Contents

CHAPTER 1 INTRODUCTION

1.1 OVERVIEW 1–1
1.1.1 Functional Units 1–1
1.1.2 Memory And System Interface 1–3
1.1.3 Instruction Set 1–4
1.1.4 DSP Performance 1–4
1.2 CORE ARCHITECTURE 1–5
1.2.1 Computational Units 1–6
1.2.2 Address Generators & Program Sequencer 1–7
1.2.3 Buses 1–8
1.3 ON-CHIP PERIPHERALS 1–8
1.3.1 Serial Ports 1–8
1.3.2 Timer 1–9
1.3.3 Host Interface Port 1–9
1.3.4 DMA Ports 1–9
1.3.5 Analog Interface 1–10
1.4 ADSP-2100 FAMILY DEVELOPMENT TOOLS 1–10
1.5 ORGANIZATION OF THIS MANUAL 1–11

CHAPTER 2 COMPUTATIONAL UNITS

2.1 OVERVIEW 2–1
2.1.1 Binary String 2–1
2.1.2 Unsigned 2–1
2.1.3 Signed Numbers: Twos-Complement 2–1
2.1.4 Fractional Representation: 1.15 2–2
2.1.5 ALU Arithmetic 2–2
2.1.6 MAC Arithmetic 2–3
2.1.7 Shifter Arithmetic 2–3
2.1.8 Summary 2–4
2.2 ARITHMETIC/LOGIC UNIT (ALU) 2–5
2.2.1 ALU Block Diagram Discussion 2–5
2.2.2 Standard Functions 2–7
2.2.3 ALU Input/Output Registers 2–8
2.2.4 Multiprecision Capability 2–8
2.2.5 ALU Saturation Mode 2–8
2.2.6 ALU Overflow Latch Mode 2–9

Contents

2.2.7 Division 2–9
2.2.8 ALU Status 2–13
2.3 MULTIPLIER/ACCUMULATOR (MAC) 2–13
2.3.1 MAC Block Diagram Discussion 2–13
2.3.2 MAC Operations 2–16
2.3.2.1 Standard Functions 2–16
2.3.2.2 Input Formats 2–18
2.3.2.3 MAC Input/Output Registers 2–18
2.3.2.4 MR Register Operation 2–18
2.3.2.5 MAC Overflow And Saturation 2–19
2.3.2.6 Rounding Mode 2–20
2.3.2.7 Biased Rounding (ADSP-217x/218x/21msp5x) 2–21
2.4 BARREL SHIFTER 2–22
2.4.1 Shifter Block Diagram Discussion 2–22
2.4.2 Shifter Operations 2–28
2.4.2.1 Shifter Input/Output Registers 2–28
2.4.2.2 Derive Block Exponent 2–29
2.4.2.3 Immediate Shifts 2–30
2.4.2.4 Denormalize 2–31
2.4.2.5 Normalize 2–33

CHAPTER 3 PROGRAM CONTROL

3.1 OVERVIEW 3–1
3.2 PROGRAM SEQUENCER 3–1
3.2.1 Next Address Select Logic 3–3
3.2.2 Program Counter & PC Stack 3–4
3.2.3 Loop Counter & Stack 3–4
3.2.4 Loop Comparator & Stack 3–5
3.3 PROGRAM CONTROL INSTRUCTIONS 3–8
3.3.1 JUMP Instruction 3–8
3.3.1.1 Register Indirect JUMPs 3–8
3.3.2 CALL Instruction 3–9
3.3.3 DO UNTIL Loops 3–9
3.3.4 IDLE Instruction 3–10
3.3.4.1 Slow IDLE 3–10
3.4 INTERRUPTS 3–11
3.4.1 Interrupt Servicing Sequence 3–14
3.4.2 Configuring Interrupts 3–14
3.4.2.1 Interrupt Control Register (ICNTL) 3–15
3.4.2.2 Interrupt Mask Register (IMASK) 3–16
3.4.2.3 Global Enable/Disable For Interrupts 3–17
3.4.2.4 Interrupt Force & Clear Register (IFC) 3–18
3.4.3 Interrupt Latency 3–18
3.4.3.1 Timer Interrupt Latency (ADSP-2101/2105/2111/2115) 3–19

Contents

3.5 STATUS REGISTERS & STATUS STACK 3–20
3.5.1 Arithmetic Status Register (ASTAT) 3–20
3.5.2 Stack Status Register (SSTAT) 3–21
3.5.3 Mode Status Register (MSTAT) 3–22
3.6 CONDITIONAL INSTRUCTIONS 3–24
3.7 TOPPCSTACK 3–25
3.7.1 TOPPCSTACK Restrictions 3–27

CHAPTER 4 DATA TRANSFER

4.1 OVERVIEW 4–1
4.2 DATA ADDRESS GENERATORS (DAGS) 4–1
4.2.1 DAG Registers 4–1
4.2.2 Indirect Addressing 4–3
4.2.2.1 Initialize L Registers To 0 For Non-Circular Addressing 4–3
4.2.3 Modulo Addressing (Circular Buffers) 4–4
4.2.4 Calculating The Base Address 4–5
4.2.4.1 Circular Buffer Base Address Example 1 4–5
4.2.4.2 Circular Buffer Base Address Example 2 4–5
4.2.4.3 Circular Buffer Operation Example 1 4–5
4.2.4.4 Circular Buffer Operation Example 2 4–6
4.2.5 Bit-Reverse Addressing 4–6
4.3 PROGRAMMNG DATA ACCESSES 4–7
4.3.1 Variables & Arrays 4–7
4.3.2 Circular Buffers 4–8
4.4 PMD-DMD BUS EXCHANGE 4–9
4.4.1 PMD-DMD Block Diagram Discussion 4–9

