO, OR, SE = –19 (negated)*

   Second Input `11110110 11010100`

   SR **`10110110 10100`**`000 00000000 00000000`

There is one additional normalization situation, requiring the HI-extended (HIX) state. This is specifically when normalizing ALU results (AR) that may have overflowed. This operation reads the arithmetic status word (ASTAT) overflow bit (AV) and the carry bit (AC) in conjunction with the value in AR. AV will be set (1) if an overflow has occurred. AC will retain the true sign of the twos-complement value.

For example, given these conditions:

AR = `11111010 00110010`

AV = `1`, indicating overflow

AC = `0`, the true sign bit of this value

1. *Detect Exponent, Modifier = HIX*

   SE gets set to `+1`

2. *Normalize, Modifier = HI, SE = 1*

   AR = `11111010 00110010`

   SR = **`0`**`1111101 00011001`

The AC bit is supplied as the sign bit, shown in bold above.

The HIX operation executes properly regardless of whether there has actually been an overflow. Consider this example.

AR = 11100011 01011011

AV = 0, indicating no overflow

AC = 0, not meaningful if AV = 0

1. *Detect Exponent, Modifier = HIX*

   SE set to −2

2. *Normalize, Modifier = HI, SE = −2*

   AR = 11100011 01011011

   SR = **10001101 01101**000 00000000 00000000

The AC bit is not used as the sign bit. A brief examination of Table 2.4 shows that the HIX mode is identical to the HI mode when AV is not set.

Data Moves 3

3.1 INTRODUCTION

This chapter describes sections of the ADSP-2100 that control the movement of data to and from the processor. These are the Data Address Generators (DAGs) and the unit for exchanging data between the Program Memory Data bus and the Data Memory Data Bus, the PMD-DMD Bus Exchange Unit.

3.2 DATA ADDRESS GENERATORS (DAGS)

The ADSP-2100 contains two independent data address generators so that both program and data memories can be accessed simultaneously. The DAGs provide indirect addressing capabilities. Both perform automatic address modification. For circular buffers, the DAGs can perform modulo address modification. The two DAGs differ: DAG1 only generates data memory addresses, but provides an optional bit-reversal capability, DAG2 can generate both data memory and program memory addresses, but has no bit-reversal capability.

While the following discussion explains the internal workings of the DAGs bear in mind that the ADSP-2100 instruction set and Cross-Software System provide a direct method for declaring buffers as circular or linear and managing the placement of the buffer in memory. Only the initializing of DAG registers needs to be explicitly programmed. See the discussion of data structures in Chapter 6, "Instruction Set Overview."

3.2.1 DAG Block Diagram Discussion

Figure 3.1 (on the following page) shows a block diagram of a single data address generator. There are three register files: the modify (M) register file, the indirect (I) register file, and the length (L) register file. Each of the register files contains four 14-bit registers which can be read from and written to via the DMD bus.

The I registers (I0-3 in DAG1, I4-7 in DAG2) contain the actual addresses used to access memory. When data is accessed in indirect mode, the address stored in the selected I register is driven out onto the appropriate address bus and becomes the memory address. With DAG1, the output

3 Data Moves

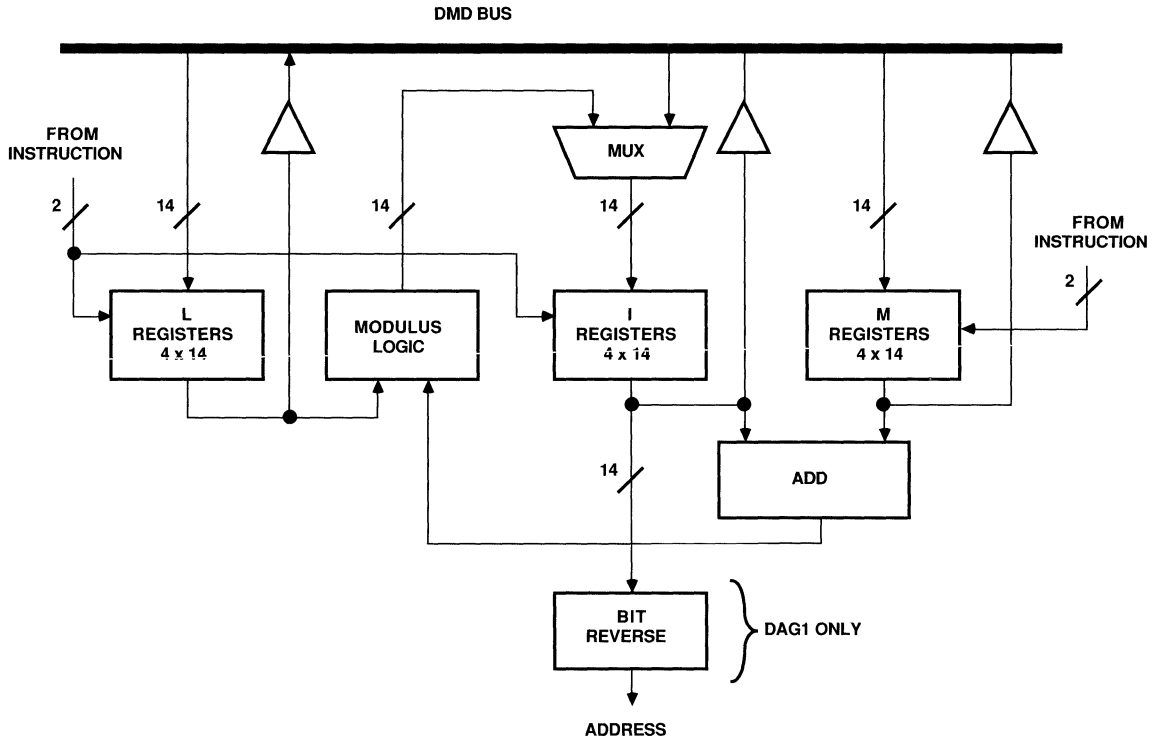


Figure 3.1 Data Address Generator Block Diagram

address can be bit-reversed before being driven onto the address bus by setting the appropriate mode bit in the mode status register (MSTAT) as discussed below. Bit-reversal facilitates radix-2 FFT addressing.

The data address generator employs a post-modify scheme; after an indirect data access, the specified M register (M0-3 in DAG1, M4-7 in DAG2) is automatically added to the specified I register, modifying it. The choice of the I and M registers are independent within each DAG. In other words, any register in the I0-3 set may be modified by any register in the M0-3 set in any combination, but not by those in DAG2 (M4-7). The modification values stored in M registers are signed numbers so that the next address can be either higher or lower.

The address generators support both linear addressing and circular addressing. The value of the L register determines which addressing scheme is used. L registers and I registers are paired and the selection of the L register (L0-3 in DAG1, L4-7 in DAG2) is determined by the I register used. Each time an I register is selected, the corresponding L register provides the modulus logic with the length information to wrap the address around if necessary. For linear buffer addressing, the modulus logic is disabled by setting the corresponding L register to zero. In this case, the modified I register value is simply the sum of the M register content and the I register content.

For circular buffer addressing, the L register is initialized with the length of the buffer. If the sum of the M register content and the I register content would cross the buffer boundary, the modified I register value is calculated by the modulus logic using the L register value (see "Modulo Addressing" below).

All data address generator registers (I, M, and L registers) are loadable and readable from the lower 14 bits of the DMD bus. Since I and L register contents are considered to be unsigned, the upper 2 bits of the DMD bus are padded with zeros when reading them. M register contents are signed; when reading an M register, the upper 2 bits of the DMD bus are sign-extended.

3.2.2 Modulo Addressing

The modulus logic implements automatic pointer wraparound for accessing circular buffers. To calculate the next address, the modulus logic uses the following information.

- The current location; found in the I register (unsigned)
- The modify value; found in the M register (signed)
- The buffer length; found in the L register (unsigned)
- The buffer base address (implicitly defined by I and L registers)

To avoid having another set of "base address" registers, the processor imposes the following restriction on the placement of circular buffers.

- *If the buffer length requires N bits to represent in binary, then the lower N bits of the buffer base address must be zero.*

With the above rule the base address can be obtained by masking out the lower N bits of the I register content in binary. Note that the ADSP-2100 Linker automatically places circular buffers at a proper address.

There is one additional restriction imposed by the modulus logic.

- *The modify value must be less than or equal to the buffer length.*

Therefore, in one operation, the modified address cannot wrap around the buffer more than once.

The modified address is calculated with the formula below.

Modified address = (I + M - B) Modulo (L) + B

Where:
I = current address
M = modify value (signed)
B = base address
L = buffer length
M ≤ L

For illustration, consider the following examples of base address calculation.

3.2.2.1. Circular Buffer Base Address Example 1

You want a circular buffer of length 8, which requires four bits to represent in binary. Valid base addresses, therefore, are multiples of 16, requiring the lower four bits of the base address to be zero: H#0000, H#0010, H#0020, H#0030 and so on (hexadecimal notation).

3.2.2.2. Circular Buffer Base Address Example 2

You want a circular buffer of length 7, which requires three bits to represent in binary. Valid base addresses, therefore, are multiples of 8, requiring the lower three bits of the base address to be zero: H#0000, H#0008, H#0010, H#0018, H#0020 and so on (hexadecimal notation).

Here are two examples of circular buffer addressing operation.

3.2.2.3. Circular Buffer Operation Example 1

Suppose that I0 = 5, M0 = 1 and L0 = 3. A length of three takes two bits to represent. By zeroing the lower two bits of the I0 register the processor determines that the base address is at 4. The next address is calculated by adding M0 to I0, resulting in an address of 6. Successive data memory

addresses using I0 for indirect addressing produce the sequence: 6, 4, 5, 6, 4, 5.... For M0 = –1 (H#3FFF), I0 would produce the sequence: 4, 6, 5, 4, 6, 5, 4....

3.2.2.4. Circular Buffer Operation Example 2

Assume that I0 = 9, M0 = –2 and L0 = 5. This example highlights the fact that the address sequence does not have to result in a "direct" hit of the buffer boundary. The 5 word buffer resides at locations 8 through 12 inclusive. The successive data memory addresses using I0 for indirect addressing cycle through the sequence: 9, 12, 10, 8, 11, 9....

3.2.3 Bit-Reverse Addressing

The bit-reverse logic is primarily intended for use in FFT computations where inputs are supplied or the outputs generated in bit-reversed order. Bit-reversing is available only on addresses generated by DAG1. The pivot point for the reversal is the midpoint of the 14-bit address, between bits 6 and 7. This is illustrated in the following chart.

	Individual DMA lines (DMA_N)													
Normal Order	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Bit-reversed	00	01	02	03	04	05	06	07	08	09	10	11	12	13

Bit-reversed addressing is a mode, enabled and disabled by setting a mode bit in the mode status register (MSTAT). When enabled, all addresses generated using indirect registers I0-3 are bit-reversed upon output. (The modified valued stored back after post-update remains in normal order.) This mode continues until the status bit is reset.

It is possible to bit-reverse address values less than 14 bits. You must determine the first address and also initialize the M register to be used with a value calculated to modify the I register bit-reversed output to the desired range. This value is:

$$2^{(14-N)}$$

Where *N* is the number of bits you wish to output reversed. For complete information get the application note *Variable Width Bit-Reversing on the*

ADSP-1410 Address Generator; it has a general discussion of this procedure. The *ADSP-2100 Applications Handbook, Volume 1* also has a complete example of this in the chapter on Fast Fourier Transforms.

3.3 PMD-DMD BUS EXCHANGE

This unit couples the program memory data bus and the data memory data bus, allowing them to transfer data in both directions. Since the program memory data (PMD) bus is 24 bits wide, while the data memory data (DMD) bus is 16 bits wide, only the upper 16 bits of PMD can be directly transferred. An internal register (PX) is always loaded with (or supplies) the additional 8 bits. This register can be directly loaded or read when the full 24 bits are required.

Note that when reading data from program memory and data memory simultaneously, there is a dedicated path from the upper 16 bits of the PMD bus to the Y registers of the computational units. This read-only path does not use the bus exchange circuit; it is the path shown on the individual computational unit block diagrams.

3.3.1 PMD-DMD Block Diagram Discussion

Figure 3.2 shows a block diagram of this circuit. There are two types of connections provided in this section.

The first type of connection is a one-way path from each bus to the other. This is implemented with two tristate buffers connecting the DMD bus with the upper 16 bits of the PMD bus. One of these two buffers is normally used when data is exchanged between the program memory and one of the registers connected to the DMD bus. This is the path used to write data to program memory; it is not shown in the individual computational unit block diagrams.

The second connection is through the PX register. The PX register is 8-bits wide and can be loaded from either the lower 8 bits of the DMD bus or the lower 8 bits of the PMD bus. Its contents can also be read to the lower 8 bits of either bus.

PX register access follows the principles described below.

From the PMD bus, the PX register is:

1. Loaded automatically whenever data (not an instruction) is read from program memory to any register.

2. Read out automatically as the lower 8 bits when data is written to program memory.

From the DMD bus, the PX register may be:

1. Loaded with a data move instruction, explicitly specifying the PX register as the destination. The lower 8 bits of the data value are used and the upper 8 are discarded.

2. Read with a data move instruction, explicitly specifying the PX register as a source. The upper 8 bits of the value read from the register are all zeroes.

Whenever any register is written out to program memory, the source register supplies the upper 16 bits. The contents of the PX register are automatically added as the lower 8 bits. If these lower 8 bits of data to be transferred to program memory (through the PMD bus) are important, the PX register should be loaded from DMD bus before the program memory write operation.

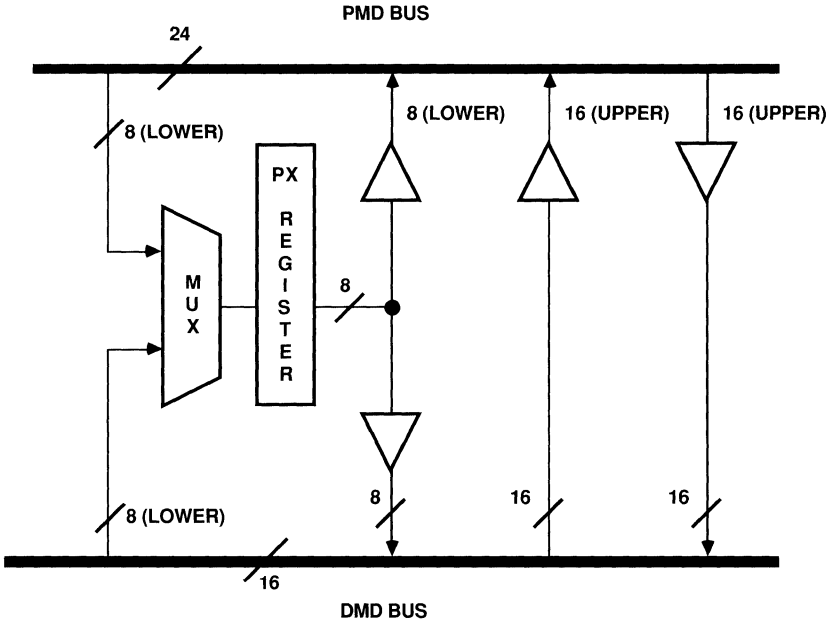


Figure 3.2 PMD–DMD Bus Exchange

Program Control 4

4.1 INTRODUCTION

This chapter describes the sections of the ADSP-2100 that control and influence the flow of your program's execution: the program sequencer, its associated status and interrupt logic and the cache memory.

4.2 PROGRAM SEQUENCER & STATUS

The program sequencer generates a stream of instruction addresses, providing flexible control of program flow. It provides for zero-overhead looping, single-cycle branching (both conditional and unconditional) and sophisticated interrupt processing. Figure 4.1, on the following page, shows a block diagram for the program sequencer and status sections of the ADSP-2100. The sections immediately below discuss individual blocks within the sequencer. The section "Sequencer Operations Illustrated" shows how the individual blocks work together to implement program flow control.

It is useful to be aware that the ADSP-2100 instruction set includes the following instructions:

- JUMP
- CALL
- RETURN FROM SUBROUTINE (RTS)
- RETURN FROM INTERRUPT (RTI)
- DO UNTIL

4.2.1 Next Address Select Logic

The sequencing logic controls the flow of ADSP-2100 program execution by outputting a program memory address onto the PMA bus from one of the following four possible sources.

- PC incrementer
- PC stack
- Instruction register
- Interrupt controller

4 Program Control

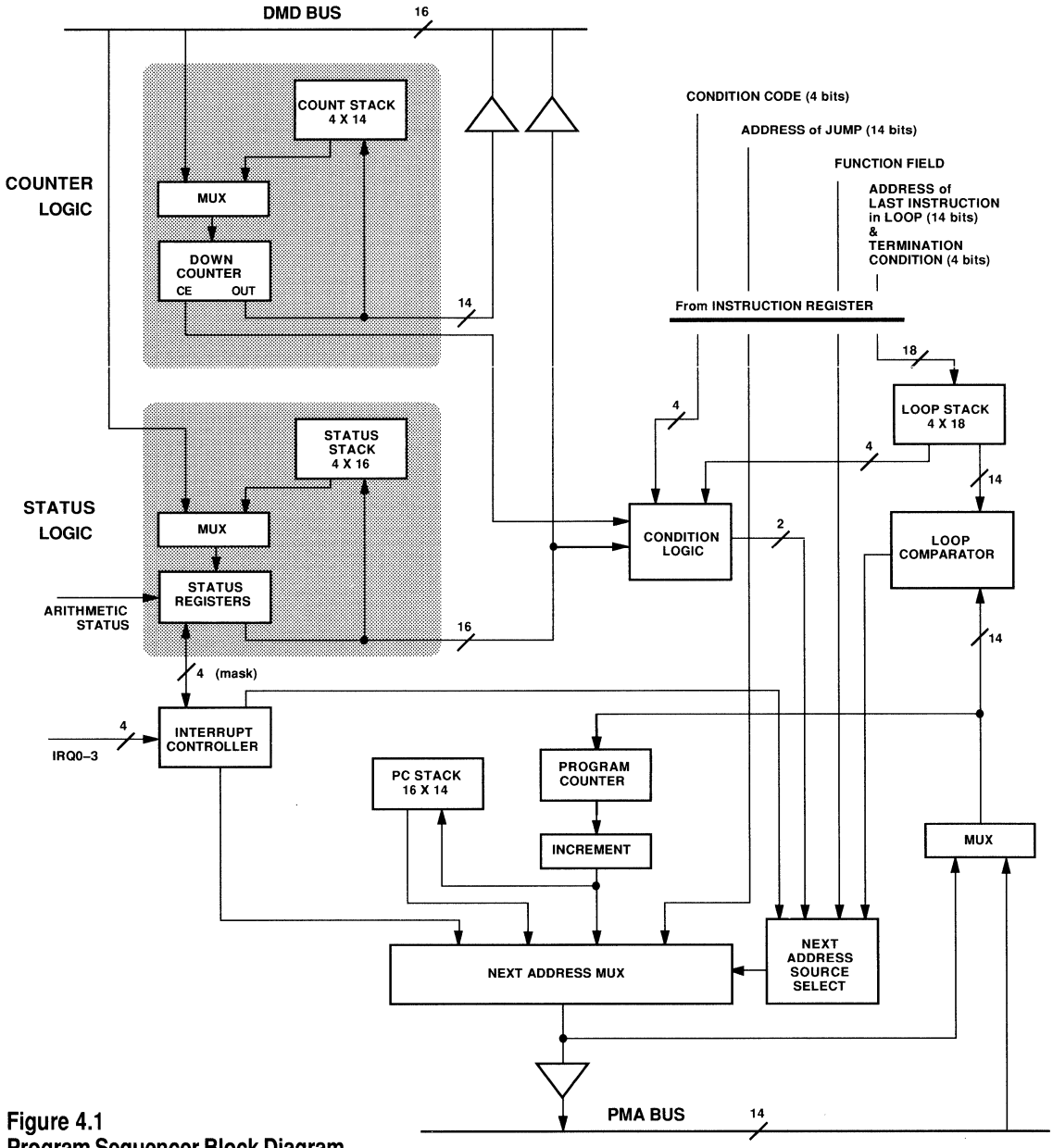


Figure 4.1
Program Sequencer Block Diagram

The next address source selector in the diagram controls which of these four sources are output from the next address multiplexer, based on outputs from the instruction register, condition logic, loop comparator, and interrupt controller. A fifth possibility for the next program memory address, although not part of the program sequencer, is DAG2 when a register indirect jump is executed.

The PC incrementer is selected as the source of the next program memory address if program flow is sequential. This is also the case when a conditional jump, return, or trap is not taken, and when a DO UNTIL loop terminates (see below for a description of the DO UNTIL construct and associated looping hardware).

The PC stack is used as the source for the next program memory address when a return from subroutine or return from interrupt is executed. The top stack value is also used as the next program memory address when returning to the top of a DO UNTIL loop.

The instruction register is selected by the next address multiplexer when a direct jump is taken. The jump address field of the instruction word itself specifies the jump address.

The interrupt controller provides the next program memory address when processing an external interrupt request. Upon recognizing an interrupt, the processor jumps to the interrupt vector location (at program memory address 0000-0003) corresponding to the active interrupt request line IRQ0-IRQ3. Control is then transferred to the interrupt service routine by means of a jump instruction.

DAG2 sources the next program memory address when executing a register indirect jump. In this case, since DAG2 is not an input to the next address multiplexer, the program counter must be loaded from the PMA bus. Note that DAG2 can also address data values in program memory via the PMA bus.

4.2.2 Program Counter and Stack

The program counter (PC) is a 14-bit register which always contains the address of the currently executing instruction. The output of the PC is fed into a 14-bit incrementer which adds 1 to the current PC value. The output of the incrementer can be selected by the next address multiplexer to fetch the next contiguous instruction. Associated with the PC is a 14-bit by 16-word PC stack that is pushed with the output of the incrementer when a CALL instruction is executed. The PC stack is also pushed when an

interrupt is processed. For interrupts, however, the incrementer is disabled so that the current PC value (instead of PC+1) is pushed. This allows the current instruction, which is aborted, to be refetched upon returning from the interrupt service routine. The pushing and popping of the PC stack occurs automatically in all of these cases. The stack can also be manually popped.

The output of the next address multiplexer is fed back to the PC, which normally reloads it at the end of each processor cycle. In the case of a register indirect jump, however, DAG2 drives the PMA bus with the next instruction address, and the PC is loaded from the PMA bus directly.

4.2.3 Down Counter and Stack

The down counter and associated count stack provide the program sequencer with a very powerful looping mechanism. The down counter is a 14-bit register with automatic post-decrement capability that is intended for controlling the flow of program loops which execute a predetermined number of times. Count values are 14-bit unsigned-magnitude values.

Before entering the loop, the counter is loaded from the lower 14 bits of the DMD bus with the desired loop count by assigning to the system variable, CNTR. The actual loop count *N* is loaded, as opposed to *N–1* which is generally required by other microprocessors to execute a loop *N* times. This is due to the operation of the counter expired (CE) status logic, which tests CE (and automatically post-decrements the counter) at the end of a DO UNTIL loop that uses CE as its termination condition. CE is tested at the beginning and the counter is decremented at the end of a processor cycle, therefore CE is asserted when the counter goes to 0001 so that the loop executes *N* times.

The counter may also be tested and decremented by a conditional jump instruction that tests CE.

The counter is not decremented when CE is checked as part of a conditional return, conditional trap, or conditional arithmetic instruction. The counter may be read directly over the DMD bus at any time without affecting its contents. When reading the counter, the upper two bits of the DMD bus will be padded with zeroes.

The count stack is a 14-bit by 4-word stack which allows the nesting of loops by storing temporarily dormant loop counts. When a new value is loaded into the counter from the DMD bus, the current counter value is automatically pushed onto the count stack as program flow enters the

inner loop. The count stack is automatically popped whenever the CE status is tested and is true, thereby resuming execution of the outer loop (if any). The count stack may also be popped manually if an early exit from a loop is taken.

