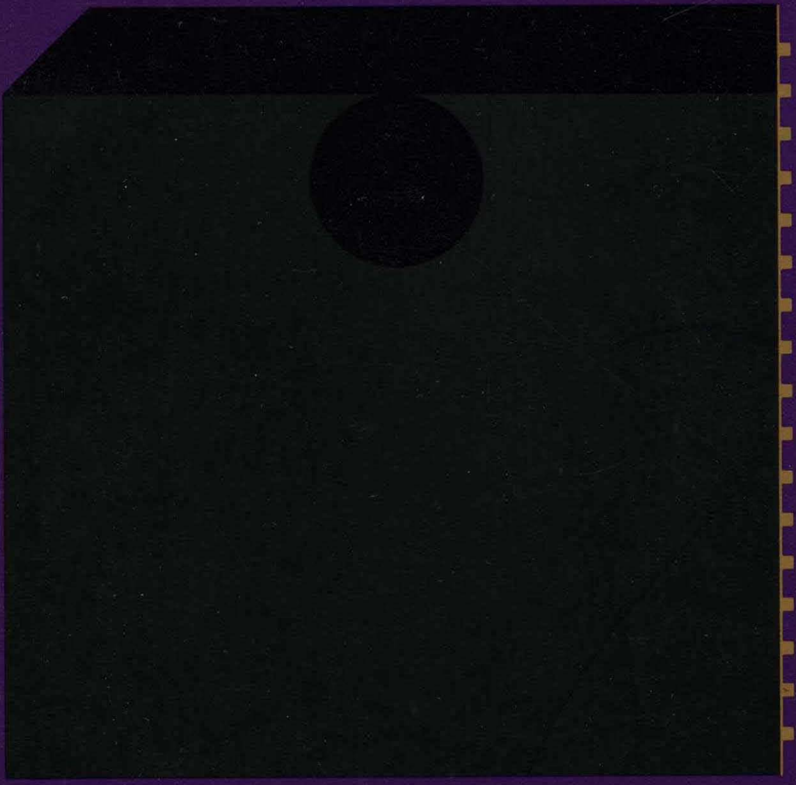
ADSP-2101/ADSP-2102 User's Manual

ANALOG DEVICES

ANALOG DEVICES



ADSP-2101
ADSP-2102

User's Manual

Architecture

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: ANALOGNORWOODMASS
- 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-2101/2102 User's Manual

February 1990

Printed in U.S.A. First Edition

Contents

Contents

Contents

Contents

Contents

Contents

FIGURES

Contents

Contents

TABLES

Introduction 1

1.1 GENERAL DESCRIPTION

The ADSP-2101 and ADSP-2102 are programmable single-chip microcomputers optimized for digital signal processing (DSP) and other high-speed numeric processing applications.

Both processors contain three computational units, two data address generators and a program sequencer, along with two serial ports, a timer, extensive interrupt capabilities and on-chip program and data memory. The ADSP-2101 has 1K words of 16-bit data memory on-chip and 2K words of 24-bit program memory on-chip. The ADSP-2102 is a mask programmable version offering any combination of RAM and ROM within the 2K word limit of the on-chip program memory. Data memory is all RAM in both versions.

In this manual, the "ADSP-2101" refers to both the ADSP-2101 and the ADSP-2102 unless otherwise noted.

The ADSP-2101 is based on the ADSP-2100 microprocessor. Like the ADSP-2100, the ADSP-2101 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 address generators and a powerful program sequencer supply addresses for on-chip or external memory access. The sequencer supports single-cycle conditional branching and executes program loops with zero overhead. Dual data 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. On-chip the ADSP-2101 uses a modified Harvard architecture in which data memory stores data and program memory stores both instructions and data. The processor can fetch an operand from on-chip data memory, an operand from on-chip program memory and the next instruction from on-chip program memory in a single cycle. (The speed of on-board memory access makes this possible and eliminates the need for cache memory as on the ADSP-2100.)

This scheme is extended off-chip via a single external memory address bus and data bus which may be used for either program or data memory access and for booting. Consequently, the processor can access external memory once in any cycle.

Boot circuitry provides for loading on-chip program memory automatically after reset with automatic wait state generation for interfacing to a single low-cost EPROM. Multiple programs can be selected and loaded from the EPROM with no additional hardware.

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

The ADSP-2101 can respond to six user interrupts. There can be up to three external interrupts, configured as edge or level sensitive. Internal interrupts can be generated from the Timer and the Serial Ports ("SPORTs"). There is also a master $\overline{RESET}$ signal.

The two serial ports provide a complete serial interface with hardware companding (data compression and expansion). Both μ-law and A-law companding are supported. The ports interface easily and directly to a wide variety of popular serial devices. Each port can generate an internal programmable clock or accept an external clock.

As a result of its architecture, the ADSP-2101 exhibits a high degree of parallelism, tailored to DSP requirements. In a single cycle, the ADSP-2101 can:

- generate the next program address
- fetch the next instruction
- perform one or two data moves
- update one or two data address pointers
- perform a computation
- receive and transmit data via the two serial ports.

Introduction 1

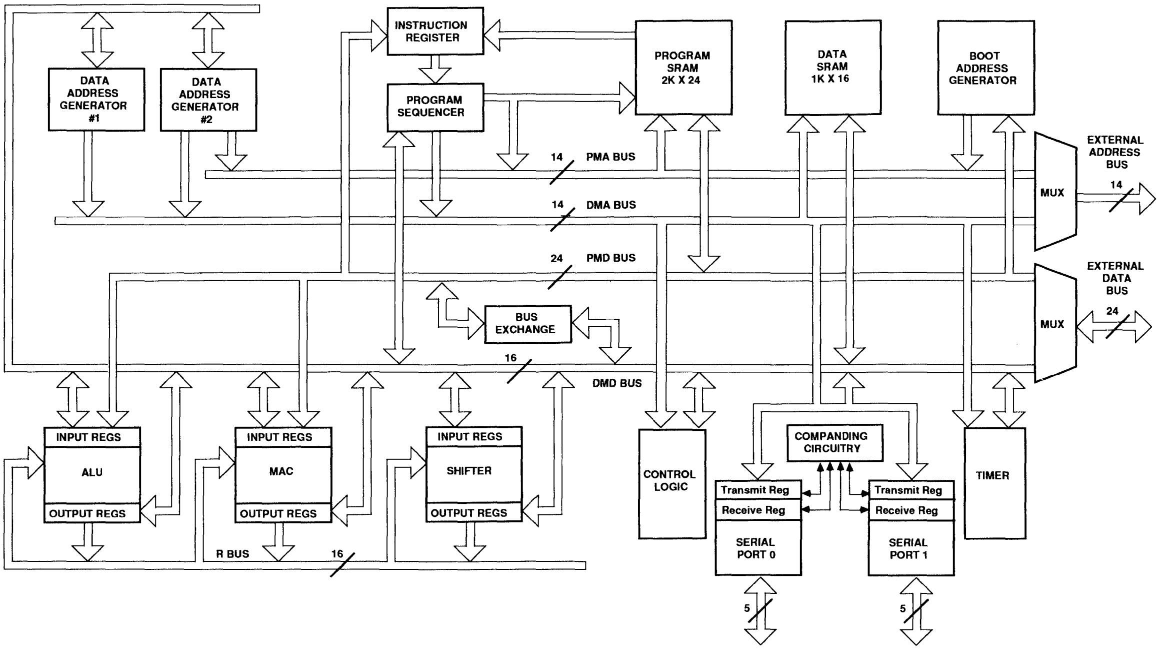


Figure 1.1 ADSP-2101/2 Internal Architecture

The instruction set is an upwardly-compatible superset of the ADSP-2100 instruction set. Chapter 9, "Instruction Set Overview" highlights the capabilities of the ADSP-2101 instruction set and shows an example program.

The ADSP-2101 instruction set provides flexible data moves and multifunction (one or more data moves with computation) instructions. Every instruction can be executed in a single processor cycle. The ADSP-2101 assembly language uses an algebraic syntax for ease of coding and readability. A comprehensive set of development tools supports program development.

1.2 SUMMARY OF ADSP-2101 KEY FEATURES

- 2K words of 24-bit program memory RAM and/or ROM on-chip
- 1K words of 16-bit data memory RAM on-chip
- Separate program and data memory buses on-chip
- Single-cycle access to on-chip program and data memory
- Dual purpose program memory for both instruction and data storage
- Automatic program boot from single byte-wide EPROM
- Programmable wait states for external program, data and boot memory spaces
- Three independent computational units: ALU, multiplier/accumulator and barrel shifter
- Provisions for multiprecision computation
- ALU and MAC saturation logic
- Zero-overhead looping
- Two double-buffered serial ports with hardware companding for μ-law and A-law
- Automatic buffering of serial port data
- Sixteen-bit programmable interval timer with 8-bit prescaler
- On-chip oscillator which can be driven from an inexpensive crystal
- Code compatible with ADSP-2100/2100A
- Simple multiprocessor interface
- 80mW low power, wait for interrupt mode
- 12.5 MHz instruction rate, 80ns per instruction
- 68-Lead PLCC/68-Pin PGA
- Complete set of hardware and software development tools

1.3 INTERNAL ARCHITECTURE

This section gives an overview of the ADSP-2101 internal architecture based on Figure 1.1. 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 stacks	4.4
• Timer	5
• Serial Ports	6

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 ADSP-2101 contains three full-function and independent computational units: an arithmetic/logic unit (ALU), a multiplier/ accumulator (MAC) and a barrel shifter (Shifter). The computational units process 16-bit data directly and provide for multiprecision computation.

The ALU performs a standard set of arithmetic and logic operations in addition to division primitives. The MAC performs single-cycle multiply, multiply/add and multiply/subtract operations. The Shifter performs logical and arithmetic shifts, normalization, denormalization, and derive exponent operations. The Shifter implements numeric format control including multiword floating point representations. The computational units are arranged side-by-side instead of serially so that the output of any unit may be the input of any unit on the next cycle. The internal result (R) bus directly connects the computational units to make this possible.

All three sections contain input and output registers which are accessible from the internal Data Memory Data (DMD) bus. 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 memory and the computational circuitry. This feature introduces one level of pipelining on input, and one level on output. The R bus allows the result of a previous computation to be used directly as the input to another computation. This avoids excessive pipeline delays when a series of different operations are performed.

Two dedicated data address generators and a powerful program sequencer ensure efficient use of these computational units.
The Data Address Generators (DAGs) provide memory addresses when memory data is transferred to or from the input/output registers. Each DAG keeps track of up to four address pointers. When a pointer is used for indirect addressing, it is post-modified by a value in a specified register. With two independent DAGs, the processor can generate two addresses simultaneously for dual operand fetches.

A length value may be associated with each pointer to implement automatic modulo addressing for circular buffers. (The circular buffer feature is also used by the serial ports for automatic data transfers. Refer to the chapter on Serial Ports for additional information.) DAG1 can supply addresses to data memory only. DAG2 can supply addresses to either the data memory or the program memory. Two independent address generators allow for simultaneous access of data stored in the program memory and data stored in the data memory.

The Program sequencer supplies instruction addresses to the program memory. The sequencer is driven by the Instruction Register which holds the currently executing instruction. The instruction register introduces a single level of pipelining into the program flow. Instructions are fetched and loaded into the instruction register during one processor cycle, and executed during the following cycle while the next instruction is prefetched. To minimize overhead cycles, the sequencer supports conditional jumps, subroutine calls and returns in a single-cycle. With an internal loop counter and loop stack, the ADSP-2101 executes looped code with zero-overhead. No explicit jump instructions are required to loop.

The programmable interval timer provides periodic interrupt generation. An 8-bit prescaler register allows the timer to decrement a 16-bit count register over a range from each cycle to every 256 cycles. An interrupt is generated when this count register reaches zero. The count register is automatically reloaded from a 16-bit period register and the count resumes immediately.

The ADSP-2101 has two bidirectional double-buffered serial ports (SPORTs) for serial communications. The SPORTs are synchronous and use framing signals to control data flow. Each SPORT can generate its serial clock internally or use an external clock. The framing sync signals may be generated internally or by an external device. Word lengths may vary from three to sixteen bits. One SPORT (SPORT0) has a multichannel capability which allows the receiving or transmitting of arbitrary data words from a 24-word or 32-word bitstream.

In addition, SPORT1 may optionally be configured as two additional external interrupt pins and the Flag Out (FO) and Flag In (FI) pins.

These components are supported by five internal buses: The PMA and DMA buses are used internally for the addresses associated with Program and Data Memory. The Program Memory Data (PMD) and Data Memory Data (DMD) buses are used for the data associated with the memory spaces. These two pairs of buses are multiplexed off chip to the external address and data buses. The $\overline{\text{BMS}}$, $\overline{\text{DMS}}$ and $\overline{\text{PMS}}$ signals select the different address spaces. The R bus is an internal bus which serves to transfer intermediate results directly between the various computational sections.

The Program Memory Address (PMA) bus is 14 bits wide allowing direct access of up to 16K words of mixed instruction code and data. The program memory data (PMD) 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 16 K 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 from program memory.

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.

1.4 ADSP-2101 DEVELOPMENT SYSTEM

The ADSP-2101 is supported with a complete set of software and hardware development tools. The ADSP-2101 Development System includes the Cross-Software Development System for software design and an Emulator for hardware debugging.

The Cross-Software Development System includes:

- System Builder

The System Builder defines the architecture of systems under development. This includes the specification of the amount of external RAM/ROM memory available and any memory-mapped I/O ports for the target hardware environment as well as the allocation of program and data memory.

- Assembler

The Assembler assembles the source code and data modules as well as supporting the high-level syntax of the instruction set. In addition to supporting a full range of system diagnostics, the Assembler provides flexible macro processing, include files, and modular code development.

- 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

The Simulator performs an interactive, instruction-level simulation of the hardware configuration described by the System Builder. It flags illegal operations and supports full symbolic assembly and disassembly.

- 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-2101 source code ready to be assembled. It also supports inline assembler code.

- In-circuit Emulator

The Emulator provides hardware debugging of the ADSP-2101 systems with stand-alone in-circuit emulation, using an ADSP-2101 in self-emulation mode. The Emulator design provides execution with little or no degradation in processor performance.

For additional information on the Development System, refer to the *ADSP-2101 Cross-Software Manual.*

1.5 MANUAL ORGANIZATION

The *ADSP-2101 User's Manual* provides the necessary information to understand and evaluate the operation of the ADSP-2101. Together with the *ADSP-2101 Data Sheet*, this manual provides all the information required to design a ADSP-2101 hardware system. For information on programming the chip, refer to the *ADSP-2101 Cross-Software Manual.*

The rest of this manual is organized as follows.

Chapter 2, "Computational Units," describes the internal architecture and function of the three computational units of the ADSP-2101: the arithmetic/logic unit, the multiplier/accumulator and the barrel shifter.

Chapter 3, "Data Moves," describes the data address generators (DAGs) and the PMD-DMD Bus Exchange Unit.

Chapter 4, "Program Control," describes the program sequencer, interrupt controller and status and condition logic.

Chapter 5, "Timer," explains the programmable interval timer.

Chapter 6, "Serial Ports," describes the two ADSP-2101 serial ports: SPORT0 and SPORT1.

Chapter 7 "System Interface," provides a description of the control interface of the ADSP-2101. Information on the software reboot function is also included.

Chapter 8, "Memory Interface," describes the three memory spaces on the ADSP-2101: data memory, program memory and boot memory. For timing characteristics, refer to the *ADSP-2101 Data Sheet.*

Chapter 9, "Instruction Set Overview," is an overview of the ADSP-2101 instruction set. All instructions are grouped by major type. Detailed programmer's reference material is in the *ADSP-2101 Cross-Software Manual;* this chapter gives enough information to understand the capabilities and flexibility of the instruction set.

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

Appendix B, "Division Exceptions," describes signed and unsigned division.

Appendix C, "Pin Information," describes the pinout of the 68-pin PGA and PLCC packages.

Appendix D, "Control/Status Registers," summarizes the contents and locations of all control and status registers in the ADSP-2101.

Computational Units ■ 2

2.1 ARITHMETIC ON THE ADSP-2101

This chapter describes the architecture and function of the three computational units of the ADSP-2101: 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-2101 handles binary arithmetic. The ADSP-2101 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 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 binary notation; sixteen bits are treated as a bit pattern. Examples of computation using this format are the logical operations: NOT, AND, OR, XOR. These ALU operations treat their operands as binary strings with no provision for sign bit or binary point placement.

2.1.2 Unsigned

Unsigned binary numbers may be thought of as positive, having nearly twice the magnitude of a signed number of the same length. The least significant words of multiple precision numbers are treated as unsigned numbers.

2.1.3 Signed Numbers: Twos-Complement

In discussions of ADSP-2101 arithmetic "signed" refers to twos-complement. Most ADSP-2101 operations presume or support twos-complement arithmetic. The ADSP-2101 does not use signed-magnitude, ones-complement, BCD or excess-n formats.

2.1.4 Fractional Representation: 1.15

The ADSP-2101 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 or Q15.) 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.

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 negative (AN) flag.

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 produces results that are binary strings. The inputs are "interpreted" according to the information given in the instruction itself (signed times signed, unsigned times unsigned, a mixture or round). The 32-bit result from the multiplier is assumed to be signed, in that it is sign-extended across the full 40-bit width of the MR register set.

The ADSP-2101 supports two modes of format adjustment: the fractional mode for fractional operands, 1.15 format (1 signed bit, 15 fractional bits), and the integer mode for integer operands, 16.0 format. When multiplying 1.15 operands, the result is 2.30 (30 fractional bits). To correct this, in the fractional mode, a left shift occurs between the multiplier product (P) and the multiplier result register (MR). This shift (1 bit to the left) causes the multiplier result to be 1.31 which can be rounded to 1.15. Figure 2.6, in the MAC section of this chapter, shows this.

In the integer mode, the left shift does not occur. For example, if the operands are in the 16.0 format, the 32-bit multiplier result would be in 32.0. A left shift would change the numerical representation. Figure 2.7 in the MAC section of this chapter shows this.

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 fractions.

2.1.8 Summary

The table below summarizes some of the arithmetic characteristics of the ADSP-2101 computational operations. In addition to the numeric types described in this section, the ADSP-2101 C Compiler supports a form of 32-bit floating-point in which one 16-bit word is the exponent and the other word is the mantissa. See the *ADSP-2101 Cross-Software Manual.*

Operation	*Arithmetic Formats*	
	Operands	*Result*
ALU		
Addition	Signed or unsigned	Interpret flags
Subtraction	Signed or unsigned	Interpret flags
Logical Operations	Binary string	same as operands
Division	Explicitly signed/unsigned	same as operands
ALU Overflow	Signed	same as operands
ALU Carry Bit	16-bit unsigned	same as operands
ALU Saturation	Signed	same as operands
MAC, Fractional Mode [ADSP-2100 Compatible]		
Multiplication (P)	1.15 Explicitly signed/unsigned	32 bits (2.30)
Multiplication (MR)	1.15 Explicitly signed/unsigned	2.30 shifted to 1.31
Mult / Add	1.15 Explicitly signed/unsigned	2.30 shifted to 1.31
Mult / Subtract	1.15 Explicitly signed/unsigned	2.30 shifted to 1.31
MAC Saturation	Signed	same as operands
MAC, Integer Mode		
Multiplication (P)	1.15 Explicitly signed/unsigned	32 bits (2.30)
Multiplication (MR)	16.0 Explicitly signed/unsigned	32.0 no shift
Mult / Add	16.0 Explicitly signed/unsigned	32.0 no shift
Mult / Subtract	16.0 Explicitly signed/unsigned	32.0 no shift
MAC Saturation	Signed	same as operands
Shifter		
Logical Shift	Unsigned / binary string	same as operands
Arithmetic Shift	Signed	same as operands
Exponent Detection	Signed	same as operands

Table 2.1 Arithmetic Formats Used by the ADSP-2101

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.

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 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 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 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. The instruction set also provides for reading AR over the PMD bus, but there is no direct connection; this operation uses the DMD-PMD bus exchange unit.

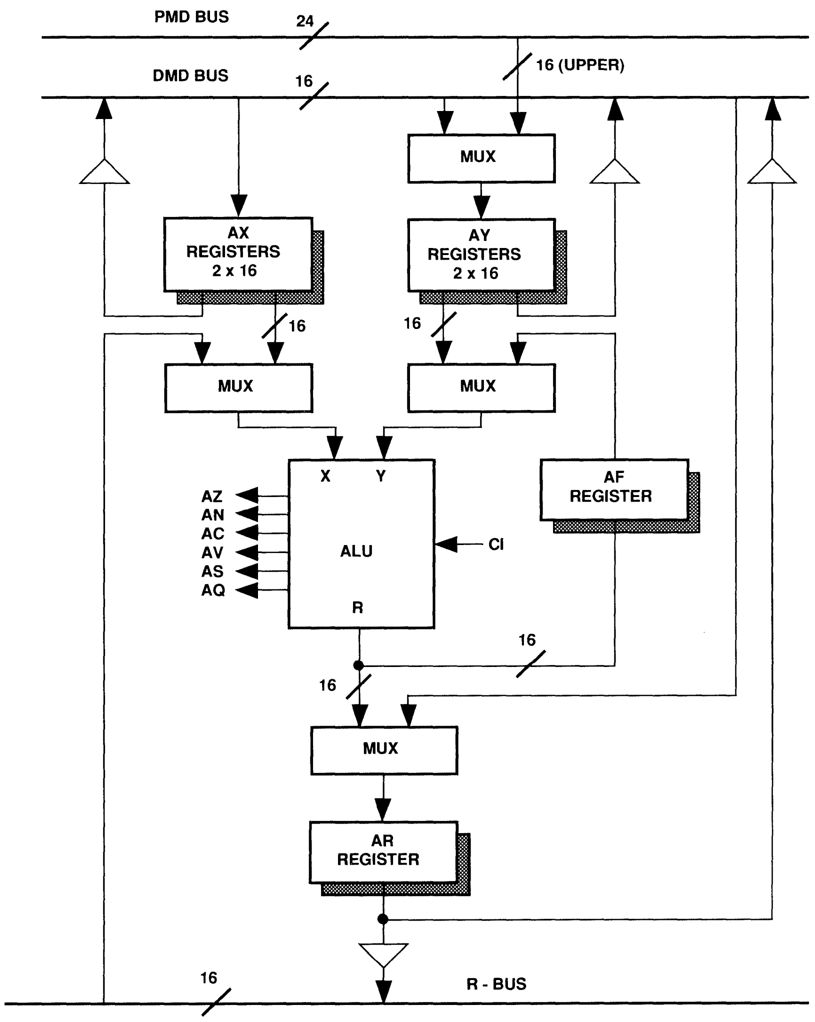


Figure 2.1 ALU Block Diagram

Any of the registers associated with 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. A register read, therefore, reads the value 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 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 (such as during an interrupt service routine) for extremely fast context switching. A new task, like 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 bit 0 in the processor mode status register (MSTAT). If this bit is a 0, the primary bank is selected; if it is a 1, the secondary bank is selected.

2.2.2 Standard Functions

The standard ALU functions 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)*

2.2.3 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 a twos-complement 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 enabled by setting bit 3 of the 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 setting bit 2 in the 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 a 16-bit quotient, executes in 16 cycles. Higher and lower precision quotients 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 start 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.

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.

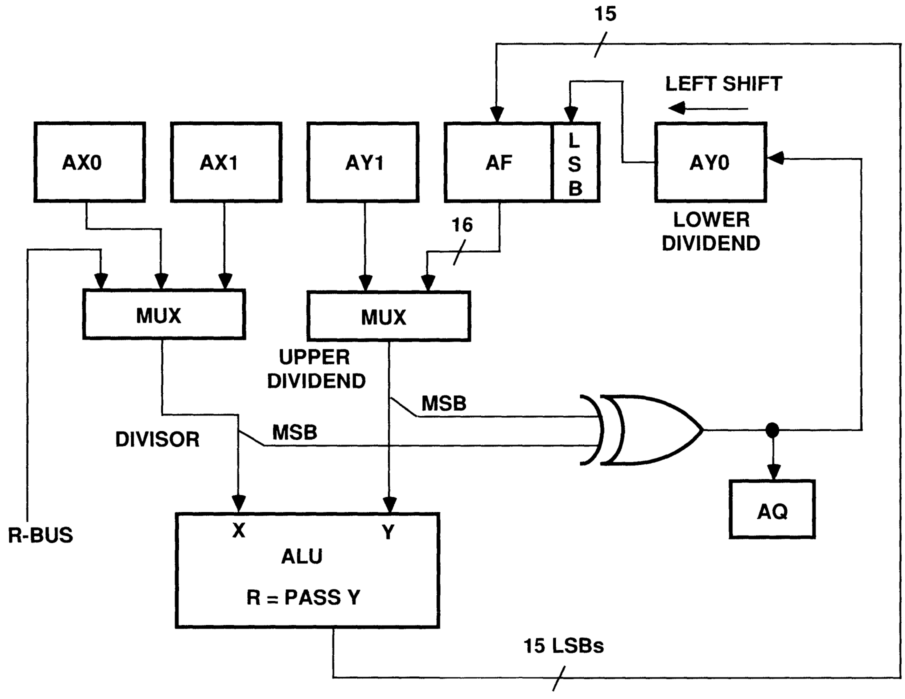


Figure 2.2 DIVS Operation

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.

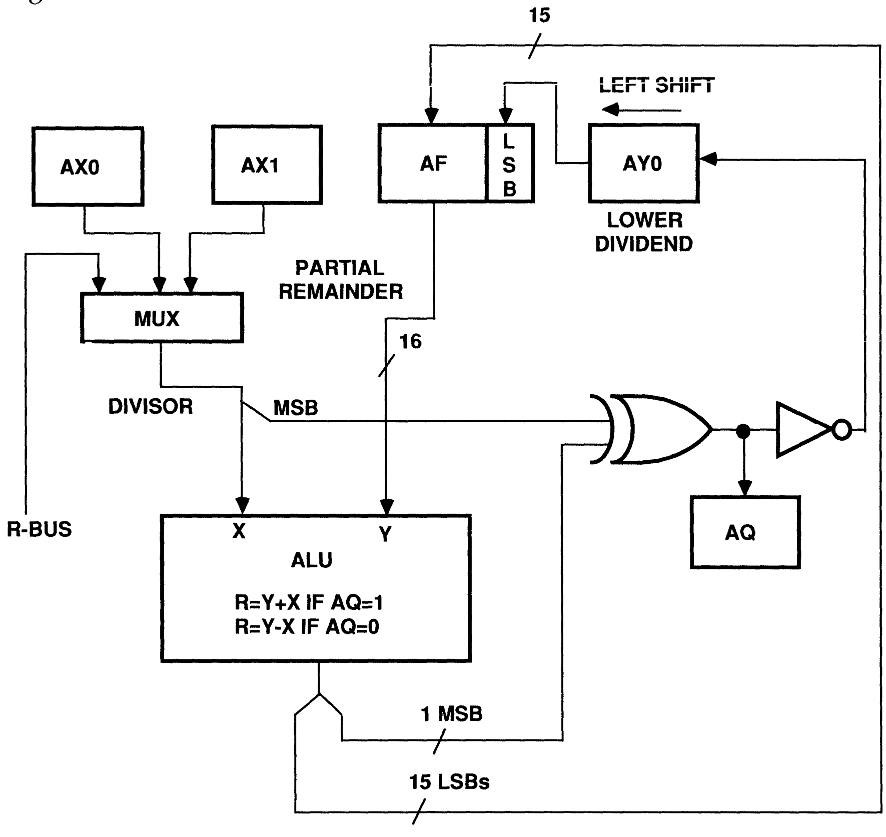


Figure 2.3 DIVQ Operation

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.*

```
Dividend            BBBBB.BBBBBBBBBBBBBBBBBBBBBBBBBBB

                 NL bits         NR bits

Divisor             BB.BBBBBBBBBBBBBB

                 DL bits         DR bits

Quotient            BBBB.BBBBBBBBBBBB

          (NL-DL+1) bits         (NR-DR-1) bits
```

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 in magnitude.