CHAPTER 5 SERIAL PORTS

5.1 OVERVIEW 5–1
5.2 BASIC SPORT DESCRIPTION 5–1
5.2.1 Interrupts 5–4
5.2.2 SPORT Operation 5–4
5.3 SPORT PROGRAMMING 5–4
5.3.1 SPORT Configuration 5–5
5.3.2 Receiving And Transmitting Data 5–6
5.4 SPORT ENABLE 5–7
5.5 SERIAL CLOCKS 5–8
5.6 WORD LENGTH 5–9
5.7 WORD FRAMING OPTIONS 5–10
5.7.1 Frame Synchronization 5–10
5.7.2 Frame Sync Signal Source 5–11
5.7.3 Normal And Alternate Framing Modes 5–13
5.7.4 Active High Or Active Low 5–14

Contents

5.8 CONFIGURATION EXAMPLE 5–15
5.9 TIMING EXAMPLES 5–16
5.10 COMPANDING AND DATA FORMAT 5–23
5.10.1 Companding Operation Example 5–24
5.10.2 Contention For Companding Hardware 5–25
5.10.3 Companding Internal Data 5–25
5.11 AUTOBUFFERING 5–26
5.11.1 Autobuffering Control Register 5–27
5.11.2 Autobuffering Example 5–28
5.12 MULTICHANNEL FUNCTION 5–30
5.12.1 Multichannel Setup 5–30
5.12.2 Multichannel Operation 5–32
5.13 SPORT TIMING CONSIDERATIONS 5–34
5.13.1 Companding Delay 5–34
5.13.2 Clock Synchronization Delay 5–34
5.13.2.1 Startup Timing 5–34
5.13.3 Internally Generated Frame Sync Timing 5–34
5.13.4 Transmit Interrupt Timing 5–36
5.13.5 Receive Interrupt Timing 5–36
5.13.6 Interrupt And Autobuffer Synchronization 5–38
5.13.7 Instruction Completion Latencies 5–38
5.13.8 Interrupt And Autobuffer Service Example 5–39
5.13.9 Receive Companding Latency 5–40
5.13.10 Interrupts With Autobuffering Enabled 5–41
5.13.11 Unusual Complications 5–42

CHAPTER 6 TIMER

6.1 OVERVIEW 6–1
6.2 TIMER ARCHITECTURE 6–1
6.3 RESOLUTION 6–3
6.4 TIMER OPERATION 6–3

Contents

CHAPTER 7 HOST INTERFACE PORT

7.1 OVERVIEW 7–1
7.2 HIP PIN SUMMARY 7–2
7.3 HIP FUNCTIONAL DESCRIPTION 7–4
7.4 HIP OPERATION 7–6
7.4.1 Polled Operation 7–7
7.4.1.1 HIP Status Synchronization 7–8
7.4.2 Interrupt-Driven Operation 7–9
7.4.3 HDR Overwrite Mode 7–9
7.4.4 Software Reset 7–10
7.5 HIP INTERRUPTS 7–10
7.6 HOST INTERFACE TIMING 7–11
7.7 BOOT LOADING THROUGH THE HIP 7–16

CHAPTER 8 ANALOG INTERFACE

8.1 OVERVIEW 8–1
8.2 A/D CONVERSION 8–2
8.2.1 Analog Input 8–2
8.2.2 ADC 8–3
8.2.2.1 Decimation Filter 8–4
8.2.2.2 High Pass Filter 8–5
8.3 D/A CONVERSION 8–6
8.3.1 DAC 8–6
8.3.1.1 High Pass Filter 8–6
8.3.1.2 Interpolation Filter 8–7
8.3.1.3 Analog Smoothing Filter & Programmable Gain Amp. .. 8–8
8.3.2 Differential Output Amplifier 8–8
8.4 OPERATING THE ANALOG INTERFACE 8–9
8.4.1 Memory-Mapped Control Registers 8–9
8.4.1.1 Analog Control Register 8–9
8.4.1.2 Analog Autobuffer/Powerdown Register 8–10
8.4.2 Memory-Mapped Data Registers 8–11
8.4.3 ADC & DAC Interrupts 8–12
8.4.3.1 Autobuffering Disabled 8–12
8.4.3.2 Autobuffering Enabled 8–13
8.5 CIRCUIT DESIGN CONSIDERATIONS 8–16
8.5.1 Analog Signal Input 8–16
8.5.2 Analog Signal Output 8–18
8.5.3 Voltage Reference Filter Capacitance 8–19

Contents

CHAPTER 9 SYSTEM INTERFACE

9.1 OVERVIEW 9–1
9.2 CLOCK SIGNALS 9–3
9.2.1 Synchronization Delay 9–3
9.2.2 1x & 1/2x Clock Considerations 9–4
9.3 RESET 9–4
9.4 SOFTWARE-FORCED REBOOTING 9–4
9.4.1 ADSP-2181 Register Values For BDMA Booting 9–13
9.5 EXTERNAL INTERRUPTS 9–14
9.5.1 Interrupt Sensitivity 9–14
9.6 FLAG PINS 9–15
9.7 POWERDOWN 9–17
9.7.1 Powerdown Control 9–18
9.7.2 Entering Powerdown 9–19
9.7.3 Exiting Powerdown 9–20
9.7.3.1 Ending Powerdown With The $\overline{\text{PWD}}$ Pin 9–20
9.7.3.2 Ending Powerdown With The $\overline{\text{RESET}}$ Pin 9–21
9.7.4 Startup Time After Powerdown 9–21
9.7.4.1 Systems Using An External TTL/CMOS Clock 9–21
9.7.4.2 Systems Using A Crystal/Internal Oscillator 9–22
9.7.5 Operation During Powerdown 9–23
9.7.5.1 Interrupts & Flags 9–23
9.7.5.2 SPORTS 9–23
9.7.5.3 HIP During Powerdown 9–24
9.7.5.4 IDMA Port During Powerdown (ADSP-2181) 9–25
9.7.5.5 BDMA Port During Powerdown (ADSP-2181) 9–26
9.7.5.6 Analog Interface (ADSP-21msp5x) 9–26
9.7.6 Conditions For Lowest Power Consumption 9–26
9.7.7 PWDACK Pin 9–29
9.7.8 Using Powerdown As A Non-Maskable Interrupt 9–30