There is an exception to the automatic pushing of the count stack. A counter load from the DMD bus does not cause a count stack push if there is no valid value in the counter, because a stack location would be wasted on the invalid counter value. There is no valid value in the counter after a system reset and also after the CE condition is tested when the count stack is empty. The "count stack empty" status bit in the SSTAT register is set whenever the number of pop operations is greater than or equal to the number of push operations (four maximum) since the last reset (ignoring overflows).

4.2.4 Loop Comparator and Stack

The DO UNTIL instruction executes a zero-overhead loop using the loop comparator and loop stack.

The loop comparator continuously compares the address of the last instruction in the loop (coded in the DO UNTIL instruction) against the next address. The address of the first instruction in the loop is maintained on top of the PC stack. When the last instruction in the loop is executed the processor conditionally jumps to the beginning of the loop, eliminating the branching overhead otherwise incurred in loop execution.

The loop stack stores the end addresses and termination conditions of temporarily dormant loops. Up to four levels can be stored. The only "extra" cycle associated with the nesting of DO UNTIL loops is the execution of the DO UNTIL instruction itself, since the pushing and popping of all stacks associated with the looping hardware is automatic. When using the counter expired (CE) status as the termination condition for the loop, another cycle is required for the initial loading of the counter. Table 4.1, on the next page, shows the termination conditions that can be used with DO UNTIL.

Syntax	*Status Condition*	*True If:*
EQ	Equal Zero	AZ = 1
NE	Not Equal Zero	AZ = 0
LT	Less Than Zero	AN .XOR. AV = 1
GE	Greater Than or Equal Zero	AN .XOR. AV = 0
LE	Less Than or Equal Zero	(AN .XOR. AV) .OR. AZ = 1
GT	Greater Than Zero	(AN .XOR. AV) .OR. AZ = 0
AC	ALU Carry	AC = 1
NOT AC	Not ALU Carry	AC = 0
AV	ALU Overflow	AV = 1
NOT AV	Not ALU Overflow	AV = 0
MV	MAC Overflow	MV = 1
NOT MV	Not MAC Overflow	MV = 0
NEG	X Input Sign Negative	AS = 1
POS	X Input Sign Positive	AS = 0
CE	Counter Expired	CE = 0 *(at loop end)*
FOREVER	Always	Always True

Table 4.1 DO UNTIL Condition Logic

These are the inverse of the conditions tested in an *IF condition* construct. That is, the termination condition for DO UNTIL NE produces the same opcode condition field (0000) as IF EQ JUMP. This difference is transparent at the source code level. The IF conditions are given in Table 4.3 which is located on page 4-25.

When a DO UNTIL instruction is executed, the 14-bit address of the last instruction and a 4-bit termination condition (both contained in the DO UNTIL instruction) are pushed onto the 18-bit by 4-word loop stack. Simultaneously, the PC incrementer output is pushed onto the PC stack. Since the DO UNTIL instruction is located just before the first instruction of the loop, the PC stack will contain the first loop instruction address, and the loop stack will contain the last loop instruction address and termination condition. The non-empty state of the loop stack activates the loop comparator which compares the address on top of the loop stack with the next address being fetched. When these two addresses are equal, the loop comparator notifies the next address source selector that the last instruction in the loop will be executed on the next cycle.

There are two possible cases depending on the type of instruction at the end of the loop. Case 1 illustrates the most typical situation. Case 2 is also allowed but involves greater program complexity for proper execution.

Case 1
If the last instruction in the loop is not a jump, call, or return, then the next address source selector will choose the next address based on the termination condition contained on top of the loop stack. If the condition is false, the top PC stack value is selected causing a jump back to the beginning of the loop. If the termination condition is true, the PC incrementer is chosen, causing execution to fall out of the loop. The loop stack, PC stack, and counter stack, (if it is being used) are then popped.

Note that conditional arithmetic instructions will be executed based on the condition explicitly stated in the instruction, with the loop sequencing controlled by the (implicit) termination condition contained on top of the stack.

Case 2
If the last instruction in the loop is a jump, call, or return, the explicitly stated instruction takes precedence over the implicit sequencing of the loop. If the condition in the instruction is false, normal loop sequencing takes place as described for Case 1.

If the condition in the instruction is true, however, program control transfers to the jump/call/return address. Any actions that would normally occur upon an end-of-loop detection will not take place: jumping to the beginning of the loop, falling out of the loop and popping the loop, PC, and counter stacks, or decrementing the counter.

Note that for a return, control is passed back to the top of the loop since the PC stack contains the beginning address of the loop.

Caution is required when ending a loop with a jump, call, or return, or when making a premature exit from a loop. Since none of the loop sequencing mechanisms are active while the jump/call/return is being performed, the loop, PC, and counter stacks will generally be left with the looping information (since they are not popped). In this situation, a manual pop of each of the relevant stacks is required to restore the original state of the processor.

Subroutine calls only pose this problem when the call is the last instruction in the loop, since a return causes program flow to transfer to the instruction just after the loop. Calls within a loop that are not the last instruction present no problem.

The only restriction concerning DO UNTIL loops is that nested loops cannot terminate on the same instruction. Since the loop comparator can only check for one loop termination at a time, falling out of an inner loop by incrementing the PC would go beyond the end address of the outer loop if they terminated on the same instruction.

4.2.5 Interrupt Controller

The interrupt controller of the ADSP-2100 allows the processor to respond to one of four external interrupts within two cycles. Because of the efficient stack and program sequencer, there is no additional latency when processing unmasked interrupts, even when interrupting DO UNTIL loops. Nesting of interrupts allows higher-priority interrupts to interrupt any lower-priority interrupt service routines that may currently be executing, also with no additional latency. Single-cycle context switching is provided by the secondary register set, and maximum flexibility is also afforded via the different modes associated with the interrupt control logic. Consult the Chapter 5, "System Interface," for more about the interrupt response.

The secondary data register set, selected by the MODE CONTROL instruction allows the contents of the primary data register set (AX0, AX1, AY0, AY1, AF, AR, MX0, MX1, MY0, MY1, MF, MR2, MR1, MR0, SI, SE, SB,SR1, and SR0) to be saved while a "fresh" set of registers may be switched in for use by the interrupt service routine. The processor cannot predict the requirements of each interrupt service routine. Consequently, you must explicitly program a context switch between the primary and secondary register banks if required.

4.2.5.1 Configuring Interrupts

There are two configuration parameters for interrupts: edge or level sensitivity and masked or unmasked operation.

The four external interrupt inputs can be individually configured as either edge- or level-sensitive. If an interrupt input is edge-sensitive, the interrupt is recognized when two successive samples of the input reveal a high-to-low transition (note that this is not, strictly speaking, a response to the transition edge). All four interrupt inputs are sampled once each processor cycle. Detection of this transition sets an internal latch

corresponding to the active interrupt request. The latch remains set until the request is serviced, then is automatically cleared. Thus an edge-sensitive interrupt signal need only be active for one processor cycle, or can remain active indefinitely. Edge-sensitive inputs generally require less external hardware than level-sensitive inputs, and allow "oddball" signals such as sampling rate clocks to be used for interrupt sources.

A level-sensitive interrupt must remain asserted until the interrupt is serviced. The interrupting device must then remove the interrupt request so that this interrupt is not serviced again. Level-sensitive inputs allow many interrupt sources to use the same interrupt level by ORing them together into a single IRQ pin.

You may also select whether nesting of interrupt service routines is allowed. All interrupt request levels may be automatically masked when an interrupt service routine is entered. Or, if desired, only equal and lower priority interrupts will be masked.

The interrupt control register (ICNTL) is set to indicate these choices. The automatic masking of interrupts described above occurs within the interrupt mask (IMASK) register. Both are described later in this chapter.

4.2.5.2 Interrupt Handling

The individual interrupt request signals are logically ANDed with the four IMASK bits and then fed to a priority encoder which selects the highest priority unmasked active request. The priorities are permanently assigned, with IRQ3 being the highest and IRQ0 being the lowest. An active output from the priority encoder causes a jump to the interrupt location, program memory address 0000 through 0003 corresponding to the interrupt level serviced.

Interrupt	*Interrupt Location*	
IRQ0	0000	Lowest priority
IRQ1	0001	
IRQ2	0002	
IRQ3	0003	Highest priority

Jump instructions to the corresponding interrupt service routines are typically stored at the interrupt location addresses, although any instruction (such as return from interrupt) may be stored there. Note that the 14-bit vector address to the interrupt service routine will be contained within the jump instruction.

Vectoring to an interrupt service routine in this manner incurs a two cycle overhead. The first overhead cycle occurs because execution of the instruction fetched during the previous cycle is aborted. (No data registers are updated by the aborted instruction.) This instruction is aborted because the PC and status stacks are pushed simultaneously with the jump to the interrupt location, and any instruction that used these (such as a CALL) would cause a conflict. For the same reason, the PC incrementer is disabled so that the current PC value is pushed onto the stack, causing the aborted instruction to be re-fetched upon returning from the interrupt service routine.

The second overhead cycle is incurred for the jump instruction (at the interrupt location) that is executed to enter the interrupt service routine.

Interrupt vectoring pushes the status stack with the current arithmetic status, mode status, and interrupt mask register contents: ASTAT, MSTAT and IMASK. (The contents of the status stack may be examined with the ADSP-2100 Simulator; ASTAT, MSTAT and IMASK are stored in this order, with the MSB of ASTAT first, and so on.) When the interrupt mask register is pushed, it is automatically loaded with a new value that reflects the status of the interrupt nesting mode bit. There is no additional overhead penalty for these operations.

After the interrupt service routine has been completed, the RTI (return from interrupt) instruction returns control to the main routine by popping the top PC stack value into the PC, while at the same time popping the status stack to restore the original machine status.

4.2.6 Sequencer Operations Illustrated

In this section, each of the major sequencer operations is illustrated and briefly described. The accompanying figures show only the parts of the program sequencer that are used in the operation described.

4.2.6.1 Linear Flow

In Figure 4.2, the typical linear flow of the program sequencer is illustrated. The instruction, identified by the instruction function field, does not branch; sequential execution is correct. The program memory address in the program counter is incremented and put on the PMA bus. This incremented address is loaded back into the program counter, as shown by the gray arrow, to begin the next cycle. This example is not conditional.

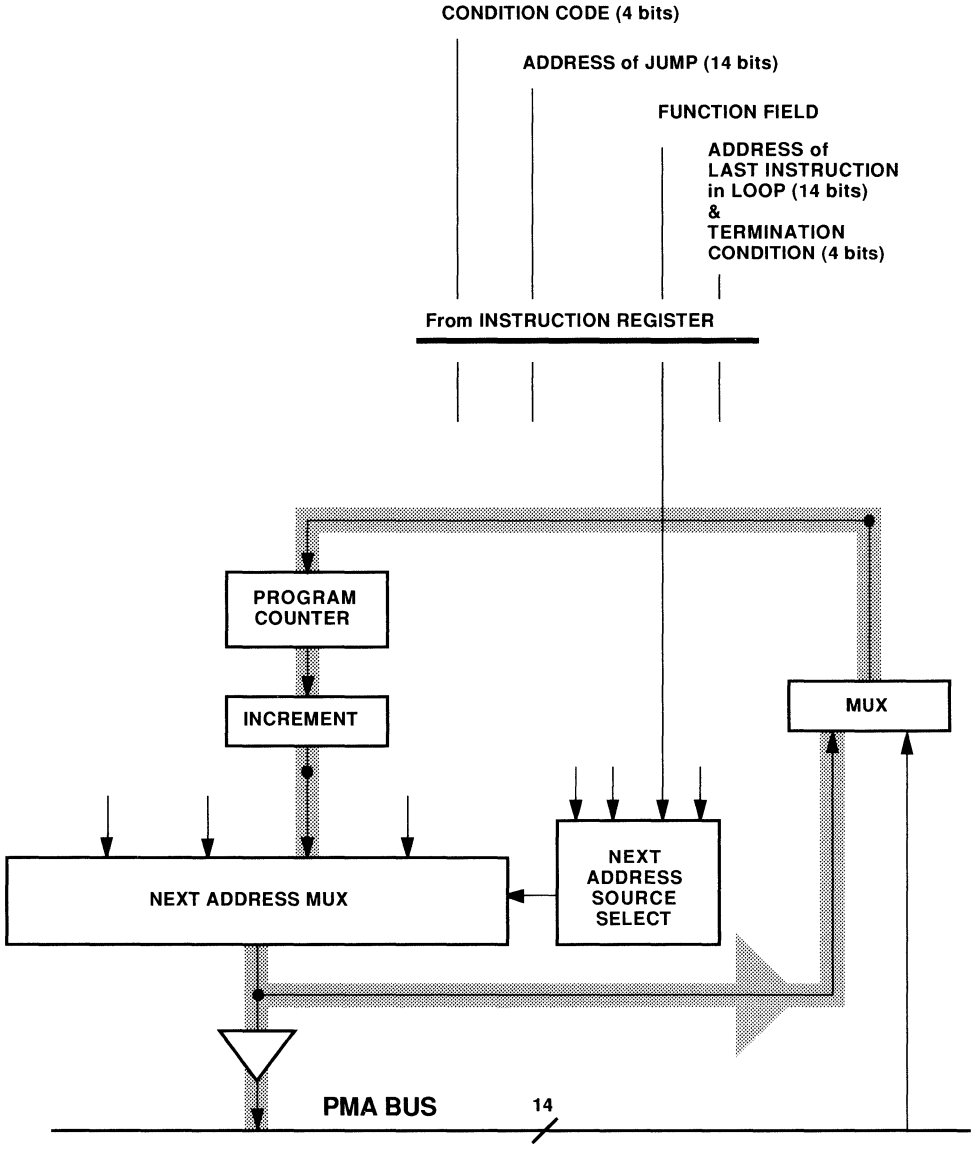


Figure 4.2 Linear Flow

4.2.6.2 JUMP Sequence

The JUMP sequence is shown in Figure 4.3. The JUMP instruction function field indicates the action to be taken. The 14-bit JUMP address is contained directly in the instruction word and is loaded directly into the next address mux. The address is put on the PMA bus and fed back to the program counter, as shown by the gray arrow, for the next cycle. This example is not conditional.

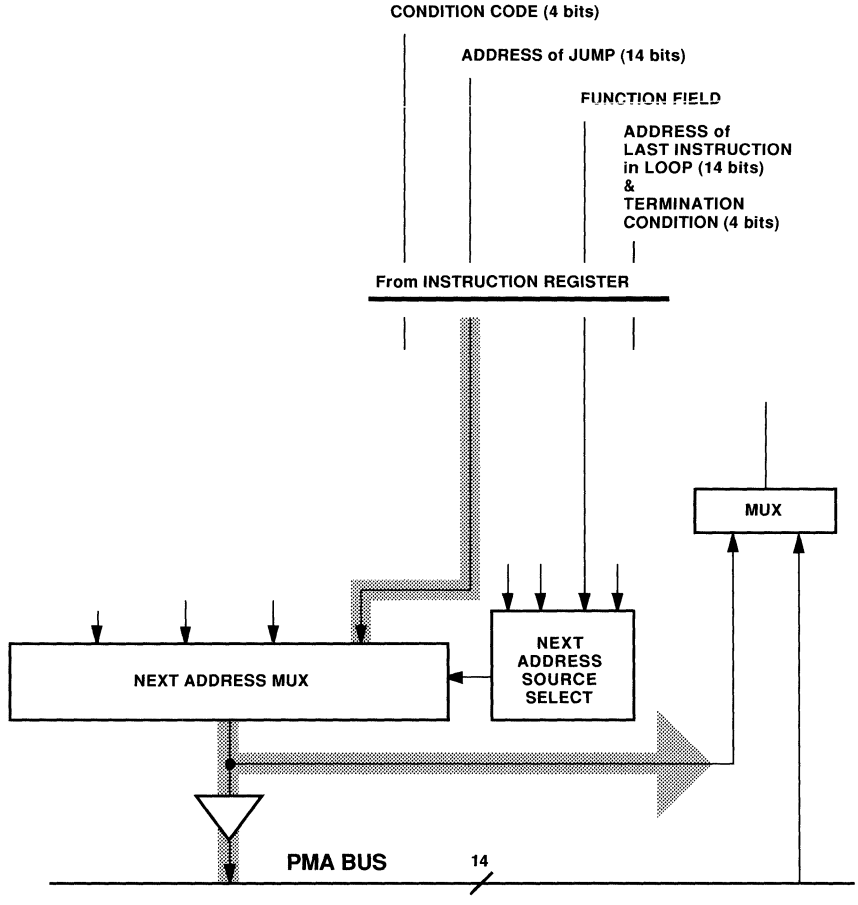


Figure 4.3 JUMP Sequence

4.2.6.3 CALL Sequence

The CALL sequence, shown in Figure 4.4, is very similar to the JUMP sequence. The address comes directly from the instruction and the address put on the PMA bus is fed back to program counter to begin the next cycle. In addition, however, the current value of the program counter is incremented and then pushed on the PC stack. Upon return from the subroutine, the PC stack is popped into the program counter and execution resumes with what would have been the next instruction if the CALL had not occurred. This example is not conditional.

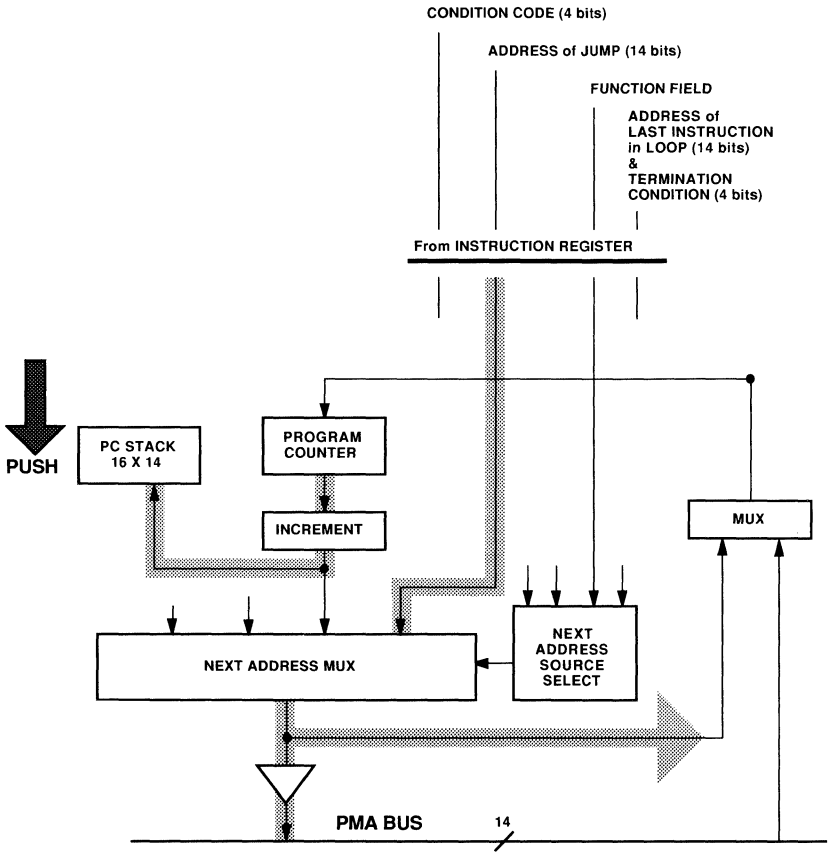


Figure 4.4 CALL Sequence

4.2.6.4 Interrupt Sequence

The interrupt sequence, shown in Figure 4.5, aborts the current instruction fetch. The interrupt is sensed by the interrupt controller and compared with the interrupt mask, IMASK. If enabled, the interrupt sequence pushes the current status registers (ASTAT, MSTAT and IMASK) onto the status

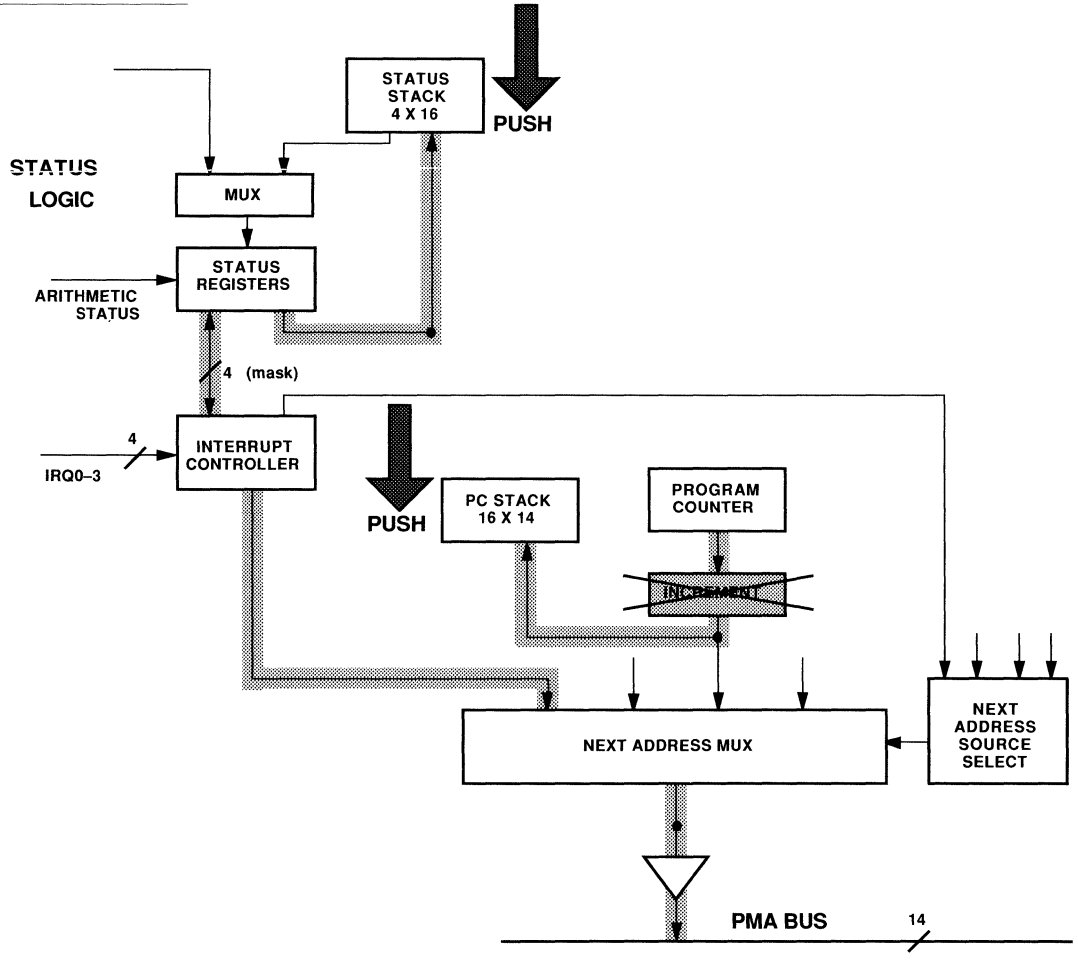


Figure 4.5 Interrupt Sequence

stack. The current program counter is pushed onto the PC stack *without* being incremented. This means that a return from the interrupt will resume with the instruction that would have been executed when the interrupt occurred, not the following one.

The interrupt controller controls the next address source selection and drives the correct one of the four possible interrupt vector addresses via the next address mux onto the PMA bus. The interrupt vector address is loaded into the program counter for the next cycle, but since the next instruction is virtually certain to be a JUMP, this action is not shown in the figure.

Upon return from the interrupt routine, the PC and status stacks are popped and execution resumes with the instruction whose fetch was aborted by the original interrupt.