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, saturation 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 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, or passes the new product directly to MR. 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)

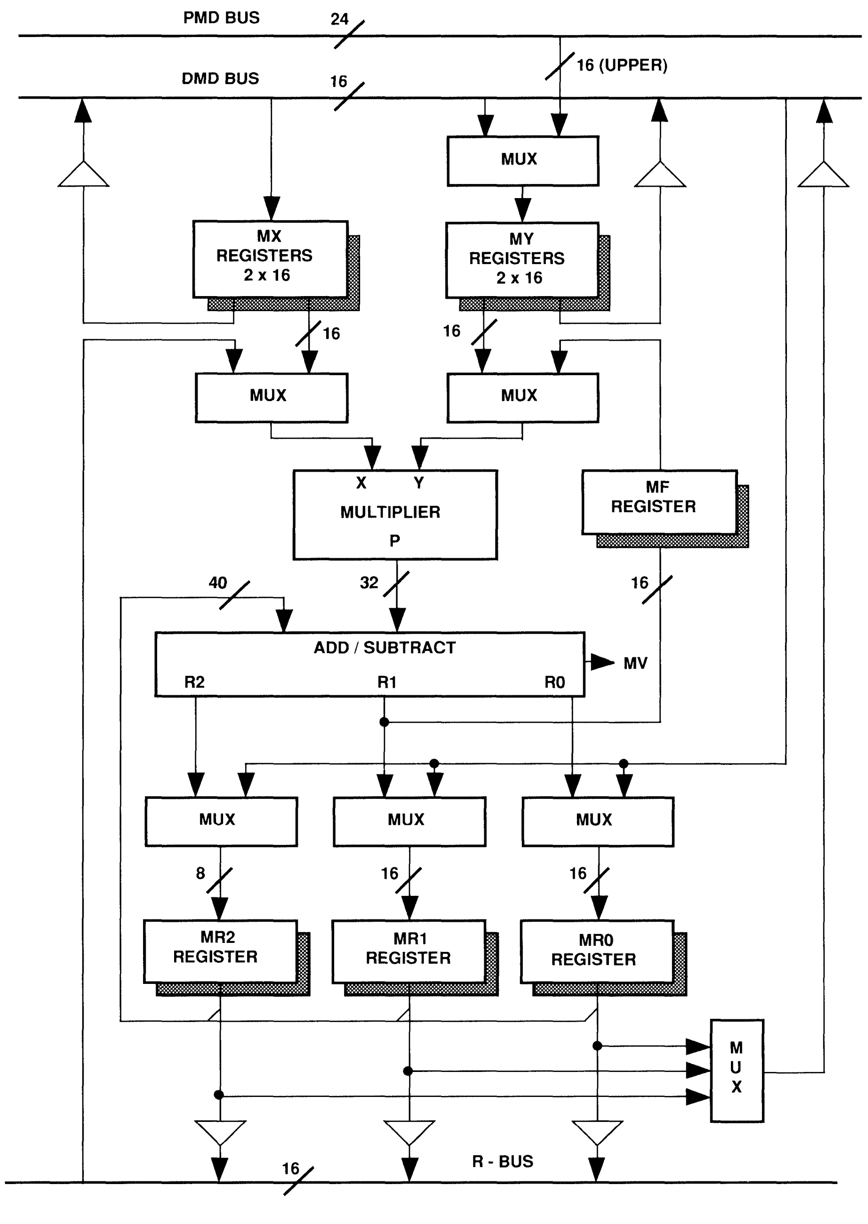


Figure 2.5 MAC Block Diagram

bits in the top nine bits of the MR register (based on twos-complement arithmetic).

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 one register can provide input to the multiplier while either one simultaneously drives the DMD bus.

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-2101 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 one register can provide input to the multiplier while either one simultaneously drives the DMD bus.

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 loaded directly from the DMD bus and output to either the DMD bus or the R bus.

Any of the registers associated with 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. A register read, therefore, reads the value 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 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 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 bit 0 in the processor mode status register (MSTAT). If this bit is a 0, the primary bank is selected; if it is a 1, the secondary bank is selected.

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

The ADSP-2101 provides two modes for the standard multiply/accumulate function: fractional mode for fractional numbers, (1.15) and integer mode for integers (16.0). The mode is selected by bit 4 of the mode status register (MSTAT). If this bit is a 1, the integer mode is selected; if it is a 0, the fractional mode is selected. In both modes, 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.

In the fractional mode, the 32-bit P output is format adjusted, that is, sign-extended and shifted one bit to the left before being added to MR. For example, 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 fractional multiplier result format is shown in Figure 2.6, on the following page.

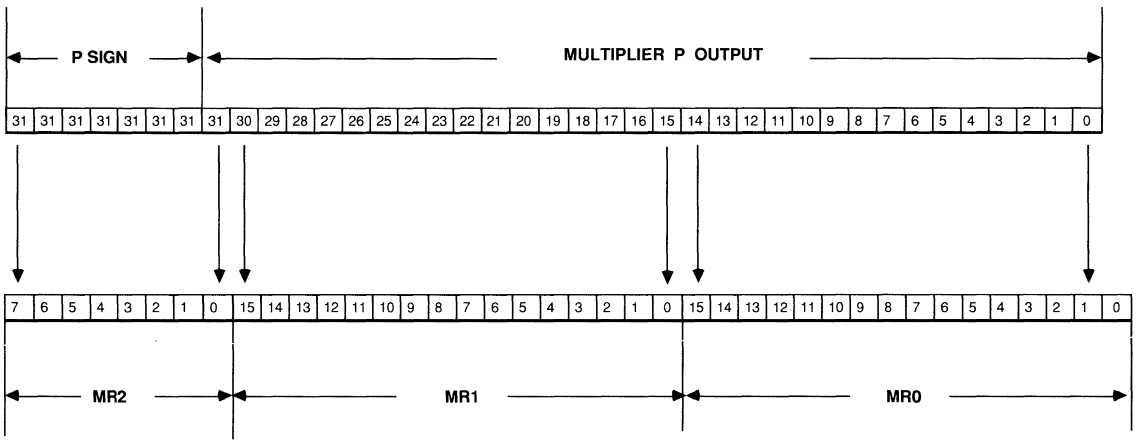


Figure 2.6 Fractional Multiplier Result Format

In the integer mode, the 32-bit P register is not shifted before being added to MR, that is, the redundant sign bit is retained. Figure 2.7 displays the integer ADSP-2101 result placement.

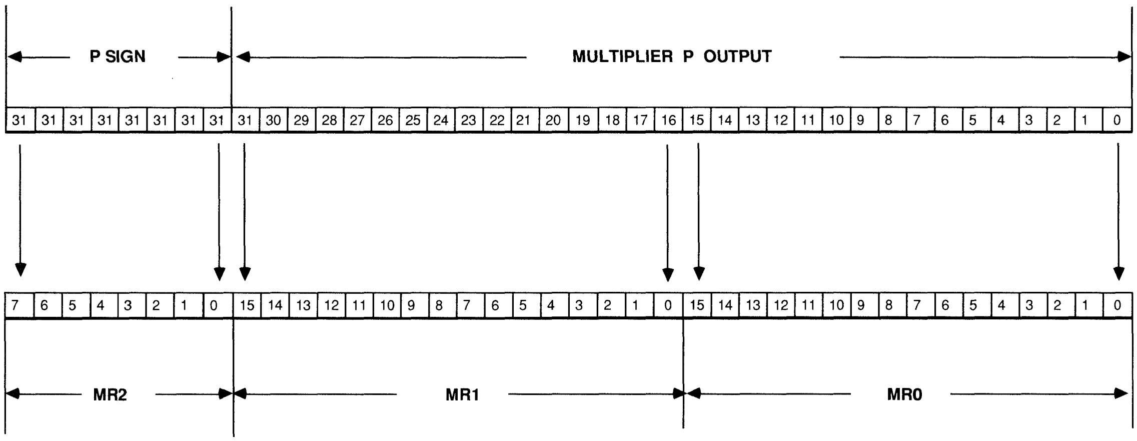


Figure 2.7 Integer Multiplier Result Format

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 loaded 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. MR1 also has an automatic sign-extend capability. When MR1 is loaded 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 load the MR2 register with a value other than MR1's sign extension, you must load MR2 after MR1 has been loaded. Loading MR0 affects neither MR1 nor MR2; no sign extension occurs in MR0 loads.

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 (MR1/MR2) boundary. That is, MV is set if the upper nine bits of MR are not all ones or all zeros.

The MR register has a saturation capability which sets MR to the maximum positive or negative 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	*MSB of MR2*	*MR content after saturation*	
0	0 or 1	no change	
1	0	00000000 0111111111111111 1111111111111111	*full-scale positive*
1	1	11111111 1000000000000000 0000000000000000	*full-scale negative*

Table 2.3 MAC Saturation Instruction Effect

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 multiplication/accumulations 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 (uniformly distributed) values. When the midway point is detected, bit 16 in the result output is forced to zero. This is called 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 lower 16 bits of 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 32-bit output. These include arithmetic shift, logical shift and normalization. The Shifter also performs 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 2.8 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 also takes as inputs AR, SR or MR via the R bus. 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 loaded 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") 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 value, that is, the value by which the block values must be shifted to normalize the largest value. 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.

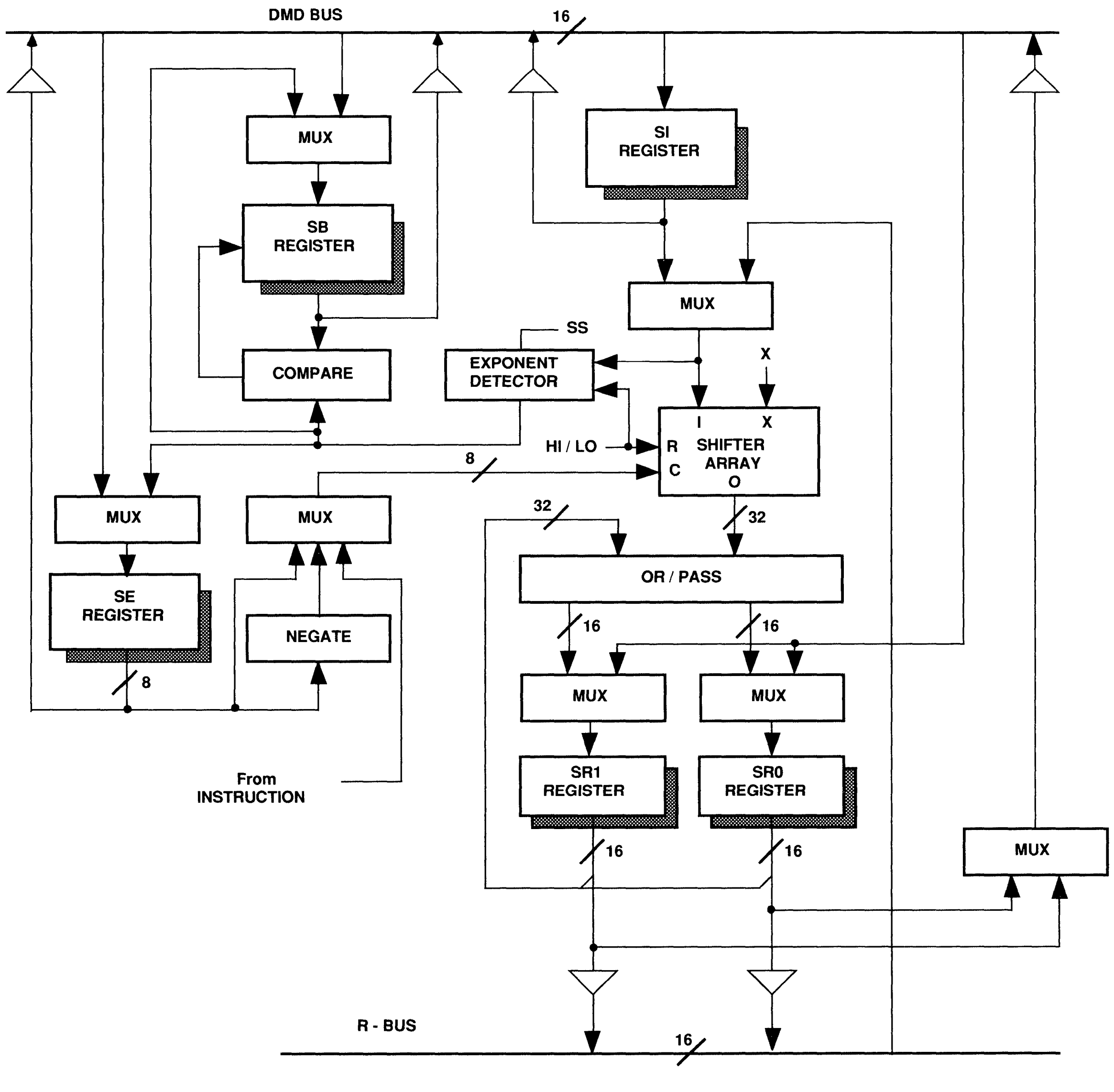


Figure 2.8 Shifter Block Diagram

Any of the SI, SE or 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 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 2.8 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 for extremely fast context switching. A new task, such as an interrupt service routine, can then be executed without transferring current states to storage.

The selection of the primary or alternate bank of registers is controlled by bit 0 in the processor mode status register (MSTAT). If this bit is a 0, the primary bank is selected; if it is a 1, the secondary bank is selected.

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). 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 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.

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

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

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 is needed in the normalized mantissa (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.

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

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

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

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 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 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	`00000`**`101 10110101 00011`**`000 000000`

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	`00000000 000`**`10110 11010100 011`**`00000`

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 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). 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

Numbers with redundant sign bits require normalizing. 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.

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*

   First Input `11110110 11010100`

   SR **10110110 10100** 000 00000000 00000000

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

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 is set (1) if an overflow has occurred. AC contains 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. When the NORM, LO operation is done, the extension bit is zero; when the NORM, HI operation is done, the extension bit is AC.

Data Moves ■ 3

3.1 INTRODUCTION

This chapter describes sections of the ADSP-2101 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-2101 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-2101 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 9, "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 index (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 becomes the memory address. With DAG1, the output address can be bit-reversed by setting the

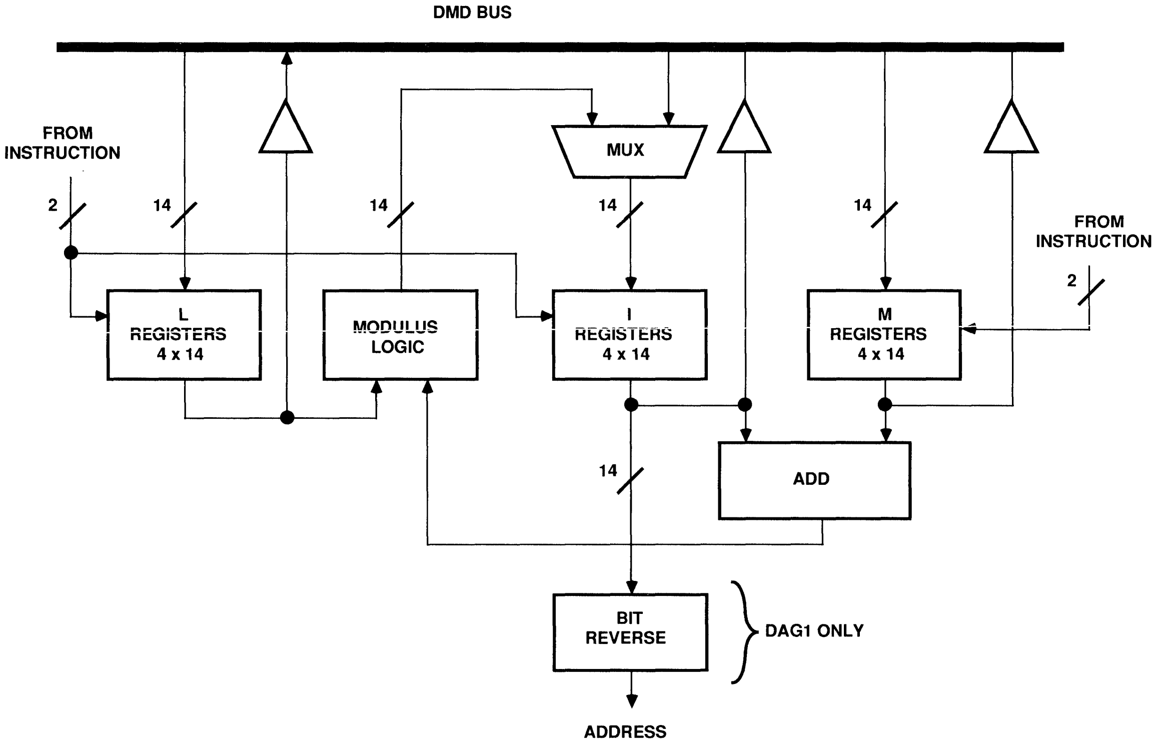


Figure 3.1 Data Address Generator Block Diagram

appropriate mode bit in the mode status register (MSTAT) as discussed below. Bit-reversal facilitates 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 added to the specified I register to generate the new I value. 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. For circular buffer addressing, the L register is initialized with length of the buffer. For linear addressing, the modulus logic is disabled by setting the corresponding L register to zero.

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. If the sum of the M register content and the I register content crosses the buffer boundary, the modified I register value is calculated by the modulus logic using the L register value.

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

From these inputs, the next address is calculated with the formula:

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

where:

I	=	current address,
M	=	modify value (signed)
B	=	base address
L	=	buffer length
M+I	=	modified address
$\lvert M \rvert$	$< L$	(this insures that the next address cannot wrap around the buffer more than once in one operation)

3.2.3 Calculating the Base Address

The above equation does not supply you with the base address. Given the length of the buffer (L), the base address is 2^n or a multiple of 2^n, where n satisfies the condition:

$2^{n-1} < L \leq 2^n$

In practice, you do not need to calculate this yourself; the Linker automatically places circular buffers at a proper address.

3.2.3.1 Circular Buffer Base Address Example 1

For example, let us assume that the buffer length is eight. According to the rule, the length of the buffer (eight) must be less than or equal to some value 2^n; n therefore, must be three or greater. The left side of the inequality specifies that the buffer length must be greater than the value 2^{n-1}; n therefore must be three or less. The only value of n that satifies both inequalities is three.

Valid base addresses are multiples of 2^n, so in this example, valid base addresses are multiples of eight: H#0008, H#0010, H#0018, and so on (hexadecimal notation).

3.2.3.2 Circular Buffer Base Address Example 2

As a second example, assume a buffer length of seven. Solving the inequalities again yields the same answer as example number one. Valid bases addresses are multiples of 2^n. In this example, valid base addresses are also multiples of eight: H#0008, H#0010, H#0018, and so on.

Note that the buffer addresses for a buffer length of seven and eight are the same. The calculation of the base address differs from the ADSP-2100. The ADSP-2101 uses memory more efficiently for buffers whose lengths are powers of two. For example, the base address for a buffer length of eight in the ADSP-2100 must be a multiple of sixteen, rather than of eight.

3.2.3.3 Circular Buffer Operation Example 1

Suppose that I0 = 5, M0 = 1 and L0 = 3. Base addresses are multiples of 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....

3.2.3.4 Circular Buffer Operation Example 2

Assume that I0 = 9, M0 = 3 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 I for indirect addressing cycle through the sequence: 9, 12, 10, 8, 11, 9...

3.2.4 Serial Ports

The Serial Port autobuffering feature uses circular buffer addressing to transfer data to or from memory and the serial ports. In this application, the automatic pointer wraparound triggers the serial port interrupt. For additional information, refer to the chapter on Serial Ports.

3.2.5 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 index 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. 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 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, you should load the PX register from DMD bus before the program memory write operation.

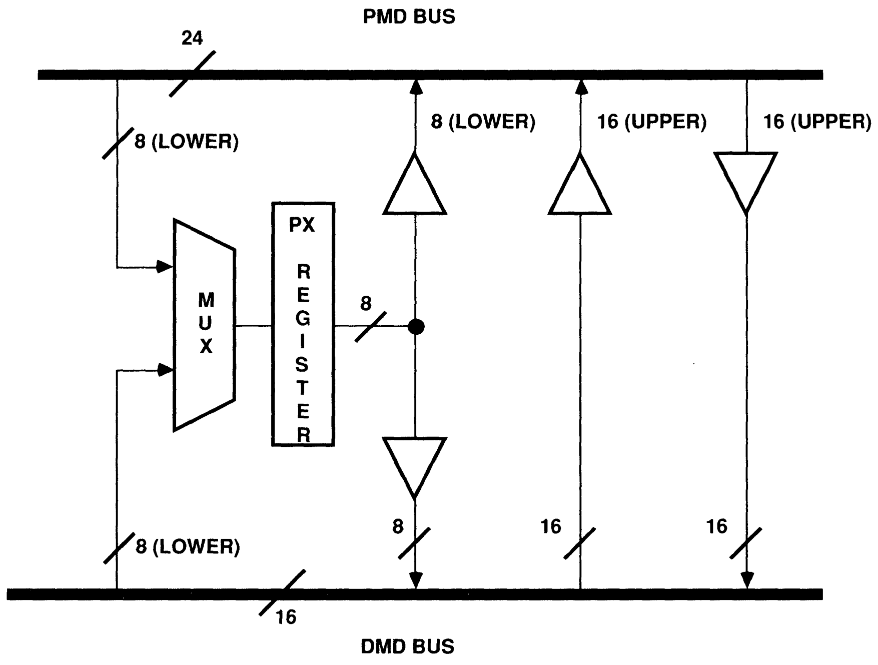


Figure 3.2 PMD–DMD Bus Exchange

Program Control ■ 4

4.1 INTRODUCTION

This chapter describes the sections of the ADSP-2101 that control and affect the flow of your program's execution: the program sequencer, its associated interrupt controller and the status and condition logic.

4.2 PROGRAM SEQUENCER

The program sequencer generates a stream of instruction addresses, and provides 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 next page) shows a block diagram for the program sequencer and status sections of the ADSP-2101. The sections immediately below discuss individual blocks within the sequencer.

It is useful to be aware that the ADSP-2101 instruction set includes the following instructions:

- JUMP
- CALL
- RETURN FROM SUBROUTINE (RTS)
- RETURN FROM INTERRUPT (RTI)
- DO UNTIL
- IDLE (Wait for interrupt)

4.2.1 Next Address Select Logic

The sequencing logic controls the flow of ADSP-2101 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

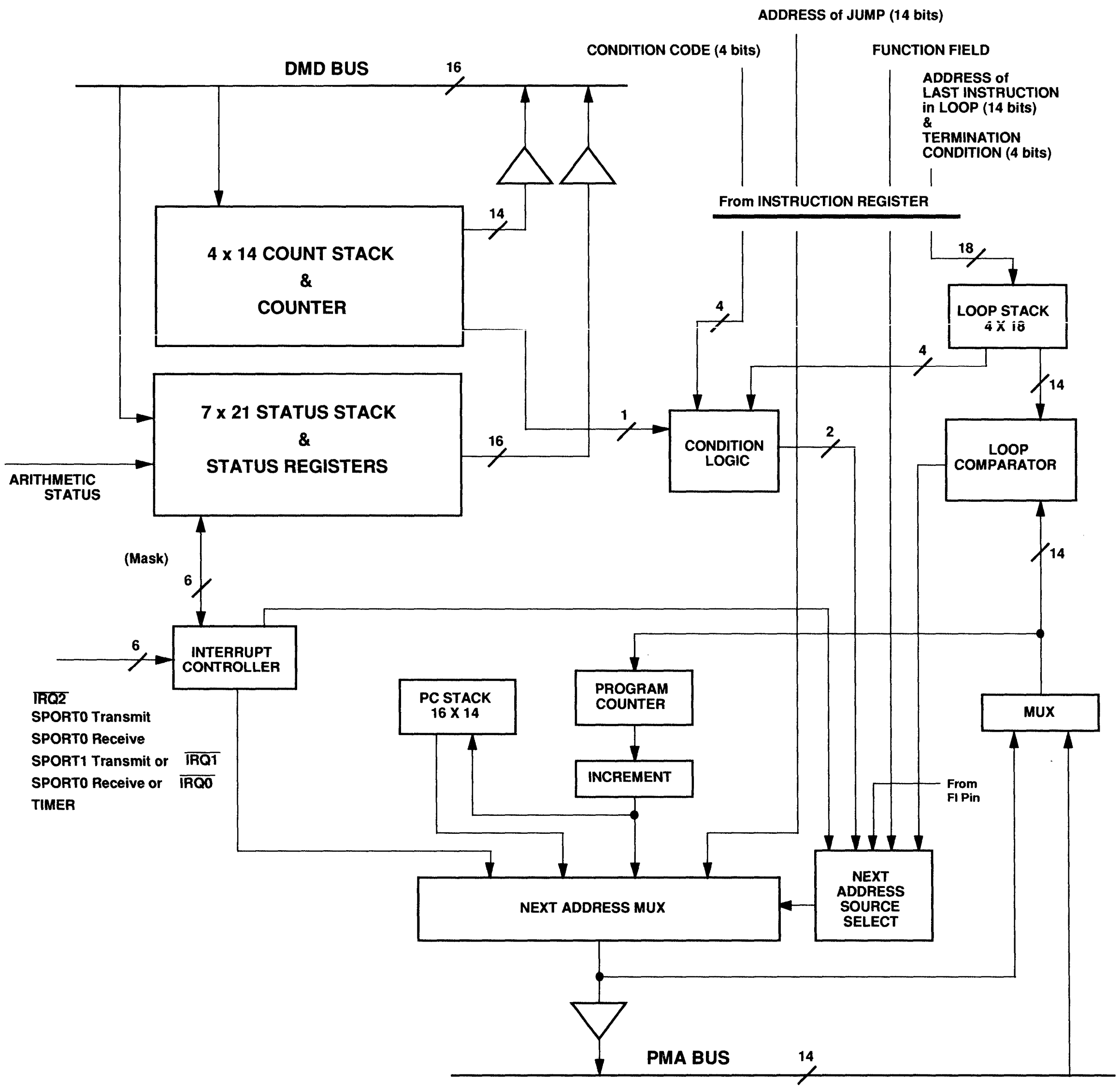


Figure 4.1 Program Sequencer Block Diagram

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 or return 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 interrupt. Upon recognizing an interrupt, the processor jumps to the interrupt vector location corresponding to the active interrupt request. The interrupt vector locations are four program memory locations apart; this allows short service routines to be coded in place. For longer routines, control is transferred to the interrupt service routine by means of a jump instruction at the interrupt vector.

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.

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 DO UNTIL is executed and 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 loading the CNTR register. 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 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 are 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. 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 are two exceptions 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).

The second exception is provided explicitly by the special purpose register mnemonic OWRCNTR. Writing a value to this register (allowed only by register-to-register transfer) rather than CNTR overwrites the counter with the new value, and nothing is pushed onto the count stack. See the instruction set overview in Chapter 9 for more information.

4.2.4 Loop Comparator and Stack

The DO UNTIL instruction initiates 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, below, 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 = 1
FOREVER	Always	Always True

Table 4.1 DO UNTIL Termination Condition Logic

The conditions in Table 4.1 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 4.4.

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 then contains the first loop instruction address, and the loop stack contains 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 address of the next instruction. 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.

At this point, there are two possible results 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 execute 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 does 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 are generally 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 correct 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.

4.3 INTERRUPT CONTROLLER

The interrupt controller of the ADSP-2101 allows the processor to respond to one of six interrupts. Depending on the configuration of SPORT1, there may be one or three interrupts generated by external devices and three or five interrupts generated internally by the serial ports and the timer. The processor responds to interrupts by shifting control to the instruction located at the appropriate interrupt vector address. Table 4.2 shows the interrupts and associated vector addresses.

SPORT1 may be configured as a serial port or alternately as two external interrupt pins, $\overline{\text{IRQ0}}$ and $\overline{\text{IRQ1}}$ (plus the Flag In and Flag Out pins and a programmable clock). Clearing the SPORT1 configuration bit in the system control register enables the interrupts and flags. See the Chapter, "System Interface," for more information about the alternate configuration of these pins.