CHAPTER 10 MEMORY INTERFACE

10.1 OVERVIEW 10–1
10.2 PROGRAM MEMORY INTERFACE 10–3
10.2.1 External Program Memory Read/Write 10–3
10.2.2 Program Memory Maps 10–5
10.2.3 ROM Program Memory Maps 10–6
10.3 DATA MEMORY INTERFACE 10–10
10.3.1 External Data Memory Read/Write 10–10
10.3.2 Data Memory Maps 10–11
10.3.3 Memory-Mapped Peripherals 10–14

Contents

10.4 BOOT MEMORY INTERFACE 10–15
10.4.1 Boot Pages 10–15
10.4.2 Powerup Boot & Software Reboot 10–16
10.4.3 Boot Memory Access 10–17
10.4.4 Boot Loading Sequence 10–17
10.5 BUS REQUEST/GRANT 10–21
10.6 ADSP-2181 MEMORY INTERFACES 10–23
10.6.1 ADSP-2181 Program Memory Interface 10–25
10.6.2 ADSP-2181 Data Memory Interface 10–30
10.6.3 ADSP-2181 Byte Memory Interface 10–32
10.6.4 ADSP-2181 I/O Memory Space 10–32
10.6.5 ADSP-2181 Composite Memory Select 10–35
10.6.6 External Memory Read – Overlays & I/O Memory 10–36
10.6.7 External Memory Write – Overlays & I/O Memory 10–37
10.7 MEMORY INTERFACE SUMMARY (ALL PROCESSORS) ... 10–37

CHAPTER 11 DMA PORTS

11.1 OVERVIEW 11–1
11.2 BDMA PORT 11–2
11.2.1 BDMA Port Functional Description 11–4
11.2.2 BDMA Control Registers 11–4
11.2.3 Byte Memory Word Formats 11–9
11.2.4 BDMA Booting 11–9
11.2.4.1 Development Software Features for BDMA Booting . 11–11
11.3 IDMA PORT 11–12
11.3.1 IDMA Port Pin Summary 11–12
11.3.2 IDMA Port Functional Description 11–14
11.3.3 Modifying Control Registers for IDMA 11–16
11.3.4 IDMA Timing 11–17
11.3.4.1 Address Latch Cycle 11–17
11.3.4.2 Long Read Cycle 11–18
11.3.4.3 Short Read Cycle 11–20
11.3.4.4 Long Write Cycle 11–21
11.3.4.5 Short Write Cycle 11–23
11.3.5 Boot Loading Through The IDMA Port 11–24
11.3.6 DMA Cycle Stealing, DMA Hold Offs, and $\overline{\text{IACK}}$ 11–25

Contents

CHAPTER 12 PROGRAMMING MODEL

12.1 OVERVIEW 12–1
12.1.1 Data Address Generators 12–2
12.1.1.1 Always Initialize L Registers 12–2
12.1.2 Program Sequencer 12–4
12.1.2.1 Interrupts 12–4
12.1.2.2 Loop Counts 12–4
12.1.2.3 Status And Mode Bits 12–5
12.1.2.4 Stacks 12–5
12.1.3 Computational Units 12–6
12.1.4 Bus Exchange 12–6
12.1.5 Timer 12–6
12.1.6 Serial Ports 12–7
12.1.7 Memory Interface & SPORT Enables 12–7
12.1.8 Host Interface 12–8
12.1.9 Analog Interface 12–8
12.2 PROGRAM EXAMPLE 12–8
12.2.1 Example Program: Setup Routine Discussion 12–10
12.2.2 Example Program: Interrupt Routine Discussion 12–11

CHAPTER 13 HARDWARE EXAMPLES

13.1 OVERVIEW 13–1
13.2 BOOT LOADING FROM HOST USING BUS REQUEST 13–2
13.3 SERIAL PORT TO CODEC INTERFACE 13–5
13.4 SERIAL PORT TO DAC INTERFACE 13–8
13.5 SERIAL PORT TO ADC INTERFACE 13–10
13.6 SERIAL PORT TO SERIAL PORT INTERFACE 13–12
13.7 80C51 INTERFACE TO HOST INTERFACE PORT 13–13

CHAPTER 14 SOFTWARE EXAMPLES

14.1 OVERVIEW 14–1
14.2 SYSTEM DEVELOPMENT PROCESS 14–2
14.3 SINGLE-PRECISION FIR TRANSVERSAL FILTER 14–4
14.4 CASCADED BIQUAD IIR FILTER 14–6
14.5 SINE APPROXIMATION 14–7
14.6 SINGLE-PRECISION MATRIX MULTIPLY 14–9
14.7 RADIX-2 DECIMATION-IN-TIME FFT 14–11
14.7.1 Main Module 14–11
14.7.2 DIT FFT Subroutine 14–13
14.7.3 Bit-Reverse Subroutine 14–18
14.7.4 Block Floating-Point Scaling Subroutine 14–19

Contents

CHAPTER 15 INSTRUCTION SET REFERENCE

15.1 QUICK LIST OF INSTRUCTIONS 15–1
15.2 OVERVIEW 15–2
15.3 INSTRUCTION TYPES & NOTATION CONVENTIONS 15–3
15.4 MULTIFUNCTION INSTRUCTIONS 15–4
15.4.1 ALU/MAC With Data & Program Memory Read 15–4
15.4.2 Data & Program Memory Read 15–6
15.4.3 Computation With Memory Read 15–6
15.4.4 Computation With Memory Write 15–6
15.4.5 Computation With Data Register Move 15–7
15.5 ALU, MAC & SHIFTER INSTRUCTIONS 15–9
15.5.1 ALU Group 15–9
15.5.2 MAC Group 15–10
15.5.3 Shifter Group 15–11
15.6 MOVE: READ & WRITE 15–12
15.7 PROGRAM FLOW CONTROL 15–14
15.8 MISCELLANEOUS INSTRUCTIONS 15–16
15.9 EXTRA CYCLE CONDITIONS 15–18
15.9.1 Multiple Off-Chip Memory Accesses 15–18
15.9.2 Wait States 15–18
15.9.3 SPORT Autobuffering & DMA 15–18
15.10 INSTRUCTION SET SYNTAX 15–19
15.10.1 Punctuation & Multifunction Instructions 15–19
15.10.2 Syntax Notation Example 15–19
15.10.3 Status Register Notation 15–20