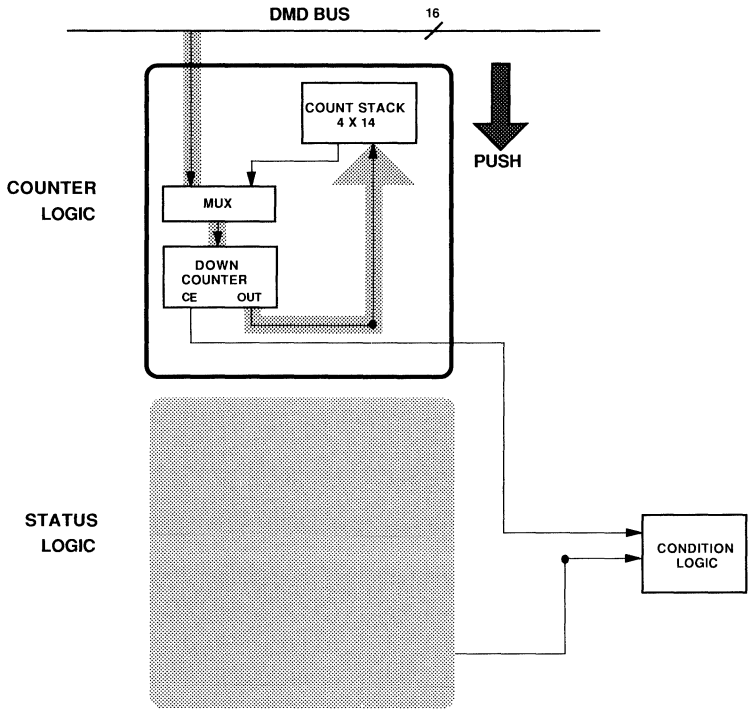


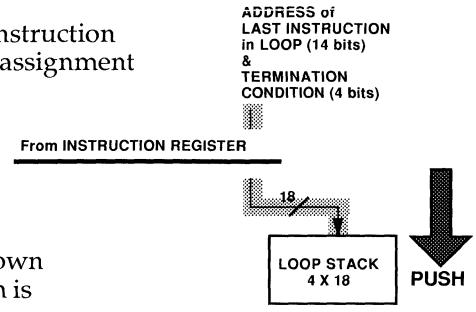
Figure 4.6A DO UNTIL: Load Counter

4.2.6.5 DO UNTIL Loop

The stages in a DO UNTIL loop are shown in Figures 4.6A through 4.6E. The sequence shown does not cover every possible case, but serves as a guide for understanding how this instruction is implemented in the ADSP-2100 program sequencer.

The example shown illustrates a DO UNTIL CE ("counter expired") version of the loop. In this case, only the counter logic is involved. If a different termination condition was used, the status logic would be used instead (see Tables 4.1 and 4.3). The balance of the DO UNTIL instruction, however, is unchanged.

In Figure 4.6A (on page 4-15), the instruction loading the counter is shown. This assignment statement, which is executed prior to entering the loop, moves the counter value over the DMD bus into the counter logic section of the program sequencer. The previous count (if any) would be pushed onto the count stack, as shown in Figure 4.6A. This push operation is omitted if the counter is empty.



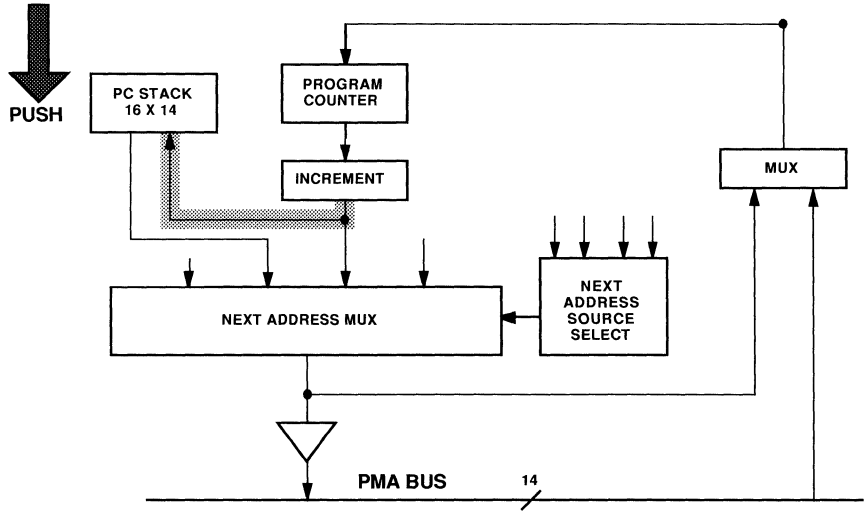


Figure 4.6B DO UNTIL: Execute "DO UNTIL"

In Figure 4.6B on the facing page, the DO UNTIL instruction itself is shown. The effect of the DO UNTIL instruction itself is only to set up the conditions for looping; no other computation occurs while this instruction is executed. This occurs only once, at the beginning of the loop.

Executing DO UNTIL pushes the output of the incremented program counter, that is, the address of the instruction immediately following the DO UNTIL itself, onto the PC stack. This is the first instruction inside the loop. On the same cycle, the loop stack is also pushed with the address of the end of the loop and the termination condition.

Within the loop, execution follows the normal linear sequence, as in Figure 4.2 above, except that the loop comparator checks the current address against the address of the last instruction loaded in the loop stack. As long as that address has not yet been reached, linear flow continues. This operation is represented in Figure 4.6C.

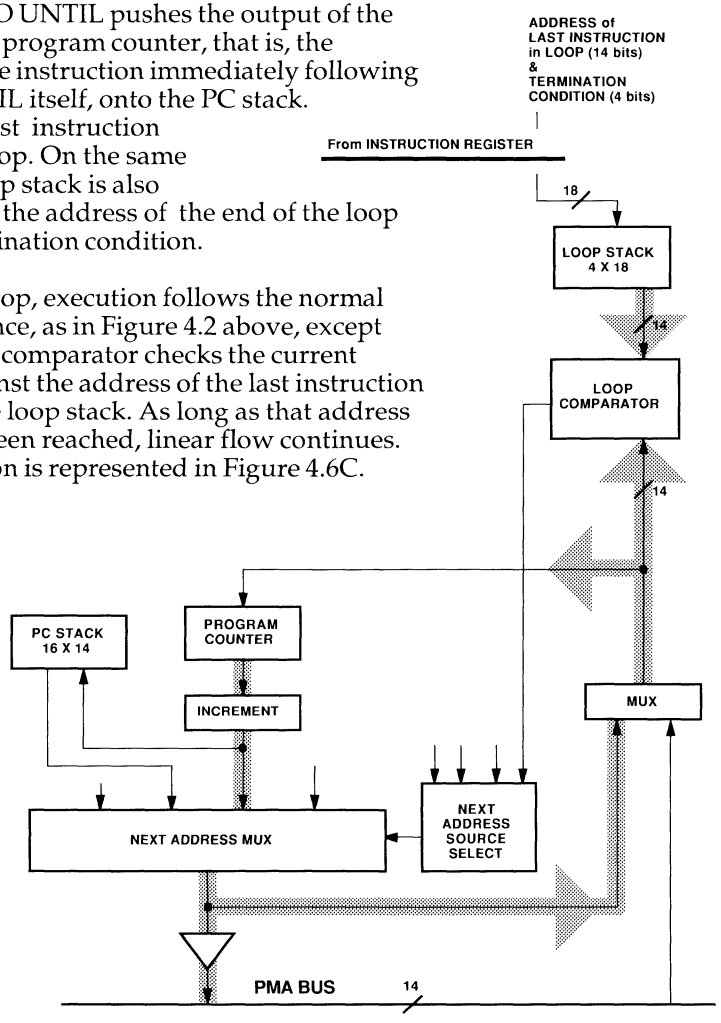


Figure 4.6C DO UNTIL: Flow Inside Loop

When the end of the loop is reached, the operations represented in Figure 4.6D. The loop comparator now finds that the current address equals the last address in the loop. This output changes the next address source select logic. Instead of using the incremented program counter, the termination condition is now evaluated. Assuming that this is only one of *N* passes through the loop, the termination condition is false and execution continues with the top of the loop. This selects the top of the PC stack as the source of the next address, effectively jumping back to the beginning of the loop. Note that the PC and Loop stacks are not popped, only read.

The condition logic, as mentioned before, may be driven by either the counter logic section or the status logic, depending on the termination condition specified.

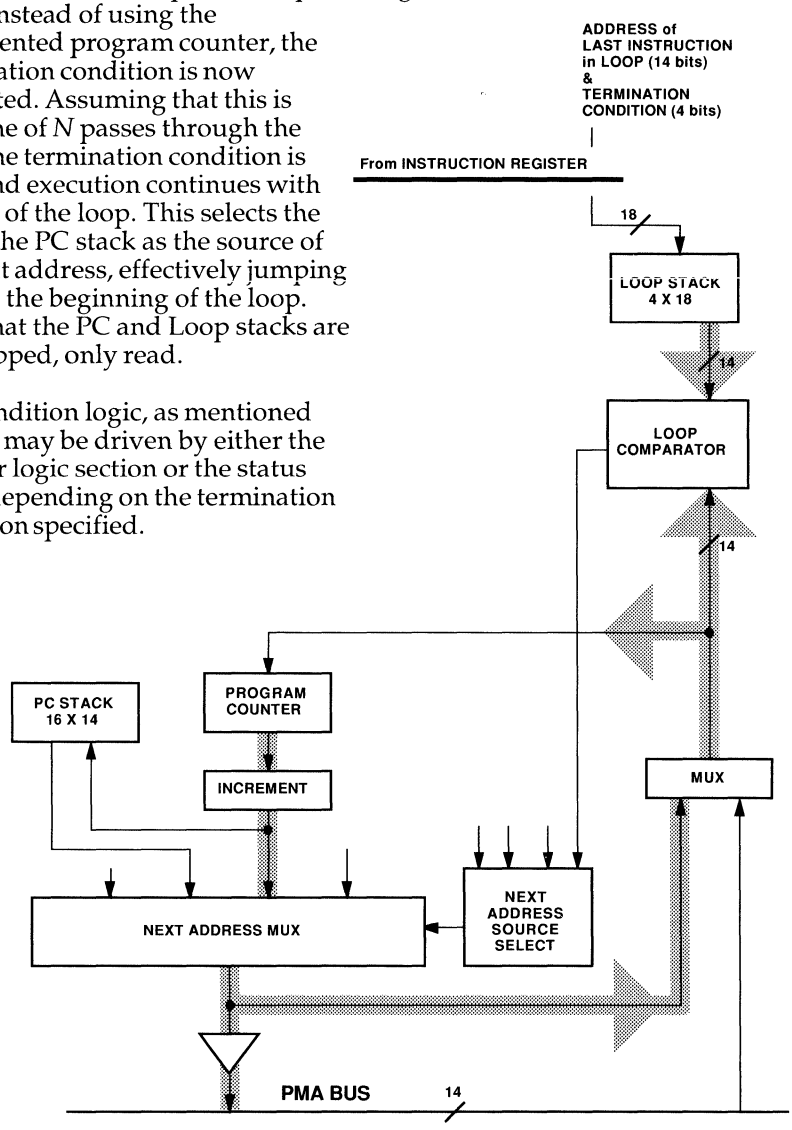


Figure 4.6D DO UNTIL: End of One Iteration

The termination of the loop on the final pass is shown in Figure 4.6E. As in the previous figure, the loop comparator signals the end of a pass through the loop, i.e. the current address and the address of the last instruction in the loop are the same. This time through, however, the termination condition is true. This means that the PC stack is popped and the next address comes from the incremented program counter, in other words, the instruction immediately following the last instruction in the loop. This carries the flow of execution out of the loop.

The loop stack and the count stack are also popped on this cycle.

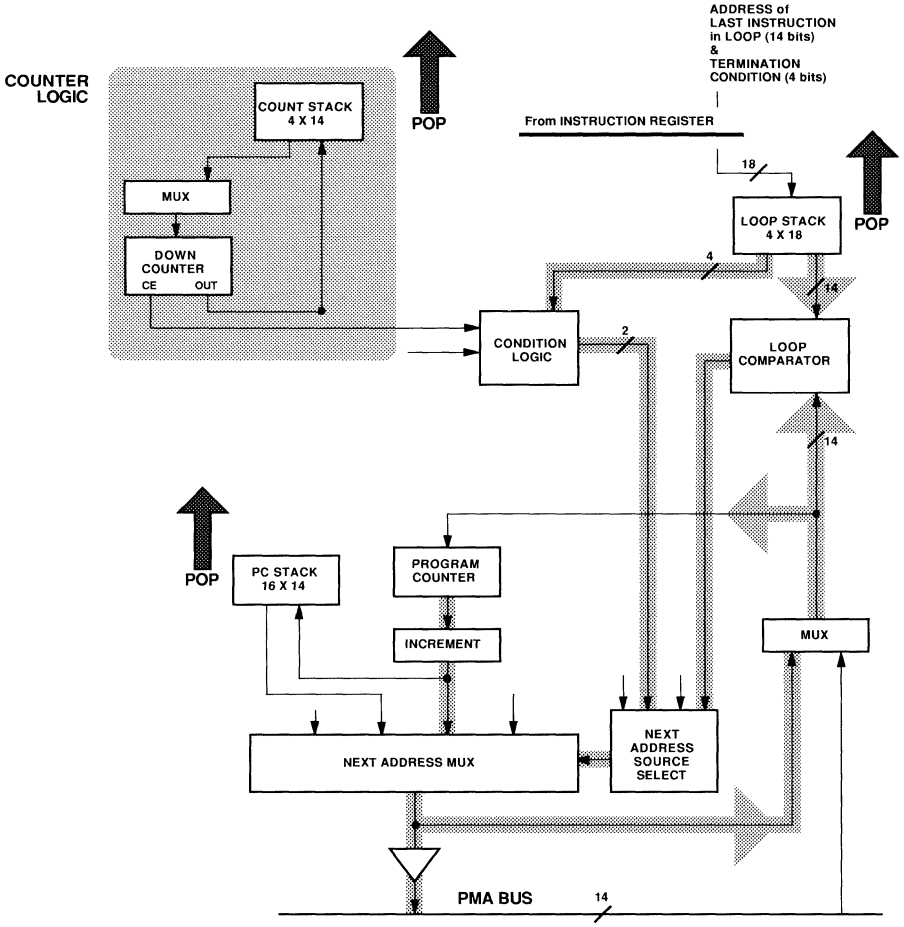


Figure 4.6E DO UNTIL: Final Iteration

4.2.6.6 Register Indirect

Figure 4.7 illustrates the case of a register indirect jump. In this case, the address is actually not being supplied by the sequencer at all. Instead, the processor is executing a jump to the label/address supplied by DAG2. DAG2 drives the address onto the PMA bus. The program counter reads the address from the PMA bus and continues normally from there.

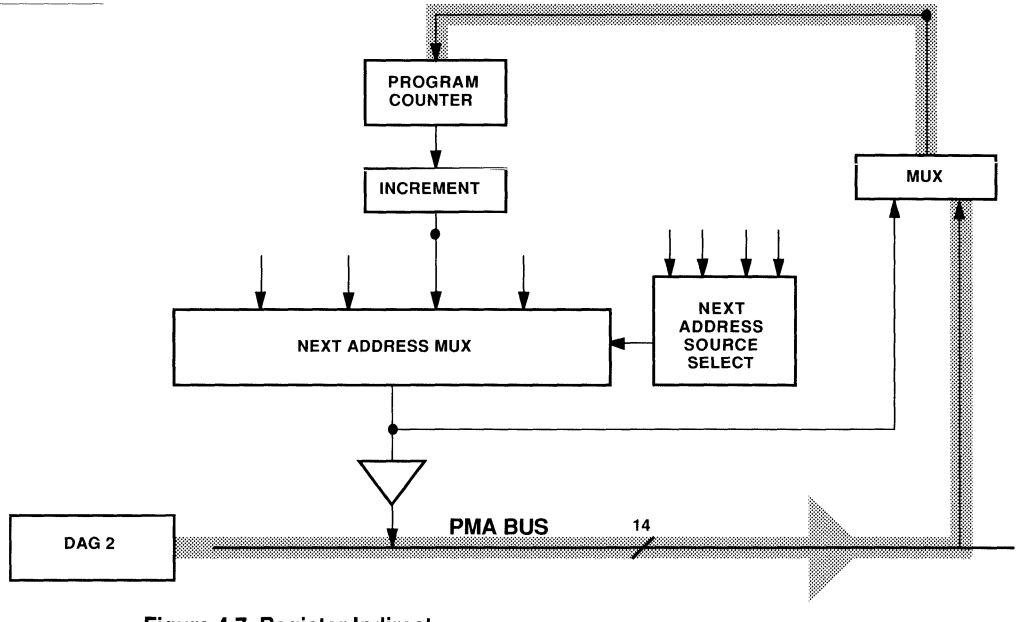


Figure 4.7 Register Indirect

4.3 STATUS REGISTERS AND STACK

The status and mode bits of the ADSP-2100 are maintained internally within five registers, each of which are independently readable over the DMD bus, and four of which can be written to from the DMD bus. These registers are:

ASTAT	Arithmetic status register
SSTAT	Stack status register (read-only)
MSTAT	Mode status register
ICNTL	Interrupt control register
IMASK	Interrupt mask register

4.3.1 Arithmetic Status Register (ASTAT)

ASTAT is 8 bits wide and holds the status information generated by the computational sections of the processor. The bits in ASTAT are defined as follows:

Bit 0	AZ	ALU result zero
Bit 1	AN	ALU result negative
Bit 2	AV	ALU overflow
Bit 3	AC	ALU carry
Bit 4	AS	ALU X input sign
Bit 5	AQ	ALU quotient flag
Bit 6	MV	MAC overflow
Bit 7	SS	Shifter input sign

The bits which express a particular condition (AZ, AN, AV, AC, MV) are all positive sense (1 = true, 0 = false). Each of the bits is automatically updated when a new status is generated by an arithmetic operation. Each bit is affected only by a subset of arithmetic operations, as defined by the following table.

Status Bit	*Updated by*
AZ, AN, AV, AC	Any ALU operation except DIVS, DIVQ
AS	ALU absolute value operation (ABS)
AQ	ALU divide operations (DIVS, DIVQ)
MV	Any MAC operation except saturate MR
SS	Shifter EXP operation

Arithmetic status is latched into the status register at the end of the cycle in which it was generated, and therefore cannot be used until the next cycle.

Loading any ALU, MAC, or Shifter input or output registers directly from the DMD bus does not affect any of the arithmetic status bits. Executing the ALU instruction PASS will set the AZ and AN bits for a given X or Y operand.

4.3.2 Stack Status Register (SSTAT)

SSTAT is 8 bits wide and holds information regarding the four internal stacks. The bits in SSTAT are defined as follows:

Bit 0 PC Stack Empty
Bit 1 PC Stack Overflow
Bit 2 Count Stack Empty
Bit 3 Count Stack Overflow
Bit 4 Status Stack Empty
Bit 5 Status Stack Overflow
Bit 6 Loop Stack Empty
Bit 7 Loop Stack Overflow

All of the bits are positive sense (1 = true, 0= false). The empty status bits indicate that the number of pop operations for the stack is greater than or equal to the number of push operations (if no stack overflow has occurred) since the last reset. The overflow status bits indicate that the number of push operations for the stack has exceeded the number of pop operations by an amount that is greater than the depth of the stack. When this occurs, the item(s) most recently pushed will be missing from the stack (old data is considered more important than new). Because of this "saturation" of the stack pointer, the stack empty status bits can be set by N sequential pop operations, where N is the depth of the stack, regardless of how many more than N sequential push operations were performed.

Since a stack overflow represents a permanent loss of information, the stack overflow status bits "stick" once they are set and subsequent pop operations have no effect on them. It is possible to have both the stack empty and stack overflow bits set for a given stack. Since SSTAT is a read-only register, write operations will have no effect on the stack status bits either. A processor reset must be executed to clear the stack overflow status.

4.3.3 Mode Status Register (MSTAT)

MSTAT is a 4-bit register that defines various operating modes of the processor. The bits in MSTAT are defined as follows:

Bit 0 Data Register Bank Select
Bit 1 Bit Reverse Mode (Data Address Generator 1 only)
Bit 2 ALU Overflow Latch Mode
Bit 3 AR Saturation Mode

All registers (including MSTAT) can be changed by moving a new value into them with any of the MOVE instructions. In contrast to the other status registers, MSTAT can also be changed with the MODE CONTROL instruction.

The data register bank select bit determines which set of data registers is currently active (0 = primary, 1 = secondary). The data registers include all of the result and input registers to the ALU, MAC, and SHIFTER: AX0, AX1, AY0, AY1, AF, AR, MX0, MX1, MY0, MY1, MF, MR2, MR1, MR0, SI, SE, SB, SR1, and SR0.

The bit-reverse mode, when enabled, bitwise reverses all addresses generated by data address generator one (DAG1). This is most useful for reordering the input or output data to a radix-2 FFT algorithm. In addition to the MODE CONTROL instruction, processor reset also disables it.

The ALU overflow latch mode causes the AV (ALU overflow) status bit to "stick" once it is set. In this mode, AV will be set by overflow and remain set, even if subsequent ALU operations do not generate overflows. AV can then only be cleared by writing a zero into it from the DMD bus.

The AR saturation mode, when set, causes AR to be saturated to the maximum positive (H#7FFF) or negative (H#8000) values when an ALU overflow occurs.

4.3.4 Interrupt Control Register (ICNTL)

ICNTL is a 5-bit register that configures the interrupt modes of the processor. These bits are all undefined after a processor reset. The bits in ICNTL are defined as follows:

Bit 0 IRQ0 Sensitivity
Bit 1 IRQ1 Sensitivity
Bit 2 IRQ2 Sensitivity
Bit 3 IRQ3 Sensitivity
Bit 4 Interrupt Nesting Mode *See Table 4.2*

The IRQ sensitivity bits determine whether a given interrupt input is edge- or level-sensitive (0 = level-sensitive, 1 = edge-sensitive).

Bit 4 determines whether nesting of interrupt service routines is allowed. When set to zero, all interrupt levels are masked automatically (IMASK set to zero) when an interrupt service routine is entered. When set to one, IMASK is set so that only equal and lower priority interrupts are masked,

permitting higher priority interrupts to interrupt the current interrupt service routine. This is graphically shown in Table 4.2 below.

4.3.5 Interrupt Mask Register (IMASK)

IMASK is a 4-bit register which enables and disables the individual interrupt levels. The IMASK register contents are automatically pushed onto the status stack when entering an interrupt service routine and popped back when returning from the routine. The configuration of IMASK upon entering the interrupt service routine is determined by bit 4 of ICNTL; it may be altered, of course, as part of the interrupt service routine itself.

The bits in IMASK are defined as follows:

Bit 0 IRQ0 Enable
Bit 1 IRQ1 Enable
Bit 2 IRQ2 Enable
Bit 3 IRQ3 Enable

The bits are all positive sense (0 = disabled, 1 = enabled). IMASK is set to zero upon a processor reset. When an interrupt is processed, the interrupt nesting mode bit determines the state of IMASK upon entering the interrupt, as shown in Table 4.2. IMASK may be read from or written to via the DMD bus.

ICNTL bit 4 = 0 (nesting disabled)

IRQ # Serviced	*IMASK contents before, pushed on stack*	*IMASK contents entering interrupt service*
0	DDDD*	0000
1	DDDD	0000
2	DDDD	0000
3	DDDD	0000

ICNTL bit 4 = 1 (nesting enabled)

IRQ # Serviced	*IMASK contents before pushed on stack*	*IMASK contents entering interrupt service*
0	DDDD	1110
1	DDDD	1100
2	DDDD	1000
3	DDDD	0000

**"DDDD" represents any pattern of ones and zeroes.*

Table 4.2 IMASK Entering Interrupt Service Routines

4.3.6 Condition Logic

The condition logic of the ADSP-2100 is used to determine whether an explicitly specified action in a conditional instruction is performed, such as a jump, trap, call, return, MAC saturation, or arithmetic operation. It also controls the implicit loop sequencing operations based upon the loop continuation condition on top of the loop stack. The condition logic takes raw status information from ASTAT and the down counter and derives a set of sixteen composite status conditions. The 4-bit condition code field of the instruction and the 4-bit loop continuation condition on the loop stack then select two of these to control whether the explicit operation in the instruction or implicit loop sequencing operation (or neither) is performed. When both are attempted, the explicitly specified operation takes precedence.

The sixteen composite status conditions, with their derivations and instruction mnemonics are given below, are for the standard IF condition statement. Consult the section on DO UNTIL and the opcodes in Appendix A for details of the termination condition usage.