Source of Interrupt	*Interrupt Vector*
$\overline{\text{IRQ2}}$ (external pin)	0004 *(highest priority)*
SPORT0 Transmit (internal)	0008
SPORT0 Receive (internal)	000C
SPORT1 Transmit (internal) or $\overline{\text{IRQ1}}$ (external)	0010
SPORT1 Receive (internal) or $\overline{\text{IRQ0}}$ (external)	0014
Timer (internal)	0018 *(lowest priority)*

Table 4.2 Interrupts & Interrupt Vector Addresses

Interrupts can also be forced under software control; see the discussion of the IFC register in the following sections.

Because of the efficient stack and program sequencer, there is no latency (beyond synchronization delay) 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. 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. You must explicitly program a context switch between the primary and secondary register banks if required.

4.3.1 Configuring Interrupts

Interrupts may be edge-sensitive or level-sensitive. Pending edge-sensitive interrupts may be cleared. Interrupts may also be individually masked. These interrupt characteristics are controlled by the ICNTL, IMASK and IFC registers.

If an interrupt input is edge-sensitive, the interrupt is latched whenever any inactive to active transition occurs. The latch remains set until the request is serviced, then is automatically cleared. It may also be cleared in software by setting the clear bit in IFC.

Thus an edge-sensitive interrupt signal need only be active long enough to be recognized or can remain active indefinitely. Edge-sensitive inputs generally require less external hardware than level-sensitive inputs, and allow 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 input by ORing them together into a single $\overline{\text{IRQ}}$ pin.

You may also select whether automatic nesting of interrupt service routines occurs. 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.

4.3.1.1 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	$\overline{\text{IRQ0}}$ Sensitivity	*(If configured)*
Bit 1	$\overline{\text{IRQ1}}$ Sensitivity	*(If configured)*
Bit 2	$\overline{\text{IRQ2}}$ Sensitivity	
Bit 3	Zero	
Bit 4	Interrupt Nesting Mode	*See Table 4.3*

The $\overline{\text{IRQ}}$ sensitivity bits determine whether a given interrupt input is edge- or level-sensitive (0 = level-sensitive, 1 = edge-sensitive). Since the timer and the SPORT interrupts are internally generated, there are no sensitivity bits for these interrupts.

Bit 4 determines whether nesting of interrupt service routines is allowed as detailed in the next paragraph.

4.3.1.2 Interrupt Mask Register (IMASK)

IMASK is a 6-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 four of ICNTL; it may be altered, of course, as part of the interrupt service routine itself.

When nesting is disabled, all interrupt levels are masked automatically (IMASK set to zero) when an interrupt service routine is entered. When nesting is enabled, IMASK is set so that only equal and lower priority interrupts are masked; higher priority interrupts remain configured as they were prior to the interrupt. This is graphically shown in Table 4.3 below.

The bits in IMASK are defined as follows:

Bit 0	Timer enable	*lowest priority*
Bit 1	$\overline{\text{IRQ0}}$ or SPORT1 receive enable	
Bit 2	$\overline{\text{IRQ1}}$ or SPORT1 transmit enable	
Bit 3	SPORT0 receive enable	
Bit 4	SPORT0 transmit enable	
Bit 5	$\overline{\text{IRQ2}}$ enable	*highest priority*

The bits are all positive sense (0 = masked, 1 = enabled). IMASK is set to zero upon a processor reset. The interrupt nesting mode bit (ICNTL) determines the state of IMASK upon entering the interrupt, as shown in Table 4.3. IMASK may be read from or written to via the DMD bus.

ICNTL bit 4 = 0 (nesting disabled)

Interrupt Level Serviced	*IMASK contents before, pushed on stack*	*IMASK contents entering interrupt service*
0 (low)	ABCDEF	000000
1	ABCDEF	000000
2	ABCDEF	000000
3	ABCDEF	000000
4	ABCDEF	000000
5 (high)	ABCDEF	000000

ICNTL bit 4 = 1 (nesting enabled)

Interrupt Level Serviced	*IMASK contents before, pushed on stack*	*IMASK contents entering interrupt service*
0 (low)	ABCDEF	ABCDE0
1	ABCDEF	ABCD00
2	ABCDEF	ABC000
3	ABCDEF	AB0000
4	ABCDEF	A00000
5 (high)	ABCDEF	000000

"ABCDEF" represents any pattern of ones and zeroes.

Table 4.3 IMASK Entering Interrupt Service Routines

4.3.1.3 Interrupt Force & Clear Register: IFC

The 12-bit IFC register is write-only and contains a bit for clearing and a bit for forcing each of the six possible interrupts in the ADSP-2101. The bits in IFC are defined as follows.

Bit 0	Timer interrupt clear
Bit 1	SPORT1 receive or $\overline{\text{IRQ0}}$ interrupt clear
Bit 2	SPORT1 transmit or $\overline{\text{IRQ1}}$ interrupt clear
Bit 3	SPORT0 receive interrupt clear
Bit 4	SPORT0 transmit interrupt clear
Bit 5	$\overline{\text{IRQ2}}$ interrupt clear
Bit 6	Timer interrupt force
Bit 7	SPORT1 receive or $\overline{\text{IRQ0}}$ interrupt force
Bit 8	SPORT1 transmit or $\overline{\text{IRQ1}}$ interrupt force
Bit 9	SPORT0 receive interrupt force
Bit 10	SPORT0 transmit interrupt force
Bit 11	$\overline{\text{IRQ2}}$ interrupt force

Pending edge-sensitive interrupts can be cleared by setting the appropriate clear bit (0-5) in IFC. Edge-triggered interrupts are normally cleared automatically when the corresponding interrupt service routine is called.

Edge-sensitive interrupts can be forced under program control by setting the force bit (6-11) corresponding to the desired interrupt. This causes the interrupt to be serviced once, unless masked. An external interrupt ($\overline{\text{IRQ0}}$, $\overline{\text{IRQ1}}$ or $\overline{\text{IRQ2}}$) must be edge-sensitive (as determined by bits in ICNTL) to be forced. The timer and SPORT interrupts behave like edge-sensitive interrupts and can be masked, cleared and forced.

4.3.2 Interrupt Controller Operation

The individual interrupt request signals are logically ANDed with the IMASK bits and then fed to a priority encoder which selects the highest priority unmasked active request. The priorities are permanently assigned. An active output from the priority encoder causes a jump to the interrupt vector location.

The latency from when an external interrupt occurs to when the first instruction of the interrupt routine is executed is at least two full cycles. The interrupt controller requires one full cycle after the cycle in which an external interrupt occurs to synchronize the interrupt internally (assuming that setup and hold times are met; see the *ADSP-2101 Data Sheet* for timing requirements). Another cycle is needed to fetch the instruction at the interrupt vector location. During this cycle, the ADSP-2101 executes a NOP instead of the instruction that would have been executed in this cycle. The address of the aborted instruction is pushed on the PC stack so that it will be fetched when the interrupt service is completed.

Interrupt vectors are four locations apart. Because of the efficiency of the ADSP-2101 assembly language, many simple service routines could be contained entirely in this space. For a longer service routine, an overhead cycle would be incurred for a JUMP instruction (at the interrupt location) to the start of the longer 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-2101 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.

After the interrupt has been serviced, 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 previous machine status.

4.4 STATUS REGISTERS AND STACK

The status and mode bits of the ADSP-2101 are maintained internally within six registers, each of which are independently readable over the DMD bus, and five of which can be written to from the DMD bus. These registers are:

ASTAT	Arithmetic status	
SSTAT	Stack status	*(read-only)*
MSTAT	Mode status	
ICNTL	Interrupt control	
IMASK	Interrupt mask	
IFC	Interrupt force and clear	*(write-only)*

The interrupt configuring status registers are described in the previous section. The other three are discussed below.

The status stack is 7 locations deep by 21 bits wide. The current ASTAT, MSTAT and IMASK values are pushed on this stack when a jump to an interrupt routine is executed and are popped upon the return from the interrupt routine. The seven stack locations accommodate nesting of all six interrupts plus one other that is used only by the ADSP-2101 Emulator.

4.4.1 Arithmetic Status Register (ASTAT)

ASTAT is eight 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 ASTAT 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 sets the AZ and AN bits for a given X or Y operand and clears AC.

4.4.2 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 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.

For example, the count stack (which is four deep) is overflowed by five successive pushes. Five successive pops will restore the stack empty condition, but cannot remove the overflow.

Since SSTAT is a read-only register, write operations have no effect on the stack status bits either. The processor must be reset to clear the stack overflow status.

4.4.3 Mode Status Register (MSTAT)

MSTAT is a seven-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
Bit 4	MAC Result Placement Mode
Bit 5	Timer Enable
Bit 6	Go Mode (Execute during Bus Grant)

MSTAT (like most registers) can be changed by moving a new value into it with any of the MOVE instructions. In contrast to the other status registers, MSTAT can also be changed with the MODE CONTROL instruction. The MODE instruction offers a high-level, self-documenting method for changing mode status bits; see Chapter 9, "Instruction Set Overview," for more information.

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 an 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 whenever an ALU overflow occurs.

The MAC result placement mode determines whether or not the left shift is made between the multiplier product and the MR register. This mode is fully discussed in Chapter 2, "Computational Units."

Setting the timer enable bit starts the timer decrementing logic. Clearing it halts the timer.

The "GO" mode allows the ADSP-2101 to continue executing instructions during a bus grant. In the microprocessor ADSP-2100 access to external memory was essential for fetching instructions and/or data. In the microcomputer ADSP-2101 this is often not true. The GO mode allows the processor to run; only if an external memory access is required does the processor halt waiting for the bus to be released.

4.5 IDLE

The ADSP-2101 IDLE instruction causes the processor to wait indefinitely in a low-power state until an interrupt occurs. When an interrupt occurs, it is serviced; then execution continues with the instruction following IDLE.

4.6 CONDITION LOGIC

The condition logic of the ADSP-2101 is used to determine whether a specified action in a conditional instruction is performed, such as a jump, 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 four-bit condition code field of the instruction and the four-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, given in Table 4.4 on the next page, are for the standard IF condition statement. In addition, the status of the Flag In (FI) pin of the processor may be used as a condition for the JUMP and CALL instructions only.

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
FLAG_IN*	Flag In	FI pin last sampled 1
NOT FLAG_IN*	Not Flag In	FI pin last sampled 0

Table 4.4 IF Condition Logic

*Only available on JUMP and CALL instructions

Timer 5

5.1 OVERVIEW

The ADSP-2101 programmable interval timer can generate periodic interrupts based on multiples of the processor's cycle time. When enabled, a 16-bit count register is decremented every *n* cycles, where *n–1* is a scaling value stored in an 8-bit register. When the value of the count register reaches zero, an interrupt is generated and the count register is reloaded from a 16-bit period register.

The scaling feature of the ADSP-2101 timer allows the 16-bit counter to generate periodic interrupts over a wide range of periods. Given a processor cycle time of 80ns, the timer can generate interrupts with periods of 80ns up to 5.24ms with a zero scale value. When scaling is used, time periods can range up to 1.34 seconds.

Timer interrupts can be masked, cleared and forced in software if desired. For additional information, refer to the section "Interrupts" in Chapter 4, "Program Control."

5.2 TIMER ARCHITECTURE

The ADSP-2101 Timer includes two 16-bit registers, TCOUNT and TPERIOD and one 8-bit register, TSCALE. The extended mode control instruction enables and disables the timer by setting and clearing bit 5 in the mode status register, MSTAT. For a description of the mode control instructions, refer to the *ADSP-2101 Cross Software Manual*. The timer registers, which are memory-mapped, are shown in Figure 5.1 (on the following page).

TCOUNT is the count register. When the timer is enabled, it is decremented as often as once every instruction cycle. When the counter reaches zero, an interrupt is generated. TCOUNT is then reloaded from the TPERIOD register and the count begins again.

5 Timer

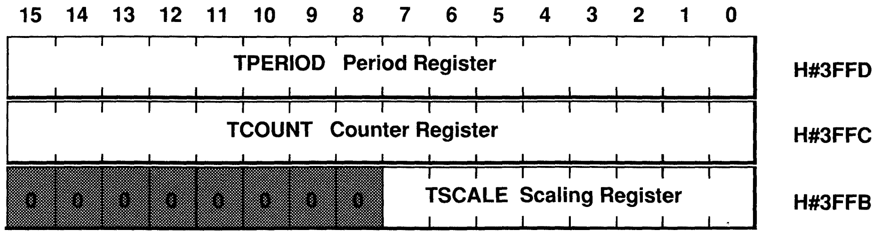


Figure 5.1 Timer Registers

TSCALE stores a scaling value that is one less than the number of cycles between decrements of TCOUNT. For example, if the value in TSCALE register is 0, the counter register decrements once every cycle. If the value in TSCALE is 1, the counter decrements once every 2 cycles. Figure 5.2 shows the timer block diagram.

5.3 RESOLUTION

TSCALE provides the capability to program longer time intervals between interrupts, extending the range of the 16-bit TCOUNT register. Table 5.1 shows the range and the relationship between period length and resolution for TPERIOD = maximum.

Cycle Time = 100ns

TSCALE	*Interrupt Every...*	*Resolution*
0	6.55ms	100ns
255	1.678s	25.6µs

Cycle Time = 80ns

TSCALE	*Interrupt Every ...*	*Resolution*
0	5 .24ms	80ns
255	1.34s	20.48µs

Table 5.1 Timer Range & Resolution

5.4 EXAMPLE

Table 5.2 shows the effect of operating the timer with TPERIOD = five, TSCALE = one and TCOUNT = five. After the timer is enabled (cycle n–1) the counter begins. Because TSCALE is one, the decrementing occurs every two cycles. The reloading of TCOUNT and continuation of the counting occurs, as shown, during the interrupt service routine.

Cycle	*TCOUNT*	*Action*
n–4		TPERIOD loaded with 5
n–3		TSCALE loaded with 1
n–2		TCOUNT loaded with 5
n–*1*	5	ENA TIMER executed
n	5	since TSCALE = 1, no decrement
n+1	5	decrement TCOUNT
n+2	4	no decrement
n+3	4	decrement TCOUNT
n+4	3	no decrement
n+5	3	decrement TCOUNT
n+6	2	no decrement
n+7	2	decrement TCOUNT
n+8	1	no decrement
n+9	1	decrement TCOUNT
n+10	0	no decrement
n+11	0	zero reached, interrupt occurs load TCOUNT from TPERIOD
n+12	5	no decrement
n+13	5	decrement TCOUNT
n+14	4	no decrement
n+15	4	decrement TCOUNT, etc..

Table 5.2 Example of Timer Operation

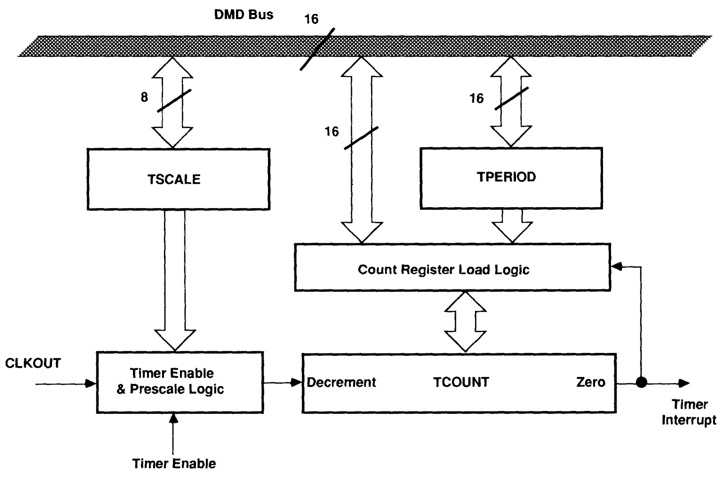


Figure 5.2 Timer Block Diagram

5.5 SUMMARY

Interrupts operate using this formula: one interrupt occurs every (TPERIOD +1) * (TSCALE +1) cycles. To set the first interrupt at a different time interval from subsequent interrupts, load TCOUNT with a different value from TPERIOD. The formula for the first interrupt is (TCOUNT+1) * (TSCALE+1).

If you write a new value to TSCALE or TCOUNT, the change is effective immediately . If you write a new value to TPERIOD, the change does not take effect until after TCOUNT is reloaded.

Serial Ports ■ 6

6.1 OVERVIEW

The ADSP-2101 has two serial ports, SPORT0 and SPORT1, that support a wide variety of serial data communications schemes and allow for several possible interprocessor communication methods in multiple ADSP-2101 systems.

Discussion of the ADSP-2101's external interface is presented in three parts. This chapter discusses the serial port interface. The next chapter discusses the memory interface and associated control lines. The following chapter discusses the control interface. Waveforms shown in this manual describe only the relationships of the signals depicted. Consult the data sheet for actual timing characteristics.

6.1.1 Basic Features of SPORTS

Each SPORT has a five-pin interface consisting of the following signal names.

Name	*Function*
SCLK	Serial clock I/O
RFS	Receive frame synchronization
TFS	Transmit frame synchronization
DR	Serial data receive
DT	Serial data transmit

Table 6.1 SPORT External Signals and Pins

Here is a brief list of the capabilities of the ADSP-2101 SPORTs. Figure 6.1, on page 6-3, shows a simplified block diagram of a single SPORT.

- Bidirectional: each SPORT has a separate transmit and receive section.

- Double-buffered: each SPORT section (both receive and transmit) has a data register accessible to the user and an internal transfer register. The double-buffering provides additional time to service the SPORT.

- Flexible clocking: each SPORT can use an external serial clock or generate its own in a wide range of frequencies.
- Flexible framing: each SPORT section (receive and transmit) can run with or without frame synchronization signals for each data word; with internally-generated or externally-generated frame synchronization signals; with active high or inverted frame signals; with either of two pulse widths / timing.
- Flexible word length: each SPORT supports serial data word lengths from three to sixteen bits.
- Companding in hardware: each SPORT provides optional A-law and μ-law companding according to CCITT recommendation G.711.
- Flexible interrupt scheme: each SPORT section (receive and transmit) can generate a unique interrupt upon completing a data word transfer or after transferring an entire buffer (see next item).
- Auto-buffering with single-cycle overhead: using the ADSP-2101 DAGs, each SPORT can receive and/or transmit an entire circular buffer of data with an overhead of only one cycle per data word. Transfers to and from the SPORT and the circular buffer are automatic in this mode and do not require additional programming. An interrupt is generated only when pointer wraparound occurs in the circular buffer.
- Multichannel capability: SPORT0 provides a multichannel interface for selective receipt and transmission of arbitrary data channels from a twenty-four or thirty-two word, time-division multiplexed, serial bitstream. This is especially useful for T1 interfaces or as a network communication scheme for multiple processors.
- Alternate configuration: SPORT1 can be configured as two external interrupt inputs, $\overline{\text{IRQ0}}$ and $\overline{\text{IRQ1}}$, and the Flag In and Flag Out signals instead of as a serial port. The internally generated serial clock may still be used in this configuration.

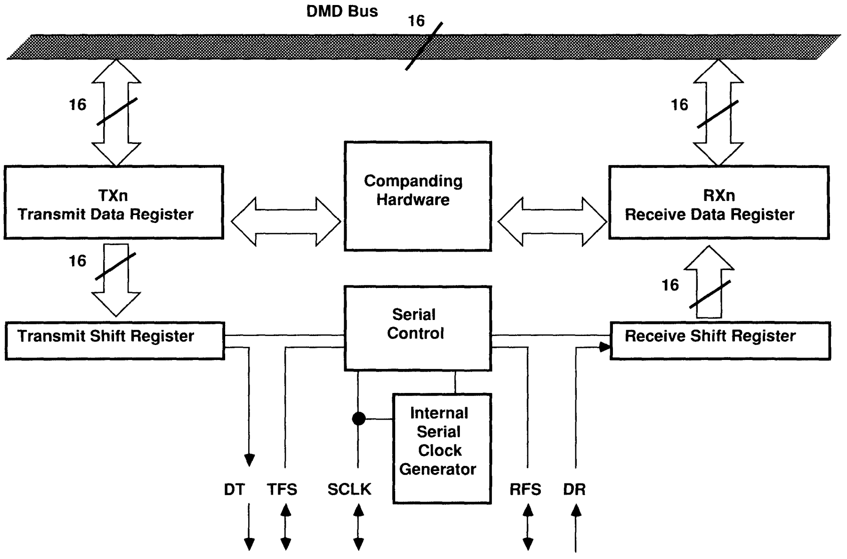


Figure 6.1 Serial Port Block Diagram

6.2 SERIAL CLOCKS

Each SPORT operates on its own serial clock signal. The serial clock (SCLK) can be internally generated or received from an external source.

The ISCLK bit in the SPORT control register determines the SCLK source. As shown in Figure 6.2 on the next page, ISCLK resets to zero, the external clock mode. External clock frequencies may be as high as the processor's cycle rate; internal clock frequencies may be as high as one-half the processor's clock rate.

When ISCLK is set, internal generation of the SCLK signal begins, whether or not the corresponding SPORT is enabled. The frequency of the internally-generated SPORT clock is a function of the value of the 16-bit serial clock divisor register, SCLKDIV, and CLKOUT. The formula is:

$$\text{SCLK}_{\text{FREQUENCY}} = \frac{\text{CLKOUT}_{\text{FREQUENCY}}}{2 \text{ x } (\text{SCLKDIV} + 1)}$$

Table 6.2 Formula for Internal SPORT Clock Frequency

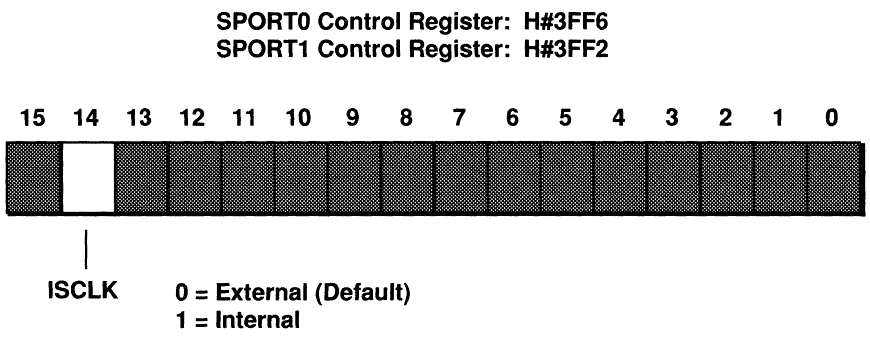


Figure 6.2 ISCLK Bit in SPORT Control Register

Table 6.3 shows how some of the commonly required SCLK frequencies can be derived from CLKOUT by SCLKDIV.

Processor Operating Frequency: 12.288MHz

SCLKDIV	*SCLK Frequency*
20479	300 Hz
5119	1200
639	9600
95	64 kHz
3	1.536 MHz
2	2.048
0	6.144

Table 6.3 Examples of Common SPORT Frequencies (Internally-Generated)

Note that the serial clock of SPORT1 (the SCLK pin) still functions when the port is being used in its alternate configuration (as FO, FI and two interrupts). In this case, SCLK is unresponsive to an external clock, but can internally generate a clock as described above.

The 16-bit SCLKDIV registers are memory-mapped. SCLKDIV for SPORT0 is located at H#3FF5 and for SPORT1 at H#3FF1.

6.3 FRAMING OPTIONS

Framing signals identify the beginning of each serial word transfer. The ADSP-2101 SPORTs have great flexibility in the ways framing signals are handled. Each SPORT has its own control register containing the control bit fields described in this section. Timing examples for the various framing options described in this section are shown later in "Waveform Examples."

Transmit and receive framing are independent of each other as well. The mnemonics "TFS" and (transmit frame synchronization) and "RFS" (receive frame synchronization) appear in the name of signals and control bits governing framing.

6.3.1 Frame Synchronization: RFSR / TFSR

Communications may be with or without frame synchronization signals for each data word. If the RFS required (RFSR) or the TFS required (TFSR) bit is zero, a frame signal is necessary to initiate communications but is ignored after the first bit is transferred. Words are then transferred continuously, unframed. If the RFSR or TFSR bit is one, a frame signal is required for every data word. These bits, shown in Figure 6.3, are both zeros at reset, requiring no frame synchronization (unframed mode).

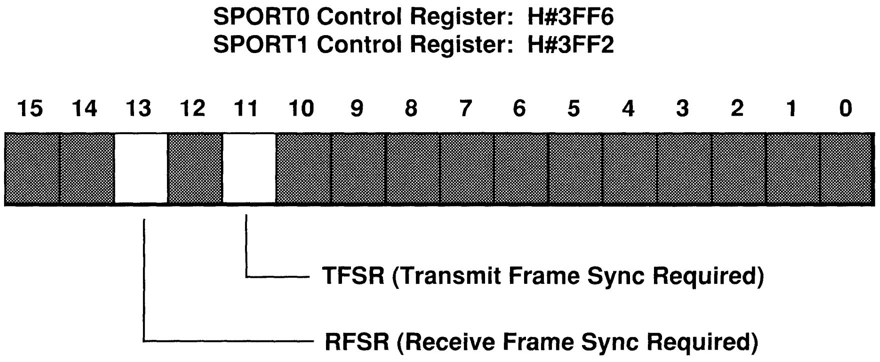

SPORT0 Control Register: H#3FF6
SPORT1 Control Register: H#3FF2
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
TFSR (Transmit Frame Sync Required)
RFSR (Receive Frame Sync Required)

Figure 6.3 Framing Required Bits in SPORT Control Register

6.3.2 External or Internal: IRFS / ITFS

The internal frame synchronization bits (IRFS and ITFS) determine whether the frame synchronization signal is generated internally or supplied externally. See Figure 6.4. Both of these bits are zeros at reset, requiring external frame synchronization signals.

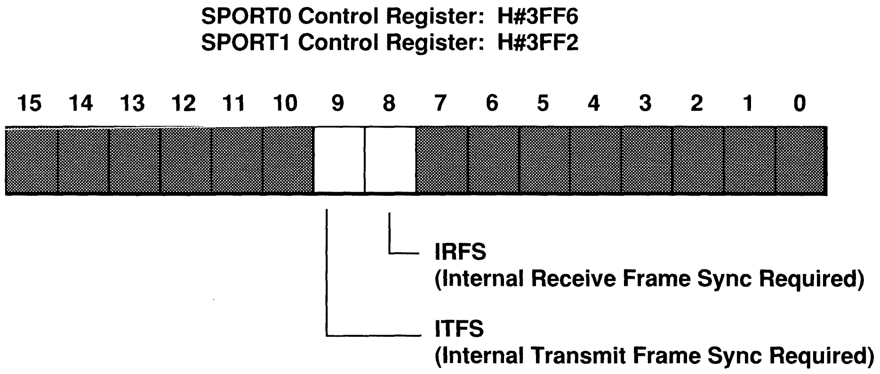

SPORT0 Control Register: H#3FF6
SPORT1 Control Register: H#3FF2
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
IRFS
(Internal Receive Frame Sync Required)
ITFS
(Internal Transmit Frame Sync Required)

Figure 6.4 Internal Framing Bits in SPORT Control Register

For transmit operations, the internal framing signal is generated by the presence of "fresh" data for transmission. After the data is loaded into the transmit register from the data register, the internal framing signal occurs at the time needed to ensure continuous data transmission after the last bit of the current word is transmitted (the exact time depends on the framing mode being used).

For receive operations, the internal framing signal is generated periodically as a function of SCLK, based on the value in the 16-bit RFSDIV register. The formula is:

Number of SCLK cycles between RFS assertions = RFSDIV + 1

Values of RFSDIV+1 that are less than the word length are not supported and may cause unpredictable operation.

The RFSDIV registers are memory-mapped. RFSDIV for SPORT0 is located at H#3FF4 and for SPORT1 at H#3FF0.

Note that RFS may be generated internally even when SCLK is supplied externally. This provides a way to divide external clocks for any purpose.