ALU Add/Add with Carry 15–23
Subtract X-Y/Subtract X-Y with Borrow 15–25
Subtract Y-X/Subtract Y-X with Borrow 15–27
AND, OR, XOR 15–29
Test Bit, Clear Bit, Set Bit, Toggle Bit 15–31
Pass/Clear 15–33
Negate 15–35
NOT 15–36
Absolute Value 15–37
Increment 15–38
Decrement 15–39
Divide 15–40
Generate ALU Status 15–42

Contents

MAC Multiply 15–43
Multiply/Accumulate 15–45
Multiply/Subtract 15–47
Clear 15–49
Transfer MR 15–50
Conditional MR Saturation 15–51

SHIFTER
Arithmetic Shift 15–52
Logical Shift 15–54
Normalize 15–56
Derive Exponent 15–58
Block Exponent Adjust 15–60
Arithmetic Shift Immediate 15–62
Logical Shift Immediate 15–64

MOVE
Register Move 15–65
Load Register Immediate 15–67
Data Memory Read (Direct Address) 15–69
Data Memory Read (Indirect Address) 15–70
Program Memory Read (Indirect Address) 15–71
Data Memory Write (Direct Address) 15–72
Data Memory Write (Indirect Address) 15–73
Program Memory Write (Indirect Address) 15–75
I/O Space Read/Write 15–76

PROGRAM FLOW
JUMP 15–77
CALL 15–78
JUMP or CALL on Flag In Pin 15–79
Modify Flag Out Pin 15–80
Return From Subroutine (RTS) 15–81
Return From Interrupt (RTI) 15–82
Do Until 15–83
IDLE 15–85

Contents

MISC
Stack Control 15–86
Mode Control 15–89
Modify Address Register 15–91
NOP 15–92
Interrupt Enable/Disable 15–93

MULTIFUNCTION
ALU/MAC/SHIFT Operation with Memory Read 15–94
ALU/MAC/SHIFT Operation with Register to Register Move . 15–98
ALU/MAC/SHIFT Operation with Memory Write 15–101
Data & Program Memory Read 15–105
ALU/MAC Operation with Data & Program Memory Read ... 15–106

APPENDIX A INSTRUCTION CODING

A.1 OPCODES A–1
A.2 ABBREVIATION CODING A–7

APPENDIX B DIVISION EXCEPTIONS

B.1 DIVISION FUNDAMENTALS B–1
B.1.1 Signed Division B–1
B.1.2 Unsigned Division B–2
B.1.3 Output Formats B–2
B.1.4 Integer Division B–3
B.2 ERROR CONDITIONS B–3
B.2.1 Negative Divisor Error B–3
B.2.2 Unsigned Division Error B–4
B.3 SOFTWARE SOLUTION B–4

APPENDIX C NUMERIC FORMATS

C.1 OVERVIEW C–1
C.2 UNSIGNED OR SIGNED: TWOS-COMPLEMENT FORMAT .. C–1
C.3 INTEGER OR FRACTIONAL C–1
C.4 BINARY MULTIPLICATION C–3
C.4.1 Fractional Mode And Integer Mode C–4
C.5 BLOCK FLOATING-POINT FORMAT C–5

Contents

APPENDIX D INTERRUPT VECTOR ADDRESSES

D.1 INTERRUPT VECTOR ADDRESSES D–1

APPENDIX E CONTROL/STATUS REGISTERS

E.1 OVERVIEW E-1

INDEX

Introduction 1

1.1 OVERVIEW

The ADSP-2100 family is a collection of programmable single-chip microprocessors that share a common base architecture optimized for digital signal processing (DSP) and other high-speed numeric processing applications. The various family processors differ principally in the type of on-chip peripherals they add to the base architecture. On-chip memory, a timer, serial port(s), and parallel ports are available in different members of the family. In addition, the ADSP-21msp58/59 processors include an on-chip analog interface for voiceband signal conversion.

This manual provides the information necessary to understand and evaluate the processors' architecture, and to determine which device best meets your needs for a particular application. Together with the data sheets describing the individual devices, this manual provides all the information required to design a DSP system. Complete reference material for programmers is also included.

1.1.1 Functional Units

Table 1.1 on the following page lists the main functional units of the ADSP-21xx architecture, and shows which functions are included on each of the processors.

- *Computational Units*—Every processor in the ADSP-2100 family contains three independent, full-function computational units: an arithmetic/logic unit (ALU), a multiplier/accumulator (MAC) and a barrel shifter. The computational units process 16-bit data directly and also provide hardware support for multiprecision computations.

- *Data Address Generators & Program Sequencer*—Two dedicated address generators and a 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 generate simultaneous addresses for dual operand fetches. Together the sequencer and data address generators keep the computational units continuously working, maximizing throughput.

1 Introduction

Feature	2101	2103	2105	2115	2111	2171	2173	2181	2183	21msp58
Arithmetic/Logic Unit	•	•	•	•	•	•	•	•	•	•
Multiply/Accumulator	•	•	•	•	•	•	•	•	•	•
Shifter	•	•	•	•	•	•	•	•	•	•
Data Address Generators	•	•	•	•	•	•	•	•	•	•
Program Sequencer	•	•	•	•	•	•	•	•	•	•
Data Memory RAM	1K	1K	512	512	1K	2K	2K	16K	16K	2K
Program Memory RAM	2K	2K	1K	1K	2K	2K	2K	16K	16K	2K
Timer	•	•	•	•	•	•	•	•	•	•
Serial Port 0 (Multichannel)	•	•	–	•	•	•	•	•	•	•
Serial Port 1	•	•	•	•	•	•	•	•	•	•
Host Interface Port	–	–	–	–	•	•	•	–	–	•
DMA Ports	–	–	–	–	–	–	–	•	•	–
Analog Interface	–	–	–	–	–	–	–	–	–	•
Supply Voltage	5V	3.3V	5V	5V	5V	5V	3.3V	5V	3.3V	5V
Instruction Rate (MIPS)	20	10	13.8	20	20	33	20	33	33	26

Table 1.1 ADSP-2100 Family Processor Features & On-Chip Peripherals

- *Memory*—The ADSP-2100 family uses a modified Harvard architecture in which data memory stores data, and program memory stores both instructions and data. All ADSP-2100 family processors contain on-chip RAM that comprises a portion of the program memory space and data memory space. The speed of the on-chip memory allows the processor to fetch two operands (one from data memory and one from program memory) and an instruction (from program memory) in a single cycle.