Syntax	*Status Condition*	*True If:*
EQ	Equal Zero	AZ = 1
NE	Not Equal Zero	AZ = 0
LT	Less Than Zero	AN .XOR. AV = 1
GE	Greater Than or Equal Zero	AN .XOR. AV = 0
LE	Less Than or Equal Zero	(AN .XOR. AV) .OR. AZ = 1
GT	Greater Than Zero	(AN .XOR. AV) .OR. AZ = 0
AC	ALU Carry	AC = 1
NOT AC	Not ALU Carry	AC = 0
AV	ALU Overflow	AV = 1
NOT AV	Not ALU Overflow	AV = 0
MV	MAC Overflow	MV = 1
NOT MV	Not MAC Overflow	MV = 0
NEG	X Input Sign Negative	AS = 1
POS	X Input Sign Positive	AS = 0
NOT CE	Not Counter Expired	CE ≠ 0
TRUE	Always True	Always True

Table 4.3 IF Condition Logic

Since arithmetic status is latched into ASTAT at the end of a processor cycle, the condition logic outputs represent conditions generated on a previous cycle.

4.4 INSTRUCTION CACHE

The instruction cache memory stores up to sixteen previously executed instructions. When an instruction requires a Program Memory Data fetch (which would conflict with the instruction fetch) the cache, if valid, is used as the source of the instruction. This section discusses the operation and programming implications of the cache.

4.4.1 Cache Memory Operation

Cache operation is transparent. No maintenance or overhead is required for either the storage or use of instructions in cache memory. Because of the significant of the ADSP-2100 cache, however, it is vital to understand how it operates.

The cache is a 24-bit by 16-word memory array. While this is a small memory space, the multifunctional nature of many ADSP-2100 instructions allows a wide variety of algorithms to be coded within this restriction; see the discussion of the instruction set in Chapter 6 and the example below.

The cache memory can be seen in the overall processor block diagram (in Chapter 1) interacting with the program sequencer and instruction register. Cache operation follows this scenario.

1. In normal operation the ADSP-2100 fetches the *(N+1)th* instruction while executing the *Nth* instruction. Each instruction fetched for the instruction register is also written into the cache. It is stored at the cache memory address specified by the four LSBs of the program memory address.

2. When the PMD bus is busy with a data transfer, the instruction register is loaded from cache. Note that, at this point, the validity of the loaded instruction has not been determined.

3. If the loaded instruction is valid, it is executed on the next cycle. If the instruction is not valid, the instruction register is cleared. An additional cycle is now required to fetch the next instruction. Validity is determined by the cache memory monitor described below.

When data can be read from program memory, the ADSP-2100 becomes, in effect, a processor with two data busses. For the multiply/accumulate operations typical of digital signal processing algorithms, this gives significant speed advantages. For program loops that can be stored completely in cache, an additional cycle penalty is only incurred on the first pass through the loop. After all instructions are in memory, the ADSP-2100 can simultaneously fetch two items of data (using the DMD and PMD busses) and one instruction (from the cache).

4.4.2 Cache Memory Monitor

The cache memory monitor logic keeps track of the program memory address range currently stored in the cache memory. One monitor register contains the number of instructions that are valid ahead of the currently executing instruction, while another register contains the number of instructions that are valid behind the currently executing instruction. The cache addressing uses only the four LSBs of the Program Memory Address.

While the cache generally contains sixteen previously executed instructions, not all of them are necessarily valid instructions because the cache memory monitor can only follow the execution of instructions that are contiguous in program memory. In effect, the cache memory monitor cannot "see" a region of memory bigger than sixteen words at a time. DO UNTIL loops and JUMPs inside the cache allow efficient utilization of the cache.

A JUMP to an address outside of the cache invalidates the entire cache. The number of valid instructions in the cache then increases until the cache fills or the program takes another out-of-cache jump. The cache memory size compared to the size of DO UNTIL or other looping constructs is one limit to keep in mind when writing programs with program memory data transfers inside the loop.

Once the cache fills, newly fetched instructions write over the oldest instructions in a circular manner due to the modulo-16 cache memory addressing.

4.4.3 Programmers' Guidelines For Cache Memory Usage

Programmers need not be aware of how the instruction cache functions in great detail. The important constraints for getting the benefit of the cache can be summarized as follows.

1. The cache can contain no more than sixteen instructions. To take advantage of the cache, loops should fit within this limit. The multifunction instruction set allows many common algorithms to be implemented within this limit.

2. To be valid, the instructions in the cache must be from a contiguous region of program memory. This means that DO UNTIL and JUMP loops can be used as long as the JUMP or loop top is within the sixteen-instruction region.

3. Cache memory is only used as the instruction source when the PMD bus is needed for data fetch that would conflict with the normal instruction fetch.

4.4.4 Cache Memory Example

Below is an ADSP-2100 subroutine that implements a simple sum-of-products FIR filter. This example illustrates three related advantages of the ADSP-2100 architecture: zero overhead looping, compact code requirements and execution speed resulting from use of the cache. This example is discussed in more detail in the *ADSP-2100 Applications Handbook, Volume 1* in the chapter on fixed-coefficient filters.

```
{           FIR Transversal Filter Subroutine

            Calling Parameters
               I0 -> Oldest input data value in delay line
               L0 = Filter length (N)
               I4 -> Beginning of filter coefficient table
               L4 = Filter length (N)
               M1,M5 = 1
               CNTR = Filter length - 1 (N-1)
```

```
                Return Values
                   MR1 = Sum of products (rounded and saturated)
                   I0 -> Oldest input data value in delay line
                   I4 -> Beginning of filter coefficient table
                Altered Registers
                   MX0,MY0,MR
                Computation Time
                   N - 1 + 5 + 2 cycles
                Coefficients & data values assumed to be 1.15 format.
}

A) fir:         MR=0, MX0=DM(I0,M1), MY0=PM(I4,M5);
B)              DO sop UNTIL CE;
C) sop:            MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M5);
D)              MR=MR+MX0*MY0(RND);
E)              IF MV SAT MR;
F)              RTS;
```

Figure 4.8 Cache Memory Program Example

For the purposes of this discussion we focus on the lines that have been labelled A) through F). The "A)" labels are not part of the ADSP-2100 instruction set and are used here only to identify each line of instruction source in this example.

Here is a description of the function of each program line:

A) Clears MR register, loads X & Y registers of MAC with operands, one from each memory. This example also uses the circular buffer addressing capabilities of the ADSP-2100.

B) Sets up DO UNTIL loop.

C) Executes a multiplication with accumulation and fetches two new operands, one from each memory. Executes *N–1* times.

D) Executes last multiplication with accumulation, rounding the result.

E) Checks for overflow and saturates if necessary.

F) Return from subroutine.

4 Program Control

Here is an overview of how this code executes a five tap FIR filter.

Cycle	*Program Bus*	*Internal Operation*	*Data Bus*
1	Fetch A	(Execute previous)	(Previous activity)
2	Fetch operand 1	Execute A	Fetch operand 1
3	Fetch B	**Idle, waiting for B**	No activity
4	Fetch C	Execute B	No activity
5	Fetch operand 2	Execute C, 1st time	Fetch operand 2
6	Fetch operand 3	Execute C, 2nd time, from cache	Fetch operand 3
7	Fetch operand 4	Execute C, 3rd time, from cache	Fetch operand 4
8	Fetch operand 5	Execute C, 4th time, from cache	Fetch operand 5
9	Fetch D	**Idle, waiting for D**	No activity
10	Fetch E	Execute D	No activity
11	Fetch F	Execute E	No activity
12	Fetch next	Execute F	No activity

Because the sequencer supports zero overhead looping, this single instruction loop runs as fast as any non-looped or straight-line version would on a different processor; in addition, two busses are available for data fetches. This also results in code that is compact; the loop requires one set-up instruction and no overhead instructions for each iteration. An *N*-tap filter requires only *N+1* cycles for the inner loop; straightline code on another processor might require *N* * 2 cycles with loop overhead included. Note that these ratios hold true for multiple instruction loops, not just for a loop of one instruction, as in this example.

The effect of the cache is quite dramatic in cycles 5 through 8; the processor executes the MAC operation and fetches two operands. (DAGs also update the address pointers to the circular buffer during each cycle.) There is no penalty for fetching the operand from program memory. This remains true for loop of up to sixteen instructions.

Finally, at the end of the loop, another cycle is consumed to load the instruction following the loop.

System Interface 5

5.1 OVERVIEW

This chapter describes how the ADSP-2100 is interfaced to your system. The chip has four major interfaces:

1. The program memory interface provides for the synchronous transfer of both instructions and data between the program memory and the processor.

2. The data memory interface provides for the synchronous transfer of data to and from the processor and, using the data memory acknowledge signal (DMACK), supports slow, memory-mapped peripherals.

   When the ADSP-2100 receives the bus request signal and responds with the bus grant signal, it relinquishes control of both program and data memory interfaces.

3. The control interface is used to halt and reset the processor and to signal an internal trap.

4. The ADSP-2100 can respond to four external interrupts which are internally prioritized, maskable and independently programmable as either edge or level-sensitive. The controls for interrupt configuration are described in Chapter 4 in the program sequencer section.

Figure 5.1, overleaf, is a basic system configuration for an ADSP-2100.

5.1.1 Note On Timing Diagrams

There are a number of "idealized" timing diagrams in this chapter; they show the logical relationship between internal clock phases of the ADSP–2100 and the external system. These timing diagrams do not show the actual specifications which factor in propagation delays. You must refer to the data sheet for that type of information. These ideal timing diagrams only provide a framework for understanding the function of the ADSP-2100.

5 System Interface

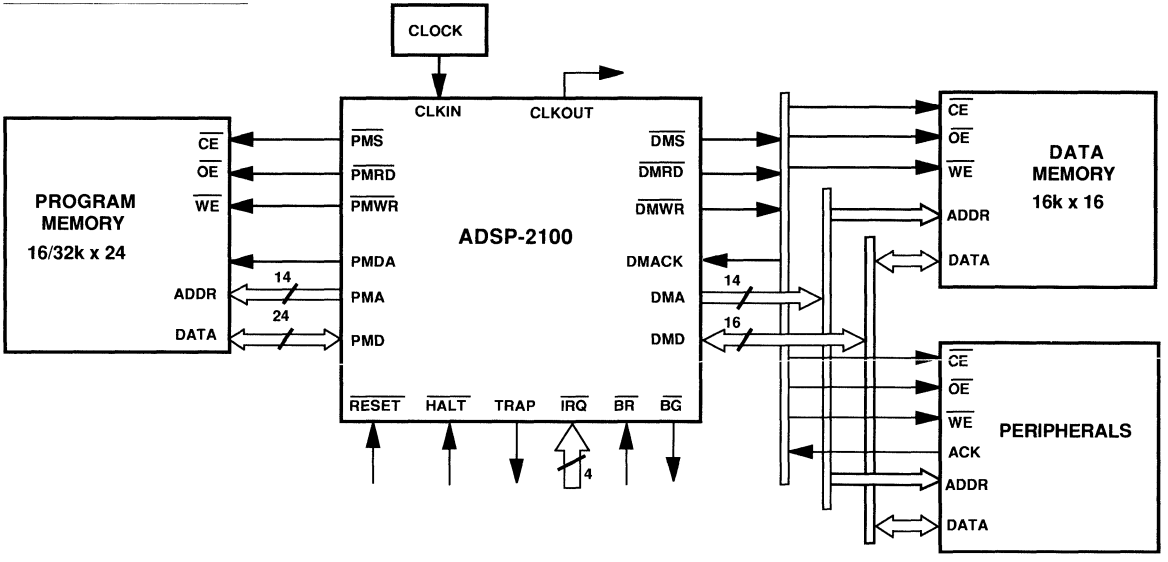


Figure 5.1 Basic System Configuration

5.1.2 Clock Signals & Processor States

The ADSP-2100 has two clock signals, CLKIN and CLKOUT. CLKIN is a master input clock to the processor that operates at four times the instruction cycle rate. The phases of CLKIN are used to define eight (1-8) distinct time periods, called the processor states, that make up an instruction cycle. This is shown in Figure 5.2. The eighth state of each instruction cycle is a dead state that provides a neutral halting point for the processor when operation is suspended. All timing diagrams annotate the phases of CLKIN with these state numbers.

CLKOUT is an output clock from the ADSP-2100 that operates at the instruction cycle rate. It is produced by dividing the frequency of CLKIN by four. The phase of CLKOUT is such that it allows external synchronization to the internal states of the processor. The falling transition of CLKOUT always occurs at the transition between states three and four while the rising edge always occurs at the transition between states seven and eight. This relationship is shown in Figure 5.2, on the facing page.

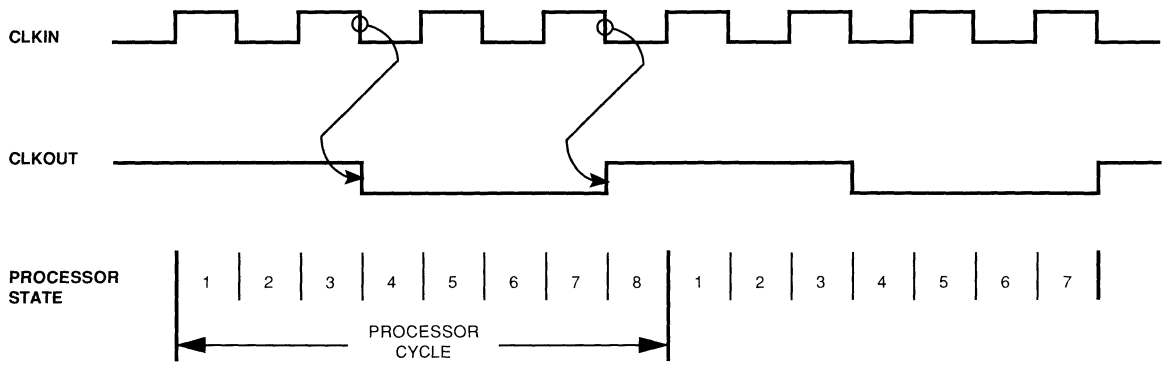


Figure 5.2 Clock Signals & Processor States

5.1.3 Synchronization Delay

The ADSP-2100 has several asynchronous inputs, namely, $\overline{\text{RESET}}$, $\overline{\text{HALT}}$, $\overline{\text{BR}}$, DMACK and $\overline{\text{IRQ0–3}}$. These inputs can be asserted in arbitrary phase to the processor clock, CLKIN. The ADSP-2100 resynchronizes them prior to recognizing them. The delay associated with resynchronization and eventual recognition is called the synchronization delay.

Different asynchronous inputs are recognized at different points in the processor cycle. For example, $\overline{\text{HALT}}$ is recognized at the end of state three but interrupt requests are recognized at the end of state seven.

Any asynchronous input must be valid prior to the recognition point. The minimum time prior to recognition (the setup time) is given on the data sheet. If an input does not meet the setup time on a given cycle, it will be recognized during the next cycle if it is held valid. Therefore, to ensure recognition of an asynchronous input, it must be asserted for at least one full processor cycle.

5.2 BUS REQUEST / GRANT

Using the bus request, $\overline{\text{BR}}$, and bus grant, $\overline{\text{BG}}$, signals, the ADSP-2100 can relinquish control of both the program and data memory interface giving direct memory access to an external device, such as a host processor.

The external device requests the bus by asserting $\overline{\text{BR}}$ (bus request). $\overline{\text{BR}}$ is

recognized at the end of the next state three and the ADSP-2100 halts in state eight of that instruction cycle. $\overline{BG}$ (bus grant) is asserted at the end of state three of what would have been the next instruction cycle, i.e. four cycles of CLKIN after the bus request is recognized. This is the normal synchronous mode of servicing this request.

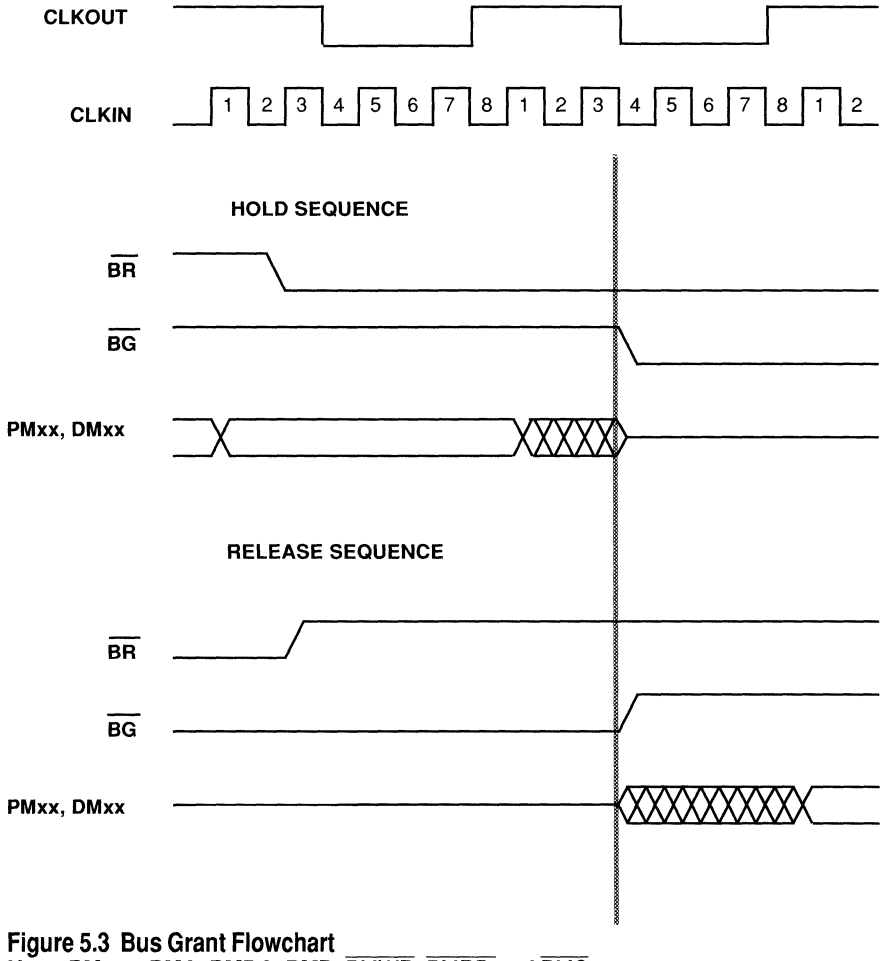


Figure 5.3 Bus Grant Flowchart
Note: PMxx = PMA, PMDA, PMD, PMWR, PMRD and PMS
DMxx = DMA, DMD, DMWR, DMRD and DMS.

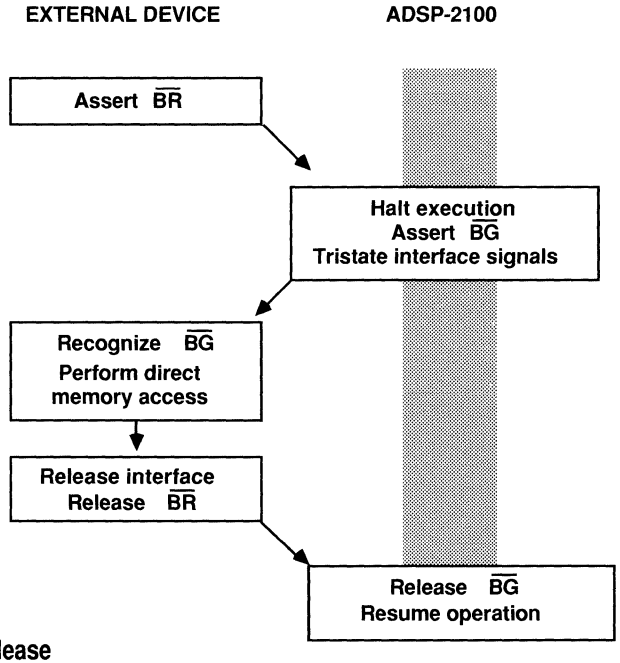


Figure 5.4 Bus Hold / Release

The ADSP-2100 tristates all the bus driving lines: PMA, PMD, PMDA, $\overline{\text{PMWR}}$, $\overline{\text{PMRD}}$ and $\overline{\text{PMS}}$ on the program memory interface and DMA, DMD, $\overline{\text{DMWR}}$, $\overline{\text{DMRD}}$ and $\overline{\text{DMS}}$ on the data memory interface. Control is then transferred to the requesting device.

The ADSP-2100's internal state is not affected by this operation. After the interface is released by the external device, normal operation resumes from the point at which it was halted. This applies uniformly to all processor operations, including the extra cycle inserted by the processor when a program memory data access is performed and the cache contents are not valid. $\overline{\text{BR}}$ can be serviced between the two cycles required for that operation if necessary.

The device returns control to the processor by releasing $\overline{\text{BR}}$. Four cycles of CLKIN after $\overline{\text{BR}}$ is recognized as released, the processor releases $\overline{\text{BG}}$ and takes over the bus, resuming with state one of the next cycle. Figure 5.3 is an operations flowchart. Figure 5.4 shows the relative timing of this cycle.

5 System Interface

5.2.1 Bus Request at $\overline{\text{RESET}}$

A bus request can be made, i.e. $\overline{\text{BR}}$ may be asserted, during a $\overline{\text{RESET}}$ of the ADSP-2100. The timing is different than shown in Figure 5.4. In this case, $\overline{\text{BG}}$ will be asserted asynchronously some time after $\overline{\text{BR}}$ is recognized. The delay is solely due to propagation delay and is much shorter than the synchronization delay seen during normal operation. Releasing $\overline{\text{BR}}$ causes $\overline{\text{BG}}$ to be de-asserted asynchronously.

$\overline{\text{BR}}$ must be removed before or coincident with the removal of $\overline{\text{RESET}}$ to ensure proper operation of the processor. In other words, $\overline{\text{BR}}$ can be asserted "during" $\overline{\text{RESET}}$ but $\overline{\text{RESET}}$ should not be asserted "during" (i.e. ending before) a $\overline{\text{BR}}$.

If the bus is requested during $\overline{\text{HALT}}$ or TRAP, the request is latched and serviced after the normal synchronization delay. The processor remains halted, but tristates the busses.

5.3 PROGRAM MEMORY INTERFACE

The program memory interface supports transfers between the ADSP-2100 and program memory using the control lines shown in Figure 5.1. The processor supplies a 14-bit address on the program memory address (PMA) bus. Data or instructions are then transferred across a 24-bit program memory data (PMD) bus. A program memory select pin, $\overline{\text{PMS}}$, indicates that the address bus is being driven and memory can be selected. $\overline{\text{PMS}}$ is asserted on processor cycles in which instructions or data are fetched from program memory. Since the ADSP-2100 always fetches either an instruction or data from program memory, in practice, $\overline{\text{PMS}}$ is asserted continuously; the only exceptions are during $\overline{\text{HALT}}$, TRAP or when the bus is tristated.

Two control lines determine the direction of the transfer. Program Memory Read, $\overline{\text{PMRD}}$, is active low indicating a memory read. $\overline{\text{PMRD}}$ is timed so that it may be used as an output enable signal. Program Memory Write, $\overline{\text{PMWR}}$, corresponds to a memory write. $\overline{\text{PMWR}}$ is timed so that it may be used as a write strobe.

The program memory can be used to store both instructions and data. The processor distinguishes between these two by asserting PMDA (program memory data access) during a data transfer. The timing of PMDA is similar to the PMA bus, allowing PMDA to be used as an additional address bit. When used as the most significant address bit, the processor

can address 32K words of program memory of which 16K is dedicated to data storage. Systems requiring less than 16K of program memory may allocate storage to mixed instructions and data without restriction. The System Builder module of the ADSP-2100 Cross-Software system allows you to define memory use in software during development. The Cross-Software system uses this definition of code and data memory allocation to drive the Simulator, Linker and PROM Splitter.

5.3.1 Program Memory Read Cycle

Program memory reads occur across the program memory interface as follows.

- The ADSP-2100 places the address on the PMA bus, sets PMDA, asserts $\overline{\text{PMS}}$, then asserts $\overline{\text{PMRD}}$. $\overline{\text{PMRD}}$ may be used as an output enable signal. $\overline{\text{PMS}}$ remains asserted without a change of state if it was asserted on the previous cycle.
- Within a specified time period (see data sheet), valid data must be placed on the PMD bus by the memory.
- The ADSP-2100 reads the data on the PMD bus.
- The ADSP-2100 removes $\overline{\text{PMRD}}$ and terminates the cycle.