6.3.3 Normal or Alternate Framing Mode: RFSW / TFSW

In the normal framing mode, the framing signal is checked at the falling edge of SCLK. If the framing signal is asserted, data is available on or latched on the *next* falling edge of SCLK, and the framing signal is not checked again until the word has been transmitted or received. If data transmission or reception is continuous, i.e., the last bit of one word is followed without a break by the first bit of the next word, then the framing signal should occur in the same SCLK cycle as the last bit of each word. See Figures 6.9 and 6.15.

The alternate framing mode is selected by setting the RFSW or TFSW bit (shown in Figure 6.5) to one. RFS or TFS should be asserted in the *same* SCLK cycle as the first bit of a word. The data bits are latched on the falling edge of SCLK, but RFS or TFS is checked only on the first bit. Internally-generated TFS and RFS signals remain asserted for the length of the serial word. Externally-generated TFS and RFS signals are only checked during the first bit time. See Figures 6.10, 6.11, 6.16 and 6.17.

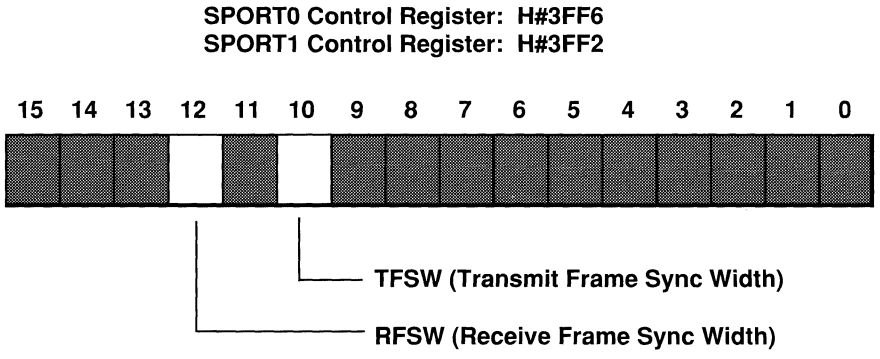

SPORT0 Control Register: H#3FF6
SPORT1 Control Register: H#3FF2
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
TFSW (Transmit Frame Sync Width)
RFSW (Receive Frame Sync Width)

Figure 6.5 Framing Width Bits in SPORT Control Register

6.3.4 Active High or Inverse Sense: INVRFS / INVTFS

The INVRFS (invert RFS) and INVTFS (invert TFS) bits in the SPORT control register are both zeros at reset, selecting the normal, active high mode of operation. When one of these bits is set, the corresponding framing signal (whatever other modes are selected) is inverted. This applies equally to internally- or externally-generated frame signals.

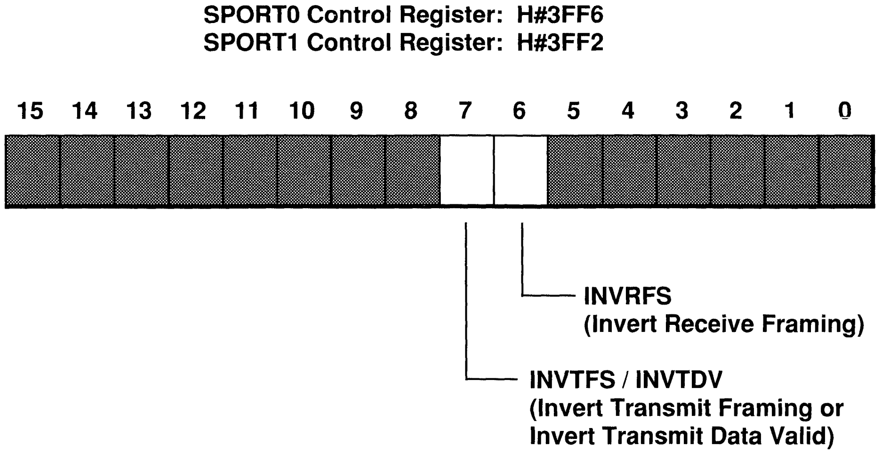

SPORT0 Control Register: H#3FF6
SPORT1 Control Register: H#3FF2
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
INVRFS
(Invert Receive Framing)
INVTFS / INVTDV
(Invert Transmit Framing or Invert Transmit Data Valid)

Figure 6.6 Active High/Low Bits in SPORT Control Register

6.4 SPORT WORD LENGTH: SLEN

Each SPORT independently handles words of three to sixteen bits. The SLEN (serial word length) field in the SPORT control register controls this according to the simple formula:

Serial Word Length = SLEN value + 1

Do not set SLEN to zero or one; these SLEN values are not permitted.

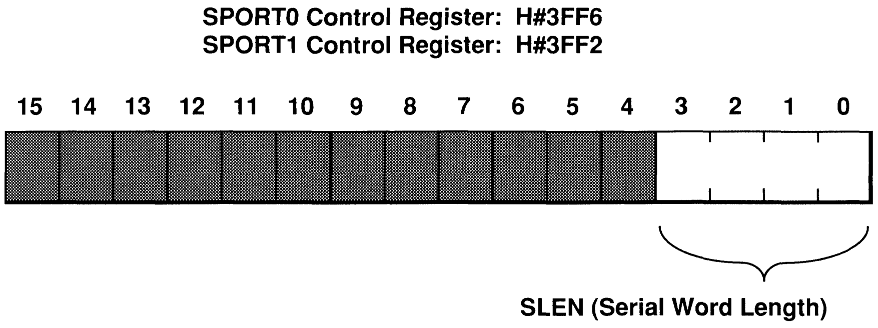

SPORT0 Control Register: H#3FF6
SPORT1 Control Register: H#3FF2
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
SLEN (Serial Word Length)

Figure 6.7 SLEN Field in SPORT Control Register

6.5 WAVEFORM EXAMPLES

Following are signal waveform examples of some combinations of the options described above. While these appear in the form of timing diagrams, they are intended to convey the relative operation of the signals, not the specifics of the actual timing. Consult the data sheet for actual timing parameters and values.

All of the following figures assume a word length of four bits, that is, SLEN = 3. Framing signals are the normal, active high form, that is, INVRFS and INVTFS = 0.

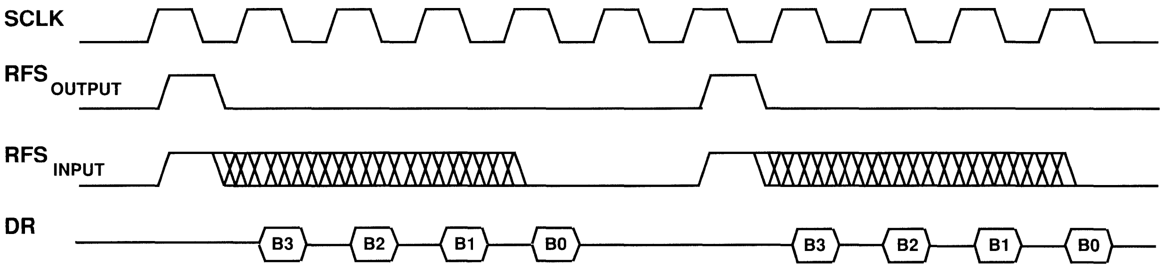


Figure 6.8 SPORT Receive, Normal Framing (Internal Framing Option and External Framing Option Both Shown)

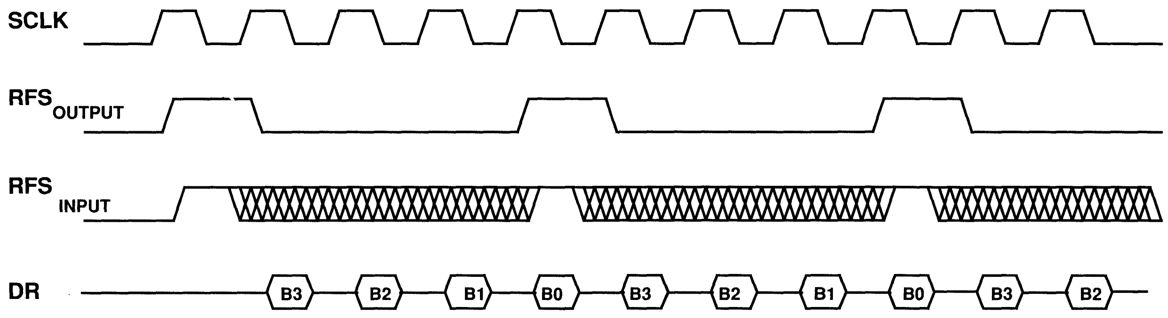


Figure 6.9 SPORT Continuous Receive, Normal Framing (Internal Framing Option and External Framing Option Both Shown)

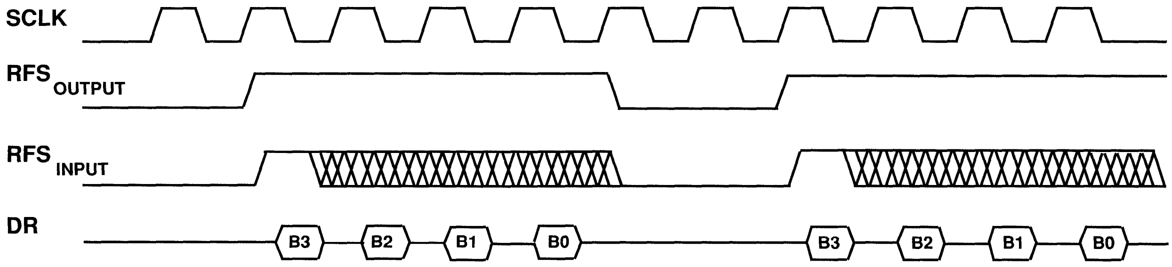


Figure 6.10 SPORT Receive, Alternate Framing (Internal Framing Option and External Framing Option Both Shown)

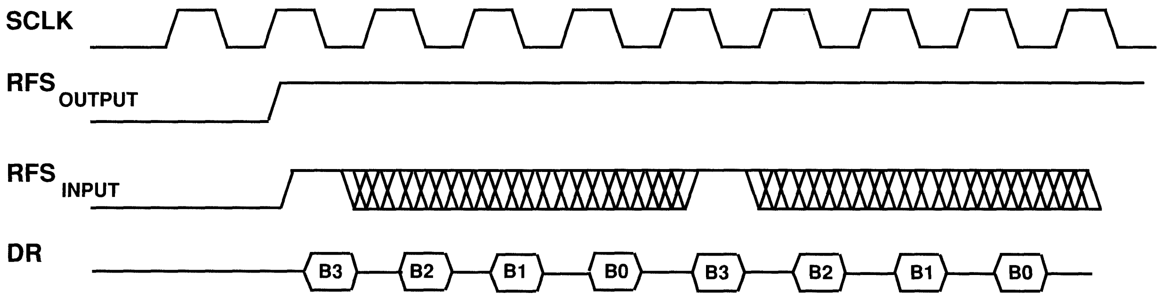


Figure 6.11 SPORT Continuous Receive, Alternate Framing (Internal Framing Option and External Framing Option Both Shown)

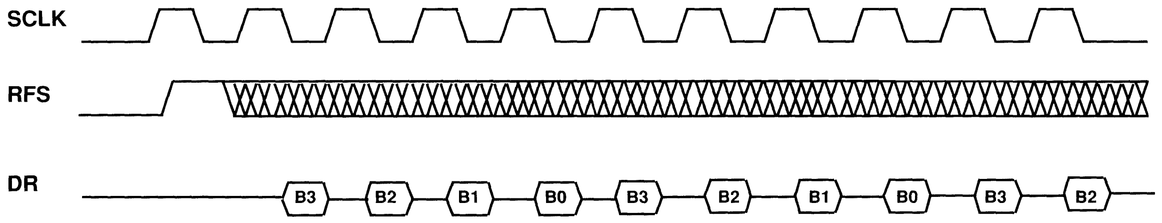


Figure 6.12 SPORT Receive, Unframed Mode, Normal Framing

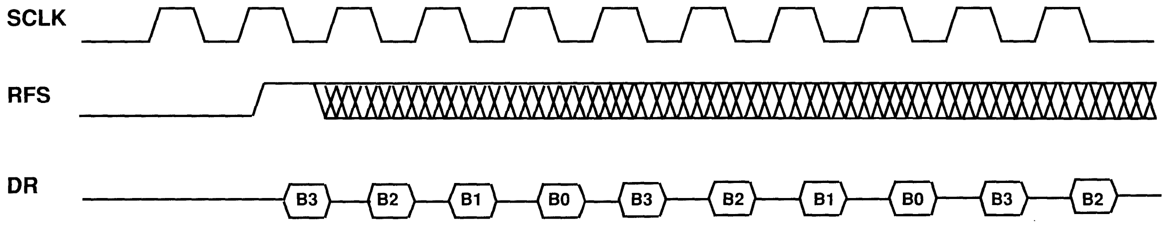


Figure 6.13 SPORT Receive, Unframed Mode, Alternate Framing

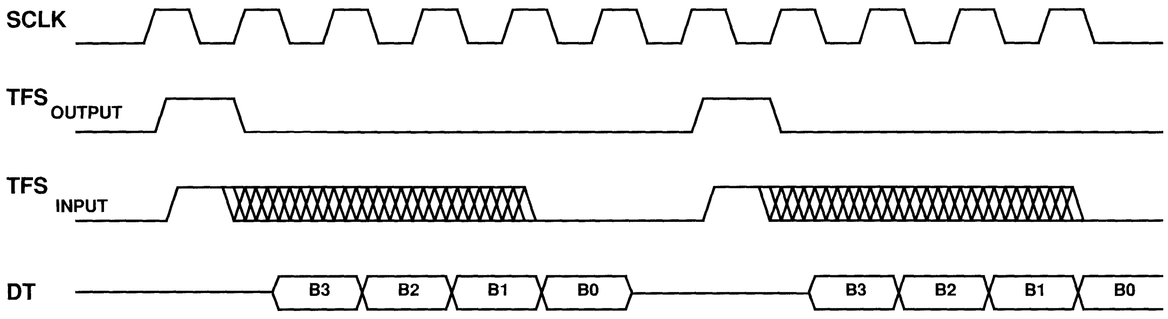


Figure 6.14 SPORT Transmit, Normal Framing (Internal Framing Option and External Framing Option Both Shown)

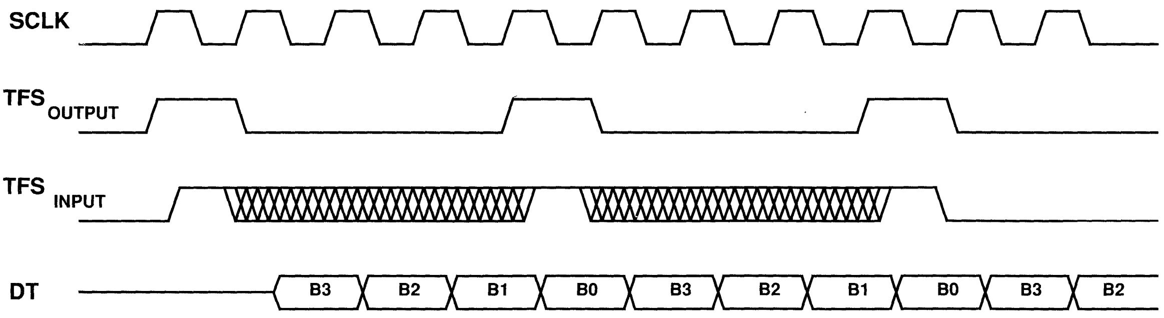


Figure 6.15 SPORT Continuous Transmit, Normal Framing (Internal Framing Option and External Framing Option Both Shown)

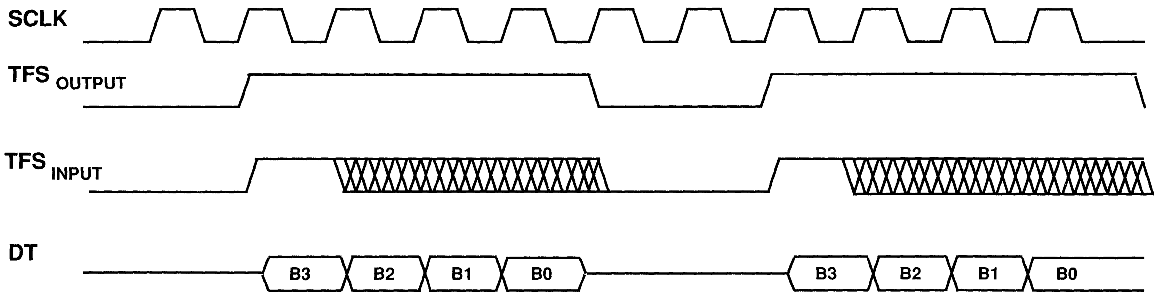


Figure 6.16 SPORT Transmit, Alternate Framing (Internal Framing Option and External Framing Option Both Shown)

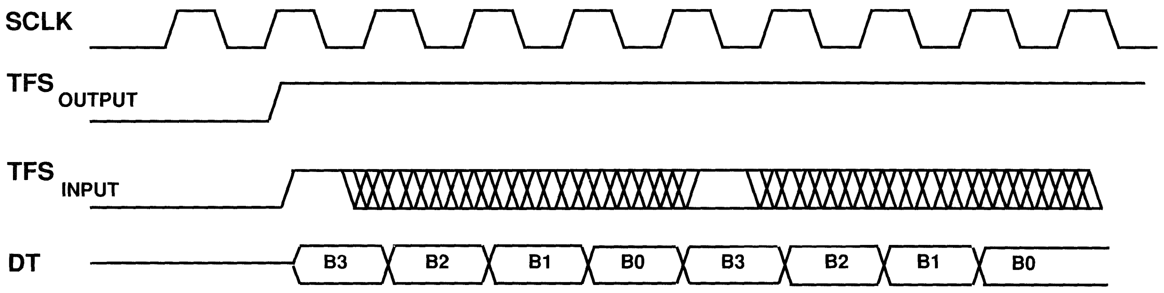


Figure 6.17 SPORT Continuous Transmit, Alternate Framing (Internal Framing Option and External Framing Option Both Shown)

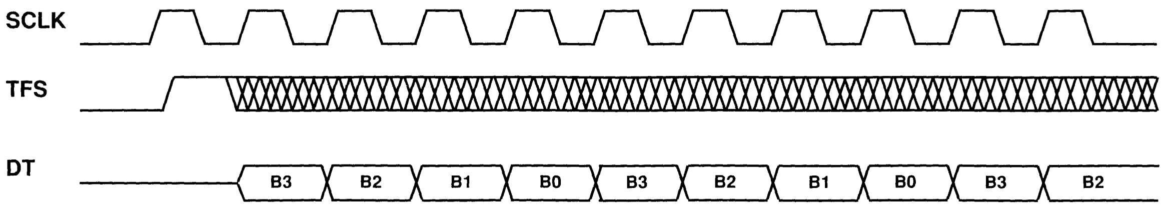


Figure 6.18 SPORT Transmit, Unframed Mode, Normal Framing

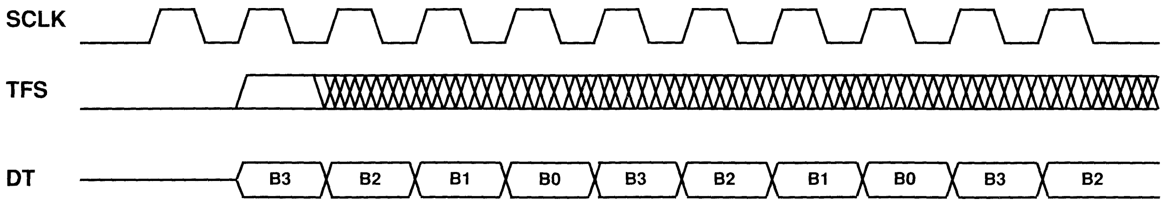


Figure 6.19 SPORT Transmit, Unframed Mode, Alternate Framing

6.6 DATA REGISTERS & COMPANDING

Each SPORT has a transmit and a receive register; SPORT0's registers are RX0 and TX0, SPORT1's are RX1 and TX1. Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding data to minimize the number of bits that must be sent. Both SPORTs share the companding hardware: one expansion and one compression operation can occur in each processor cycle. In the event of contention, SPORT0 has priority. The ADSP-2101 supports both of the widely used algorithms for companding: A-law and μ-law. The type of companding can be independently selected for each SPORT.

Figure 6.1 shows the two data registers associated with each SPORT. These registers, TXn and RXn, are identified by name in the ADSP-2101 assembly language, not memory-mapped.

TXn and RXn can be read and written (like other non-data registers) with the following instructions: read/write to data memory (direct address), load non-data immediate, and internal (register-to-register) moves.

See Appendix A, Instruction Coding, for additional information and consult the instruction set reference found in the *ADSP-2101 Cross-Software Manual.*

6.6.1 Simple Operation Example

There are two ways to generate the SPORT interrupts, after the transmisstion or receipt of 1) each data word or 2) each complete buffer of data words. This section discusses the first method. Section 6.7, "Interrupts & Autobuffering," discusses the second.

Writing to the TXn register readies the SPORT for transmission; the TFS signal initiates it. The value in TXn is written to the internal transmit register and, after framing synchronization has occurred (if required), the bits are sent, MSB first.

When the first bit has been transferred, the SPORT generates the transmit interrupt. TXn is now available for the next piece of data, even though the transmission of the first is not complete.

In the receiving section, bits accumulate as they are received in an internal receive register. When a complete word has been received, it is written to the RXn register and the receive interrupt for that SPORT is generated.

6.6.2 Companding & Data Format: DTYPE

Companding is done according to the CCITT G.711 recommendation. Companding and data format are controlled by the DTYPE field in the SPORT control register (shown in Figure 6.20) and described in Table 6.4.

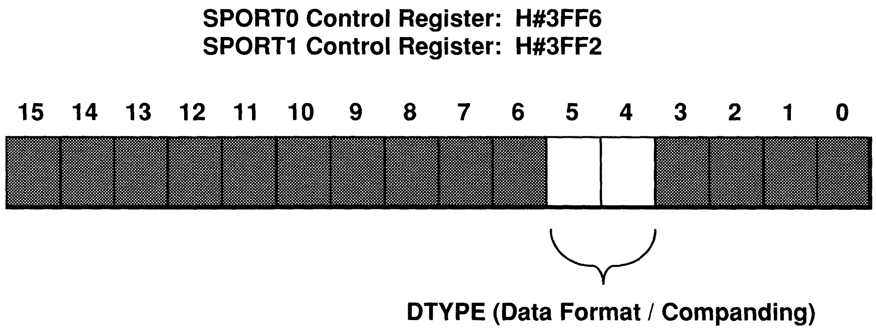

SPORT0 Control Register: H#3FF6
SPORT1 Control Register: H#3FF2
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
DTYPE (Data Format / Companding)

Figure 6.20 DTYPE Field in SPORT Control Register

DTYPE	*Format of data*
`00`	Right justify, zero fill unused MSBs
`01`	Right justify, sign extend into unused MSBs
`10`	Compand using µ-law
`11`	Compand using A-law

Table 6.4 DTYPE Field Values

6.6.2.1 Companding Internal Data

When companding is enabled, valid data in RXn is the right-justified, sign-extended, expanded value of the eight LSBs received. Likewise, a write to TXn causes the 16-bit value to be compressed to eight LSBs (sign-extended to the width of the transmit word) before being written to the internal transmit register. If the 16-bit value is greater than the 13-bit A-law or 14-bit µ-law maximum, it is automatically compressed to the maximum value.

Because the values in the RXn and TXn registers are actually companded "in place" it is possible to use the companding hardware internally, without any transmission at all, such as for debugging and testing. This requires a single cycle of overhead.

With companding enabled:

1. Write data to TXn (compression is calculated).
2. Wait for one cycle (TXn is written with compressed value)
3. Read TXn (it returns the eight-bit compressed data)

Exactly the same approach works for expanding data, using RXn instead of TXn.

6.6.3 Companding Operation Example

With hardware companding, interfacing to a codec requires little additional programming effort. See the codec hardware interfacing example in the last section of this chapter.

Here is a typical sequence of operations for transmitting companded data:

- Write data to the TXn register
- The value in TXn is compressed
- The compressed value is written back to TXn
- After the frame sync signal has occurred (if required), TXn is written to the internal transmit register and the bits are sent, MSB first.

As soon as the SPORT has started to send the second bit of the current word, TXn can be written with the next word, even though transmission of the first is not complete. When the first bit has been transferred, the SPORT generates the transmit interrupt to indicate that TXn is ready for the next data word. If the framing signal is being provided externally, the next word must be written to TXn early enough to allow for compression before the next framing signal arrives.

Here is a typical sequence of operations for receiving companded data:

- Bits accumulate as received in the internal receive register
- When a complete word is received, it is written to RXn
- The value in RXn is expanded
- The expanded value is written back to RXn

The receive interrupt for that SPORT is then generated.

6.6.4 Contention For Companding Hardware

Since both SPORTs share the companding hardware, only one compression and one expansion operation can take place during a single machine cycle. If contention arises, such as when two expansions need to occur in the same cycle, SPORT0 has priority, while SPORT1 is forced to wait one cycle.

The effects of contention, however, are usually small. The instruction set does not support loading both TX0 and TX1 in the same cycle; consequently these operations will be naturally out of phase for contention in many cases. The overhead cycle for the receive operation occurs prior to the receive interrupt and does not increase the time needed to service the interrupt, although it does affect the interval between interrupts.

6.7 INTERRUPTS & AUTOBUFFERING

Four interrupts (out of the six available) are assigned to serial port activity: each SPORT has a receive and a transmit interrupt. The priority of these interrupts is shown below in Table 6.5.

Highest	SPORT0 Transmit
	SPORT0 Receive
	SPORT1 Transmit
Lowest	SPORT1 Receive

Table 6.5 SPORT Interrupt Priorities

For complete details about how interrupts are handled in the ADSP-2101, see the Interrupts section in Chapter 4, "Program Control."

6.7.1 Autobuffering Operation

Autobuffering provides a mechanism that allows an entire block of serial data to be received or transmitted before an interrupt is generated. Service routines can operate on the entire block of data, rather than on a single word, resulting in a significant reduction of overhead. Autobuffering uses the circular buffer addressing of the DAGs. (Refer back to Chapter 3, "Data Moves," for information on the DAGs.)

With autobuffering enabled, each serial data word is transferred (if multichannel operation is enabled, only active words are transferred) to or from data memory in a single overhead cycle. (Autobuffering to program memory is not supported.) This "overhead" cycle occurs independently of the instructions being executed and effectively suspends execution for one cycle (or more, if wait states are required) when it happens. No interrupt is generated for these individual data word transfers.

This transfer is not an operation that can be executed directly in the assembly language. The transfer could be *expressed* in ADSP-2101 assembly language as:

```
DM(I,M)  = RX0
    or                          Equivalent Instructions Only
TX0  = DM(I,M)
```

The I and M registers used in the transfer are selected by fields in the SPORT's autobuffer control register.

The processor waits for the current instruction to finish before inserting the overhead cycle. A delay in the autobuffer transfer occurs if the transfer is required during an instruction executing in multiple cycles (for wait states, for example). If the transfer is required when the ADSP-2101 is waiting in an IDLE state, the transfer is executed and the processor returns to IDLE.

When a data word transfer causes the circular buffer pointer to wrap, the SPORT interrupt is generated. The interrupt occurs, in other words, after the complete buffer has been transmitted or received.

Aside from the completion of an instruction requiring multiple cycles, the automatic transfer of individual data words has the highest priority of any operation short of $\overline{\text{RESET}}$, including all interrupts. Thus, it is possible for an autobuffer transfer to increase the latency of an interrupt response if the interrupt happens to coincide with the transfer. Up to four autobuffered transfers can occur; they are prioritized exactly as the SPORT interrupts shown in Table 6.5 above. In the worst case that all four autobuffer transfers are required at about the same time, interrupt latency could increase by four cycles.

6.7.2 Autobuffering Control Register

In autobuffering mode, the interrupt is only generated when the modification of a specified I register (in the DAG) by the modify value in the specified M register (in the DAG) causes a modulus overflow. This corresponds to pointer wraparound in the circular buffer or, in other words, detection of the end of the buffer.