- *Serial Ports*—The serial ports (SPORTs) provide a complete serial interface with hardware companding for data compression and expansion. Both μ-law and A-law companding are supported. The SPORTs interface easily and directly to a wide variety of popular serial devices. Each SPORT can generate a programmable internal clock or accept an external clock. SPORT0 includes a multichannel option.

- *Timer*—A programmable timer/counter with 8-bit prescaler provides periodic interrupt generation.

- *Host Interface Port*—The Host Interface Port (HIP) allows direct connection (with no glue logic) to a host processor. The HIP is made up of 16 data pins and 11 control pins. The HIP is extremely flexible and has provisions to allow simple interface to a variety of host processors. For example, the Motorola 68000, the Intel 8051, or another ADSP-2100 family processor can be easily connected to the HIP.

- *DMA Ports*—The ADSP-2181's Internal DMA Port (IDMA) and Byte DMA Port (BDMA) provide efficient data transfers to and from internal memory. The IDMA port has a 16-bit multiplexed address and data bus and supports 24-bit program memory. The IDMA port is completely asynchronous and can be written to while the ADSP-2181 is operating at full speed. The byte memory DMA port allows boot loading and storing of program instructions and data.

- *Analog Interface*—The ADSP-21msp58/59 processors include on-chip circuitry for mixed analog and digital signal processing. This circuitry includes an analog-to-digital converter (ADC), a digital-to-analog converter (DAC), analog and digital filters, and a parallel interface to the processor's core. The converters use sigma-delta technology to capture data samples from a highly oversampled signal.

The ADSP-2100 family architecture exhibits a high degree of parallelism, tailored to DSP requirements. In a single cycle, any device in the family 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.

In that same cycle, processors which have the relevant functional units can also:

- Receive and/or transmit data via the serial port(s).
- Receive and/or transmit data via the host interface port.
- Receive and/or transmit data via the DMA ports.
- Receive and/or transmit data via the analog interface.

1.1.2 Memory And System Interface

In each ADSP-21xx processor, four on-chip buses connect internal memory with the other functional units: Data Memory Address bus, Data Memory Data bus, Program Memory Address bus, and Program Memory Data bus. A single external address bus and and a single external data bus are extended off-chip; these buses can be used for either program or data memory access.

External devices can gain control of the processor's buses with the bus request and grant signals ($\overline{BR}$, $\overline{BG}$). The ADSP-21xx processors can continue running while the buses are granted to another device, as long as an external memory operation is not required.

1 Introduction

The ADSP-21xx processors support memory-mapped peripherals with programmable wait state generation.

Boot circuitry provides for loading on-chip program memory automatically after reset. This can be done either through the memory interface from a single low-cost EPROM, through the host interface port from a host processor, or through the BDMA port of the ADSP-2181. Multiple programs can be selected and loaded with no additional hardware.

ADSP-2100 family processors differ in their response to interrupts. In all cases, however, the program sequencer allows the processor to respond with minimum latency. Interrupts can be nested with no additional latency. External interrupts can be configured as edge- or level-sensitive. Internal interrupts can be generated from the timer, the host interface port, the serial ports, and the BDMA port.

1.1.3 Instruction Set

The ADSP-2100 family shares a single unified instruction set designed for upward compatibility with higher-integration devices. The ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors have a number of additional and enhanced instructions.

The ADSP-2100 family instruction set provides flexible data moves. Multifunction instructions combine one or more data moves with a computation. Every instruction can be executed in a single processor cycle. The assembly language uses an algebraic syntax for readability and ease of coding. A comprehensive set of software and hardware tools supports program development.

1.1.4 DSP Performance

Signal processing applications make special performance demands which distinguish DSP architectures from other microprocessor and microcontroller architectures. Not only must instruction execution be fast, but DSPs must also perform well in each of the following areas:

- *Fast and Flexible Arithmetic*—The ADSP-2100 family base architecture provides single-cycle computation for multiplication, multiplication with accumulation, arbitrary amounts of shifting, and standard arithmetic and logical operations. In addition, the arithmetic units allow for any sequence of computations so that a given DSP algorithm can be executed without being reformulated.

- *Extended Dynamic Range*—Extended sums-of-products, common in DSP algorithms, are supported in the multiply/accumulate units of the ADSP-2100 family. A 40-bit accumulator provides eight bits of protection against overflow in successive additions to ensure that no loss of data or range occurs; 256 overflows would have to occur before any data is lost. Special instructions are provided for implementing block floating-point scaling of data.

- *Single-Cycle Fetch of Two Operands*—In extended sums-of-products calculations, two operands are needed on each cycle to feed the calculation. All members of the ADSP-2100 family are able to sustain two-operand data throughput, whether the data is stored on-chip or off.

- *Hardware Circular Buffers*—A large class of DSP algorithms, including digital filters, requires circular data buffers. The ADSP-2100 family base architecture includes hardware to handle address pointer wraparound, simplifying the implementation of circular buffers both on- and off-chip, and reducing overhead (thereby improving performance).

- *Zero-Overhead Looping and Branching*—DSP algorithms are repetitive and are most logically expressed as loops. The program sequencer in the ADSP-2100 family supports looped code with zero overhead, combining excellent performance with the clearest program structure. Likewise, there are no overhead penalties for conditional branches.