A timing diagram for this operation is shown in Figure 5.5, on the next page.

5.3.2 Program Memory Write Cycle

Program memory is written with the following sequence of operations.

- The ADSP-2100 places the address on the PMA bus, sets PMDA and asserts $\overline{\text{PMS}}$; $\overline{\text{PMS}}$ remains asserted if already asserted.
- The ADSP-2100 places data on the PMD bus and asserts $\overline{\text{PMWR}}$ for writing.
- The ADSP-2100 removes $\overline{\text{PMWR}}$ and terminates the cycle after a fixed time period.

Figure 5.5, on the next page, depicts the timing diagram for the program memory write operation. Note that $\overline{\text{PMWR}}$ may be used as a write strobe.

5 System Interface

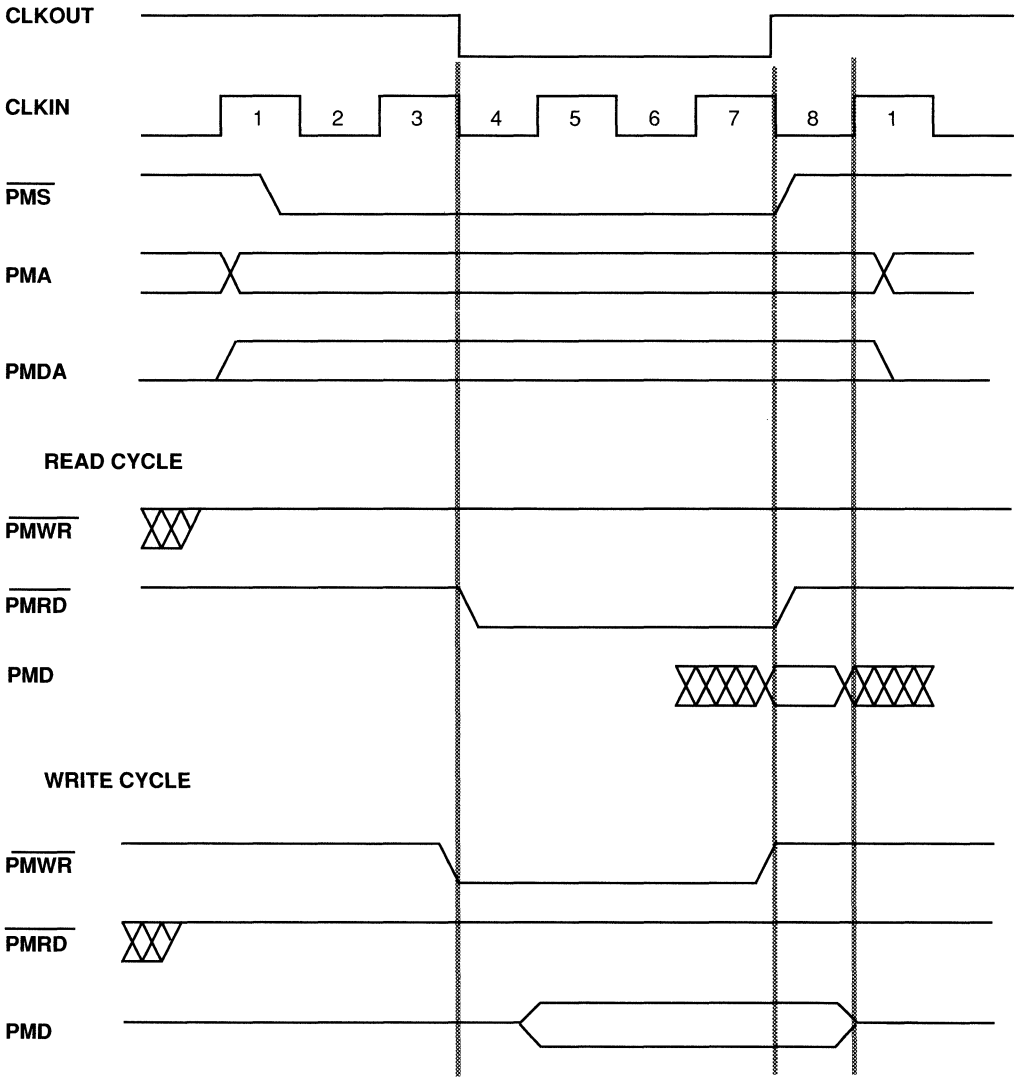


Figure 5.5 Program Memory Read / Write

5.4 DATA MEMORY INTERFACE

The data memory interface supports transfers between the ADSP-2100 and data memory using the control lines shown in Figure 5.1. The processor supplies a 14-bit address on the data memory address (DMA) bus, allowing it to address up to 16K words of data memory. Data is then transferred across a 16-bit data memory data (DMD) bus. Its operation is similar to the program memory interface operation with the following exceptions.

1. There is no equivalent signal to PMDA since only data is stored in data memory.

2. The data memory interface supports slow, memory-mapped peripherals via the DMACK signal.

Data memory access cycles begin with the processor providing a 14-bit address on the DMA bus. A data memory select pin, $\overline{\text{DMS}}$, indicates that the address bus is being driven and memory can be selected. $\overline{\text{DMS}}$ is asserted on processor cycles in which data memory is accessed. $\overline{\text{DMS}}$ remains asserted without a change of state or "glitch" on successive cycles that access data memory. Two control lines determine the direction of the transfer. Data Memory Read, $\overline{\text{DMRD}}$, is active low indicating a memory read. $\overline{\text{DMRD}}$ is timed so that it may be used as an output enable signal. Data Memory Write, $\overline{\text{DMWR}}$, corresponds to a memory write. $\overline{\text{DMWR}}$ is timed so that it may be used as a write strobe.

The data memory access cycle is completed by returning DMACK (data memory acknowledge) to the ADSP-2100. The processor checks DMACK at the end of processor state six. If DMACK is not valid, state seven is extended by one full processor cycle time. This is repeated until DMACK is asserted. With this procedure the ADSP-2100 can readily share memory with slow, memory-mapped peripherals. Of course, during normal full speed memory accesses, DMACK is returned prior to the end of state six and the access completes in a single processor cycle.

Note that the wait for DMACK can prevent the processor from responding to other signals. Interrupts, bus requests and $\overline{\text{HALT}}$ are latched but not serviced during extra cycles required while waiting for DMACK.

5 System Interface

Figure 5.6 shows the timing of typical data memory read and write cycles. Figure 5.7 shows a read cycle stretched one cycle while waiting for DMACK.

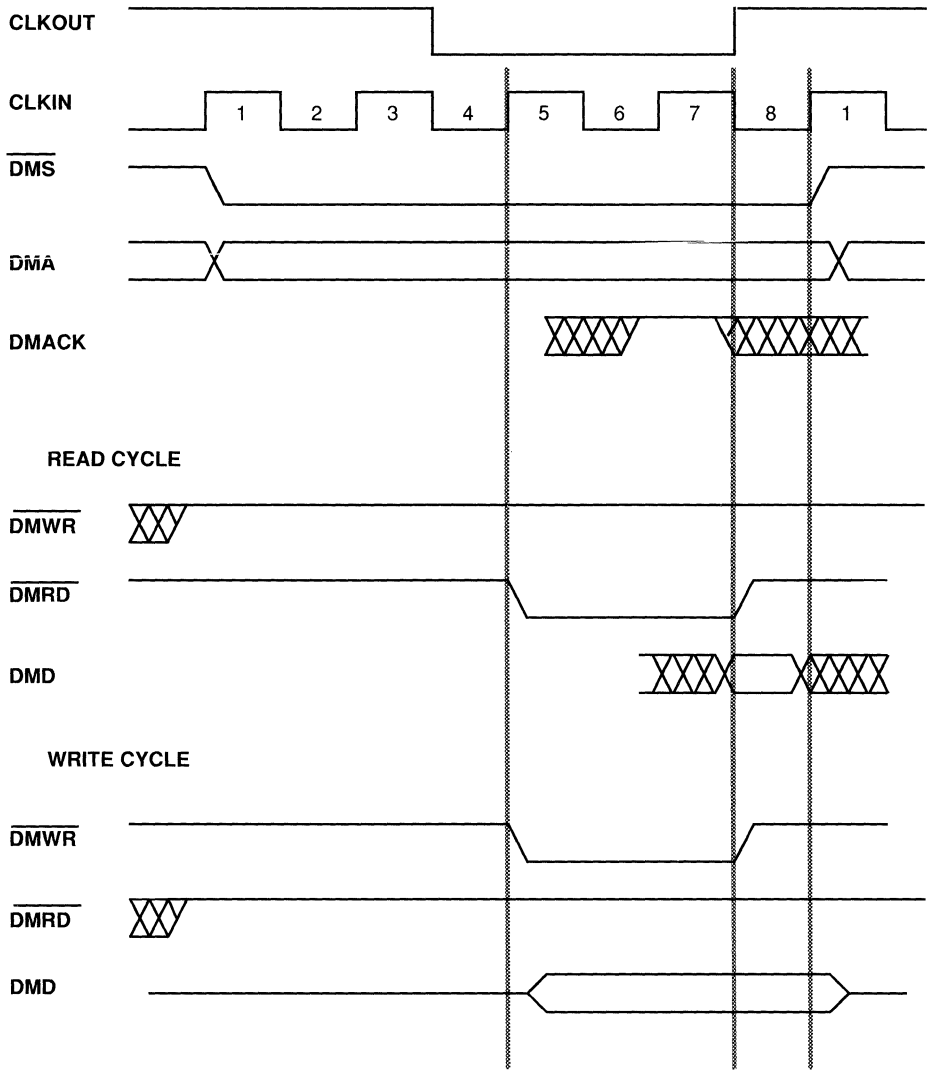


Figure 5.6 Data Memory Read / Write

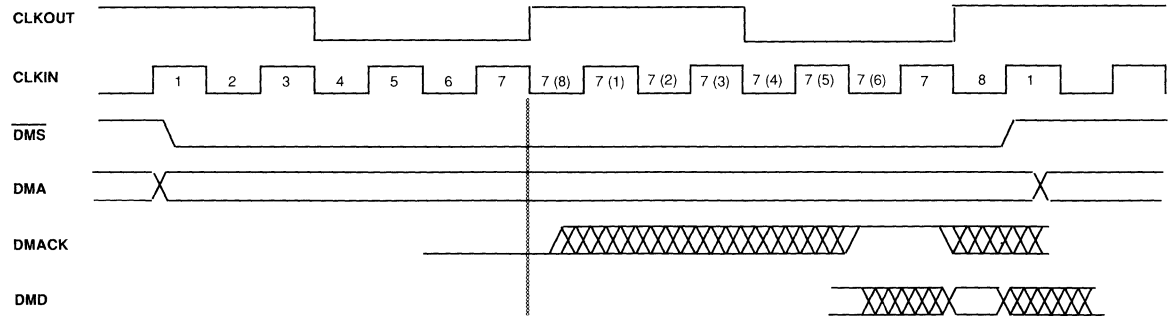


Figure 5.7 Data Memory Read Extended by DMACK

5.4.1 Data Memory Read Cycle

Data memory reads occur across the data memory interface as follows.

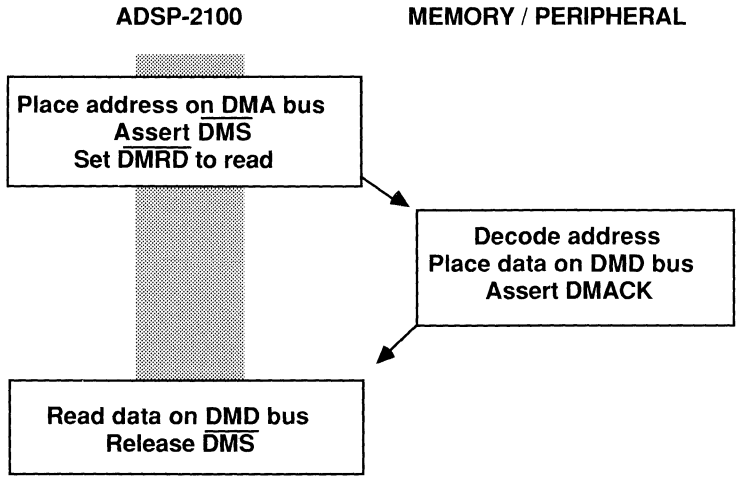


Figure 5.8 Data Memory Read Flowchart

A timing diagram for this operation is shown in Figure 5.6.

5.4.2 Data Memory Write Cycle

Data memory is written with the following sequence of operations.

Figure 5.6 depicts the timing diagram for the data memory write operation. Note that $\overline{\text{DMWR}}$ may be used as a write strobe.

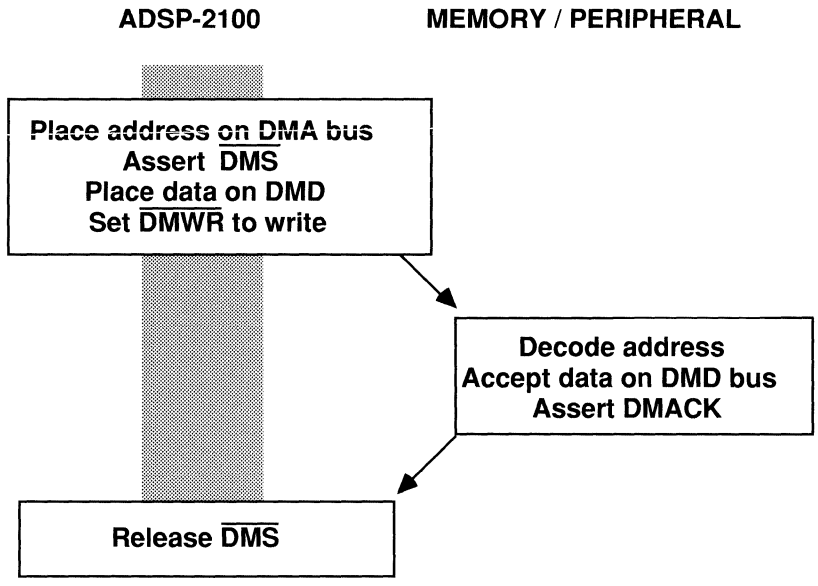


Figure 5.9 Data Memory Write Flowchart

5.5 CONTROL INTERFACE

The control interface consists of three asynchronous signals, $\overline{\text{RESET}}$, $\overline{\text{HALT}}$ and TRAP. These signals allow external control over the activities of the ADSP-2100.

5.5.1 $\overline{\text{RESET}}$

$\overline{\text{RESET}}$ performs a hardware reset on the processor. It must be asserted after power-up to initialize the processor to a known state prior to initiating any program execution. $\overline{\text{RESET}}$ performs the following functions.

- Initializes the internal clock generator.
- Resets all stack pointers (PC stack, Counter stack, Status stack).
- Clears cache memory monitor (invalidates contents).
- Clears all latches ($\overline{\text{IRQ}}$, $\overline{\text{HALT}}$).
- If there is no pending $\overline{\text{BR}}$, PMA is driven with 0004.
- If $\overline{\text{BG}}$ is asserted, all busses remain tristated during $\overline{\text{RESET}}$.
- Masks all interrupts (IMASK = 0000). Note: ICNTL is undefined.
- Clears the MSTAT register. This disables the ALU overflow bit, ALU AR register saturation mode, bit-reversal of DAG1 addresses and use of the alternate register bank.

$\overline{\text{RESET}}$ is recognized on any rising edge of CLKIN and must be asserted for at least four CLKIN cycles. The processor remains in state four during $\overline{\text{RESET}}$. CLKOUT remains low during this period. Upon releasing $\overline{\text{RESET}}$, the processor goes from state four to state five on the second rising edge of CLKIN following release.

Multiple processors operating from the same CLKIN can be synchronized by employing a common $\overline{\text{RESET}}$ line, since then they will all go from state four to state five simultaneously. However in this case, $\overline{\text{RESET}}$ must be externally synchronized to CLKIN first.

See also the discussion of bus request and bus grant during $\overline{\text{RESET}}$ in section 5.2 of this chapter.

5.5.2 $\overline{\text{HALT}}$

$\overline{\text{HALT}}$ is used to temporarily suspend processor operation. It is recognized at the end of state three of the processor cycle. The processor will stop in state eight of the current cycle if it was performing a program memory instruction fetch when the $\overline{\text{HALT}}$ was recognized or in state eight of the

following processor cycle if it was performing a program memory data fetch. In the latter case, the second cycle is a forced program memory instruction fetch even if the instruction is available from cache. Hence, the processor is always halted on a program memory instruction fetch and the controlling device can observe the address where execution was terminated on the PMA bus. Normal processor operation is resumed when the $\overline{\text{HALT}}$ line is released. You must ensure that DMACK is HI when $\overline{\text{HALT}}$ is released.

$\overline{\text{HALT}}$ and $\overline{\text{RESET}}$ are recognized at different points in the processor cycle but they share the common characteristic of synchronization delay mentioned above.

If $\overline{\text{HALT}}$ is asserted during a bus grant, it is latched but not serviced until the ADSP-2100 regains control of the bus; the processor is halted by the bus grant already, of course. Likewise, if $\overline{\text{HALT}}$ is asserted during a wait for DMACK, it is latched but not serviced. Once the processor is halted, $\overline{\text{BR}}$ is latched and serviced normally.

5.5.3 TRAP

The TRAP signal is generated by the processor whenever a TRAP instruction is executed. It is asserted on the transition between state seven and eight of that cycle and the processor is halted in state eight. The PMA bus provides the address of the instruction that follows the TRAP instruction. The TRAP signal will remain active until $\overline{\text{HALT}}$ is asserted. Upon recognition of $\overline{\text{HALT}}$, the ADSP-2100 releases TRAP but remains in a halt state.

If $\overline{\text{BR}}$ is asserted during TRAP it is latched and serviced in the normal sequence, that is, after the required synchronization delay.

Normal operation is resumed when $\overline{\text{HALT}}$ is released as shown in Figure 5.10.

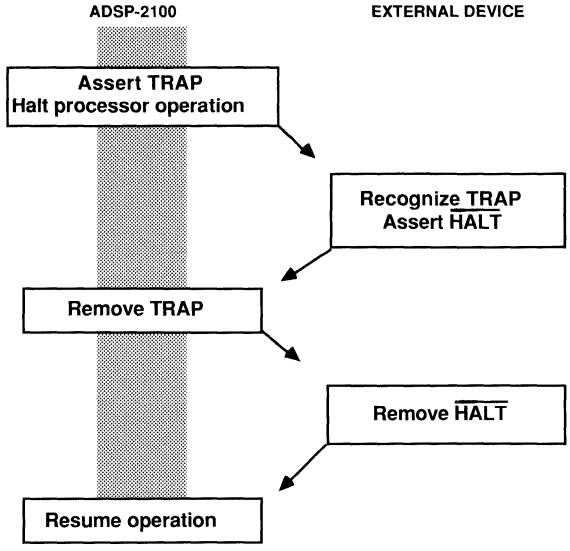


Figure 5.10 TRAP Flowchart

5.6 INTERRUPT OPERATION

The ADSP-2100 supports four prioritized, individually maskable interrupts that can be either level or edge-triggered. Additional information about interrupt masking can be found in Chapter 4, "Program Control."

Level-sensitive interrupts operate by asserting an interrupt request line (active low) until the request is recognized by the processor. The ADSP-2100 checks the interrupt request lines in processor state seven. Once recognized, the request must be negated before returning from the interrupt service routine to prevent being reserviced.

In contrast, edge-triggered interrupt requests are recognized when an inactive-to-active transition occurs on the interrupt line. The ADSP-2100 recognizes a transition by comparing the state of the request line in processor state seven on two successive cycles. Therefore, to guarantee recognition of an asynchronous interrupt, the request must be greater than one processor cycle in duration. The request is latched internal to the processor so that the request line may be held at any level for an arbitrarily long period between interrupts. This latch is automatically cleared when the interrupt is serviced.

5 System Interface

Edge-triggered interrupts require less external hardware compared to level-sensitive requests since there is never a need to hold or negate the request. However, level-sensitive interrupts provide improved noise immunity. Furthermore, many interrupting devices may share a single level-sensitive request line on a wired-OR basis which allows for easy system expansion.

An interrupt request is deemed valid if it is not masked (determined by IMASK) and a higher priority request is not pending. Valid requests invoke an interrupt service sequence that vectors the processor to address 0000 through 0003 for $\overline{IRQ0}$ through $\overline{IRQ3}$ respectively. The interrupt request is recognized at the end of state seven of the processor cycle. There is a synchronization delay associated with the interrupt request lines.

If an interrupt occurs during the two cycles required to execute a program memory data access with invalid cache, it is not recognized between the two cycles, only before or after. Interrupts are latched, but not serviced, during $\overline{HALT}$, TRAP, bus grant ($\overline{BG}$) and while waiting for DMACK. Remember that in order to service an interrupt, the processor must be running and executing instructions.

The masking of interrupts upon entering the interrupt service routine is determined solely by bit 4 of the ICNTL register; see the discussion and Table 4.2 in Chapter 4.

Figure 5.11 shows the interrupt service timing. Edge-sensitive and level-sensitive interrupt requests are serviced similarly except that in the former case, the request line state must be compared on two successive processor cycles to determine the occurrence of an edge. Edge-sensitive interrupts may remain low indefinitely, while level-sensitive interrupts must be removed before executing the RTI instruction.

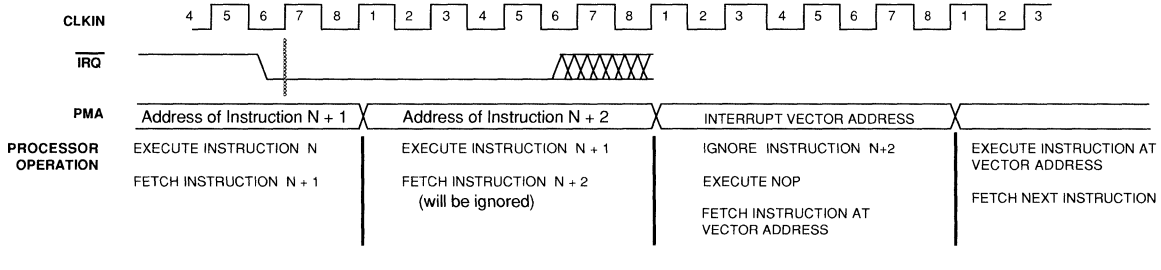


Figure 5.11 Interrupt Service Timing

5.7 PIN DESCRIPTION

This section summarizes the pin description of the processor by interface. When groups of pins are identified with subscripts, as in PMD23-0, the highest numbered pin (PMD23) is the MSB.

Pin Name	*Type*	*Tristate?*	*Function*
Clocks:			
CLKIN	Input	No	Master input clock operating at four times the processor instruction rate. Nominally 50% duty cycle. The phases of CLKIN define the eight internal processor states making up one instruction cycle.
CLKOUT	Output	No	Output clock operating at the processor instruction rate with a 50% duty cycle. Synchronized to the internal processor states.
Interrupt Request Lines:			
$\overline{\text{IRQ3-0}}$	Input	No	Interrupt Request lines that may be either edge triggered or level sensitive. Interrupts are prioritized and individually maskable.
Control Interface:			
$\overline{\text{RESET}}$	Input	No	Master Reset must be asserted for at least four CLKIN cycles to assure proper reset. When $\overline{\text{RESET}}$ is released, execution begins at program memory location 0004.
$\overline{\text{HALT}}$	Input	No	Used to halt the processor. All control signals become inactive and the address and data buses are driven for observation.