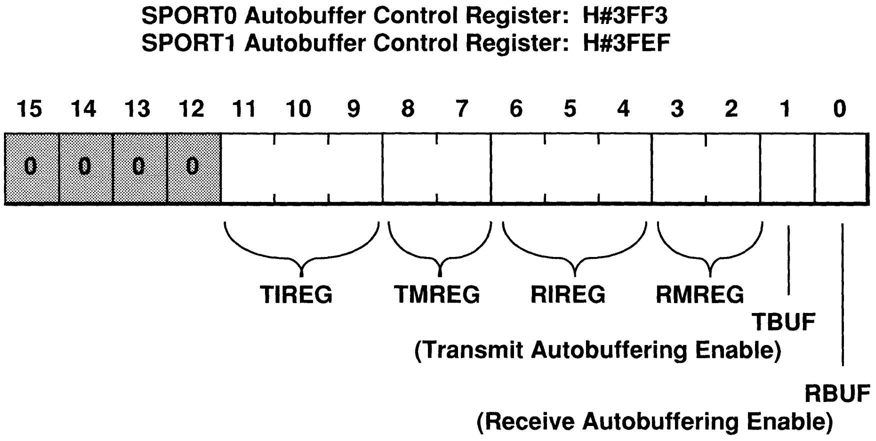

SPORT0 Autobuffer Control Register: H#3FF3
SPORT1 Autobuffer Control Register: H#3FEF
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0
TIREG
TMREG
RIREG
RMREG
TBUF
(Transmit Autobuffering Enable)
RBUF
(Receive Autobuffering Enable)

Figure 6.21 SPORT Autobuffer Control Register

The I and M registers are identified in the SPORTn autobuffer control register. Autobuffering is enabled separately for receive and transmit by the RBUF and TBUF bits in the same autobuffered control register.

TIREG identifies the I register associated with the transmit buffer and TMREG identifies the M register for this same buffer. The rules governing the pairing of I and M registers are the same as for other DAG operations. You can only mix I and M registers within the same DAG, such as I0-I3 with M0-M3. Consequently, once a specific (0-7) I register has been selected, only two bits are necessary to indicate the associated M register.

6.8 MULTICHANNEL OPERATION

SPORT0 also supports a multichannel function. In the multichannel mode of operation the SPORT automatically selects enabled words from a twenty-four or thirty-two word block of time-division multiplexed serial data.

In single-channel mode, receive and transmit framing identifies the start of a single word or continuous stream, with independent receive and transmit operation. In the multichannel mode, the receive frame synch signal, RFS, identifies the start of a twenty-four or thirty-two word block of serial data with the receiver and transmitter operating at the same time.

6.8.1 Multichannel Set Up

Multichannel operation is enabled by a bit in SPORT0's control register. Other control bits in this register have a meaning in multichannel mode that is different from single-channel operation, as shown in Figure 6.22.

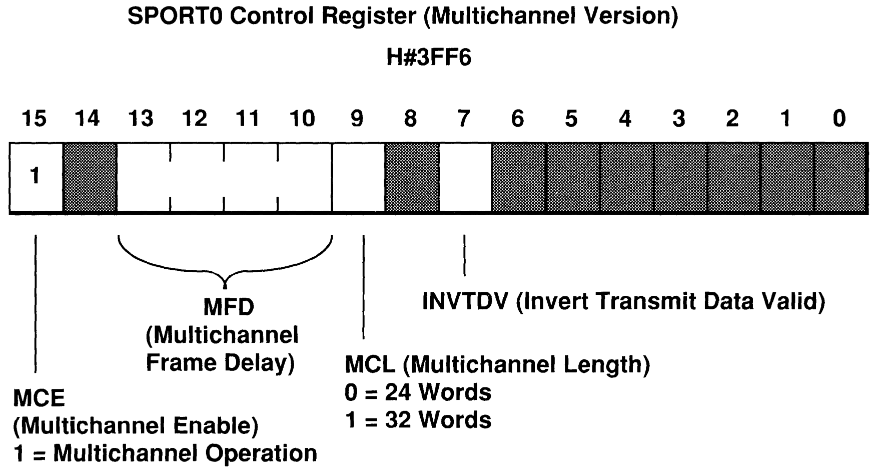

SPORT0 Control Register (Multichannel Version)
H#3FF6
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1
MFD (Multichannel Frame Delay)
INVTDV (Invert Transmit Data Valid)
MCL (Multichannel Length)
0 = 24 Words
1 = 32 Words
MCE (Multichannel Enable)
1 = Multichannel Operation

Figure 6.22 SPORT0 Control Register (Multichannel Version)

Multichannel operation becomes active when a one is written into the multichannel enable bit of the control word, MCE. MCE is zero after $\overline{\text{RESET}}$.

The state of the multichannel length bit (MCL) determines whether the block length is twenty-four or thirty-two words. A zero selects twenty-four word blocks, a one, thirty-two word blocks.

Multichannel frame delay (MFD) is a four-bit field specifying the number of serial clock cycles between the frame signal and the first data bit. This allows the ADSP-2101 to work with T1 interfaces of different types. Figure 6.23 shows a variety of delays.

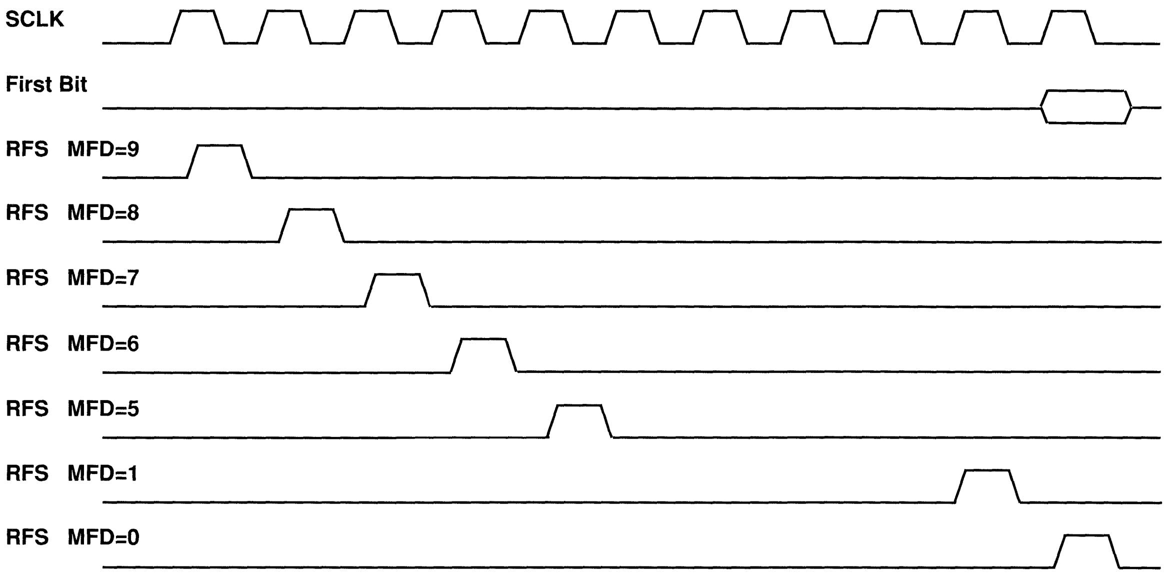


Figure 6.23 SPORT Multichannel Frame Delay Examples

You designate the active words for the receive and transmit operations independently by setting any combination of bits in the thirty-two-bit receive and transmit enable register, each made up of two contiguous sixteen-bit registers, as shown in Figure 6.24. For example, setting bit zero selects word zero, bit twelve selects word twelve and so on.

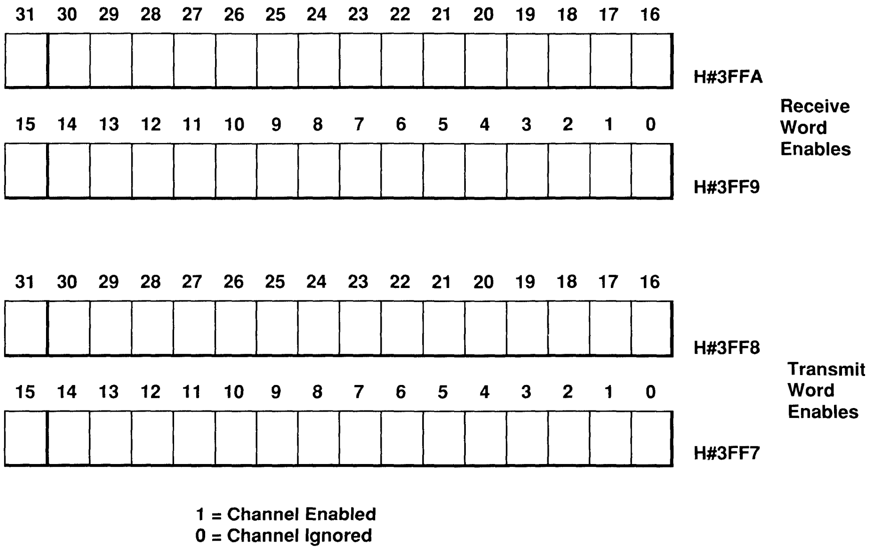


Figure 6.24 SPORT0 Multichannel Word Enable Registers

6.8.2 Multichannel Operation

In general, most aspects of SPORT0 control operate normally in the multichannel mode. Specifically word length (SLEN), internal or external framing (IRFS), inverting the frame signal (INVRFS), companding (DTYPE) and setting up and using autobuffering are all unchanged in their meanings when used in the multichannel mode.

Receive word time slots which are not active are ignored; that is, no interrupts are generated for these words, no autobuffering occurs and no data is written to the RX0 register. Transmit word time slots which are not active tristate the data transmit (DT) pin.

The TFS signal functions as a transmit data valid (TDV) signal in multichannel mode. Whenever the processor transmits (an active word) TDV is asserted; its logic is controlled by the same invert bit (INVTFS) bit, referred to as INVTDV in this context. If INVTDV is one, TDV is active low. TDV can be used to enable additional buffer logic, if required.

Figure 6.25 shows the start of a multichannel transfer. As in our earlier examples, word length is four bits (SLEN=3) and framing is active high. Multichannel frame delay (MFD) is one SCLK cycle. For the purpose of illustration, words zero and two are selected for receiving and words one and two are selected for transmission.

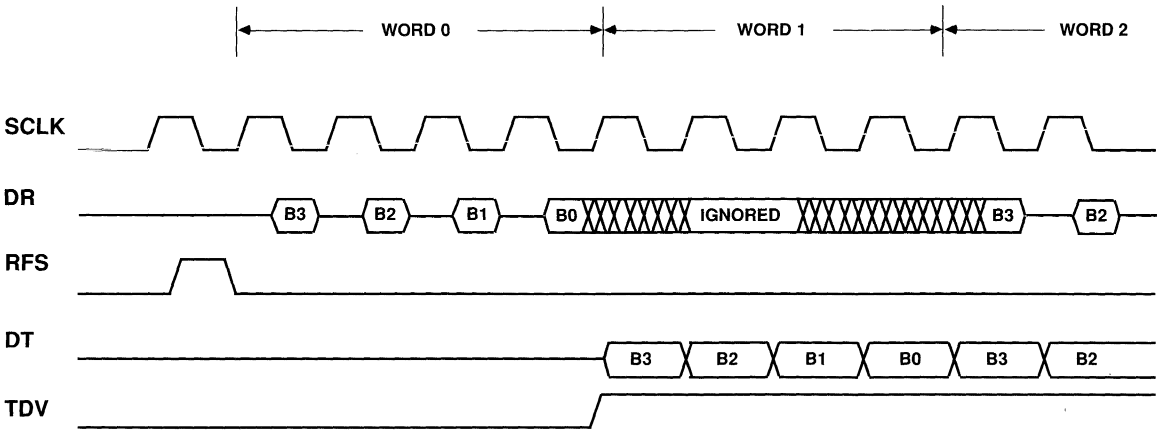


Figure 6.25 Start of Multichannel Operation

Figure 6.26 shows a complete twenty-four word cycle in the multichannel mode, with complete words represented in the waveforms instead of individual bits. Receiving is active for all words and transmitting is active for words 0–3, 8–11 and 16–19 only.

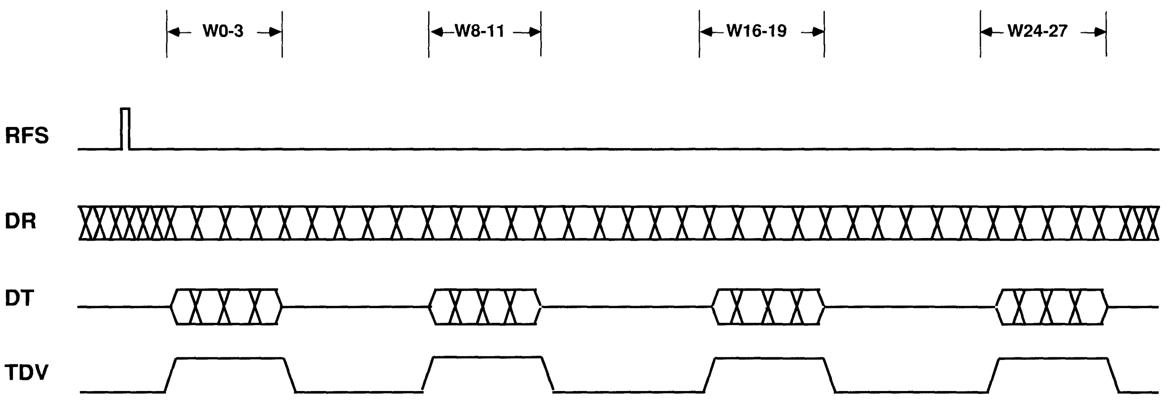


Figure 6.26 Complete Multichannel Example

6.9 SPORT ENABLE AND CONFIGURATION

SPORT0 and SPORT1 are enabled by bits 12 and 11, respectively, in the system control register (see Figure 6.27 on the next page). Each bit must be set for its corresponding serial port to operate.

SPORT1 can be configured as two external interrupt inputs, $\overline{\text{IRQ0}}$ and $\overline{\text{IRQ1}}$, and the Flag In and Flag Out signals instead of as a serial port. If bit 10 of the sytem control register is a 0, the SPORT1 pins are defined in the alternate configuration listed below, regardless of whether SPORT1 is enabled or disabled. SCLK may still be used as internally generated serial clock in this configuration.

Name	*Alternate Function*
SCLK1	Serial clock (output only)
RFS1	$\overline{\text{IRQ0}}$
TFS1	$\overline{\text{IRQ1}}$
DR1	FI (Flag In)
DT1	FO (Flag Out)

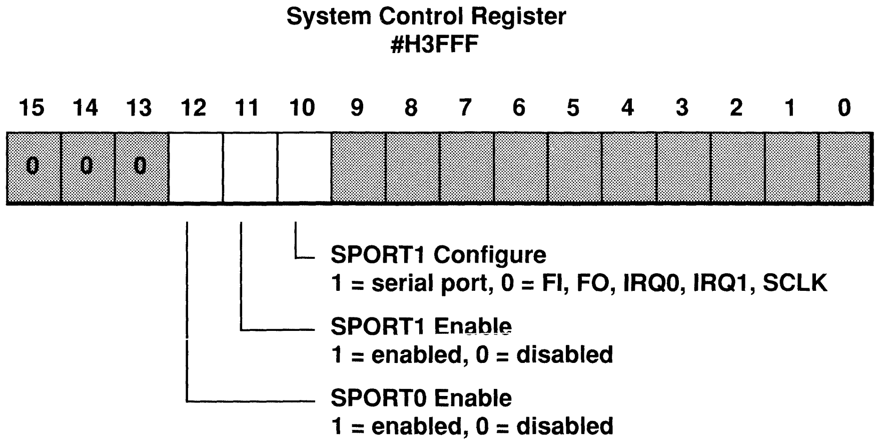


Figure 6.27 SPORT Enables in System Control Register

6.10 SPORT HARDWARE INTERFACING

Figure 6.28 below shows an ADSP-2101 communicating with two PCM codecs via the two SPORTs. In this example, each SPORT generates its own serial clock and framing signal. The transmit section of each SPORT is using the receive frame synch for framing. The two codecs operate independently and could be using different companding algorithms if necessary.

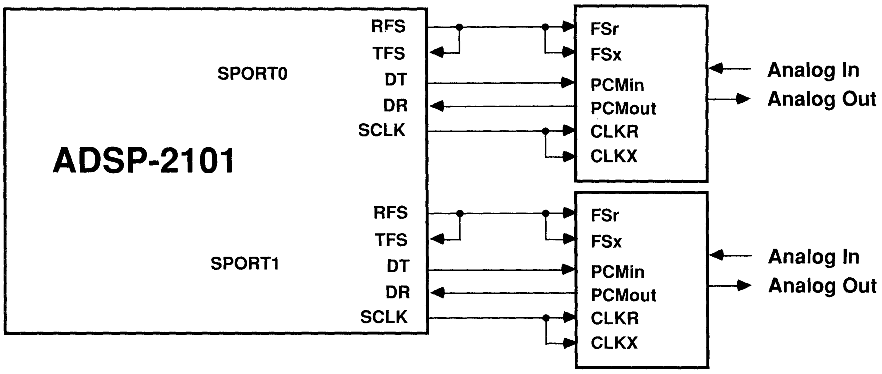


Figure 6.28 ADSP-2101 With Two Codecs

Figure 6.29 shows the ADSP-2101 interfacing to a T1 PCM interface chip. This figure shows two ADSP-2101s, although even more can easily be added because of the simplicity of sharing channels in the multichannel operating mode. As many ADSP-2101s as necessary could be connected to handle the computation for all twenty-four channels of the T1 link. A similar ADSP-2101 configuration can be used for a 32-channel CEPT link.

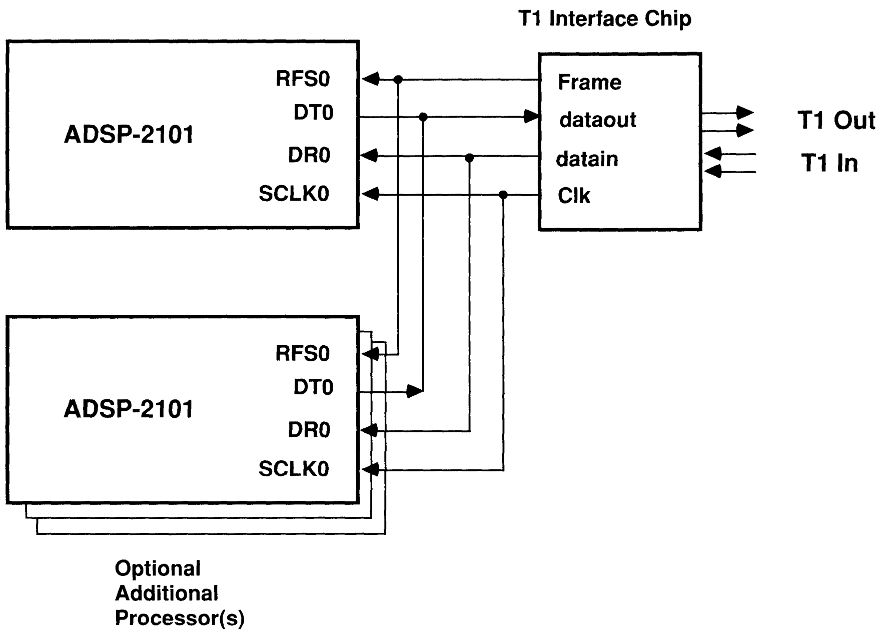


Figure 6.29 ADSP-2101s With T1 Interface in Multichannel Mode

Figure 6.30 shows three ADSP-2101 devices communicating through the multichannel function of SPORT0. All three devices must have their SPORT0 control registers initialized in software for multichannel operation of the same length (32 or 24 channels). The topmost ADSP-2101 illustrated is programmed to generate the RFS0 and SCLK0 signals for the other two, which are programmed to receive RFS0 and SCLK0 externally. Each device transmits on a different channel (software must ensure this to prevent contention on the common DT0/DR0 line). For example, processor #1 may transmit on channel 0 and receive on channel 1. Processor #2 completes the bidirectional communication by receiving on channel 0 and transmitting on channel 1. Processor #3 may transmit on channel 12 to both #1 and #2, which are set up to receive on channel 12. This type of multiprocessor communication can be expanded to more ADSP-2101 devices, up to the number that can be accommodated with the maximum of 32 channels available.

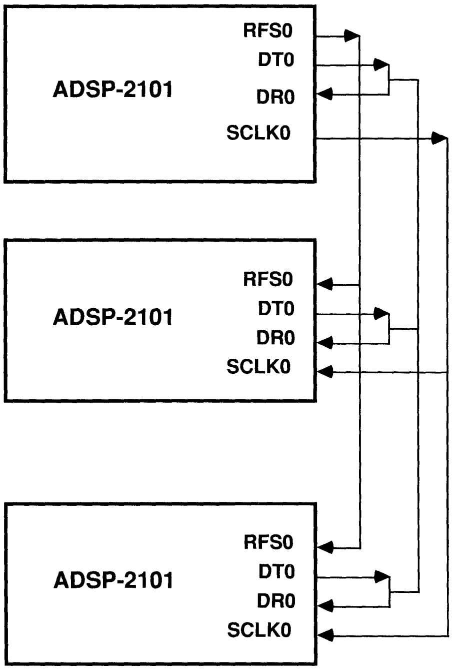


Figure 6.30 Using Multichannel Mode for Interprocessor Communication

System Interface 7

7.1 INTRODUCTION

This chapter describes the control interface of the ADSP-2101. The processor has clock input pin, CLKIN, a crystal output pin, XTAL and a clock output, CLKOUT, a $\overline{\text{RESET}}$ line for resetting the processor, a Flag In pin, FI, and a Flag Out pin, FO, and interrupt lines.

This chapter describes only the logical relationships of control signals; you must consult the data sheet for actual timing characteristics.

The discussion of the ADSP-2101's external interface is presented in three parts. The following chapter discusses the memory interface and associated control lines, Bus Request and Bus Grant. The previous chapter on serial ports discusses the serial port interface. These interfaces are shown in Figure 7.1, on the next page.

7.2 CLOCK SIGNALS & PROCESSOR STATES

The ADSP-2101 may be operated with a clock oscillator input to the CLKIN pin or with a crystal connected across CLKIN (input) and XTAL (crystal output). If an oscillator is used, XTAL must not be connected. See Figure 7.2. The processor uses a phase-locked loop to generate internal phases and the clock output signal, CLKOUT.

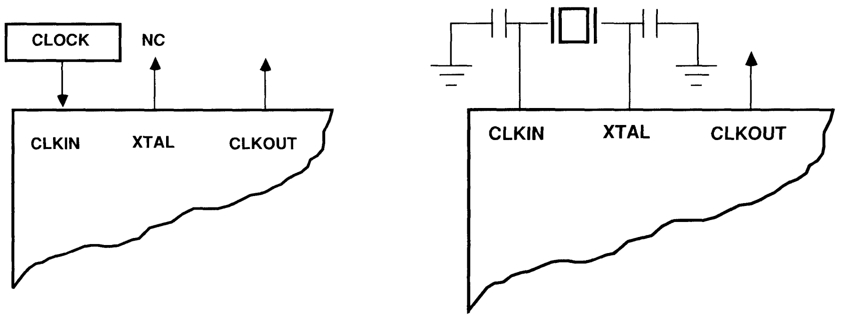
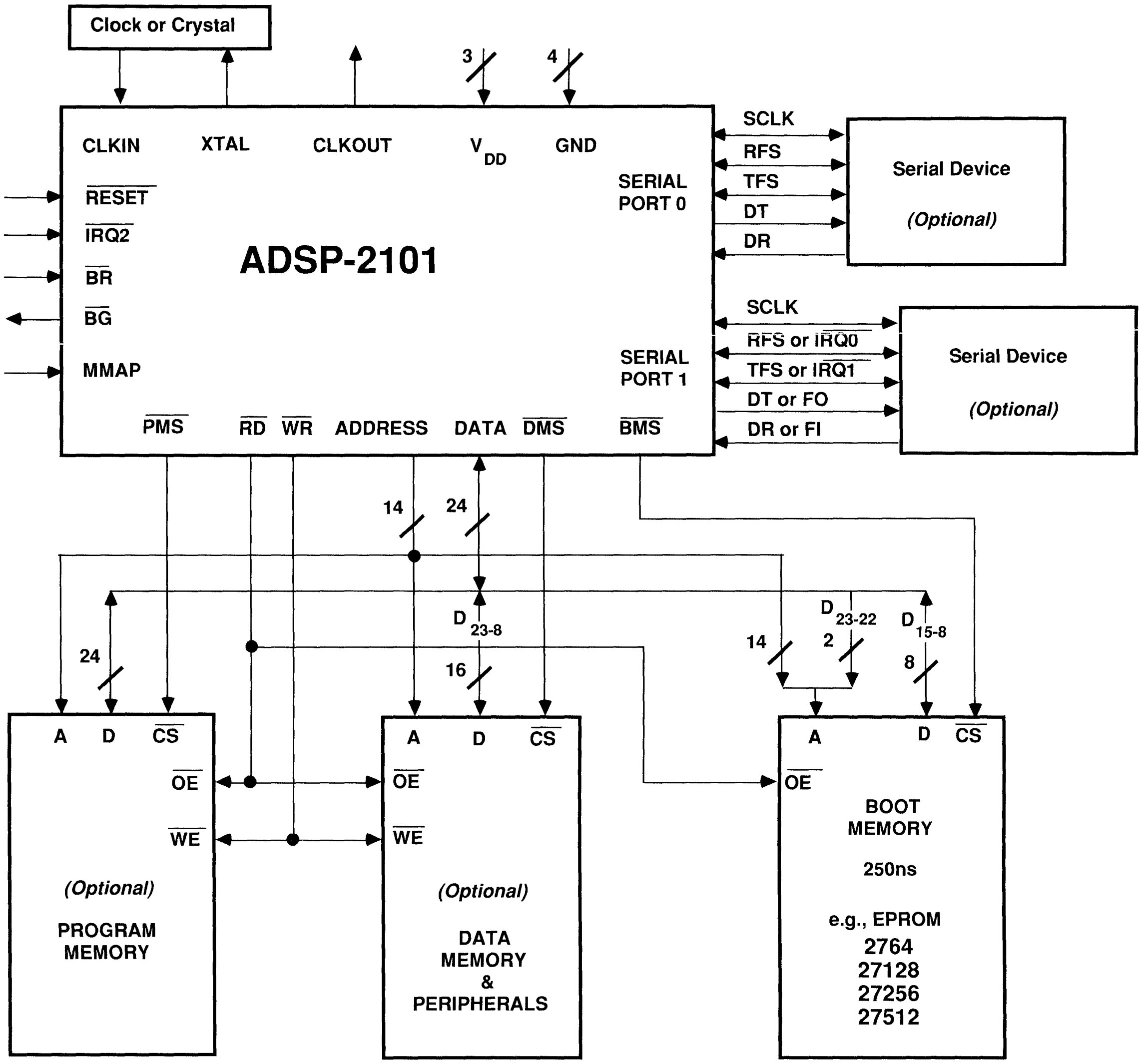


Figure 7.2 Clock or Crystal Configuration

7 System Interface



NOTE: The two MSBs of the Boot EPROM Address are also the two MSBs of the Data Bus. This is only required for the 27256 and 27512.

Figure 7.1 ADSP-2101 Basic System

CLKIN is a master input clock to the processor that operates at the instruction cycle rate. CLKOUT is an output clock from the ADSP-2101 that operates at the instruction cycle rate. The rising edge of CLKOUT is aligned with the rising edge of CLKIN. The relationship between the phases of CLKOUT and the four internal time periods, called the processor states, that make up an instruction cycle is shown in Figure 7.3. The falling transition of CLKOUT always occurs at the transition between states two and three while the rising edge always occurs at the transition between states four and one.