1.2 CORE ARCHITECTURE

This section describes the core architecture of the ADSP-2100 family, as shown in Figure 1.1. Each component of the core architecture is described in detail in different chapters of this manual, as shown below:

Arithmetic/logic unit (ALU)	Chapter 2, *Computation Units*
Multiplier/accumulator (MAC)	Chapter 2, *Computation Units*
Barrel shifter	Chapter 2, *Computation Units*
Program sequencer	Chapter 3, *Program Control*
Status registers and stacks	Chapter 3, *Program Control*
Two data address generators (DAGs)	Chapter 4, *Data Transfer*
PMD-DMD bus exchange (PX registers)	Chapter 4, *Data Transfer*

1 Introduction

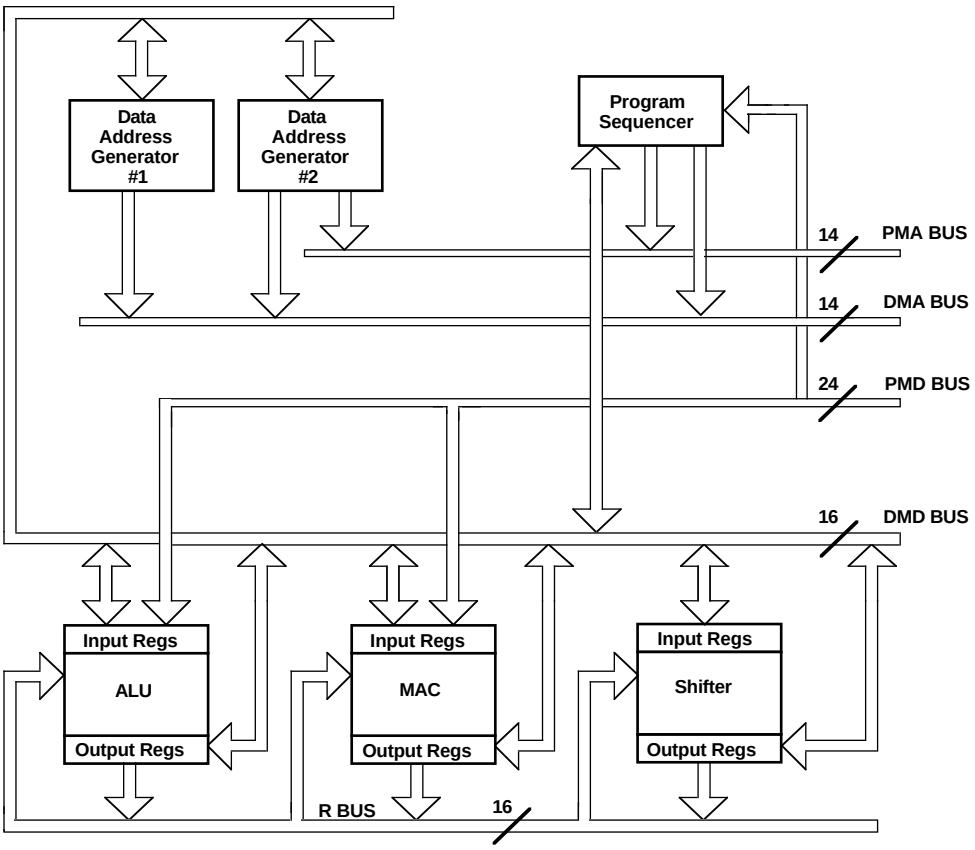


Figure 1.1 Base Architecture

1.2.1 Computational Units

Every processor in the ADSP-2100 family contains three independent, full-function computational units: an arithmetic/logic unit (ALU), a multiplier/accumulator (MAC) and a barrel shifter. The computation units process 16-bit data directly and provide hardware support for multiprecision computation as well.

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 units 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.

1.2.2 Address Generators & Program Sequencer

Two dedicated data address generators and a powerful program sequencer ensure efficient use of the computational units. The data address generators (DAGs) provide memory addresses when memory data is transferred to or from the input or 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 Serial Ports chapter for additional information.)

DAG1 can supply addresses to data memory only; DAG2 can supply addresses to either data memory or program memory. When the appropriate mode bit is set in the mode status register (MSTAT), the output address of DAG1 is bit-reversed before being driven onto the address bus. This feature facilitates addressing in radix-2 Fast Fourier Transform (FFT) algorithms.

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 processor executes looped code with zero overhead. No explicit jump instructions are required to loop.

1.2.3 Buses

The processors have five internal buses. The program memory address (PMA) and data memory address (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. The buses are multiplexed into a single external address bus and a single external data bus; the $\overline{\text{BMS}}$, $\overline{\text{DMS}}$ and $\overline{\text{PMS}}$ signals select the different address spaces. The R bus transfers intermediate results directly between the various computational units.

The PMA bus is 14 bits wide allowing direct access of up to 16K words of mixed instruction code and data. The PMD bus is 24 bits wide to accommodate the 24-bit instruction width.

The 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 DMD bus provides a path for the contents of any register in the processor to be transferred to any other register or to any 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, when necessary.

1.3 ON-CHIP PERIPHERALS

This section describes the additional functional units which are included in various processors of the ADSP-2100 family.

1.3.1 Serial Ports

Most ADSP-21xx processors have 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 serial port, SPORT0, has a multichannel capability that allows the receiving or transmitting of arbitrary data words from a 24-word or 32-word bitstream. The other

serial port, SPORT1, may optionally be configured as two additional external interrupt pins ($\overline{IRQ1}$ and $\overline{IRQ0}$)and the Flag Out (FO) and Flag In (FI) pins.

1.3.2 Timer

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.

1.3.3 Host Interface Port (ADSP-2111, ADSP-2171, ADSP-21msp5x)

The host interface port (HIP) is a parallel I/O port that allows for an easy connection to a host processor. Through the HIP, an ADSP-21xx DSP can be used as a memory-mapped peripheral of the host. The HIP operates in parallel with and asynchronous to the ADSP-21xx's computational core and internal memory. The host interface port consists of registers through which the ADSP-21xx and the host processor pass data and status information. The HIP can be configured for: an 8-bit data bus or 16-bit data bus; a multiplexed address/data bus or separate address and data buses; and separate read and write strobes or a read/write strobe and a data strobe.