5 System Interface

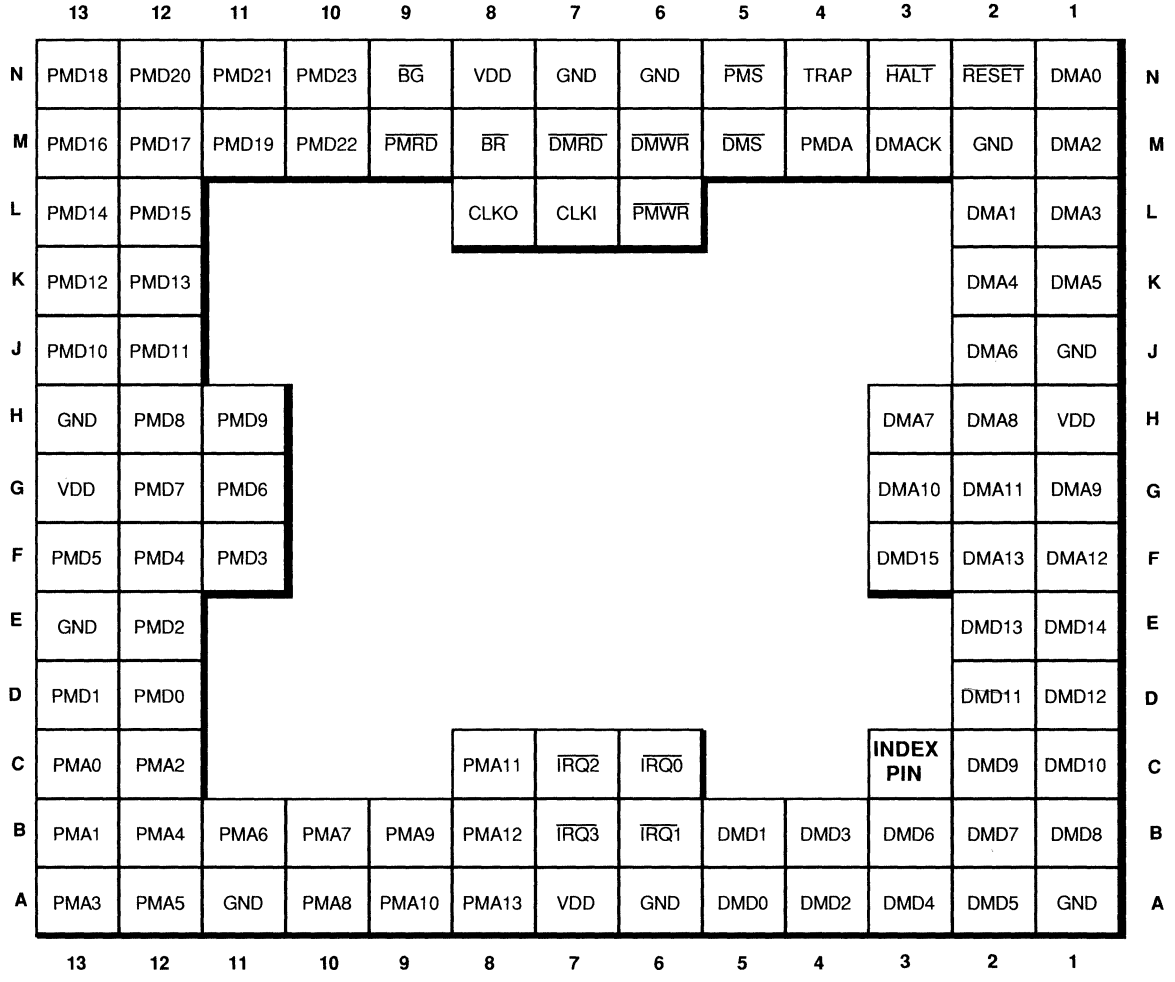
Pin Name	*Type*	*Tristate?*	*Function*
TRAP	Output	No	Used to indicate the execution of a Trap instruction. Remains asserted until $\overline{\text{HALT}}$ is asserted by an external device.
$\overline{\text{BR}}$	Input	No	Bus Request used by an external device to request control of the program and data memory interface. Upon receiving $\overline{\text{BR}}$ the processor halts execution at the completion of the current cycle and relinquishes the program and data memory interface by tristating PMA, PMD, $\overline{\text{PMS}}$, $\overline{\text{PMWR}}$, $\overline{\text{PMRD}}$, PMDA, DMA, DMD, $\overline{\text{DMS}}$, $\overline{\text{DMRD}}$ and $\overline{\text{DMWR}}$. The processor regains control when $\overline{\text{BR}}$ is released.
$\overline{\text{BG}}$	Output	No	Bus Grant. Acknowledges a bus request ($\overline{\text{BR}}$), indicating that the external device may take control. $\overline{\text{BG}}$ is held asserted until $\overline{\text{BR}}$ is released.
Program Memory Interface:			
PMA13-0	Output	Yes	Program Memory Address Bus; tristated when $\overline{\text{BG}}$ is asserted.
PMD23-0	Bidirectional	Yes	Program Memory Data Bus; tristated when $\overline{\text{BG}}$ is asserted.
$\overline{\text{PMS}}$	Output	Yes	Program Memory Select signals a program memory access on the PM interface. Also usable as a chip select signal for external memories. Tristated when $\overline{\text{BG}}$ is asserted.

Pin Name	*Type*	*Tristate?*	*Function*
$\overline{\text{PMRD}}$	Output	Yes	Program Memory Read indicates a read operation on the PM interface. Also usable as a read strobe or output enable signal. Tristated when $\overline{\text{BG}}$ is asserted.
$\overline{\text{PMWR}}$	Output	Yes	Program Memory Write establishes the direction of data transfer on the PM interface. Also usable as a write strobe. Tristated when $\overline{\text{BG}}$ is asserted.
PMDA	Output	Yes	Program Memory Data Access used to distinguish instruction and data fetches from PM. Asserted high when data, as opposed to instruction, is accessed. Also usable as a fifteenth PM address bit. Tristated when $\overline{\text{BG}}$ is asserted.
Data Memory Interface:			
DMA13-0	Output	Yes	Data Memory Address Bus; tristated when $\overline{\text{BG}}$ is asserted.
DMD15-0	Bidirectional	Yes	Data Memory Data Bus; tristated when $\overline{\text{BG}}$ is asserted.
$\overline{\text{DMS}}$	Output	Yes	Data Memory Select signals the a Data Memory Access on the Data Memory interface. Also usable as a chip select signal for external memories. Tristated when $\overline{\text{BG}}$ is asserted

5 System Interface

Pin Name	*Type*	*Tristate?*	*Function*
$\overline{\text{DMRD}}$	Output	Yes	Data Memory Read indicates a read operation on the Data Memory interface. Also usable as a read strobe or output enable signal. Tristated when $\overline{\text{BG}}$ is asserted.
$\overline{\text{DMWR}}$	Output	Yes	Data Memory Write indicates a write operation on the Data Memory interface. Also usable as a write strobe. Tristated when $\overline{\text{BG}}$ is asserted
DMACK	Input	No	Data Memory Acknowledge signal used for asynchronous transfers across the DM interface. Indicates that data memory or memory-mapped peripherals are ready for data transfer. If DMACK is not asserted when checked by the processor, wait states are automatically generated until DMACK is asserted.
Supply Rails:			
VDD	Supply		Power supply rail nominally +5VDC. There are four VDD pins
GND	Ground		Power supply return. There are nine GND pins

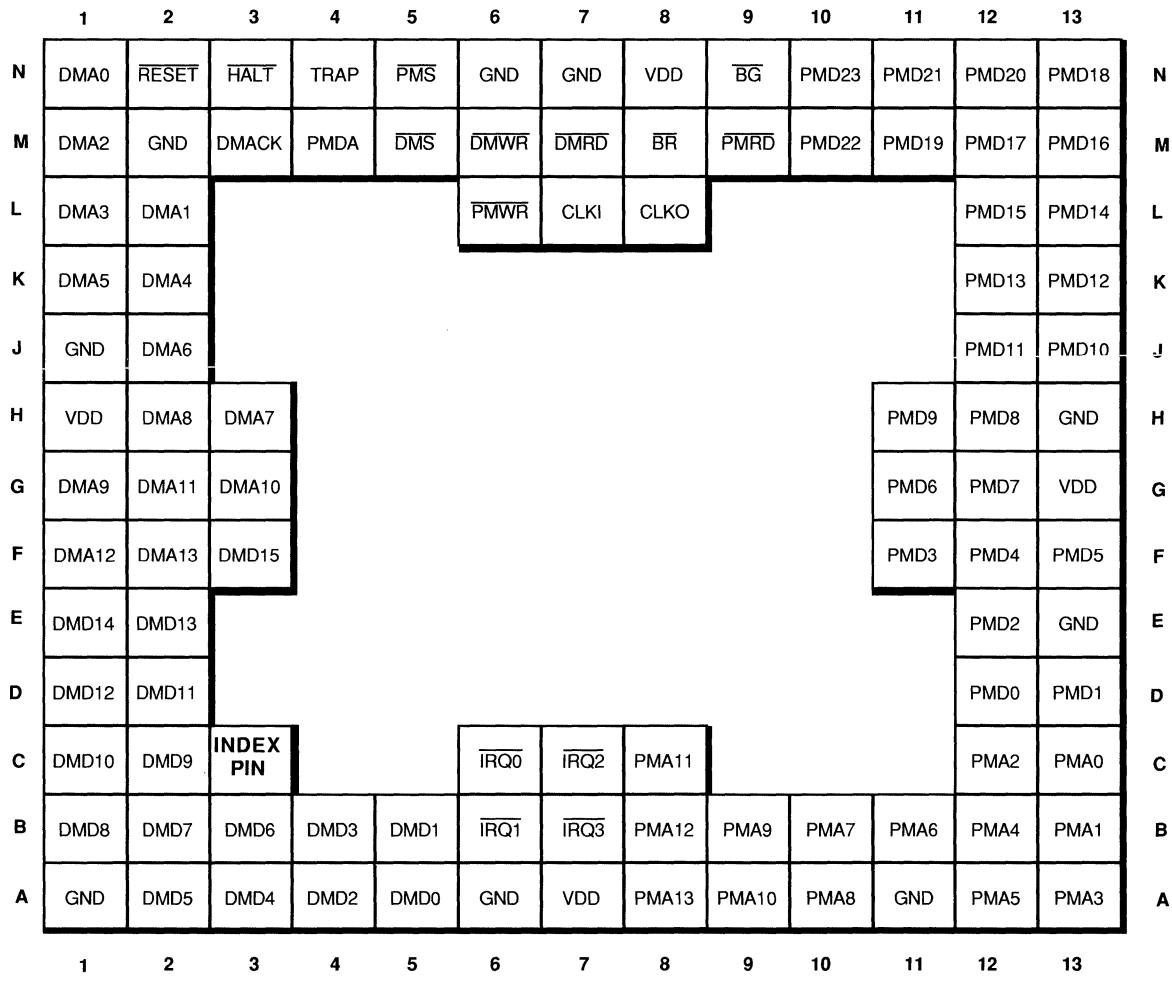
The ADSP-2100 has 100 pins plus an Index pin. Refer to Figure 5.12 and 5.13 for a detailed pinout.



	13	12	11	10	9	8	7	6	5	4	3	2	1	
N	PMD18	PMD20	PMD21	PMD23	$\overline{BG}$	VDD	GND	GND	$\overline{PMS}$	TRAP	$\overline{HALT}$	$\overline{RESET}$	DMA0	N
M	PMD16	PMD17	PMD19	PMD22	$\overline{PMRD}$	$\overline{BR}$	$\overline{DMRD}$	$\overline{DMWR}$	$\overline{DMS}$	PMDA	DMACK	GND	DMA2	M
L	PMD14	PMD15				CLKO	CLKI	$\overline{PMWR}$				DMA1	DMA3	L
K	PMD12	PMD13										DMA4	DMA5	K
J	PMD10	PMD11										DMA6	GND	J
H	GND	PMD8	PMD9								DMA7	DMA8	VDD	H
G	VDD	PMD7	PMD6								DMA10	DMA11	DMA9	G
F	PMD5	PMD4	PMD3								DMD15	DMA13	DMA12	F
E	GND	PMD2										DMD13	DMD14	E
D	PMD1	PMD0										DMD11	DMD12	D
C	PMA0	PMA2				PMA11	$\overline{IRQ2}$	$\overline{IRQ0}$			INDEX PIN	DMD9	DMD10	C
B	PMA1	PMA4	PMA6	PMA7	PMA9	PMA12	$\overline{IRQ3}$	$\overline{IRQ1}$	DMD1	DMD3	DMD6	DMD7	DMD8	B
A	PMA3	PMA5	GND	PMA8	PMA10	PMA13	VDD	GND	DMD0	DMD2	DMD4	DMA5	GND	A
	13	12	11	10	9	8	7	6	5	4	3	2	1	

Figure 5.12 ADSP-2100 Pins, Top View, Pins Down

5 System Interface



	1	2	3	4	5	6	7	8	9	10	11	12	13	
N	DMA0	$\overline{\text{RESET}}$	$\overline{\text{HALT}}$	TRAP	$\overline{\text{PMS}}$	GND	GND	VDD	$\overline{\text{BG}}$	PMD23	PMD21	PMD20	PMD18	N
M	DMA2	GND	DMACK	PMDA	$\overline{\text{DMS}}$	$\overline{\text{DMWR}}$	$\overline{\text{DMRD}}$	$\overline{\text{BR}}$	$\overline{\text{PMRD}}$	PMD22	PMD19	PMD17	PMD16	M
L	DMA3	DMA1				$\overline{\text{PMWR}}$	CLKI	CLKO				PMD15	PMD14	L
K	DMA5	DMA4										PMD13	PMD12	K
J	GND	DMA6										PMD11	PMD10	J
H	VDD	DMA8	DMA7								PMD9	PMD8	GND	H
G	DMA9	DMA11	DMA10								PMD6	PMD7	VDD	G
F	DMA12	DMA13	DMD15								PMD3	PMD4	PMD5	F
E	DMD14	DMD13										PMD2	GND	E
D	DMD12	DMD11										PMD0	PMD1	D
C	DMD10	DMD9	INDEX PIN			$\overline{\text{IRQ0}}$	$\overline{\text{IRQ2}}$	PMA11				PMA2	PMA0	C
B	DMD8	DMD7	DMD6	DMD3	DMD1	$\overline{\text{IRQ1}}$	$\overline{\text{IRQ3}}$	PMA12	PMA9	PMA7	PMA6	PMA4	PMA1	B
A	GND	DMD5	DMD4	DMD2	DMD0	GND	VDD	PMA13	PMA10	PMA8	GND	PMA5	PMA3	A
	1	2	3	4	5	6	7	8	9	10	11	12	13	

Figure 5.13 ADSP-2100 Pins, Bottom View, Pins Up

Instruction Set Overview 6

6.1 INTRODUCTION

This chapter provides an overview of the instruction set used to program the ADSP-2100 and the ADSP-2100 development system software. It provides enough information to understand the nature of programming the ADSP-2100 and the capabilities of the instruction set itself including a programming example at the end of the chapter. This chapter is *not* a complete programmer's reference section.

For actual software development, you must have the *ADSP-2100 Cross-Software Manual* which contains a detailed instruction reference section and a complete guide to the development tools: System Builder, Assembler, Linker, Simulator, PROM Splitter and C Compiler. The two volume *ADSP-2100 Applications Handbook* presents many program examples with source code and discussion; these programs are also available on IBM PC diskettes.

The chip's instruction set is tailored to the computation-intensive algorithms common in DSP applications. For example, sustained single-cycle multiplication / accumulation operations are possible. The instruction set provides full control of the ADSP-2100's three computational units: the ALU, MAC and Shifter. Arithmetic instructions can process single-precision 16-bit operands directly with provisions for multiprecision operations.

The high-level syntax of the ADSP-2100 source code is both efficient and readable. Unlike many assemblers, the ADSP-2100 source code uses an algebraic notation for arithmetic operations and for data moves. There is no performance penalty for this easy-to-read source code. Each program statement assembles into one 24-bit opcode which executes in a single cycle. There are no multicycle instructions in the ADSP-2100 instruction set.

In addition to JUMP and CALL, the control instructions support conditional execution of most arithmetic and a DO UNTIL looping instruction. Two addressing modes are supported for external memory fetches. Direct addressing uses immediate values; indirect addressing uses

6 Instruction Set Overview

the two data address generators (DAGs). All immediate instructions provide the full width for an immediate data (16 bits) or address (14 bits) field.

The 24-bit instruction word allows a high degree of parallelism in performing operations. The instruction set allows for a single-cycle execution of any of the following combinations:

- any ALU, MAC or Shifter operation (may be conditional)
- any register to register move
- any data memory read or write
- a computation with any register to register move
- a computation with any memory read or write
- a computation with a read from both of the two external memories.

The ADSP-2100 instruction set provides the programmer with maximum flexibility. The instruction set provides unrestricted moves from any register to any other register, or from almost any register to/from either external memory. For combining operations, almost any ALU, MAC or Shifter operation may be combined with any register-to-register move or with a register move to or from either external memory.

6.2 INSTRUCTION TYPES

The ADSP-2100 instruction set is grouped into the following categories:

- Multifunction
- Computational: ALU, MAC, Shifter
- Move
- Program Flow/Control
- Miscellaneous

The multifunction instructions best illustrate the power of the ADSP-2100 architecture. In this overview, we begin by examining this group of instructions.

In each section of this chapter are find tables summarizing the syntax of each instruction group. Here is the notation used in those tables.

Instruction Set Overview 6

Square Brackets [ ]	Anything within square brackets is an optional part of the instruction statement.
Parallel Lines | |	Lists of parameters enclosed by parallel vertical lines require the choice of one parameter from among the operands listed.
CAPITAL LETTERS	denote reserved words. These are instruction words, register names and operand selections.
parameters	are shown in small letters and denote an operand in the instruction for which there are numerous choices. For example, the parameter *yop* might have as its choices in the actual instruction: MY0, MY1 or MF.
<data>	denotes an immediate value. Immediate data values may be symbolic names for constants or literal numeric values in binary, octal, hexadecimal or decimal format. The default is decimal.
<reg>	refers to any accessible register; see Table 6.6.
<dreg>	refers to any data register; see Table 6.6.
<address>	denotes an immediate value of an address to be coded in the instruction. The address may be either an immediate value or a LABEL.

6.2.1 Multifunction Instructions

Multifunction operations exploit the inherent parallelism of the ADSP-2100 architecture by providing combinations of data moves, memory reads and memory writes and computation in a single-cycle.

6.2.1.1 ALU/MAC with Data & Program Memory Read

Perhaps the most common single operation in DSP algorithms is the sum of products, like the following:

- Fetch two operands (such as a coefficient and a data point)
- Multiply them and sum the result with previous products

6 Instruction Set Overview

The ADSP-2100 can execute both data fetches and the multiplication/accumulation in a single-cycle. Typically, such a repetitive series can be expressed in ADSP-2100 source code in just a few program lines. Since the cache memory stores up to sixteen contiguous instructions, most loops of this type can execute with sustained single-cycle throughput. An example of such an instruction is:

```
MR=MR+MX0*MY0(UU), MX0=DM(I0,M1), MY0=PM(I4,M5);
```

The first clause of this instruction (up to the first comma) says that MR, the MAC result register, gets the sum of its previous value plus the product of the (current) X and Y input registers of the MAC (MX0 and MY0) both treated as unsigned (UU). Note the simple assignment statement form of the source code.

In the second and third clauses of this multifunction instruction two new operands are fetched. One is fetched from the data memory (DM) pointed to by index register zero (I0, post modified by the value in M1) and the other is fetched from the program memory location (PM) pointed to by I4 (post-modified by M5 in this instance). Note that indirect memory addressing uses a syntax similar to array indexing, with DAG registers providing the index values. Any I register may be paired with any M register within the same DAG.

As discussed in Chapter 2, "Computational Units," registers are read at the beginning of the cycle and written at the end of the cycle. The operands present in the MX0 and MY0 registers at the beginning of the instruction cycle are multiplied and added to the MAC result register, MR. The new operands fetched at the end of this same instruction overwrite the old operands after the multiplication has taken place and are available for computation on the following cycle. You may, of course, load any data registers in conjunction with the computation, not just MAC registers with a MAC operation as in our example.

The computational part of this multifunction instruction may be any unconditional ALU instruction except division or any MAC instruction. Certain other restrictions apply. The X operand must come from Data Memory and the Y operand must come from Program Memory. The result of the computation must go to the result register (MR or AR) not to the feedback register (MF or AF).

6.2.1.2 Data & Program Memory Read

This instruction is a special case of the instruction above, in which the computation is left out. It executes only the dual fetch as shown below.

```
AX0=DM(I2,M0), AY0=PM(I4,M6);
```

In this example, we have used the ALU input registers as the destination. As with the previous multifunction instruction, X operands must come from Data Memory and Y operands from Program Memory.

6.2.1.3 Computation With Memory Read

If a single memory read is performed, instead of the dual memory read of the previous two multifunction instructions, a wider range of computations can be executed. The legal computations include all ALU operations except division, all MAC operations and all Shifter operations except SHIFT IMMEDIATE. Computation must be unconditional.

An example of this instruction is:

```
AR=AX0+AY0, AX0=DM(I0,M3);
```

Here an addition is performed in the ALU while a single operand is fetched from Data Memory. Similar restrictions apply to this instruction as applied to previous multifunction instructions. The value of AX0, used as a source for the computation, is the value at the beginning of the cycle. The data read operation loads a new value into AX0 by the end of the cycle. For this same reason, the destination register (AR in the example above) cannot be the destination for the memory read. If that were legal, the result of the computation would be overwritten by the memory read.

6.2.1.4 Computation With Memory Write

This instruction is quite similar to the immediately preceding one: the order of the clauses in the instruction line, however, is reversed. First the memory write is performed, then the computation as shown below.

```
DM(I0,M0)=AR, AR=AX0+AY0;
```

Again, the value of the source register for the memory write (AR in the example) is the value at the beginning of the instruction. The computation loads a new value into the same register; this is the value in AR at the end of this instruction. Reversing the order of the clauses of the instruction is illegal; it would imply that the result of the computation is written to

memory when, in fact, the previous value of the register is what is written. There is no requirement that the same register be used in this way although this will usually be the case in order to pipeline operands to the computation.

The restrictions on computation operations are identical to those above. All ALU operations except division, all MAC operations and all Shifter operations except SHIFT IMMEDIATE are legal. Computation must be unconditional.

6.2.1.5 Computation With Data Register Move

This final multifunction instruction performs a data register to data register move in parallel with a computation. Most of the restrictions applying to the previous two instructions apply to this instruction.

```
AR=AX0+AY0, AX0=MR2;
```

Here an ALU addition operation occurs while a new value is loaded into AX0 from MR2. As before, the value of AX0 at the beginning of the instruction is the value used in the computation. The move may be from or to all ALU, MAC and Shifter input and output registers except the feedback registers (AF and MF).

The move loads the same register with the new value by the end of the cycle. All ALU operations except division, all MAC operations and all Shifter operations except SHIFT IMMEDIATE are legal. Computation must be unconditional. A complete list of data registers is in Table 6.6.

Here is a table showing the legal combinations for multifunction instructions.