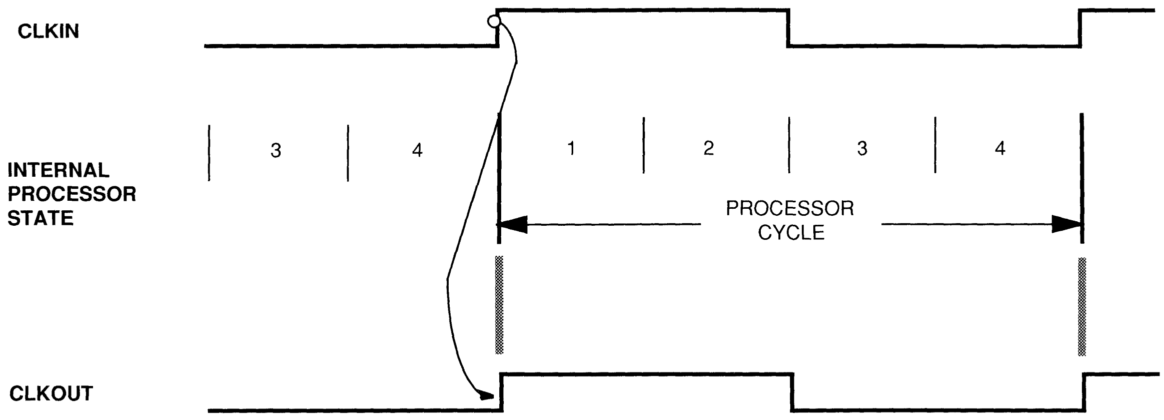


Figure 7.3 Clock Signals & Processor States

7.2.1 Synchronization Delay

The ADSP-2101 has several asynchronous inputs, namely, $\overline{\text{RESET}}$, FI and $\overline{\text{IRQ0–2}}$. These inputs can be asserted in arbitrary phase to the processor clock, CLKIN. The ADSP-2101 synchronizes them prior to recognizing them. The delay associated with recognition is called the synchronization delay.

Any asynchronous input must be valid prior to the recognition point to be recognized in a particular cycle. 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 plus setup and hold time. The minimum time prior to recognition (the setup and hold time) is given in the *ADSP-2101 Data Sheet.*

7.2.2 Clock Considerations

The ADSP-2101 requires only a 1X frequency clock signal which is used by an on-chip phase-locked loop to generate the higher frequency internal processor clock signals and CLKOUT. Because these clocks are generated based on the rising edge of CLKIN, there is no ambiguity about the phase relationship of two processors sharing the same input clock. Multiple processor synchronization is very easy as a result.

Using a 1X frequency input clock with the phase-locked loop to generate the various internal clocks imposes certain restrictions. The CLKIN must be valid long enough to achieve phase lock before $\overline{\text{RESET}}$ can be removed; see the *ADSP-2101 Data Sheet* for details. Also, the CLKIN cannot be stopped or changed on the fly.

7.3 $\overline{\text{RESET}}$

$\overline{\text{RESET}}$ halts execution and returns all registers to a state defined in Table 7.1 below. The contents of all RAM are unchanged after $\overline{\text{RESET}}$, except as shown in this table.

When $\overline{\text{RESET}}$ is released the booting sequence described in the memory interface chapter (based on the state of the MMAP pin) takes place, except in ROM-based processors (ADSP-2102).

It is possible to force the processor to reboot in software by setting the BFORCE bit to one. This is not the same as a $\overline{\text{RESET}}$ in that some registers and data values are preserved.

During booting (and rebooting) all interrupts including serial port interrupts are masked, and autobuffering is disabled. The serial ports remain active; one transfer – from internal shift register to data register – can occur for each serial port before there are overrun problems.

The timer runs during a reboot. A timer interrupt happening during the reboot is masked. Thus, if more than one occurs during the reboot, the processor latches only the first. A timer overrun can occur.

Table 7.1 shows the state of the processor control registers, fields and bits after a $\overline{\text{RESET}}$ and after a software reboot. The values of any registers not listed are undefined at reset and unchanged by a reboot.

Control Field	*Description*	*$\overline{\text{RESET}}$*	*Reboot*
Data Registers			
PX	PX register	undefined	undefined
Status Registers			
IMASK	Interrupt service enables	0	0
ASTAT	Arithmetic status	0	0
MSTAT	Mode status	0	no change
SSTAT	Stack status	H#55	H#55
ICNTL	Interrupt control	undefined	no change
Control Registers (Memory-mapped)			
BWAIT	Boot memory wait states	3	no change
BPAGE	Boot page	0	no change
SPORT1 configure	Configuration	1	no change
SPE0	SPORT0 enable	0	no change
SPE1	SPORT1 enable	0	no change
DWAIT0–4	Data memory wait states	7	no change
PWAIT	Program memory wait	7	no change
TCOUNT	Timer count register	undefined	operates during reboot
TPERIOD	Timer period register	undefined	no change
TSCALE	Timer scale register	undefined	no change
Serial Port Control Registers (Memory-mapped, one set per SPORT)			
ISCLK	Internal serial clock	0	no change
RFSR, TFSR	Frame sync required	0	no change
RFSW, TFSW	Frame sync width	0	no change
IRFS, ITFS	Internal frame sync	0	no change
INVRFS, INVTFS	Invert frame sense	0	no change
DTYPE	Companding type, format	0	no change
SLEN	Serial word length	0	no change
SCLKDIV	Serial clock divide	undefined	no change
RFSDIV	RFS divide	undefined	no change
Multichannel word enable bits		undefined	no change
MCE	Multichannel enable	0	no change
MCL	Multichannel length	0	no change
MFD	Multichannel frame delay	0	no change
INVTDV	Invert transmit data valid	0	no change
FO	Flag Out value	undefined	no change
RBUF, TBUF	Autobuffering enable	0	0
TIREG, RIREG	Autobuffer I index	undefined	no change
TMREG, RMREG	Autobuffer M index	undefined	no change

Table 7.1 ADSP-2101 State After $\overline{\text{RESET}}$ or Software Reboot

7.4 INTERRUPTS

The ADSP-2101 supports one or three prioritized, individually maskable external interrupts that can be either level or edge-triggered. The processor also supports internal interrupts from the Timer and SPORTs, which are discussed in those chapters. Additional information about interrupt masking, set-up and operation can be found in Chapter 4, "Program Control." The $\overline{IRQ2}$ interrupt is always supported; $\overline{IRQ1}$ and $\overline{IRQ0}$ are alternate uses of SPORT1's interface, available when SPORT1 is not being used as a serial port.

7.4.1 Edge & Level Sensitivity

Level-sensitive interrupts operate by asserting the interrupt request line (active low) until the request is recognized by the processor. Once recognized, the request must be removed before unmasking the interrupt to prevent being reserviced.

In contrast, edge-triggered interrupt requests are latched when any HI-to-LO transition occurs on the interrupt line. The ADSP-2101 latches the interrupt 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.

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, many interrupting devices may share a single level-sensitive request line which allows for easy system expansion.

An interrupt request gets serviced when 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 the vector addresses shown in Table 4.2. There is a synchronization delay associated with the interrupt request lines (and with internal interrupts).

If an interrupt occurs during the extra cycles required to execute an instruction that accesses external memory more than once, it is not recognized between the cycles, only before or after. Interrupts are latched, but not serviced, during bus grant ($\overline{BG}$) unless the GO mode is enabled. Remember that in order to service an interrupt, the processor must be running and executing instructions, if only the IDLE instruction.

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

7.4.2 Interrupt Operation

Figure 7.4 shows the interrupt service timing. Edge-sensitive and level-sensitive interrupt requests are serviced similarly. Edge-sensitive interrupts may remain LO indefinitely, while level-sensitive interrupts must be removed before executing the RTI instruction or the same interrupt immediately recurs.

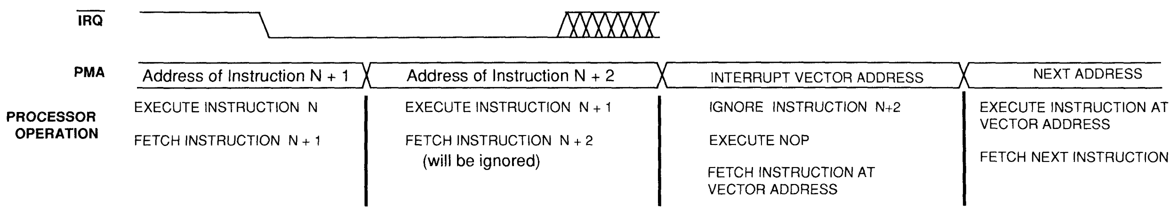


Figure 7.4 Interrupt Service Timing

7.5 FLAG IN & FLAG OUT PINS

In addition to the $\overline{IRQ1}$ and $\overline{IRQ0}$ pins, the alternate configuration of SPORT1 provides the ADSP-2101 with a Flag In (FI) and a Flag Out (FO) pin. In the alternate configuration, the DR1 pin is redefined as Flag In and the DT1 pin as Flag Out. Clearing the SPORT1 configuration bit in the system control register selects the alternate configuration.

FI may be used to control the branching of your program, using the IF FLAG_IN and IF NOT FLAG_IN conditions for the JUMP and CALL instructions. These condition statements evaluate based on the last state of the FI pin; FLAG_IN is true if FI last sampled as one and false if zero.

FO may be set, toggled, or cleared in software to signal events or conditions to any other device such as a host processor. The flag out control instruction, which is conditional, supports the SET, RESET or TOGGLE actions. These operations allow programs executing on the ADSP-2101 to control the state of this output pin as needed. The state of FO is also available as a read-only bit of the SPORT1 control register.

Memory Interface ▣ 8

8.1 INTRODUCTION

Figure 8.1 (on the next page) shows a complete ADSP-2101 system with external memories and peripherals.

The ADSP-2101 has three separate memory spaces: data memory, program memory and boot memory. Boot memory is only active during the loading of program code from an external device (ROM, EPROM or RAM typically). Data memory is a single address space, some on the chip and the rest external. Likewise, program memory consists of a single address space, some on the chip and the rest external.

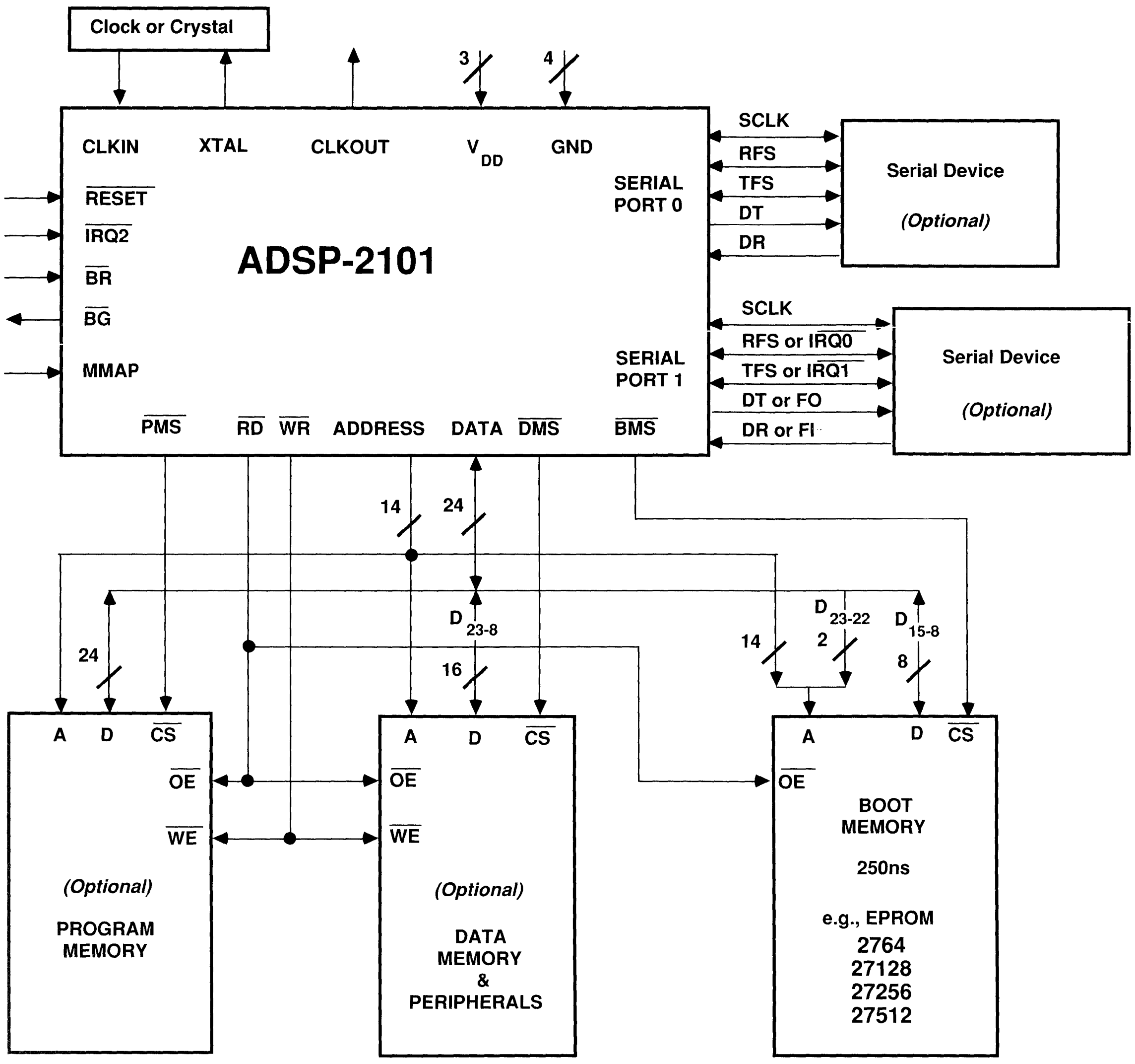
The program memory address bus (PMA) and the data memory address bus (DMA) are multiplexed into one bus and driven off chip. Likewise, the program memory data bus (PMD) and the data memory data bus (DMD) are multiplexed into one bus and driven off chip. The sixteen MSBs of the external data bus are used as the DMD bus. In other words, D_{23-8} are used for DMD_{15-0}.

The $\overline{PMS}$, $\overline{DMS}$ and $\overline{BMS}$ signals indicate which memory is being accessed. Because program memory and data memory buses are shared, if more than one off-chip transfer needs to be made in the same instruction there will be an overhead cycle required. There is no overhead if just one off-chip access with no wait states occurs in any instruction.

All external memories may have automatic wait state generation associated with them. Wait states – each equal to one cycle – are programmable; the defaults are given in each section of this chapter. The ADSP-2101 can grant control of the external buses to another device using the bus request ($\overline{BR}$) and bus grant ($\overline{BG}$) signals.

The discussion of the ADSP-2101's external interface is presented in three parts. This chapter discusses the memory interface and associated control lines. The previous chapter discusses the system/control interface and the chapter on serial ports discusses the serial port interface. Only the relationships are described; you must consult the *ADSP-2101 Data Sheet* for actual timing characteristics.

8 Memory Interface



NOTE: The two MSBs of the Boot EPROM Address are also the two MSBs of the Data Bus. This is only required for the 27256 and 27512.

Figure 8.1 ADSP-2101 System Block Diagram

8.2 BOOT MEMORY INTERFACE

The ADSP-2101 has 2K of 24-bit (3-byte) wide internal program memory. It can load the entire 2K or some fraction of it during a boot sequence. To interface to inexpensive EPROM, the processor loads instructions one byte at a time.

Booting is only possible when the MMAP pin is logical 0. If the MMAP pin is logical 1, the boot sequence does not occur.

$\overline{BR}$ is recognized during the booting sequence. The bus is granted after completion of loading the current byte. $\overline{BR}$ during booting may be used to implement booting under control of a host processor.

8.2.1 Boot Pages

Although 2K words of 3-byte wide program memory require only 6K bytes of storage, boot memory is organized into eight pages which are each 8K bytes long. Every fourth byte of a page is an "empty" byte, except the first one, which contains the page length. The page length is read first and then bytes are loaded from the top of the page downwards. This results in shorter booting times for shorter pages.

The length of the boot page is given as:

pagelength = (number of 24-bit PM words/8) – 1

That is, a page length of 0 causes the boot address generator to generate byte addresses for 8 words which reside in 32 sequential EPROM locations.

The ADSP-2101 PROM Splitter, part of the ADSP-2101 Cross-Software development tools, calculates the proper page length for your program and orders the bytes of your program as shown in Figure 8.2 on the next page.

8.2.2 Powerup Boot and Software Reboot

Upon reset, the ADSP-2101 boot sequence occurs if the MMAP pin is 0. The boot sequence on powerup or hardware reset always loads boot page 0. After reset, boot loading can occur from any one of up to 8 different boot pages. The boot page select field (BPAGE) in the ADSP-2101 memory-mapped register at location 0x3FFF (see Figure 8.3 on the next page) specifies which boot page is to be loaded. To boot the ADSP-2101 from a specific boot page, set BPAGE to the desired page number and, in

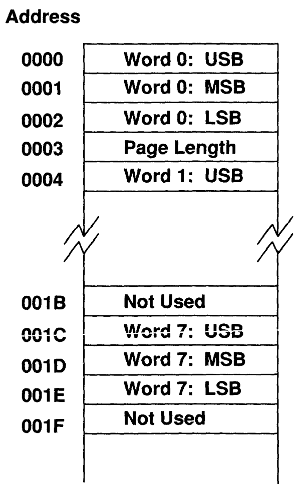


Figure 8.2 EPROM Contents

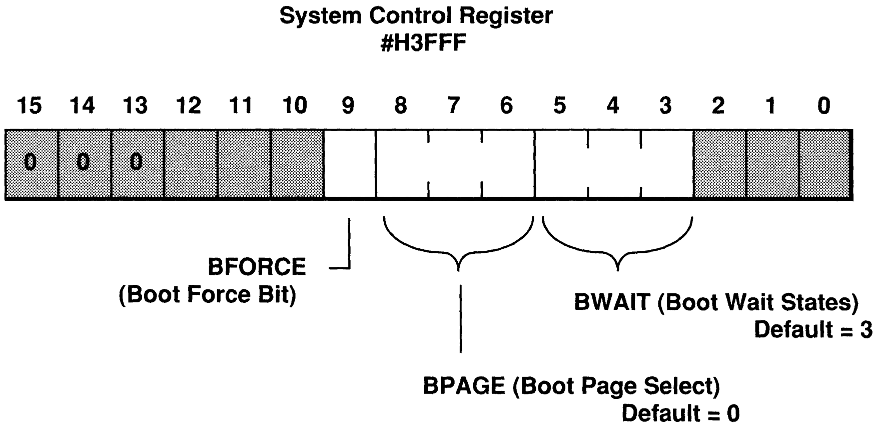


Figure 8.3 Boot Control Fields in System Control Register

the same memory-mapped register, set the boot force bit (BFORCE). When the boot force bit is set, the (software-forced) booting sequence starts. Except for the page selection, there is no difference between a software-forced boot sequence and a reset boot sequence.

Table 7.1, in the system interface chapter, shows the state of the processor control registers after a reset and after a software reboot. Essentially, the processor's control state is saved, but stacks are cleared and execution restarts at the restart vector.

8.2.3 Boot Memory Access

The ADSP-2101 can boot its internal memory from a single byte-wide 250 ns EPROM, such as the 2764 and 27512. The number of wait states for the boot memory access is located in the BWAIT field of the ADSP-2101 memory-mapped register located at address 0x3FFF (Figure 8.3). This field can be set to any value from 0 to 7 in order to generate 0 to 7 wait states. The default value at reset is three wait states.

Timing of the boot memory access is identical to that of external program memory or external data memory accesses, except that the active strobe is $\overline{\text{BMS}}$ rather than $\overline{\text{PMS}}$ or $\overline{\text{DMS}}$. To address eight pages of 8K bytes each, 16 bits are needed. The least significant 14 bits are output on the 14-bit ADSP-2101 address bus, and the most significant 2 bits are output on the 2 MSBs of the data bus during a boot memory access. Data is read from the data bus on the middle eight bits.

8.2.4 Boot Loading Sequence

The order in which the ADSP-2101 loads data into its internal memory during a boot operation is unimportant in most applications. The boot loading sequence is explained in this section for those instances in which the order is relevant (when booting from a host instead of an EPROM, for example).

To execute the boot operation, the boot address generator generates the appropriate byte addresses and loads the ADSP-2101 internal program memory with the contents of the EPROM. The ADSP-2101 internal program memory is loaded beginning with the high addresses. For example, assume that eight 24-bit words are loaded into the ADSP-2101 during the booting process. The first word written into the ADSP-2101 program memory is written to address 0007. The last word loaded is written to internal program memory address 0000.

The boot address is made up of several values, as shown in Figure 8.4: the 3-bit page number (from BPAGE in the system control register); the 8-bit page length, which is always read first, from the fourth byte of the page; three ones (111); and a 2-bit code whose value determines which byte of the word is being addressed.

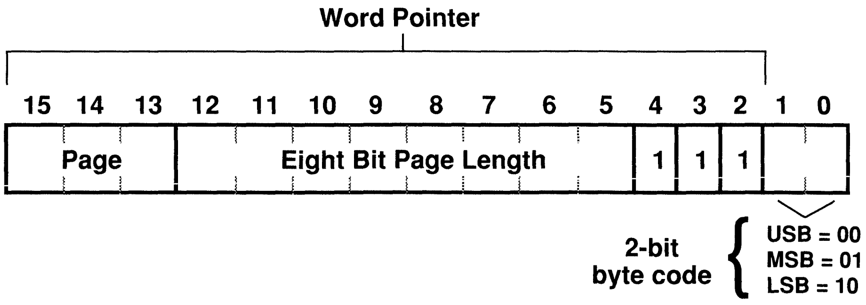


Figure 8.4 Boot Address

The last 24-bit word (the last instruction or program memory data value) is loaded into the ADSP-2101 first. The byte loading order is: upper byte, lower byte, middle byte. The word pointer is then decremented. This addresses the second-to-last 24-bit word in the EPROM.

For example, to boot from page 0 the shortest allowable page (eight 24-bit words corresponding to a page length of 0), the following addresses would be generated:

- The first address generated is 0003 which reads the page length.
- The next address generated in this example is address 001C. This is the upper byte of the last word.
- The byte code is then updated to specify the lower byte (the final two bits are 10) and the address generated is 001E.
- The byte address changes again, this time to address the middle byte (the two bit code is 01) and the address generated is 001D.
- Once all three bytes are loaded, the word counter is decremented. The three succeeding byte addresses generated are 0018, 001A, and 0019.
- The word counter is decremented again and the next set of byte addresses generated is 0014, 0016, and 0015. This process continues until word 0 is loaded.

The contents of the EPROM, the byte addresses and the order of addresses generated is summarized in Figure 8.5.

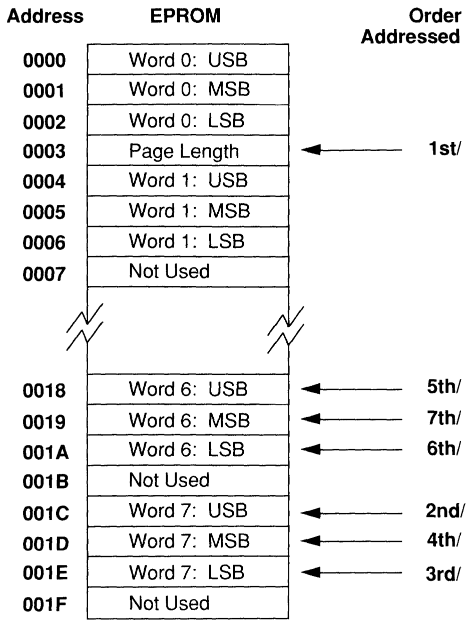


Figure 8.5 Boot Loading Order

8.3 PROGRAM MEMORY INTERFACE

The ADSP-2101 addresses 16K of 24-bit wide program memory, 2K on-board and up to 14K external using the control lines shown in Figure 8.1. The processor supplies a 14-bit address on the program memory address bus (PMA) which is multiplexed off-chip. Instructions or data are transferred across the 24-bit program memory data (PMD) bus which is also multiplexed off-chip. A program memory select pin, $\overline{\text{PMS}}$, indicates that the address bus is being driven with a program memory address and memory can be selected.

Two control lines indicate the direction of the transfer. Memory read ($\overline{\text{RD}}$) is active low signaling a read and memory write ($\overline{\text{WR}}$) is active low for a write operation. Typically, you would connect $\overline{\text{PMS}}$ to $\overline{\text{CE}}$, $\overline{\text{RD}}$ to $\overline{\text{OE}}$ and $\overline{\text{WR}}$ to $\overline{\text{WE}}$ of your memory.

8.3.1 Program Memory Read / Write

The on-chip program memory access is transparent to the outside memory interface. Off-chip program memory access happens in this sequence:

1. The ADSP-2101 places the address on the PMA bus, which is multiplexed off-chip.

2. $\overline{\text{PMS}}$ and then $\overline{\text{RD}}$ or $\overline{\text{WR}}$ are asserted.

3. Within a specified time, data is placed on the data bus, multiplexed to the internal PMD bus.

4. The data is read or written and $\overline{\text{RD}}$ (or $\overline{\text{WR}}$) then $\overline{\text{PMS}}$ is deasserted.

The basic read and write cycles are illustrated in Figure 8.6. Part A shows zero wait states and Part B shows the effect of one wait state.

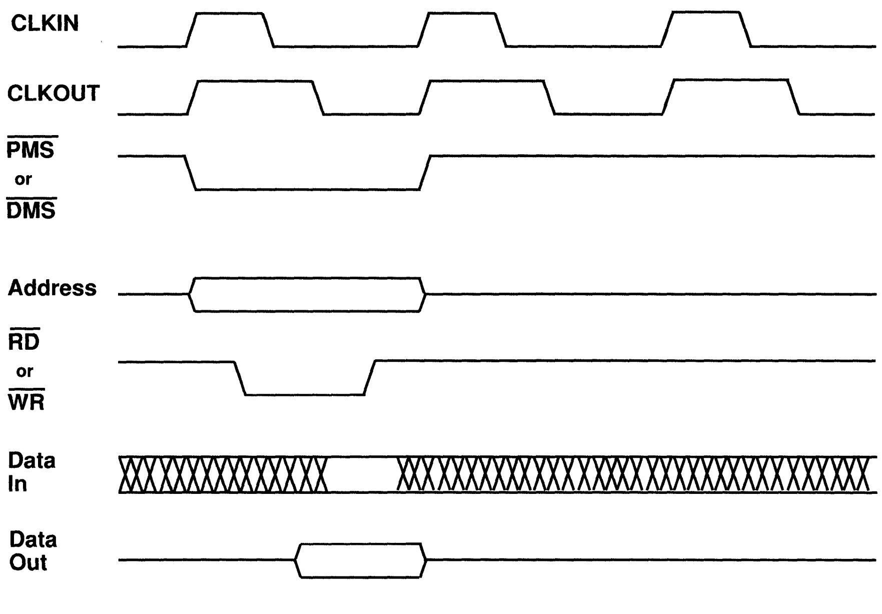


Figure 8.6a. Program & Data Memory Read & Write Operations, No Wait States

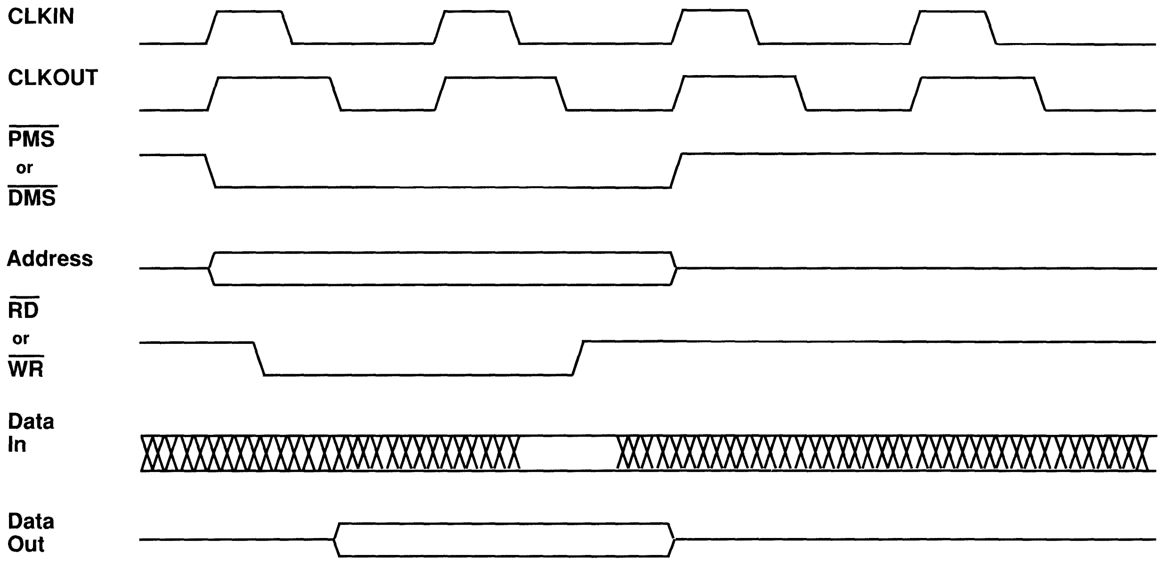


Figure 8.6b Program & Data Memory Read & Write Operations, One Wait State

External program memory has a programmable wait state field (PWAIT) in the system control register, as shown in Figure 8.7. PWAIT defaults to seven wait states for program memory access on power-up.