1.3.4 DMA Ports (ADSP-2181)

The ADSP-2181 contains two DMA ports, and Internal DMA Port and a Byte DMA Port. The IDMA port provides an efficient means of communication between a host system and the DSP. The port is used to access the on-chip program memory and data memory of the DSP with only one cycle per word of overhead. The IDMA port has a 16-bit multiplexed address and data bus and supports 24-bit program memory. The IDMA port is completely asynchronous and can be written to while the ADSP-2181 is operating at full speed.

The internal memory address is latched and then automatically incremented after each IDMA transaction. An external device can therefore access a block of sequentially addressed memory by specifying only the starting address of the block.

The byte memory DMA controller allows loading and storing of program instructions and data using the byte memory space. The BDMA circuit is able to access the byte memory space while the processor is operating normally and steals only one processor cycle per 8-, 16- or 24-bit word transferred.

1 Introduction

1.3.5 Analog Interface

The analog interface of the ADSP-21msp58/59 consists of input amplifiers and a 16-bit sigma-delta analog-to-digital converter (ADC) as well as a sigma-delta digital-to-analog converter (DAC) and a differential output amplifier.

1.4 ADSP-2100 FAMILY DEVELOPMENT TOOLS

The ADSP-2100 family is supported with a complete set of software and hardware development tools. The ADSP-2100 Family Development System includes software utilities for program development and EZ Tools for hardware/software debugging.

The Development Software includes:

- *System Builder*—The System Builder defines the architecture of your hardware system. This includes the specification of the amount of external memory available and any memory-mapped I/O ports.

- *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 programmer compatible files.

- *C Compiler*—The C Compiler reads ANSI C source and outputs ADSP-2100 family source code ready to be assembled. It also supports inline assembler code.

The EZ-ICE® emulators provide hardware-based debugging of ADSP-21xx systems. The EZ-ICEs perform stand-alone, in-circuit emulation with little or no degradation in processor performance.

The EZ-LAB® evaluation boards are low-cost, basic hardware platforms for running example applications.

For additional information on the development tools, refer to the *ADSP-2100 Family Development Tools Data Sheet.*

1.5 ORGANIZATION OF THIS MANUAL

This manual is organized as follows.

Chapters 2, 3, and 4 describe the core architectural features shared by all members of the ADSP-2100 family:

- Chapter 2, "Computational Units," describes the functions and internal organization of the arithmetic/logic unit (ALU), the multiplier/accumulator (MAC), and the barrel shifter.
- Chapter 3, "Program Control," describes the program sequencer, interrupt controller and status and condition logic.
- Chapter 4, "Data Transfer," describes the data address generators (DAGs) and the PMD-DMD bus exchange unit.

Chapters 5, 6, 7, and 8 describe the additional functional units included in different members of the ADSP-2100 family. (See Table 1.1 for a list of the functions included in each device.)

- Chapter 5, "Serial Ports," describes the serial ports, SPORT0 and SPORT1.
- Chapter 6, "Timer," explains the programmable interval timer.
- Chapter 7, "Host Interface Port," describes the operation of the host interface port, including boot loading and software reset.
- Chapter 8, "Analog Interface," describes the operation and the internal architecture of the ADSP-21msp58/59's analog interface.

Chapters 9 and 10 describe the behavior of the ADSP-21xx processors from the point of view of external memory and control logic:

- Chapter 9, "System Interface," discusses the issue of system clocking, and describes the processors' control interface, the software reboot function, and the powerdown mode.

- Chapter 10, “Memory Interface,” describes the data and program memory spaces. This chapter describes both internal and external memory, including the use of boot memory space. A special section is devoted to the ADSP-2181, since its memory interface differs from that of the other family processors.

Chapter 11, “DMA Ports,” describes the operation of the ADSP-2181’s IDMA and BDMA features.

Chapter 12, “Programming Model,” gives a functional description of the processor resources—such as registers—as they appear in software.

Chapter 13, “Hardware Examples,” gives examples of system designs using the ADSP-21xx processors. Each example illustrates the solution to a different system design issue, using block diagrams, explanatory text, and programs or timing diagrams as needed.

Chapter 14, “Software Examples,” provides illustrative code for some important DSP and numerical algorithms.

Chapter 15, “Instruction Set Reference,” provides a detailed description of each ADSP-21xx instruction.

The Appendices provide reference material and further details on specific issues:

- Appendix A, “Instruction Coding,” gives the complete set of opcodes and specifies the bit patterns for choices within each field of the instruction word.

- Appendix B, “Division Exceptions,” describes signed and unsigned division.

- Appendix C, “Numeric Formats,” describes the fixed-point numerical formats directly supported by the ADSP-2100 family, discusses block floating-point arithmetic, and tells how to handle the results of multiplication for operands of various formats.

- Appendix D, “Interrupt Vector Addresses,” lists the interrupt vectors of each family processor.

- Appendix E, “Control/Status Registers,” summarizes the processors’ control and status registers.

Computational Units 2

2.1 OVERVIEW

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

Every device in the ADSP-2100 family is a 16-bit, fixed-point machine. Most operations assume a twos-complement number representation, while others assume unsigned numbers or simple binary strings. Special features support multiword arithmetic and block floating-point. Details concerning the various number formats supported by the ADSP-2100 family are given in Appendix C.

In ADSP-2100 family arithmetic, signed numbers are always in twos-complement format. The processors do not use signed-magnitude, ones-complement, BCD or excess-n formats.

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-2100 family arithmetic, “signed” refers to twos-complement. Most ADSP-2100 family operations presume or support twos-complement arithmetic. The ADSP-2100 family does not use signed-magnitude, ones-complement, BCD or excess-n formats.