Unconditional Computations		*Data Move DM=DAG1*	*Data Move PM=DAG2*
None or any ALU (except Division) or MAC		DM read	PM read
Any ALU except Division Any MAC Any Shift except Immediate	} {	DM read	—
		—	PM read
		DM write	—
		—	PM write
		Register To Register	

Table 6.1 Summary of Valid Combinations For Multifunction Instructions

Instruction Set Overview 6

Multifunction Instructions

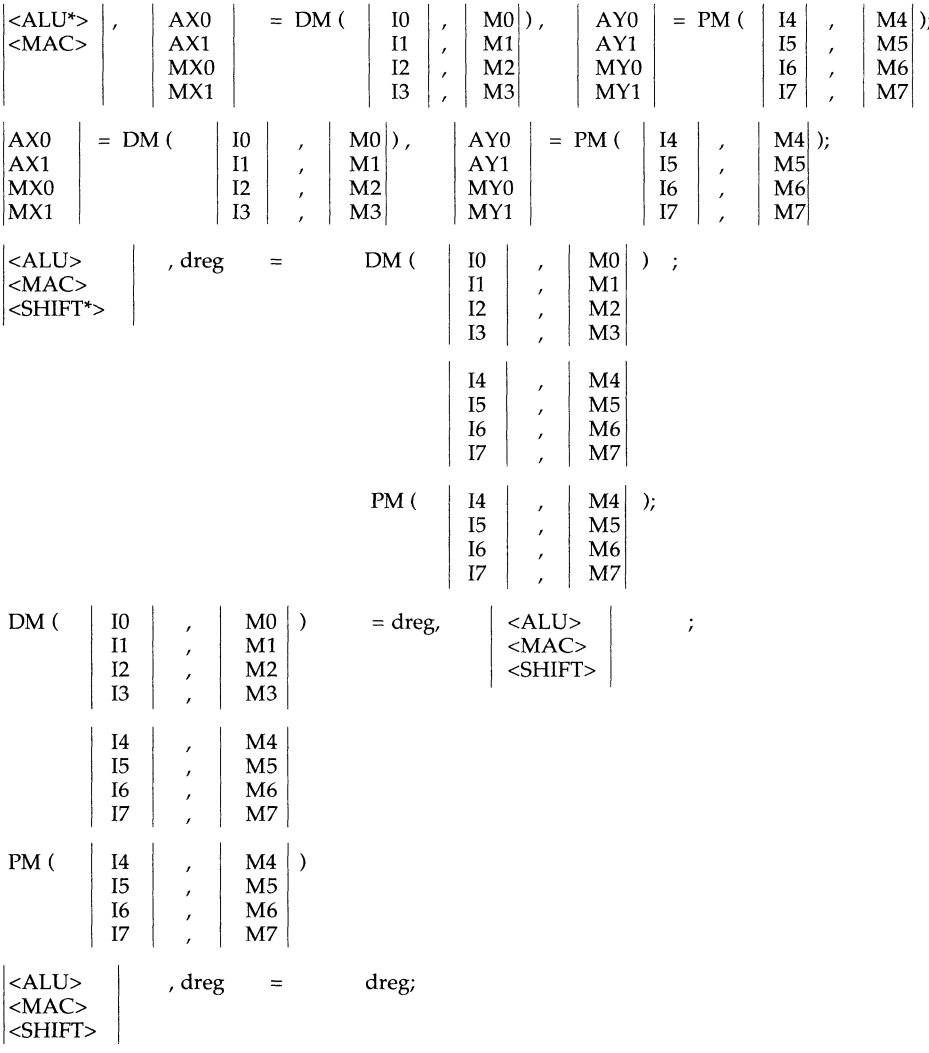


Table 6.2 Multifunction Instructions
*All computation is unconditional; ALU Division and Shift Immediate operations prohibited

6 Instruction Set Overview

6.2.2 ALU, MAC and Shifter Instructions

This group of commands execute all the computation. All of these instructions can be executed conditionally except the ALU division instructions and the Shifter SHIFT IMMEDIATE instructions.

6.2.2.1 ALU Group

Here is a example of one of the ALU instructions, Add/Add with Carry:

```
IF AC AR=AX0+AY0+C;
```

The (optional) conditional expression, IF AC, tests the ALU Carry bit (AC); if there is a carry from the previous instruction, this instruction executes, otherwise a NOP occurs and execution continues with the next instruction. The algebraic expression, AR=AX0+AY0+C, means that the ALU result register (AR) gets the value of the ALU X input and Y input registers plus the value of the carry-in bit.

Here is a summary list of all ALU instructions. In this list, *condition* stands for all the possible conditions that can be tested and *xop* and *yop* stand for the registers that can be specified as input for the ALU. The conditional clause is optional and is enclosed in square brackets to show this.

ALU Instructions

[IF condition]	AR AF	=	xop	+ yop + C + yop + C		;
[IF condition]	AR AF	=	xop	– yop – yop + C – 1		;
[IF condition]	AR AF	=	yop	– xop – xop + C – 1		;
[IF condition]	AR AF	=	xop	AND OR XOR	yop	;
[IF condition]	AR AF	=	PASS	xop yop –1 0 1		;

[IF condition]	AR / AF	=	–	xop / yop	;
[IF condition]	AR / AF	=	NOT	xop / yop	;
[IF condition]	AR / AF	=	ABS	xop	;
[IF condition]	AR / AF	=	yop	+ 1	;
[IF condition]	AR / AF	=	yop	– 1	;

DIVS yop, xop ;
DIVQ xop ;

Table 6.3 ALU Instructions

6.2.2.2 MAC Group

Here is an example of one of the MAC instructions, Multiply/Accumulate:

```
IF NOT MV MR=MR+MX0*MY0(UU);
```

The conditional expression, IF NOT MV, tests the MAC overflow bit. If the condition is not true, a NOP is executed. The expression MR=MR+MX0*MY0 is the multiply/accumulate operation: the multiplier result register (MR) gets the value of itself plus the product of the X and Y input registers selected. The modifier in parentheses (UU) treats the operands as unsigned. There can be only one such modifier selected from the available set. (SS) means both are signed, while (US) and (SU) mean that either the first or second operand is signed; (RND) means to round the result.

6 Instruction Set Overview

Here is a summary list of all MAC instructions. In this list, *condition* stands for all the possible conditions that can be tested and *xop* and *yop* stand for the registers that can be specified as input for the MAC.

MAC Instructions

```
[IF condition]  | MR |  =  xop * yop    (  | SS  |    );
                | MF |                     | SU  |
                                           | US  |
                                           | UU  |
                                           | RND |

[IF condition]  | MR |  =  MR + xop * yop    (  | SS  |    );
                | MF |                          | SU  |
                                                | US  |
                                                | UU  |
                                                | RND |

[IF condition]  | MR |  =  MR – xop * yop    (  | SS  |    );
                | MF |                          | SU  |
                                                | US  |
                                                | UU  |
                                                | RND |

[IF condition]  | MR |  =  0;
                | MF |

[IF condition]  | MR |  =  MR [ ( RND ) ];
                | MF |

IF MV SAT MR;
```

Table 6.4 MAC Instructions

6.2.2.2 Shifter Group

Here is an example of one of the Shifter instructions, Normalize:

```
IF NOT CE SR = SR OR NORM SI (HI);
```

The conditional expression, IF NOT CE, tests the counter. If the condition is not true, a NOP is executed. The destination of all shifting operations is the Shifter Result register, SR. In this example, SI, the Shifter Input register, is the operand. The amount and direction of the shift is controlled by the signed value in the SE register in all shift operations except an immediate shift.

The "SR OR" modifier (which is optional) logically ORs the result with the current contents of the SR register; this allows you to construct a 32-bit value in SR from two 16-bit pieces. "NORM" is the operator and "(HI)" is the modifier that determines whether the shift is relative to the HI or LO half of SR.

The "SR OR" modifier may be omitted. Omitting it is the "PASS" option, although there is no actual PASS modifier.

Here is a summary list of all Shifter instructions. In this list, *condition* stands for all the possible conditions that can be tested.

Shifter Instructions

[IF condition]	SR	=	[SR OR] ASHIFT xop (	HI / LO);
[IF condition]	SR	=	[SR OR] LSHIFT xop (	HI / LO);
[IF condition]	SR	=	[SR OR] NORM xop (	HI / LO);
[IF condition]	SE	=	EXP xop	(HI / LO / HIX);
[IF condition]	SB	=	EXPADJ xop;	
SR	=		[SR OR] ASHIFT xop BY <data>	(HI / LO);
SR	=		[SR OR] LSHIFT xop BY <data>	(HI / LO);

Table 6.5 Shifter Instructions

6 Instruction Set Overview

6.2.3 MOVE: Read & Write

MOVE instructions move data to and from data registers and external memory. ADSP-2100 registers may be viewed as divided into two groups, referred to as *reg* which includes almost all registers and *dreg* or data registers, which is a subset. Only the program counter (PC) and the ALU and MAC feedback registers (AF and MF) are not accessible.

The Table 6.1 shows which registers belong to these groups.

Accessible Registers: ***reg***	*Data Registers:* ***dreg***
SB	AX0, AX1, AY0, AY1, AR
PX	MX0, MX1, MY0, MY1, MR0, MR1, MR2
I0 – I7, M0 – M7, L0 – L7	SI, SE, SR0, SR1
CNTR	
ASTAT, MSTAT, SSTAT	
IMASK, ICNTL	

Table 6.6 ADSP-2100 Register Set: *reg* & *dreg*

MOVE Instructions

```
reg     =   reg;

reg     =   DM (<address>);

dreg    =   DM ( | I0 | , | M0 | );
                 | I1 | , | M1 |
                 | I2 | , | M2 |
                 | I3 | , | M3 |

                 | I4 | , | M4 |
                 | I5 | , | M5 |
                 | I6 | , | M6 |
                 | I7 | , | M7 |

DM ( | I0 | , | M0 | )   =  | dreg   |  ;
     | I1 | , | M1 |        | <data> |
     | I2 | , | M2 |
     | I3 | , | M3 |

     | I4 | , | M4 |
     | I5 | , | M5 |
     | I6 | , | M6 |
     | I7 | , | M7 |
```

```
DM (<address>) =        reg ;

reg      =      <data>;

dreg     =      PM ( | I4 | , | M4 | );
                     | I5 | , | M5 |
                     | I6 | , | M6 |
                     | I7 | , | M7 |

PM ( | I4 | , | M4 | )      =      dreg ;
     | I5 | , | M5 |
     | I6 | , | M6 |
     | I7 | , | M7 |
```

Table 6.7 MOVE Instructions

6.2.4 Program Flow Control

Program Flow Control on the ADSP-2100 is simple and powerful. The discussion of the Program Sequencer in this manual gives an example of each type of control statement. Here is an example of one such statement.

```
IF EQ JUMP my_label;
```

JUMP, of course, is a familiar construct from many other processors. *My_label* is any identifier you wish to use as a label for the entry point of the code jumped to. Instead of the label, an index register in DAG2 may be explicitly used.

If the counter condition (CE, NOT CE) is to be used, an assignment to CNTR must be executed to initialize the counter value.

The default scope for any label is the module in which it is declared. The Assembler directive .ENTRY makes a label "visible" as an entry point for routines outside the module. Conversely, the .EXTERNAL directive makes it possible to use a label declared in another module.

On the next page is the summary of all program flow control instructions.

6 Instruction Set Overview

Program Flow Control Instructions

[IF condition]	JUMP	(I4) (I5) (I6) (I7) <address>	;
[IF condition]	CALL	(I4) (I5) (I6) (I7) <address>	;
[IF condition]	RTS ;		
[IF condition]	RTI ;		
DO <address> [UNTIL termination] ;			
[IF condition]	TRAP ;		

Table 6.8 Program Flow Control Instructions

6.2.5 Miscellaneous Instructions

There are four Miscellaneous instructions. NOP, of course, is a no operation instruction. The PUSH/POP instruction allows you to explicitly set or control the status, counter, PC and loop stacks; interrupt servicing automatically pushes and pops some of these stacks.

The Enable/Disable instruction turns on and off four modes of operation: bit-reversal on DAG1, latching ALU overflow, saturating ALU results and choosing the primary or shadow register set.

The MODIFY instruction modifies the address pointer in the I register selected with the value in the selected M register, without performing any actual memory access. As always, the I and M registers must be from the same DAG; any of I0-I3 may be used only with one from M0-M3 and the same for I4-I7 and M4-M7.

Miscellaneous Instructions

```
NOP;

| PUSH | STS  [, POP CNTR] [, POP PC] [, POP LOOP] ;
| POP  |

| ENA | BIT_REV     , | ENA | AV_LATCH     , | ENA | AR_SAT     , | ENA | SEC_REG ;
| DIS |               | DIS |               | DIS |               | DIS |

MODIFY ( | I0 | , | M0 | ) ;
         | I1 | , | M1 |
         | I2 | , | M2 |
         | I3 | , | M3 |

         | I4 | , | M4 |
         | I5 | , | M5 |
         | I6 | , | M6 |
         | I7 | , | M7 |
```

Table 6.9 Miscellaneous Instructions

6.3 DATA STRUCTURES

The ADSP-2100 Cross-Software supports the declaration and use of a simple set of data structures: one-dimensional arrays and ports. The array may be a single value or multiple values. In addition, the array may be used as a circular buffer. Here is a brief discussion of each instance with an example of how they are declared and used. Complete syntax for these and other directives is given in the *ADSP-2100 Cross-Software Manual.*

6.3.1 Arrays

Arrays are the basic data structure in the ADSP-2100 instruction set. In ADSP-2100 literature, the words "array" and the expression "data buffer" are used interchangeably. Arrays are declared with Assembler directives and can be referenced indirectly and by name, can be initialized from immediate values in a directive or from external data files and can be linear or circular with automatic wraparound.

An array is declared with a directive such as

```
.VAR/DM coefficients[128];
```

This declares an array of 128 16-bit values located in data memory (DM). The special operators ^ and % reference the address and length, respectively, of the array. It could be referenced as shown below.

```
I0 = ^coefficients;  {point to address of buffer}
MX0=DM(I0,M0);       {load MX0 from buffer}
```

These instructions load a value into MX0 from the beginning of the *coefficients* buffer in data memory. With the automatic post-modify of the DAGs, you could execute the second of these instructions in a loop and continuously advance through the buffer.

Alternatively, when you only need to address the first location, you can directly use the buffer name as a label in many circumstances, such as

```
MX0=DM(coefficients);
```

The Linker substitutes the actual address for the label. It is also possible to initialize a complete array/buffer from a data file, using the INIT directive.

```
.INIT coefficients : <filename.dat>;
```

This reads the values from the file *filename.dat* into the array at link time.

An array or data buffer with a length of one behaves like a simple single-word variable.

6.3.2 Circular Arrays/Buffers

A common requirement in DSP is the circular buffer. This is directly implemented by the ADSP-2100 DAGs, using the L (length) registers. First, you must declare the buffer as circular:

```
.VAR/DM/CIRC coefficients[128];
```

This identifies it to the Linker for placement on the proper address boundary. Next, you must initialize the L register, typically using the % operator (or a constant) and, in the example below, the I register.

```
L0 = %coefficients;  {length of circular buffer}
I0 = ^coefficients;  {point to address of buffer}
```

Now a statement like

```
MX0=DM(I0,M0);        {load MX0 from buffer}
```

in a loop, cycles continuously through *coefficients* and wraps around automatically. L registers should be initialized to zero for buffers of any length that are not circular.

6.3.3 Ports & Memory-Mapping

The .PORT directive in the System Builder module allows you to refer to a specific hardware address with an identifier of your choosing as shown here. This capability makes it easy to interface to memory-mapped peripherals, such as converters.

```
.PORT/ABS=16382 converter_in;
```

After declaring the same identifier in the Assembler, a value can be read directly from the port with a statement like

```
SI = DM(converter_in);
```

This loads the SI register with the value present at the address specified in the System Builder. (The Linker reads the Architecture Description file produced by the System Builder to obtain the actual address for the label.) You can change the hardware address of the port without having to rewrite your program.

6 Instruction Set Overview

6.4 PROGRAM EXAMPLE

Below are three listings, showing an example of an FIR filter program written for the ADSP-2100 with discussion of each section of the program. This FIR filter program demonstrates much of the conceptual power of the ADSP-2100 architecture and instruction set. More complex programs would, of course, exercise many additional features of the language.

```
   .MODULE/ROM/ABS=0        main_routine;
A  .INCLUDE                 <const.h>;              {include file of constants}
B{ .VAR/DM/RAM/CIRC         data_buffer[taps];      {length "taps" defined in const.h}
   .VAR/PM/RAM/CIRC         coefficient[taps];
   .GLOBAL                  data_buffer, coefficient; {these buffers are global}
   .EXTERNAL fir_start;                             {external routine entry label}
   .INIT                    coefficient:<coeff.dat>; {initialize with values in file}

   {start code section}
   {load interrupt vector addresses}
                            JUMP fir_start;         {vectored address of interrupt 0}
C ------------------------  RTI;                    {no interrupt 1}
                            RTI;                    {no interrupt 2}
                            RTI;                    {no interrupt 3}
   {initializations}
                            L0=%data_buffer;        {setup circular buffer length}
                            L1=0;
                            L2=0;
                            L3=0;
D ------------------------  L4=%coefficient;        {setup circular buffer length}
                            L5=0;
                            L6=0;
                            L7=0;
                            M0=1;                   {set modifier registers}
                            M4=1;

                            I0 = ^data_buffer;      {point to buffer start}
                            I4 = ^coefficient;      {point to table start}

                            CNTR = %data_buffer;    {set CNTR with buffer length}
                            DO clear UNTIL CE;      {setup loop}
   clear:                      DM(I0,M0)=0;         {initialize data buffer}

                            IMASK = B#1111;         {now activate interrupts}
   mainloop:                JUMP mainloop;          {infinite loop, waits for interrupts}

   .ENDMOD;
```

Setup and Main Loop Routine

```
.CONST                  taps = 15, taps_less_one = 14;
```

Include File, Constant Initialization

Figure 6.10 Program Example Listing 1, Main Routine & Constants File

6.4.1 Example Program: Setup Routine Discussion

The setup and main loop routine, Figure 6.10, does initialization and then loops continuously. The filter itself is interrupt-driven. When an interrupt occurs (this example uses interrupt zero) control shifts to the subroutine.

Line A shows that the constant declarations are contained in a separate file.

Section B shows the directives defining the two circular buffers: one in data memory RAM (used to hold a delay line of samples) and one in program memory RAM (used to store coefficients for the filter). The coefficients are actually loaded from an external file by the Linker. These values can be changed without reassembling; only another linking pass is required.

Section C shows the setup of the interrupts. Interrupt vectors are the first four memory locations in the ADSP-2100 program memory space. Since this module (first line) is located at absolute address zero, the first four instructions occupy these interrupt vector locations. Only the first location is used; it jumps to the subroutine. The other three are placeholders; program execution begins at address 004.

Section D sets up the Index, Length and Modify registers used to address the circular buffer. A non-zero value for length activates the modulus logic. Each time the interrupt occurs, the pointers advance one position "around" the buffer.

6 Instruction Set Overview

```
    .MODULE/ROM  fir_routine;                          {relocatable fir interrupt module}

    .INCLUDE                <const.h>;                 {include constant declaration file}
E   .PORT                   ad_sample;                 {AD port you defined in Systm Bldr}
    .PORT                   da_data;                   {DA port you defined in Systm Bldr}
    .ENTRY                  fir_start;                 {make label visible outside module}
    .EXTERNAL               data_buffer, coefficient;  {make globals accessible in module}

    {interrupt service routine code section}

    FIR_START:              CNTR = taps_less_one;      {N-1 passes within DO UNTIL}
                            SI = DM(ad_sample);        {read from port}
                            DM(I0,M0) = SI;            {transfer data to buffer}
F                           MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0);
                                                       {set up multiplier for loop}
G                           DO convolution UNTIL CE;   {CE = counter expired}
    convolution:                MR=MR+MX0*MY0(SS), MY0=PM(I4,M4), MX0=DM(I0,M0);
                                                       {MAC these, fetch next}
                            MR=MR+MX0*MY0(RND);        {Nth pass with rounding}
                            IF MV SAT MR;              {saturate if overflowed}
                            DM(da_data) = MR1;         {write to port}

                            RTI;                       {return from interrupt}
    .ENDMOD;
```

Figure 6.11 Program Example Listing 2, Interrupt Routine

6.4.2 Example Program: Interrupt Routine Discussion

This subroutine reads the sample and transfers it to the next location in the circular buffer (overwriting the oldest sample). Then all samples and coefficients are multiplied and the products are accumulated to produce the next output value. The subroutine checks for overflow and saturates the output value to the appropriate full scale then writes the result to the converter out port and returns.

The lines labelled with E declare the symbolic names used to reference the A/D and D/A ports. You select these names and define their hardware locations with the System Builder, a tool described in detail in the *ADSP-2100 Cross-Software Manual.*

The subroutine begins by loading the counter register CNTR. The new sample is read from the port into the SI register; the choice of SI is of no particular significance. Then, the data is written into the data buffer. Because of the automatic circular buffer addressing, the new data overwrites the oldest sample. The *N* most recent samples are always in the buffer.

Line F zeroes the multiplier result register (MR) and fetches the first two operands. Because this instruction access both memories, a one-cycle overhead occurs to fetch the next instruction.

G labels the loop itself, consisting of only two lines, one setting up the loop and one instruction "inside" the loop. The MAC instruction multiplies and accumulates the previous set of operands while fetching the next ones from each memory. This instruction also accesses both memories and so another one cycle penalty will be incurred. However, since the loop fits entirely the cache, this overhead penalty happens only once. The loop executes out of cache after the instruction has been loaded once.

The last MAC instruction (the first one outside the loop) performs the final multiplication/accumulation using the round modifier (RND) instead of the signed-by-signed modifier (SS).

Instruction Coding A

A.1 OPCODES

Here is a summary of the complete instruction set of the ADSP-2100. Following the list of types and codes shown immediately below is a key to the abbreviations used. Any instruction codes not shown are reserved for future use.