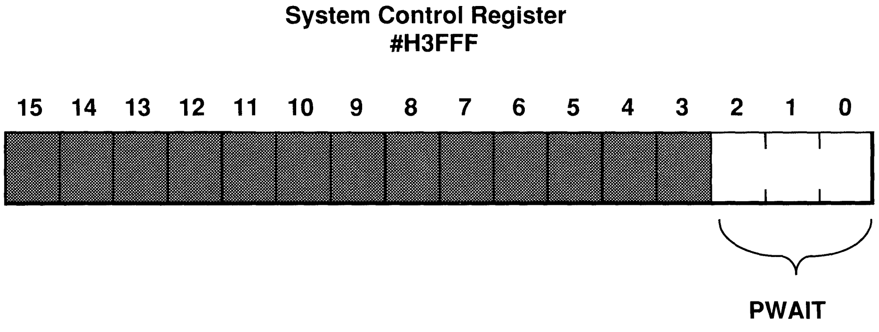


Figure 8.7 Program Memory Wait State Field In System Control Register

8.3.2 Program Memory Map

Depending on the state of the MMAP pin, the program memory space is configured as one of the two layouts shown in Figure 8.8.

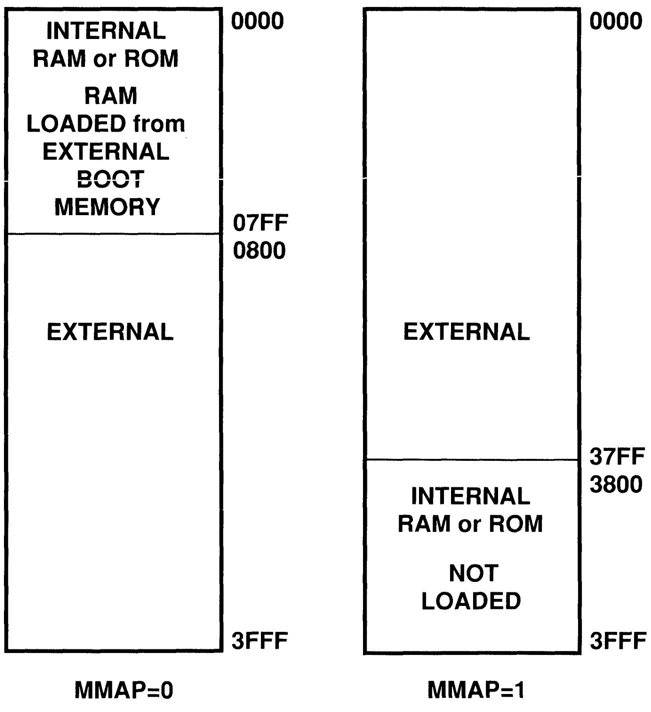


Figure 8.8 Program Memory Configurations

The 16K program memory space can hold instructions and data intermixed in any combination. The Linker determines where to place relocatable code and data segments. You may specify absolute address placement for any module or data structure, including the code for the restart and interrupt vector locations.

Table 8.1 shows the location of these vectors.

Program Memory Address	*Use*
0000	Restart Vector
0004	$\overline{\text{IRQ2}}$
0008	SPORT0 Transmit
000C	SPORT0 Receive
0010	SPORT1 Transmit or $\overline{\text{IRQ1}}$
0014	SPORT1 Receive or $\overline{\text{IRQ0}}$
0018	Timer Interrupt

Table 8.1 Program Memory Restart / Interrupt Vectors

Internal program memory RAM is fast enough to supply an instruction and data in the same cycle eliminating the need for cache memory as in the ADSP-2100. Consequently, the ADSP-2101, if operating entirely from on-chip memories, can fetch two operands and the next instruction on every cycle. The ADSP-2101 can also fetch any one of these three from external memory with no performance penalty.

8.4 DATA MEMORY INTERFACE

The ADSP-2101 addresses up to 16K of 16-bit data memory using the control signals shown in Figure 8.1. On-chip data memory is 1K in size beginning at H#3800. In addition, control registers are memory-mapped into the upper 1K of data memory address space. The top 1K of data memory is reserved for future expansion; 14K of data memory is available for user data storage.

The processor supplies a 14-bit address on the data memory address bus (DMA) which is multiplexed off-chip. Data is transferred across the upper 16 bits of the 24-bit memory data bus which is also multiplexed off-chip. A data memory select pin, $\overline{\text{DMS}}$, indicates that the address bus is being driven with a data memory address and memory can be selected.

Two control lines indicate the direction of the transfer. Memory read ($\overline{\text{RD}}$) is active low signaling a read and memory write ($\overline{\text{WR}}$) is active low for a write operation. Typically, you would connect $\overline{\text{DMS}}$ to $\overline{\text{CE}}$, $\overline{\text{RD}}$ to $\overline{\text{OE}}$ and $\overline{\text{WR}}$ to $\overline{\text{WE}}$ of your memory.

8.4.1 Data Memory Read/Write

The on-chip data memory access is transparent to the outside memory interface. Off-chip data memory access requires the same sequence as for off-chip program data memory, namely:

1. The ADSP-2101 places the address on the DMA bus, which is multiplexed off-chip.

2. $\overline{DMS}$ and then $\overline{RD}$ or $\overline{WR}$ are asserted.

3. Within a specified time, data is placed on the data bus, multiplexed to the internal DMD bus.

4. The data is read or written and $\overline{RD}$ (or $\overline{WR}$) then $\overline{DMS}$ are deasserted.

The basic read and write cycles are illustrated in Figure 8.6 in the preceding section.

8.4.2 Data Memory Map

Data memory configurations are shown in Figure 8.10, on the facing page. Each of the five zones of off-chip data memory has its own programmable wait state. Wait states are extra cycles that the ADSP-2101 either waits before latching data (on a read) or drives the data (on a write). This means that one zone of memory could be used for working with memory-mapped peripherals of one speed while another zone was used with faster or slower peripherals. Similarly, slower and faster memories can be used for different purposes, as long as they are located in different zones of the data memory map.

The data memory wait state control register, shown in Figure 8.9 has a separate field for each zone of external memory. Each 3-bit field contains the number (0-7) of wait states for the corresponding zone of memory.

Data Memory Wait State Control Register
#H3FFE

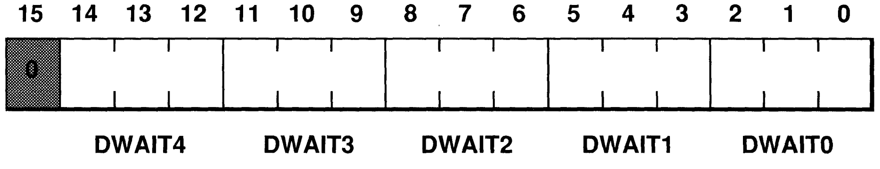


Figure 8.9 Data Memory Wait State Control Register

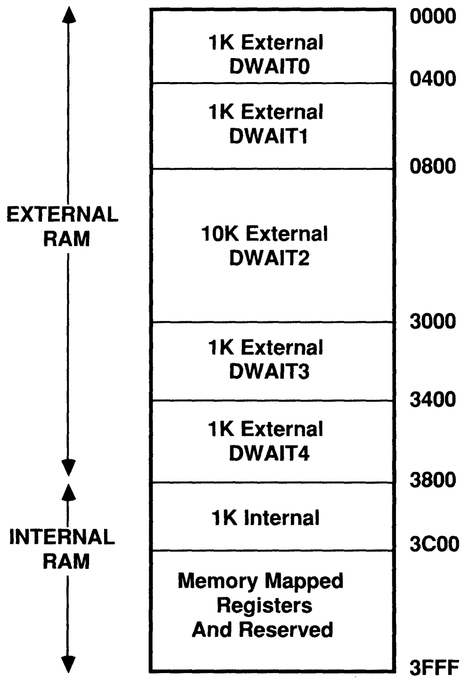


Figure 8.10 Data Memory Configuration

8.4.3 Parallel & Memory-Mapped Peripherals

Peripherals requiring parallel communications and other types of devices can be mapped into external data memory. Communication takes the form of reading and writing the memory locations associated with the device. Some A/D and D/A converters require this type of interface. The PORT directives in the System Builder and Assembler modules of the Cross-Software support this mapping. Communication with a memory-mapped device consists simply of reading and writing the appropriate locations. By matching the access times of the external devices to the wait states specified for their zone of data memory, you can easily interface a variety of devices.

8.5 BUS REQUEST / GRANT

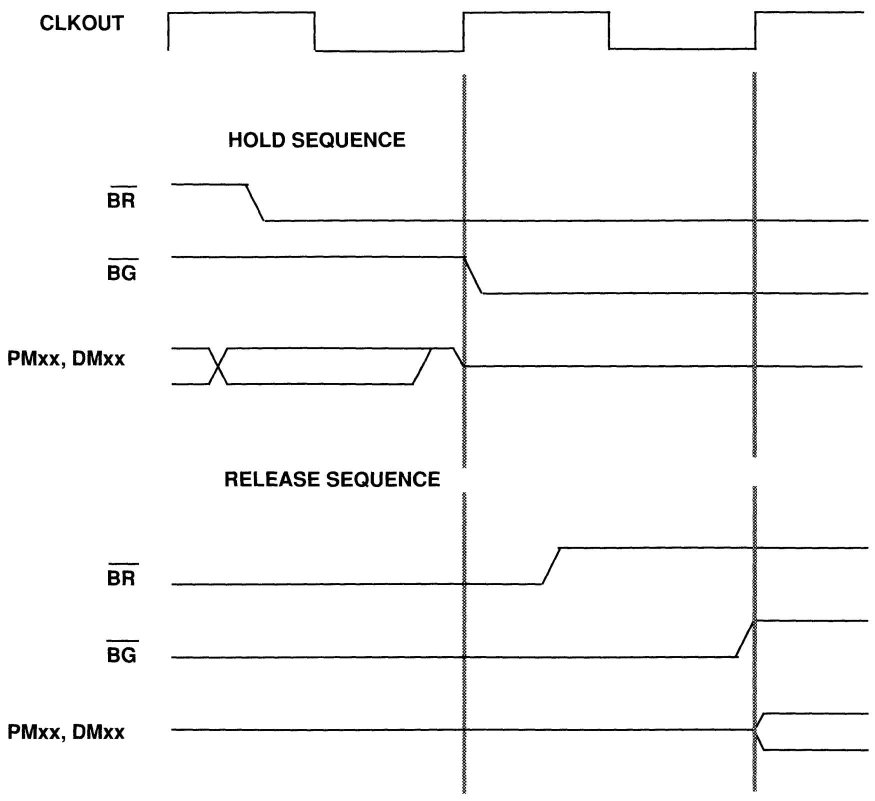
Using the bus request, $\overline{BR}$, and bus grant, $\overline{BG}$, signals, the ADSP-2101 can relinquish control of the external memory interface giving access to an external device, such as a host processor. If the GO mode is enabled, the ADSP-2101 continues to execute instructions using on-chip program and data memory. The processor halts only when it must access external memory. If the GO mode is not enabled, the processor always halts before granting the bus.

The external device requests the bus by asserting $\overline{BR}$. $\overline{BR}$ is a synchronous input with setup and hold requirements specified in the *ADSP-2101 Data Sheet*. When $\overline{BR}$ is recognized, the ADSP-2101 halts if necessary and tristates fourteen address bus lines, twenty-four data bus lines, $\overline{WR}$, $\overline{RD}$, $\overline{PMS}$, $\overline{DMS}$ and $\overline{BMS}$. Control is then transferred to the requesting device by asserting $\overline{BG}$. If the processor is in the middle of an instruction requiring the access of both external program and external data memory (requiring two consecutive cycles of external bus use) and the second access has not yet begun, $\overline{BG}$ is granted in between the two accesses. The second access is performed after the bus request is removed.

Even if the processor has to halt, its internal state is not affected by granting the bus. After the bus request is released by the external device, normal operation resumes from the point at which it was halted. This applies uniformly to all processor operations.

The external device returns control to the ADSP-2101 by releasing $\overline{BR}$. After $\overline{BR}$ is recognized as released, the processor releases $\overline{BG}$ and takes over the bus. Figure 8.11 shows the relative timing of this cycle.

During reset, $\overline{BR}$ is recognized and the bus is granted in the same manner as during normal operation. $\overline{BR}$ is also recognized during the booting sequence. The bus is granted after completion of loading the current byte, including any wait states. $\overline{BR}$ during booting may be used to implement booting under control of a host processor.



NOTE: PMXX stands for PMA, PMD, $\overline{WR}$, $\overline{RD}$ and $\overline{PMS}$.
DMxx stands for DMA, DMD, $\overline{WR}$, $\overline{RD}$ and $\overline{DMS}$.

Figure 8.11 Bus Hold / Release

8.6 MEMORY INTERFACE SUMMARY

Table 8.2 summarizes the states of the memory interface pins for various combinations of program memory and data memory accesses. Table 8.3 summarizes the states of the memory interface and control pins during reset, booting and bus grant.

Access	$\overline{PMS}$	$\overline{DMS}$	$\overline{BMS}$	$\overline{RD}$	$\overline{WR}$	*Address*	*Data*
Internal program memory only	high	high	high	high	high	tristated*	tristated
Internal data memory only	high	high	high	high	high	tristated	tristated
Internal program memory, external data memory	high	low	high	low (for read)	low (for write)	DM address	DM data
Internal data memory, external program memory	low	high	high	low (for read)	low (for write)	PM address	PM data

* ADSP-2101 Emulator does not tristate the address bus.

Table 8.2 Pin States During Memory Accesses

Operation	*Address*	*Data*	$\overline{PMS}$ $\overline{DMS}$ $\overline{BMS}$	$\overline{RD}$ $\overline{WR}$	*CLKOUT*	*SPORTs FO*	$\overline{BG}$
Reset	tristated	tristated	high	high	active	tristated	high
Auto Booting after Reset	active	active	$\overline{BMS}$ active $\overline{PMS}$, $\overline{DMS}$ high	$\overline{RD}$ active $\overline{WR}$ high	active	tristated	high
$\overline{BR}$ Asserted during Normal operation, Booting or Go Mode	tristated	tristated	tristated	tristated	active	active	low
$\overline{BR}$ Asserted during Reset	tristated	tristated	tristated	tristated	active	tristated	low

Table 8.3 Pin States During Reset, Booting and Bus Grant

Instruction Set Overview 9

9.1 INTRODUCTION

This chapter provides an overview of the instruction set used to program the ADSP-2101 and of the ADSP-2101 development system software. It provides enough information to understand the nature of programming the ADSP-2101 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.

For software development, you must have the *ADSP-2101 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 3-volume *ADSP-2100 Applications Handbook* presents many ADSP-2100 program examples with source code and discussion; these programs are also available on IBM PC diskettes. Individual Applications Notes detail ADSP-2101 programs.

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-2101'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-2101 source code is both readable and efficient. Unlike many assemblers, the ADSP-2101 instruction set uses an algebraic notation for arithmetic operations and for data moves resulting in highly readable source code. There is no performance penalty for this; each program statement assembles into one 24-bit instruction which executes in a single cycle. There are no multicycle instructions in the ADSP-2101 instruction set. (If memory access times require it or contention for off-chip memory occurs, overhead cycles will be required, but all instructions can otherwise execute in a single cycle.)

In addition to JUMP and CALL, the control instructions support conditional execution of most calculations and a DO UNTIL looping instruction. Return from interrupt (RTI) and return from subroutine (RTS) are also provided.

The ADSP-2101 also provides the IDLE instruction for idling the processor until an interrupt occurs. IDLE puts the processor into a low-power state while waiting for interrupts.

Two addressing modes are supported for memory fetches. Direct addressing uses immediate values; indirect addressing uses the two data address generators (DAGs).

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 data register to data register move
- a computation with any memory read or write
- a computation with a read from two memories.

The ADSP-2101 instruction set provides the programmer with maximum flexibility. The instruction set provides moves from any register to any other register, or from most registers to/from either 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 internal or external memory.

9.2 INSTRUCTION TYPES

The ADSP-2101 instruction set is grouped into the following categories:

- Multifunction
- Computational: ALU, MAC, Shifter

Instruction Set Overview 9

- Move
- Program Flow/Control
- Miscellaneous

The multifunction instructions best illustrate the power of the ADSP-2101 architecture. In this overview, we begin by examining this group of instructions.

In each section of this chapter you will find tables summarizing the syntax of each ADSP-2101 instruction group. Here is the notation used in those tables.

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 9.6.
<dreg>	refers to any data register; see Table 9.6.
<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.

9.2.1 Multifunction Instructions

Multifunction operations exploit the inherent parallelism of the ADSP-2101 architecture by providing combinations of data moves, memory reads and memory writes and computation in a single-cycle.

9.2.1.1 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

The ADSP-2101 can execute both data fetches and the multiplication/accumulation in a single-cycle. Typically, a loop of multiply/accumulates can be expressed in ADSP-2101 source code in just two program lines. Since the on-chip program memory is fast enough to provide an operand and the next instruction in a single cycle, loops of this type can execute with sustained single-cycle throughput. An example of such an instruction is:

```
MR=MR+MX0*MY0(SS), MX0=DM(I0,M0), 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 signed (SS). 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 M0) 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 except saturation. Certain other restrictions apply: the next X operand must be loaded into MX0 from data memory and the new Y operand must be loaded into MY0 from program memory (internal and external memory are identical at the level of the instruction set). The result of the computation must go to the result register (MR or AR) not to the feedback register (MF or AF).

9.2.1.2 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 (internal or external memory in either case).

9.2.1.3 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. The restrictions are similar to those for 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, there would be a conflict.

9.2.1.4 Computation With Memory Write

The computation with memory write instruction is similar in structure 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 and invokes an assembler warning; 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.

9.2.1.5 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) and SB.

In the example, the data register move loads the AX0 register with the new value at 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 appears in Table 9.6. A complete list of the permissible *xops* and *yops* for the computational operations is given in the *ADSP-2101 Cross-Software Manual.*

Table 9.1 shows the legal combinations for multifunction instructions. You may combine operations on the same row with each other.

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 9.1 Summary of Valid Combinations For Multifunction Instructions

Multifunction Instructions

<ALU*>	,	AX0	= DM (	I0	,	M0	),	AY0	= PM (	I4	,	M4	);
<MAC>		AX1		I1	,	M1		AY1		I5	,	M5	
		MX0		I2	,	M2		MY0		I6	,	M6	
		MX1		I3	,	M3		MY1		I7	,	M7	

AX0	= DM (	I0	,	M0	),	AY0	= PM (	I4	,	M4	);
AX1		I1	,	M1		AY1		I5	,	M5	
MX0		I2	,	M2		MY0		I6	,	M6	
MX1		I3	,	M3		MY1		I7	,	M7	

<ALU>	, dreg	=	DM (	I0	,	M0	)	;
<MAC>				I1	,	M1		
<SHIFT*>				I2	,	M2		
				I3	,	M3		
				I4	,	M4		
				I5	,	M5		
				I6	,	M6		
				I7	,	M7		
			PM (	I4	,	M4	)	
				I5	,	M5		
				I6	,	M6		
				I7	,	M7		

DM (	I0	,	M0	)	= dreg,	<ALU>	;
	I1	,	M1			<MAC>	
	I2	,	M2			<SHIFT>	
	I3	,	M3				
	I4	,	M4				
	I5	,	M5				
	I6	,	M6				
	I7	,	M7				
PM (	I4	,	M4	)			
	I5	,	M5				
	I6	,	M6				
	I7	,	M7				

<ALU>	, dreg	=	dreg;
<MAC>			
<SHIFT>			

Table 9.2 Multifunction Instructions
*All computation is unconditional; ALU Division and Shift Immediate operations prohibited

9.2.2 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.

9.2.2.1 ALU Group

Here is an example of one ALU instruction, 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. A complete list of the permissible *xops* and *yops* is given in the *ADSP-2101 Cross-Software Manual.* A complete list of conditions is in Table 4.1 in this manual.

ALU Instructions

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

[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 9.3 ALU Instructions

9.2.2.2 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 (implicitly signed) result.

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. A complete list of the permissible *xops* and *yops* is given in the *ADSP-2101 Cross-Software Manual.*

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 9.4 MAC Instructions

9.2.2.3 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 "not counter expired" condition. If the condition is false, a NOP is executed. The destination of all shifting operations is the Shifter Result register, SR. (The destination of exponent detection instructions is SE or SB, as shown below.) 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. Positive values cause left shifts; negative values cause right shifts.

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 (16-bit) half of SR. If "SR OR" is omitted, the result is passed directly into SR.

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 9.5 Shifter Instructions

9.2.3 MOVE: Read & Write

MOVE instructions move data to and from data registers and external memory. ADSP-2101 registers are 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.

Table 9.6 shows which registers belong to these groups. Many of the ADSP-2101 system control registers are memory-mapped; these registers are read and written as memory locations instead of with register names.

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	
TX0, TX1, RX0, RX1	
IFC	

Table 9.6 ADSP-2101 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 9.7 MOVE Instructions

9.2.4 Program Flow Control

Program Flow Control on the ADSP-2101 is simple but powerful. Here is an example of one 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 destination jumped to. Instead of the label, an index register in DAG2 may be explicitly used. 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.

If the counter condition (CE, NOT CE) is to be used, an assignment to CNTR must be executed to initialize the counter value. JUMP and CALL permit the additional conditionals "FLAG_IN" and "NOT FLAG_IN" to be used for branching on the state of the FI pin, but only with direct addressing, not with DAG2 as the address source.

RTS (return from subroutine) and RTI (return from interrupt) provide for conditional return from CALL or interrupt vectors respectively.

The FO pin (Flag Out) can be set, cleared or toggled; while this instruction does not alter the flow of your program, it provides a control structure for multiprocessor communication and is therefore included in this group.

The IDLE statement provides a way to wait for interrupts. IDLE causes the processor to wait in a low-power state until an interrupt occurs. When an interrupt is serviced, control returns to the instruction following the IDLE statement. IDLE uses less power than loops created with JUMP.

Here is a summary of all program flow control instructions. *Condition* and *termination* are described in Chapter 4, Tables 4.1 and 4.4.

Program Flow Control Instructions

```
[IF condition]      JUMP     |   (I4)      |  ;
                             |   (I5)      |
                             |   (I6)      |
                             |   (I7)      |
                             | <address>   |

IF | FLAG_IN     |            JUMP             <address> ;
   | NOT FLAG_IN |

[IF condition]      CALL     |   (I4)      |  ;
                             |   (I5)      |
                             |   (I6)      |
                             |   (I7)      |
                             | <address>   |

IF | FLAG_IN     |            CALL             <address> ;
   | NOT FLAG_IN |

| SET    |
| RESET  |          FLAG_OUT        ;
| TOGGLE |

[IF condition]      RTS ;

[IF condition]      RTI ;

DO <address>  [UNTIL termination] ;

IDLE;
```

Table 9.8 Program Flow Control Instructions

9.2.5 Miscellaneous Instructions

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

The Mode Control (enable/disable) instructions turn on and off several modes of operation. The instruction governs modes common to the ADSP-2100 (bit-reversal on DAG1, latching ALU overflow, saturating the ALU result register, choosing the primary or shadow register set) and the ADSP-2101 extended mode controls (GO mode for continued operation during Bus Grant, multiplier shift mode for fractional or integer arithmetic and timer enabling).

A single ENA or DIS can be followed by any number of mode identifiers, separated by commas; ENA and DIS can also be repeated. All seven modes can be enabled, disabled or changed in a single instruction.

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. If circular buffering is in use, modulus logic applies (See Chapter 3, "Data Moves," for more information).

Miscellaneous Instructions

```
NOP;

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

| ENA |       | BIT_REV  |      [ , ]       ;
| DIS |       | AV_LATCH |
              | AR_SAT   |
              | SEC_REG  |
              | G_MODE   |
              | M_MODE   |
              | TIMER    |

MODIFY (    | I0 |  ,  | M0 |  ) ;
            | I1 |  ,  | M1 |
            | I2 |  ,  | M2 |
            | I3 |  ,  | M3 |
            ----------------
            | I4 |  ,  | M4 |
            | I5 |  ,  | M5 |
            | I6 |  ,  | M6 |
            | I7 |  ,  | M7 |
```

Table 9.9 Miscellaneous Instructions

9.3 DATA STRUCTURES

The ADSP-2101 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-2101 Cross-Software Manual.*

9.3.1 Arrays

Arrays are the basic data structure in the ADSP-2101 instruction set. In ADSP-2101 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. This feature is supported only in the ADSP-210X Simulators even though data cannot be loaded directly into on-chip data memory by the hardware booting sequence.

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

9.3.2 Circular Arrays/Buffers

A common requirement in DSP is the circular buffer. This is directly implemented by the ADSP-2101 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 and M register.

```
L0 = %coefficients;  {length of circular buffer}
I0 = ^coefficients;  {point to address of buffer}
M0 = 1;              {increment by 1 location each time}
```

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.

9.3.3 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= H#800 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.

9.4 PROGRAM EXAMPLE

Below are three listings, showing an example of an FIR filter program written for the ADSP-2101 with discussion of each section of the program. This FIR filter program demonstrates much of the conceptual power of the ADSP-2101 architecture and instruction set. More complex programs would, of course, use many additional features of the language.