Type 1: ALU / MAC with Data & Program Memory Read

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	PD		DD		AMF					Yop		Xop			PM I		PM M		DM I		DM M	

Type 2: Data Memory Write (Immediate Data)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	G	DATA																I		M	

Type 3: Read /Write Data Memory (Direct Address)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	D	RGP		ADDR														REG			

Type 4: ALU / MAC with Data Memory Read / Write

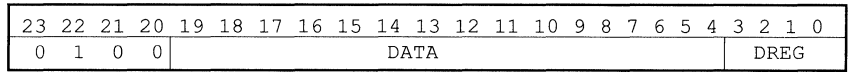
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	G	D	Z	AMF					Yop		Xop			DREG				I		M	

Type 5: ALU / MAC with Program Memory Read / Write

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	D	Z	AMF					Yop		Xop			DREG				I		M	

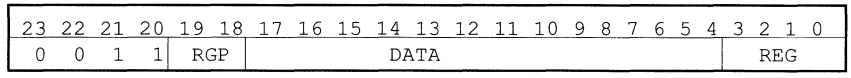
A Instruction Coding

Type 6: Load Data Register Immediate



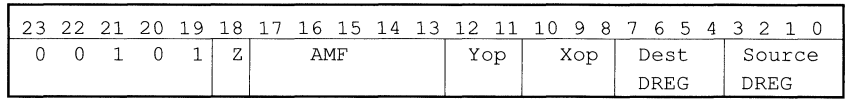
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	DATA																DREG			

Type 7: Load Non-Data Register Immediate



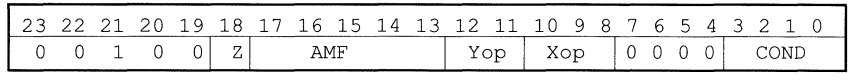
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	RGP		DATA														REG			

Type 8: ALU / MAC with Internal Data Register Move



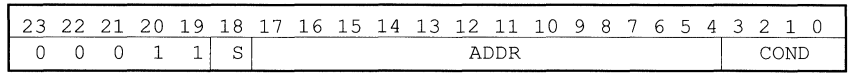
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Z	AMF					Yop		Xop			Dest DREG				Source DREG			

Type 9: Conditional ALU / MAC



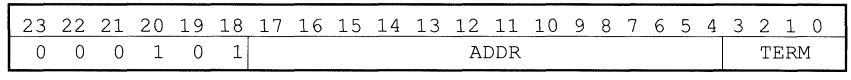
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					Yop		Xop			0	0	0	0	COND			

Type 10: Conditional Jump (Direct Address)



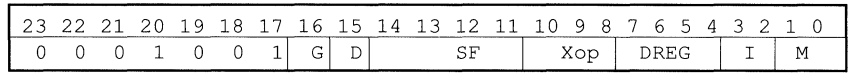
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	S	ADDR														COND			

Type 11: Do Until



23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	ADDR														TERM			

Type 12: Shift with Data Memory Read / Write



23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	G	D	SF				Xop			DREG				I	M		

Instruction Coding A

Type 13: Shift with Program Memory Read / Write

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	D	SF				Xop			DREG				I		M	

Type 14: Shift with Internal Data Register Move

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	x	SF				Xop			Dest DREG				Source DREG			

Type 15: Shift Immediate

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	SF				Xop			exponent							

Type 16: Conditional Shift

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

Type 17: Internal Data Move

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	0	DST RGP		SRC RGP		Dest REG				Source REG			

Type 18: Mode Control

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	0	0	0	0	AS MCC		OL MCC		BR MCC		SR MCC		0	0	0	0

Type 19: Conditional Jump (Indirect Address)

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	I		0	S	COND			

A Instruction Coding

Type 20: Conditional Return

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	T	COND			

Type 21: Modify Address Register

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	G	I		M	

Type 22: Conditional Trap

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	COND			

Type 23: DIVQ

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	0	0	1	0	Xop			0	0	0	0	0	0	0	0

Type 24: DIVS

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	Yop		Xop			0	0	0	0	0	0	0	0

Type 25: Saturate MR

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Type 26: Stack Control

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	PP		LP	CP	SPP

Instruction Coding A

Type 27: Reserved

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Type 28: Reserved

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Type 29: Reserved

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Type 30: No Operation

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A.2 ABBREVIATION CODING

AMF ALU / MAC Function codes

0 0 0 0 0 No operation

MAC Function codes

0 0 0 0 1	X * Y	(RND)	
0 0 0 1 0	MR + X * Y	(RND)	
0 0 0 1 1	MR – X * Y	(RND)	
0 0 1 0 0	X * Y	(SS)	Clear when y = 0
0 0 1 0 1	X * Y	(SU)	
0 0 1 1 0	X * Y	(US)	
0 0 1 1 1	X * Y	(UU)	
0 1 0 0 0	MR + X * Y	(SS)	
0 1 0 0 1	MR + X * Y	(SU)	
0 1 0 1 0	MR + X * Y	(US)	
0 1 0 1 1	MR + X * Y	(UU)	

A Instruction Coding

0 1 1 0 0	MR – X * Y	(SS)
0 1 1 0 1	MR – X * Y	(SU)
0 1 1 1 0	MR – X * Y	(US)
0 1 1 1 1	MR – X * Y	(UU)

ALU Function codes

1 0 0 0 0	Y	Clear when y = 0
1 0 0 0 1	Y + 1	
1 0 0 1 0	X + Y + C	
1 0 0 1 1	X + Y	X when y = 0
1 0 1 0 0	NOT Y	
1 0 1 0 1	– Y	
1 0 1 1 0	X – Y + C – 1	
1 0 1 1 1	X – Y	
1 1 0 0 0	Y – 1	
1 1 0 0 1	Y – X	– X when y = 0
1 1 0 1 0	Y – X + C – 1	
1 1 0 1 1	NOT X	
1 1 1 0 0	X AND Y	
1 1 1 0 1	X OR Y	
1 1 1 1 0	X XOR Y	
1 1 1 1 1	ABS X	

COND Status Condition codes

0 0 0 0	Equal	EQ
0 0 0 1	Not equal	NE
0 0 1 0	Greater than	GT
0 0 1 1	Less than or equal	LE
0 1 0 0	Less than	LT
0 1 0 1	Greater than or equal	GE
0 1 1 0	ALU Overflow	AV
0 1 1 1	NOT ALU Overflow	NOT AV
1 0 0 0	ALU Carry	AC
1 0 0 1	Not ALU Carry	NOT AC
1 0 1 0	X input sign negative	NEG
1 0 1 1	X input sign positive	POS
1 1 0 0	MAC Overflow	MV
1 1 0 1	Not MAC Overflow	NOT MV
1 1 1 0	Not counter expired	NOT CE
1 1 1 1	Always	TRUE

Instruction Coding A

CP Counter Stack Pop codes

0	No change
1	Pop

D Memory Access Direction codes

0	Read
1	Write

DD Double Data Fetch Data Memory Destination codes

0 0	AX0
0 1	AX1
1 0	MX0
1 1	MX1

DREG Data Register codes

0 0 0 0	AX0
0 0 0 1	AX1
0 0 1 0	MX0
0 0 1 1	MX1
0 1 0 0	AY0
0 1 0 1	AY1
0 1 1 0	MY0
0 1 1 1	MY1
1 0 0 0	SI
1 0 0 1	SE
1 0 1 0	AR
1 0 1 1	MR0
1 1 0 0	MR1
1 1 0 1	MR2
1 1 1 0	SR0
1 1 1 1	SR1

G Data Address Generator codes

0	DAG1
1	DAG2

A Instruction Coding

I Index Register codes

G =	0	1
0 0	I0	I4
0 1	I1	I5
1 0	I2	I6
1 1	I3	I7

LP Loop Stack Pop codes

0	No Change
1	Pop

M Modify Register codes

G =	0	1
0 0	M0	M4
0 1	M1	M5
1 0	M2	M6
1 1	M3	M7

MCC Mode Control codes

SR:	Secondary register bank mode
BR:	Bit-reverse mode
OL:	ALU overflow latch mode
AS:	AR register saturate mode

0 0	No change
0 1	No change
1 0	Deactivate
1 1	Activate

PD Double Data Fetch Program Memory Destination codes

0 0	AY0
0 1	AY1
1 0	MY0
1 1	MY1

Instruction Coding A

PP PC Stack Pop codes

0	No Change
1	Pop

REG Register codes

RGP =	00	01	10	11
0 0 0 0	AX0	I0	I4	ASTAT
0 0 0 1	AX1	I1	I5	MSTAT
0 0 1 0	MX0	I2	I6	SSTAT
0 0 1 1	MX1	I3	I7	IMASK
0 1 0 0	AY0	M0	M4	ICNTL
0 1 0 1	AY1	M1	M5	CNTR
0 1 1 0	MY0	M2	M6	SB
0 1 1 1	MY1	M3	M7	PX
1 0 0 0	SI	L0	L4	
1 0 0 1	SE	L1	L5	
1 0 1 0	AR	L2	L6	
1 0 1 1	MR0	L3	L7	
1 1 0 0	MR1			
1 1 0 1	MR2			
1 1 1 0	SR0			
1 1 1 1	SR1			

S Jump Type codes

0	Jump
1	Jump Subroutine

A Instruction Coding

SF Shifter Function codes

0 0 0 0	LSHIFT	(HI, PASS)
0 0 0 1	LSHIFT	(HI, OR)
0 0 1 0	LSHIFT	(LO, PASS)
0 0 1 1	LSHIFT	(LO, OR)
0 1 0 0	ASHIFT	(HI, PASS)
0 1 0 1	ASHIFT	(HI, OR)
0 1 1 0	ASHIFT	(LO, PASS)
0 1 1 1	ASHIFT	(LO, OR)
1 0 0 0	NORM	(HI, PASS)
1 0 0 1	NORM	(HI, OR)
1 0 1 0	NORM	(LO, PASS)
1 0 1 1	NORM	(LO, OR)
1 1 0 0	EXP	(HI)
1 1 0 1	EXP	(HIX)
1 1 1 0	EXP	(LO)
1 1 1 1	Block Exponent Adjust	

SPP Status Stack Push/Pop codes

0 0	No change
0 1	No change
1 0	Push
1 1	Pop

T Return Type codes

0	Return from Subroutine
1	Return from Interrupt

TERM Termination codes for DO UNTIL

0 0 0 0	Not equal	NE
0 0 0 1	Equal	EQ
0 0 1 0	Less than or equal	LE
0 0 1 1	Greater than	GT
0 1 0 0	Greater than or equal	GE
0 1 0 1	Less than	LT
0 1 1 0	Not ALU Overflow	NOT AV
0 1 1 1	ALU Overflow	AV
1 0 0 0	Not ALU Carry	NOT AC

1 0 0 1	ALU Carry	AC
1 0 1 0	X input sign positive	POS
1 0 1 1	X input sign negative	NEG
1 1 0 0	Not MAC Overflow	NOT MV
1 1 0 1	MAC Overflow	MV
1 1 1 0	Counter expired	CE
1 1 1 1	Always	FOREVER

X X Operand codes

0 0 0	X0	(SI for Shifter)
0 0 1	X1	(Not for Shifter)
0 1 0	AR	
0 1 1	MR0	
1 0 0	MR1	
1 0 1	MR2	
1 1 0	SR0	
1 1 1	SR1	

Y ALU/MAC Y Operand codes

0 0	Y0	
0 1	Y1	
1 0	F	(feedback register)
1 1	zero	

Z ALU/MAC Result Register codes

0	Result register
1	Feedback register

Division Exceptions B

B.1 DIVISION FUNDAMENTALS

The ADSP-2100's instruction set contains two instructions for implementing a non-restoring divide algorithm. These instructions take as their operands twos-complement or unsigned numbers, and in 16 cycles produce a truncated quotient of 16 bits. For most numbers and applications, these primitives produce the correct results. However, there are certain situations where results produced will be off by one LSB. This appendix documents these situations, and presents alternatives for producing the correct results.

Computing a 16-bit fixed point quotient from two numbers is accomplished by 16 executions of the DIVQ instruction for unsigned numbers. Signed division uses the DIVS instruction first, followed by 15 DIVQs. Whichever division you perform, both input operands must be of the same type (signed or unsigned) and produce a result of the same type.

These two instructions are used to implement a conditional add/subtract, non-restoring division algorithm. As its name implies, the algorithm functions by adding or subtracting the divisor to/from the dividend. The decision as to which operation is perform is based on the previously generated quotient bit. Each add/subtract operation produces a new partial remainder, which will be used in the next step.

The phrase non-restoring refers to the fact that the final remainder is not correct. With a restoring algorithm, it is possible, at any step, to take the partial quotient, multiply it by the divisor, and add the partial remainder to recreate the dividend. With this non-restoring algorithm, it is necessary to add two times the divisor to the partial remainder if the previously determined quotient bit is zero. It is easier to compute the remainder using the multiplier than in the ALU.

B.1.1 Signed Division

Signed division is accomplished by first storing the 16-bit divisor in an X register (AX0, AX1, AR, MR2, MR1, MR0, SR1, or SR0). The 32-bit dividend must be stored in two separate 16-bit registers. The lower 16-bits must be stored in AY0, while the upper 16-bits can be in AY1 or AF.

B Division Exceptions

The DIVS primitive is executed once, with the proper operands (ex. DIVS AY1, AX0) to compute the sign of the quotient. The sign bit of the quotient is determined by XORing (exclusive-or) the sign bits of each operand. The entire 32-bit dividend is shifted left one bit. The lower 15 bits of the dividend with the recently determined sign bit appended are stored in AY0, while the lower 15 bits of the upper word, with the MSB of the lower word appended is stored in AF.

To complete the division, 15 DIVQ instructions are executed. Operation of the DIVQ primitive is described below

B.1.2 Unsigned Division

Computing an unsigned division is done like signed division, except the first instruction is not a DIVS, but another DIVQ. The upper word of the dividend must be stored in AF, and the AQ bit of the ASTAT register must be set to zero before the divide begins.

The DIVQ instruction uses the AQ bit of the ASTAT register to determine if the dividend should be added to, or subtracted from the partial reminder stored in AF&AY0. If AQ is zero, a subtract occurs. A new value for AQ is determined by XORing the MSB of the divisor with the MSB of the dividend. The 32-bit dividend is shifted left one bit, and the inverted value of AQ is moved into the LSB.

B.1.3 Output Formats

As in multiplication, the format of a division result is based on the format of the input operands. The division logic has been designed to work most efficiently with fully fractional numbers, those most commonly used in fixed-point DSP applications. A signed, fully fractional number uses one bit before the binary point as the sign, with 15 (or 31 in double precision) bits to the right, for magnitude.

If the dividend is in M.N format (M bits before the decimal point, N bits after), and the divisor is O.P format, the quotient's format will be (M-O+1).(N-P-1). As you can see, dividing a 1.31 number by a 1.15 number will produce a quotient whose format is (1-1+1).(31-15-1) or 1.15. Before dividing two numbers, you must ensure that the format of the quotient will be valid. For example, if you attempted to divide a 32.0 number by a 1.15 number the result would attempt to be in (32-1+1).(0-15-1) or 32.-16 format. This cannot be represented in a 16-bit register!

In addition to proper output format, you must insure that a divide overflow does not occur. Even if a division of two numbers produces a

legal output format, it is possible that the number will overflow, and be unable to fit within the constraints of the output. For example, if you wished to divide a 16.16 number by a 1.15 number, the output format would be (16-1+1).(16-15-1) or 16.0 which is legal. Now assume you happened to have 16384 (H#4000) as the dividend and .25 (H#2000) as the divisor, the quotient would be 65536, which does not fit in 16.0 format. This operation would overflow, producing an erroneous results.

Input operands can be checked before division to ensure that an overflow will not result. If the magnitude of the upper 16 bits of the dividend is larger than the magnitude of the divisor, an overflow will result.

B.1.4 Integer Division

One special case of division that deserves special mention is integer division. There may be some cases where you wish to divide two integers, and produce an integer result. It can be seen that an integer-integer division will produce an invalid output format of (32-16+1).(0-0-1), or 17.-1.

To generate an integer quotient, you must shift the dividend to the left one bit, placing it in 31.1 format. The output format for this division will be (31-16+1).(1-0-1), or 16.0. You must ensure that no significant bits are lost during the left shift, or an invalid result will be generated.

B.2 ERROR SITUATIONS

Although the ADSP-2100 divide primitives work in most instances, there are two cases where an invalid, or inaccurate result can be generated. The first case involves signed division by a negative number. If you attempt to use a negative number as the divisor, the quotient generated may be one LSB less than the correct result. The other case concerns unsigned division by a divisor greater than h#7FFF. If the divisor in an unsigned division exceeds H#7FFF, an invalid quotient will be generated.

B.2.1 Negative Divisor Error

The quotient produced during a divide involving a negative divisor will generally be one LSB less than the correct result. The divide algorithm implement in ADSP-2100 hardware does not correctly compensate for the twos-complement format of a negative number, causing this inaccuracy.

There is one case where this discrepancy does not occur. If the result of the division operation should equal H#8000, then it will be correctly represented, and not be one LSB off.

There are several ways to correct for this error. But before changing any code, you should determine if one LSB error in you quotient is significant problem. In some cases, the LSB is small enough to be insignificant. If you find it necessary have exact results, two solutions are apparent.

One way would be to avoid division by a negative number. If your divisor is negative, take its absolute value, and invert the sign of the quotient after division. This will produce the correct result.

Another technique would be to check the result by multiplying the quotient by the divisor. Compare this value with the dividend, if they are off by more than the value of the divisor, increase the quotient by one.

B.2.2 Unsigned Division Error

Unsigned divisions can produce erroneous results if the divisor is greater than H#7FFF. You should not attempt to divide two unsigned numbers if the divisor has a one in the MSB. If it is necessary to perform a such a division, both operands should be shifted right one bit. This will maintain the correct orientation of operands.

Shifting both operands may result in a one LSB error in the quotient. This can be solved by multiplying the quotient by the original (not shifted) divisor. Subtract this value from the original dividend to calculate the error. If the error is greater than the divisor, add one to the quotient, if it is negative, subtract one from the quotient.

B.3 SOFTWARE SOLUTION

Each of the problems mentioned in this Appendix can be compensated for in software. Listing 1 shows the module *divide_solution*. This code can be used to divide two signed or unsigned numbers to produce the correct quotient, or an error condition.

In addition to correcting the problems mentioned, this module provides a check for division overflow and computes the remainder following the division.

Since many applications do not require complete error checking, the code has been designed so you can remove tests that are not necessary for your project. This will decrease memory requirements, as well as increase execution speed.

The module *signed_div* expects the 32-bit dividend to be stored in AY1&AY0, and the divisor in AX0. Upon return either the AR register

holds the quotient and MR0 holds the remainder, or the overflow flag is set. The entire routine takes at most 27 cycles to execute. If an exception condition exists, it may return sooner. The first two instructions store the dividend in the MR registers, the absolute value of the dividend's MSW in AF, and the divisor's absolute value in AR.

The code block labeled *test_1* checks for division by H#8000. Attempting to take the absolute value of H#8000 produces an overflow. If the AV flag is set (from taking the absolute value of the divisor), then the quotient is –AY1. This can produce an error if AY1 is H#8000, so after taking the negative of AY1, the overflow flag is checked again. If it is set control is returned to the calling routine, otherwise the remainder is computed. If it is not necessary to check for a divisor of H#8000, this code block can be removed.

The code block labeled *test_2* checks for a division overflow condition. The absolute value of the divisor is subtracted from the absolute value of the dividend's MSW. If the divisor is less then the dividend, it is likely an overflow will occur. If the two are equal in magnitude, but different in sign, the result will be H#8000, so this special case is checked. If your application does not require an overflow check, this code block can be removed. If you decide to remove *test_2* be sure to change the JUMP address in *test_1* to *do_divs*, instead of *test_2*.

After error checking, the actual division is performed. Since the absolute value of the divisor has been stored in AR, this is used as the X-operand for the DIVS instruction. Fifteen DIVQ instructions follow, computing the rest of the quotient. The correct sign for the quotient is determined, based on the AS flag of the ASTAT register. Since the MR register contains the original dividend, the remainder can be determine by a multiply subtract operation. The divisor times the quotient is subtracted from MR to produce the remainder in MR0.

The last step before returning is to clear the ASTAT register which may contain an overflow flag produced during the divide.

The subroutine *unsigned_div* is very similar to *signed_div*. MR1 and AF are loaded with the MSW of the dividend, MR0 is loaded with the dividend LSW and the divisor is passed into AR. Since unsigned division with a large divisor (>H#7FFF) is prohibited, the MSB of the divisor is checked. If it contains a one, the overflow flag is set, and the routine returns to the caller. Otherwise *test_11* checks for a standard divide overflow.

B Division Exceptions

In *test_11* the divisor is subtracted from the MSW of the dividend. If the result is less then zero division can proceed, otherwise the overflow flag is set. If you wish to remove *test_11*, be sure to change the JUMP address in *test_10* to *do_divq*.

The actual unsigned division is performed by first clearing the AQ bit of the ASTAT register, then executing 16 DIVQ instructions. The remainder is computed, after first setting MR2 to zero. This is necessary since MR1 automatically sign-extends into MR2. Also, the multiply must be executed with the unsigned switch. To ensure that the overflow flag is clear, ASTAT is set to zero before returning.

In both subroutines, the computation of the remainder requires only one extra cycle, so it is unlikely you would need to remove it for speed. If it is a problem to have the multiply registers altered, remove the multiply/subtract instruction just before the return, and remove the register transfers to MR0 and MR1 in the first two multifunction instructions. Be sure to remove the MR2=0; instruction in the *unsigned_div* subroutine also.

```
.MODULE/ROM  Divide_solution;
{
This module can be used to generate correct results when using the divide primitives of
the ADSP-2100. The code is organized in sections.  This entire module can be used to
handle all error conditions, or individual sections can be removed to increase
execution speed.

Entry Points
   signed_div   Computes 16-bit signed quotient
   unsigned_div Computes 16-bit unsigned quotient
Calling Parameters
   AX0 = 16-bit divisor
   AY0 = Lower 16 bits of dividend
   AY1 = Upper 16 bits of dividend
Return Values
   AR = 16-bit quotient
   MR0 = 16-bit remainder
   AV flag set if divide would overflow
Altered Registers
   AX0, AX1, AR, AF, AY0, AY1, MR, MY0
Computation Time: 30 cycles
}
```

Listing B.1 Division Error Routine (continues on next page)

```
.ENTRY          signed_div, unsigned_div;

signed_div:     MR0=AY0,AF=AX0+AY1;         {Take divisor's absolute value}
                MR1=AY1, AR=ABS AX0;        {See if divisor and dividend have}
                                            {same magnitude, different sign}
test_1:         IF NE JUMP test_2;          {If divisor non-zero, do test 2}
                ASTAT=H#4;                  {Divide by zero, so overflow}
                RTS;                        {Return to calling program}

test_2:         IF NOT AV JUMP test_3;      {If divisor H#8000, then the}
                AY0=AY1, AF=ABS MR1;        {quotient is simply -AY1}
                IF NOT AV JUMP recover_sign;
                ASTAT=H#4;                  {H#8000 divided by H#8000,}
                RTS;                        {so overflow}

test_3:         AF=PASS AF;                 {Check for division overflow}
                IF NE JUMP test_4;          {Not equal, jump test 4}
                AY0=H#8000;                 {Quotient equals -1}
                ASTAT=H#0;                  {Clear AS bit of ASTAT}
                JUMP recover_sign;          {Compute remainder}

test_4:         AF=ABS MR1;                 {Get absolute of dividend}
                AR=ABS AX0;                 {Restore AS bit of ASTAT}
                AF=AF-AR;                   {Check for division overflow}
                IF LT JUMP do_divs;         {If Divisor>Dividend do divide}
                ASTAT=H#4;                  {Division overflow}
                RTS;
```

Listing B.1 Division Error Routine (continues on next page)

B Division Exceptions

```
do_divs:        DIVS AY1, AR; DIVQ AR;      {Compute sign of quotient}
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;
                DIVQ AR; DIVQ AR;

recover_sign:   MY0=AX0,AR=PASS AY0;        {Put quotient into AR}
                IF NEG AR=-AY0;             {Restore sign if necessary}
                MR=MR-AR*MY0 (SS);          {compute remainder dividend neg}
                RTS;                        {Return to calling program}

unsigned_div:   MR0=AY0, AF=PASS AY1;       {Move dividend MSW to AF}
                MR1=AY1, AR=PASS AX0;       {Is MSB set?}

test_10:        IF GT JUMP test_11;         {No, so check overflow}
                ASTAT=H#4;                  {Yes, so set overflow flag}
                RTS;                        {Return to caller}

test_11:        AR=AY1-AX0;                 {Is divisor<dividend?}
                IF LT JUMP do_divq;         {No, so go do unsigned divide}
                ASTAT=H#4;                  {Set overflow flag}
                RTS;

do_divq:        ASTAT=0;                    {Clear AQ flag}
                DIVQ AX0; DIVQ AX0;         {Do the divide}
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;
                DIVQ AX0; DIVQ AX0;

uremainder:     MR2=0;                      {MR0 and MR1 previous set}
                MY0=AX0, AR=PASS AY0;       {Divisor in MXO, Quotient in AR}
                MR=MR-AR*MY0 (UU);          {Determine remainder}
                RTS;                        {Return to calling program}
.ENDMOD;
```

Listing B.1 Division Error Routine

Index

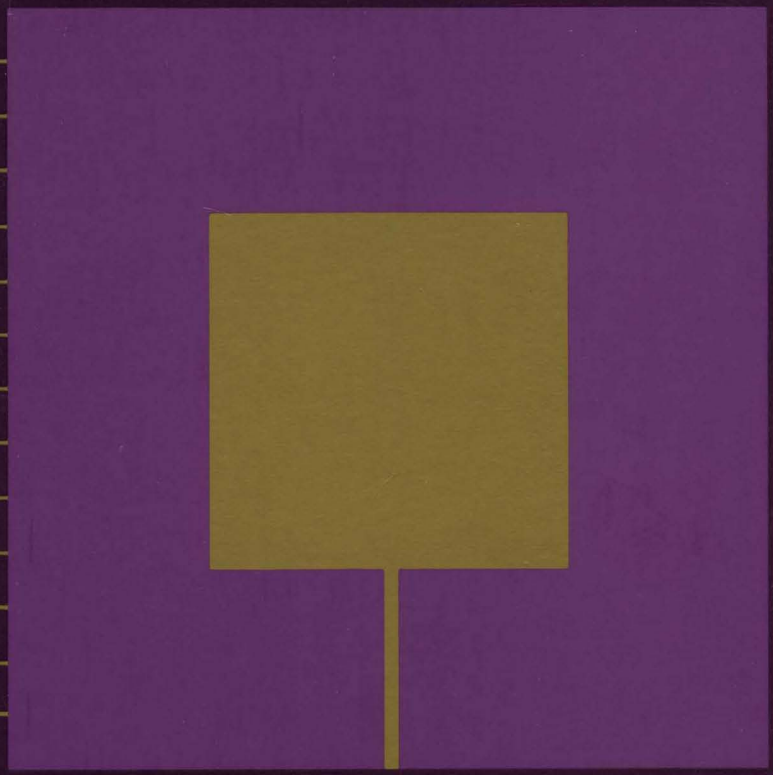
GUIDE

Boldface, in this index, denotes the major entry for the item indexed. The notation "&c" means "and following pages." Figures and tables appear in italics.

Index

Index

Index



ANALOG DEVICES

Analog Devices
Digital Signal Processing Division
One Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
(617) 329-4700

E971c-4-10/89