```
  {ADSP-2101  FIR Filter routine
  I/O uses serial port 0
  Internally generated serial clock
  12.288 MHz processor clock rate divided to 1.536MHz serial clock
  Serial clock divided to 8KHz frame sampling rate}

  .MODULE/RAM/ABS=0 main_routine;                 {program loaded    }
                                                  {from EPROM MMAP=0}
A .INCLUDE          <const.h>;
B .VAR/DM/RAM/ABS=H#3800/CIRC   data_buffer[taps];      {data values internal}
  .VAR/PM/RAM/CIRC          coefficient[taps];

  .GLOBAL           data_buffer, coefficient;
  .EXTERNAL         fir_start;
  .INIT             coefficient:<coeff.dat>;

  {code starts here}
  {load interrupt vector addresses}

                        JUMP restarter; nop; nop; nop;     {restart interrupt}
C                       RTI; nop; nop; nop;                {sampling interrupt IRQ2}
                        RTI; nop; nop; nop;                {SPORT0 Transmit int}
                        JUMP fir_start; nop; nop; nop;     {SPORT0 Receive int}
                        RTI; nop; nop; nop;                {SPORT1 Transmit int}
                        RTI; nop; nop; nop;                {SPORT1 Receive int}
                        RTI; nop; nop; nop;                {TIMER interrupt}

  {initializations}

D restarter:
                        L0 = %data_buffer;          {setup circular buffer length}
                        L4 = %coefficient;          {setup circular buffer length}

                        M0 = 1;                     {modify=1 for increment}
                        M4 = 1;                     {through buffers}

                        I0 = ^data_buffer;          {point to data start}
                        I4 = ^coefficient;          {point to coeff start}

E                       CNTR = %data_buffer;        {clear data buffer}
                        DO clear UNTIL CE;
  clear:                DM(I0,M0)=0;
```

```
                                I1 = H#3FEF;             {point to last DM reg.}
                                                         {for initialization}

                                DM(I1,M0)=H#0000;        {SPORT1 AUTOBUFF Disabled}
                                DM(I1,M0)=H#0000;        {SPORT1 RFSDIV not used}
                                DM(I1,M0)=H#0000;        {SPORT1 SCLKDIV not used}
                                DM(I1,M0)=H#0000;        {SPORT1 CNTL Disabled}
                                DM(I1,M0)=H#0000;        {SPORT0 AUTOBUFF Disabled}
                                DM(I1,M0)=191;           {Divide for 8KHz RFS}
                                DM(I1,M0)=H#0003;        {1.536MHz Internal Serial CLK}

                                DM(I1,M0)=H#6927;        {Multichannel disabled}
                                                         {int. gen serial clock}
                                                         {Receive frame sync required}
                                                         {Receive width 0}
                                                         {Transmit frame sync required}
                                                         {Transmit width 0}
                                                         {int Transmit frame sync disabled}
                                                         {int Receive frame sync enabled}
                                                         {u-law companding}
                                                         {8 bit words}
                                DM(I1,M0)=0000;          {Transmit multichannels}
                                DM(I1,M0)=0000;
                                DM(I1,M0)=0000;          {Receive multichannels}
                                DM(I1,M0)=0000;
                                DM(I1,M0)=0000;          {Timer not used, cleared}
                                DM(I1,M0)=0000;
                                DM(I1,M0)=0000;
                                DM(I1,M0)=H#7000;        {DM wait states }
                                                         {H#3400 - H#37FF 7 waits}
                                                         {all else 0 waits}
                                DM(I1,M0)=H#1000         {SPORT0 enabled}
                                                         {Boot page 0, 0 PM waits}
                                                         {0 Boot waits}
                                ICNTL = H#00;
                                IMASK = H#0018;          {enable SPORT0 interrupt only}
mainloop:                       IDLE;                    {wait for interrupt}
                                JUMP mainloop;

.ENDMOD;
```

Setup and Main Loop Routine

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

Include File, Constant Initialization

Figure 9.1 Program Example Listing 1, Main Routine & Constants File

9.4.1 Example Program: Setup Routine Discussion

The setup and main loop routine, Figure 9.1, on the previous page, does initialization and then uses the IDLE instruction to wait continuously until the receive interrupt from SPORT0 is received. The filter is interrupt-driven. When the interrupt occurs 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 in on-chip memory: 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. Since this module (first line of listing) is located at absolute address zero, the first instruction is placed at the restart vector: 0000. The first location is the restart vector instruction, which jumps to the routine *restarter*. Interrupts that are not used are filled with a return from interrupt instruction followed by NOPs. (Since only one interrupt will be enabled, this is only a thorough programming practice rather than a necessity.) The SPORT0 receive interrupt vector jumps to the interrupt service routine, which is too long to fit in the four locations available.

Section D, *restarter*, 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. The routine, *clear*, zeroes all values in the data memory buffer.

Section E, after *clear*, initializes all the system control registers. The use of the data address generator points out that most of the ADSP-2101 system control registers can be written in ascending order because of the memory map layout. Many bits are cleared for features that are not used in this example, including SPORT1, multichannel features, autobuffering features, etc. See Appendix D for a summary listing of control register initialization.

SPORT0 is set up to generate the serial clock internally at 1.536MHz, based on an assumed processor clock rate of 12.288MHz. The RFS and TFS signals are both required and the RFS signal is generated internally at

8kHz, while the TFS signal comes from the external device in communication with the processor.

Finally, SPORT0 is enabled and the interrupts are enabled. Now the IDLE instruction puts the processor into a wait for interrupts. After the return from interrupt instruction, execution resumes at the instruction following the IDLE instruction. Once these setup instructions have been executed, all further activity takes place in the interrupt service routine, described below.

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

    .INCLUDE    <const.h>;                      {include constant declarations}
    .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 = RX0;                       {read from SPORT0}
                           DM(I0,M0) = SI;                 {transfer data to buffer}
G   ---------------------- MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0);
                                                           {set up multiplier for loop}
H   ---------------------- 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}
                           TX0 = MR1;                      {write to sport}

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

Figure 9.2 Program Example Listing 2, Interrupt Routine

9.4.2 Example Program: Interrupt Routine Discussion

This subroutine transfers the received data 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 transmit section of SPORT0 and returns.

Section F identifies the module (which is relocatable rather than placed at an absolute address), includes the same file of constants, and makes the entry point visible to the main routine with the .ENTRY line. Likewise, the .EXTERNAL line makes the main routine labels visible in the interrupt routine.

The subroutine begins by loading the counter register CNTR. The new sample is read from SPORT0's receive data register, RX0, 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 G zeroes the multiplier result register (MR) and fetches the first two operands. This instruction access both program and data memory but still executes in a single cycle because of the ADSP-2101 internal architecture.

H identifies 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 without overhead.

The final value is transferred back to SPORT0, to the transmit data register, TX0, to be sent to the communicating device.

Instruction Coding ▣ A

A.1 OPCODES

Here is a summary of the complete instruction set of the ADSP-2101. 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 (Immediate 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 (Immediate 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	0	SF				Xop			Dest REG				Source REG			

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	TI		MM		AS		OL		BR		SR		GM		0	0

Explanation of these codes can be found together alphabetically under "Mode Control" in the next section.

A Instruction Coding

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			

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: 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	1	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

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

Instruction Coding A

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	

Type 27: Call or Jump on Flag In

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	Address												Addr		FIC	S
								12 LSBs												2 MSBs			

Type 28: Flag Out 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	0	0	1	0	0	0	0	0	0	0	0	0	0	0	FO		COND			

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

Type 31: Idle

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

A Instruction Coding

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)	
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	FOREVER

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

FIC FI condition code

1	latched FI is 1	FLAG_IN
0	latched FI is 0	NOT FLAG_IN

FO Mode Control codes for Flag Out pin

FO:	Set, clear, or toggle the output Flag.
0 0	No change
0 1	Toggle
1 0	Clear
1 1	Set

G Data Address Generator codes

0	DAG1
1	DAG2

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

Mode Control codes

SR:	Secondary register bank
BR:	Bit-reverse mode
OL:	ALU overflow latch mode
AS:	AR register saturate mode
MM:	Alternate Multiplier placement mode
GM:	GOMode; enable means go if possible
TI:	Timer enable

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

PP PC Stack Pop codes

0	No Change
1	Pop

Instruction Coding A

REG Register codes

Codes not assigned are reserved for future use.

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 (read only)
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	RX0
1 0 0 1	SE	L1	L5	TX0
1 0 1 0	AR	L2	L6	RX1
1 0 1 1	MR0	L3	L7	TX1
1 1 0 0	MR1	–	–	IFC (write only)
1 1 0 1	MR2	–	–	OWRCNTR
1 1 1 0	SR0	–	–	–
1 1 1 1	SR1	–	–	–

S Jump Type codes

0	Jump
1	Jump Subroutine

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	Derive Block Exponent	

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

X X Operand codes

0 0 0	X0 (SI for shifter)
0 0 1	X1 (invalid 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 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-2101'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 sixteen cycles produce a truncated quotient of sixteen 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 fifteen DIVQs. Regardless of which 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 either AY1, or AF.

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 fifteen bits of the dividend with the recently determined sign bit appended are stored in AY0, while the lower fifteen 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 and 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 fifteen (or thirty-one 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-2101 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-2101 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 twenty-seven 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. 15 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.

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 sixteen 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-2101. 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
```

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

```
    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
}

.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, dividend have same magnitude}

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 AY1;       {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 AY1;                {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)

```
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

Pin Information C

C.1 PIN DESCRIPTION

The ADSP-2101 is available in a 68-pin PGA and a 68-lead PLCC. Here is a description of each pin or group of pins.

Pin Group Name	*# of Pins*	*Function*
Address	14	Address output for program, data and boot memory spaces
Data	24	Data I/O pins for program and data memories. Input only for Boot memory space, with two MSBs used as Boot space addresses.
$\overline{\text{RESET}}$	1	Processor reset input
$\overline{\text{IRQ2}}$	1	External interrupt request #2 input
$\overline{\text{BR}}$	1	External bus request input
$\overline{\text{BG}}$	1	External bus grant output
$\overline{\text{PMS}}$	1	External program memory select
$\overline{\text{DMS}}$	1	External data memory select
$\overline{\text{BMS}}$	1	Boot memory select
$\overline{\text{RD}}$	1	External memory read enable output
$\overline{\text{WR}}$	1	External memory write enable output
MMAP	1	Memory Map select
CLKIN, XTAL	2	External clock input or quartz crystal I/O
CLKOUT	1	Processor clock output
SPORT0	5	Serial Port Zero I/O pins
SPORT1	5	Serial Port One I/O pins
	or	
$\overline{\text{IRQ1}}$ (TFS1)	1	External interrupt request #1 input
$\overline{\text{IRQ0}}$ (RFS1)	1	External interrupt request #0 input
SCLK1	1	Programmable clock output
FO (DT1)	1	Flag output pin
FI (DR1)	1	Flag input pin
GND	4	Ground pins
VDD	3	Power Supply

Table C.1 ADSP-2101 Pin List

C.2 PINOUT

The ADSP-2101 pinouts for the 68-lead PGA package and the 68-contact PLCC package are listed below. For more package information, see the *ADSP-2101 Data Sheet.*

PIN CONFIGURATION

PGA Number	Pin Name	PGA Number	Pin Name
B1	GND	K11	$\overline{WR}$
B2	D19	K10	$\overline{RD}$
C1	D20	J11	DT0
C2	D21	J10	TFS0
D1	D22	H11	RFS0
D2	D23	H10	GND
E1	VDD	G11	DR0
E2	MMAP	G10	SCLK0
F1	$\overline{BR}$	F11	DT1
F2	$\overline{IRQ2}$	F10	TFS1
G1	$\overline{RESET}$	E11	RFS1
G2	A0	E10	DR1
H1	A1	D11	SCLK1
H2	A2	D10	VDD
J1	A3	C11	D0
J2	A4	C10	D1
K1	VDD	B11	D2
L2	A5	A10	D3
K2	A6	B10	D4
L3	GND	A9	D5
K3	A7	B9	D6
L4	A8	A8	D7
K4	A9	B8	D8
L5	A10	A7	D9
K5	A11	B7	D10
L6	A12	A6	D11
K6	A13	B6	GND
L7	$\overline{PMS}$	A5	D12
K7	$\overline{DMS}$	B5	D13
L8	$\overline{BMS}$	A4	D14
K8	$\overline{BG}$	B4	D15
L9	XTAL	A3	D16
K9	CLKIN	B3	D17
L10	CLKOUT	A2	D18
C3	Index		

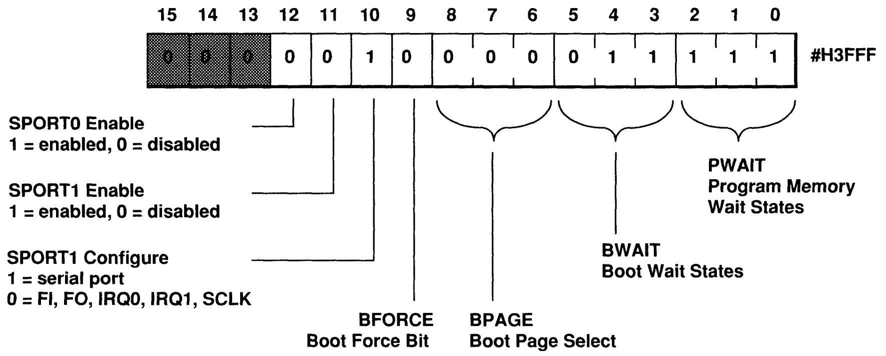
PLCC Number	Pin Name	PLCC Number	Pin Name
1	D11	35	A12
2	GND	36	A13
3	D12	37	$\overline{PMS}$
4	D13	38	$\overline{DMS}$
5	D14	39	$\overline{BMS}$
6	D15	40	$\overline{BG}$
7	D16	41	XTAL
8	D17	42	CLKIN
9	D18	43	CLKOUT
10	GND	44	$\overline{WR}$
11	D19	45	$\overline{RD}$
12	D20	46	DT0
13	D21	47	TFS0
14	D22	48	RFS0
15	D23	49	GND
16	VDD	50	DR0
17	MMAP	51	SCLK0
18	$\overline{BR}$	52	DT1
19	$\overline{IRQ2}$	53	TFS1
20	$\overline{RESET}$	54	RFS1
21	A0	55	DR1
22	A1	56	SCLK1
23	A2	57	VDD
24	A3	58	D0
25	A4	59	D1
26	VDD	60	D2
27	A5	61	D3
28	A6	62	D4
29	GND	63	D5
30	A7	64	D6
31	A8	65	D7
32	A9	66	D8
33	A10	67	D9
34	A11	68	D10

Control/Status Registers D

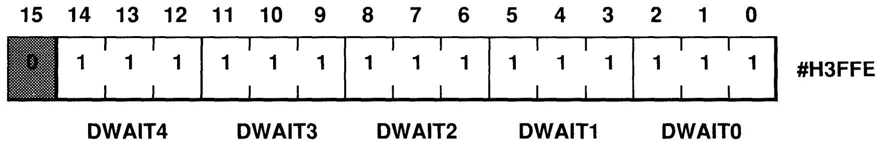
D.1 INTRODUCTION

This appendix shows bit definitions for 1) the memory-mapped control registers and 2) other control and status registers in the ADSP-2101. The memory-mapped registers are listed in descending address order. Default bit values at reset are shown; if no value is shown, the bit is undefined at reset. Reserved bits are shown on a gray field. These bits should always be written with zeros.

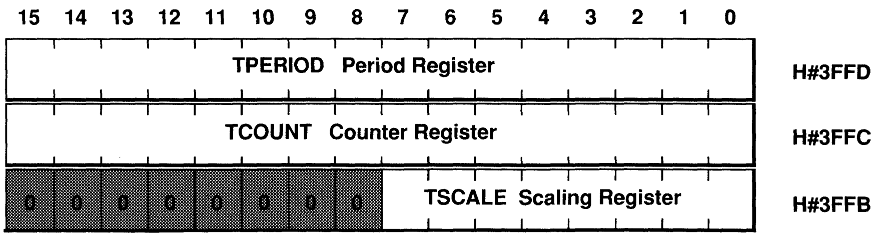
System Control Register



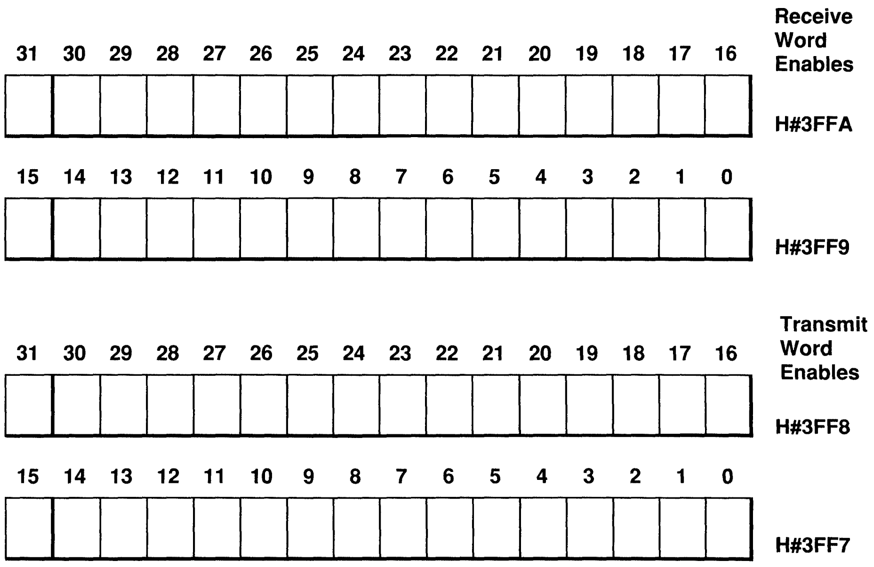
Data Memory Wait State Control Register



D Control/Status Registers

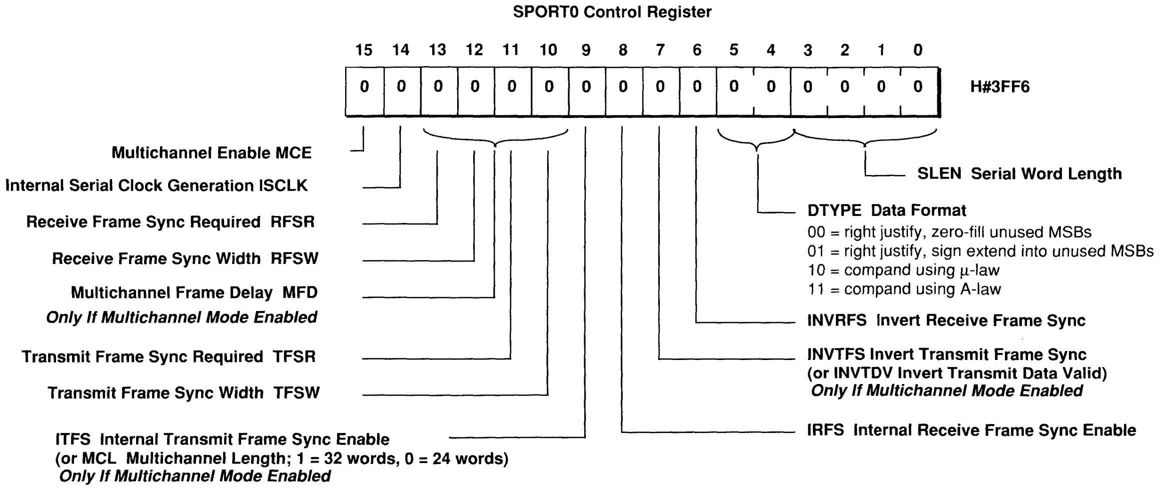


SPORT0 Multichannel Word Enable Register

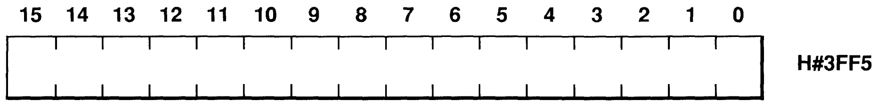


1 = Channel Enabled
0 = Channel Ignored

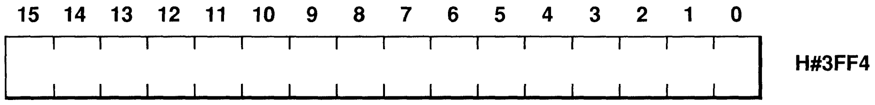
Control/Status Registers D


SPORT0 Control Register
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
H#3FF6
Multichannel Enable MCE
Internal Serial Clock Generation ISCLK
Receive Frame Sync Required RFSR
Receive Frame Sync Width RFSW
Multichannel Frame Delay MFD
Only If Multichannel Mode Enabled
Transmit Frame Sync Required TFSR
Transmit Frame Sync Width TFSW
ITFS Internal Transmit Frame Sync Enable
(or MCL Multichannel Length; 1 = 32 words, 0 = 24 words)
Only If Multichannel Mode Enabled
SLEN Serial Word Length
DTYPE Data Format
00 = right justify, zero-fill unused MSBs
01 = right justify, sign extend into unused MSBs
10 = compand using μ-law
11 = compand using A-law
INVRFS Invert Receive Frame Sync
INVTFS Invert Transmit Frame Sync
(or INVTDV Invert Transmit Data Valid)
Only If Multichannel Mode Enabled
IRFS Internal Receive Frame Sync Enable

SPORT0 SCLKDIV
Serial Clock Divide Modulus

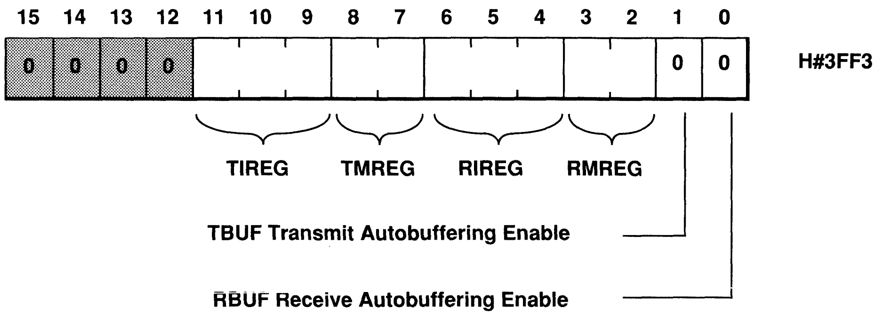


SPORT0 RFSDIV
Receive Frame Sync Divide Modulus

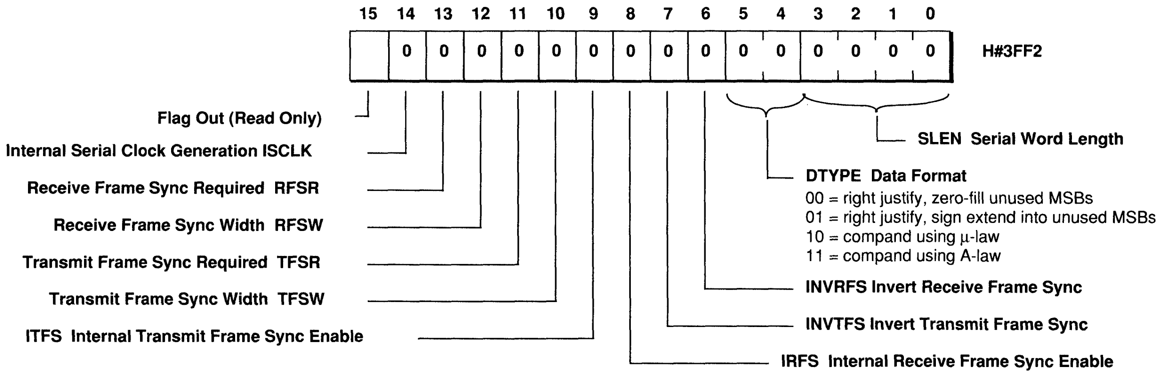


D Control/Status Registers

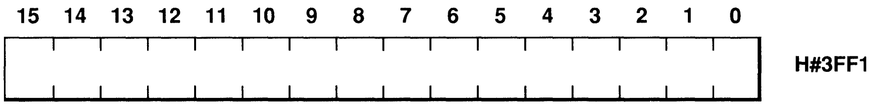
SPORT0 Autobuffer Control Register



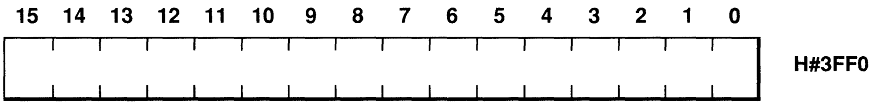
SPORT1 Control Register


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
H#3FF2
Flag Out (Read Only)
Internal Serial Clock Generation ISCLK
Receive Frame Sync Required RFSR
Receive Frame Sync Width RFSW
Transmit Frame Sync Required TFSR
Transmit Frame Sync Width TFSW
ITFS Internal Transmit Frame Sync Enable
SLEN Serial Word Length
DTYPE Data Format
00 = right justify, zero-fill unused MSBs
01 = right justify, sign extend into unused MSBs
10 = compand using µ-law
11 = compand using A-law
INVRFS Invert Receive Frame Sync
INVTFS Invert Transmit Frame Sync
IRFS Internal Receive Frame Sync Enable

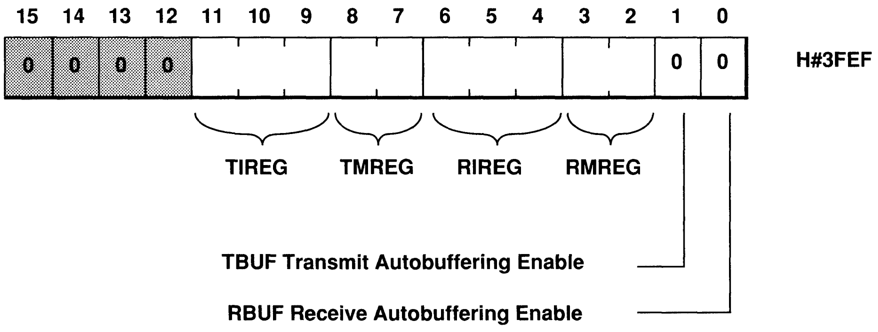
SPORT1 SCLKDIV
Serial Clock Divide Modulus



SPORT1 RFSDIV
Receive Frame Sync Divide Modulus

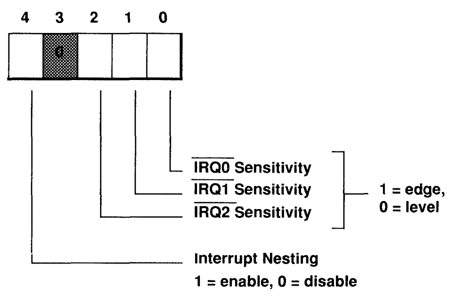


SPORT1 Autobuffer Control Register

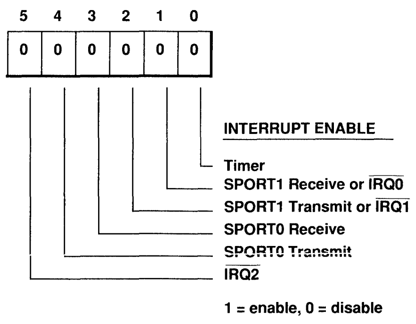


D Control/Status Registers

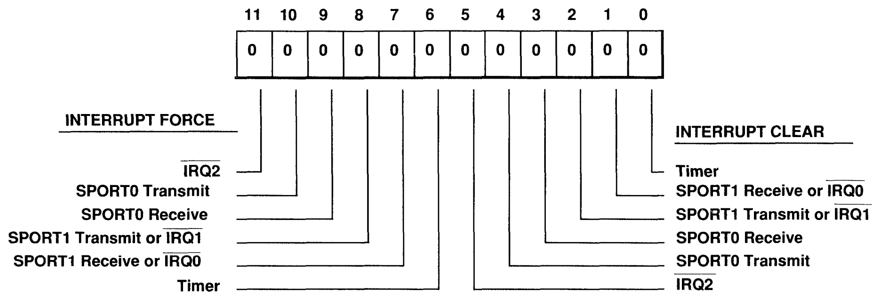
ICNTL


4 3 2 1 0
0
$\overline{\text{IRQ0}}$ Sensitivity
$\overline{\text{IRQ1}}$ Sensitivity
$\overline{\text{IRQ2}}$ Sensitivity
1 = edge,
0 = level
Interrupt Nesting
1 = enable, 0 = disable

IMASK



IFC



Source of Interrupt	**Interrupt Vector**
$\overline{\text{IRQ2}}$ (external pin)	0004 *(highest priority)*
SPORT0 Transmit (internal)	0008
SPORT0 Receive (internal)	000C
SPORT1 Transmit (internal) or $\overline{\text{IRQ1}}$ (external)	0010
SPORT1 Receive (internal) or $\overline{\text{IRQ0}}$ (external)	0014
Timer (internal)	0018 *(lowest priority)*

ASTAT

Bit	Default	Name	Description
7	0	SS	Shifter Input Sign
6	0	MV	MAC Overflow
5	0	AQ	ALU Quotient
4	0	AS	ALU X Input Sign
3	0	AC	ALU Carry
2	0	AV	ALU Overflow
1	0	AN	ALU Result Negative
0	0	AZ	ALU Result Zero

SSTAT

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

MSTAT

Bit	Default	Description
6	0	Go Mode Enable
5	0	Timer Enable
4	0	MAC Result Placement 0 = fractional, 1 = integer
3	0	AR Saturation Mode Enable
2	0	ALU Overflow Latch Mode Enable
1	0	Bit Reverse Mode Enable (DAG1)
0	0	Data Register Bank Select 0 = primary, 1 = secondary

Index

Index

ANALOG DEVICES

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

E1368-8-2/90