2 Computational Units

2.1.4 Fractional Representation: 1.15

ADSP-2100 family arithmetic is optimized for numerical values in a fractional binary format denoted by 1.15 (“one dot fifteen”). In the 1.15 format, there is one sign bit (the MSB) and fifteen fractional bits representing values from –1 up to one LSB less than +1.

Figure 2.1 shows the bit weighting for 1.15 numbers. Below are examples of 1.15 numbers and their decimal equivalents.

1.15 Number	*Decimal Equivalent*
0x0001	0.000031
0x7FFF	0.999969
0xFFFF	–0.000031
0x8000	–1.000000

-2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

Figure 2.1 Bit Weighting For 1.15 Numbers

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 arithmetic. 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 arithmetic. 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 a rounding operation). 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-2100 family 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 the processor multiplies two 1.15 operands, the result is a 2.30 (2 sign bits, 30 fractional bits) number. In the fractional mode, the MAC automatically shifts the multiplier product (P) left one bit before transferring the result to the multiplier result register (MR). This shift causes the multiplier result to be in 1.31 format, which can be rounded to 1.15 format. Figure 2.7, 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 format. A left shift is not needed; it would change the numerical representation. Figure 2.8 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 Computational Units

2.1.8 Summary

Table 2.1 summarizes some of the arithmetic characteristics of ADSP-2100 family operations. In addition to the numeric types described in this section, the ADSP-2100 Family 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-2100 Family C Tools Manual*.

Operation	*Arithmetic Formats*	
ALU	*Operands*	*Result*
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		
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

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 2.2, on the following page, 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. Please see the "Instruction Set Reference" chapter of this manual for information on how each instruction affects the ALU flags.

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.

2 Computational Units

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.

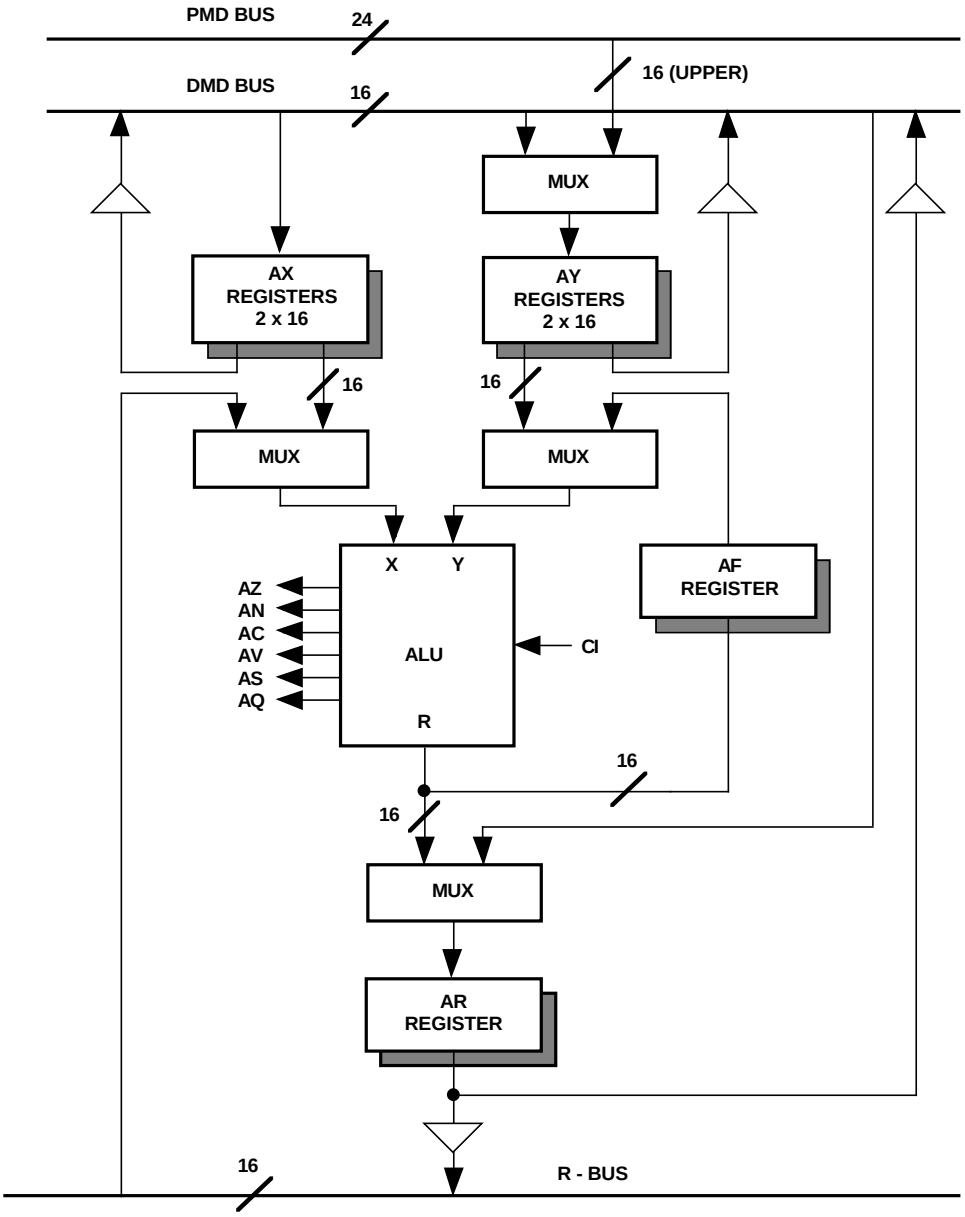


Figure 2.2 ALU Block Diagram

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 Chapter 15, "Instruction Set Reference" for an illustration of this same-cycle read and write.

The ALU contains a duplicate bank of registers, shown in Figure 2.2 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.

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 <u>*X input port*</u>	*Source for* <u>*Y input port*</u>	*Destination for* <u>*R output port*</u>
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 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" (+ C) operation is intended for adding the upper portions of multiprecision numbers. The "subtract with borrow" (C – 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 the maximum negative or positive 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 AR when 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 different from 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.

When the ALU saturation mode is used, only the AR register saturates; if the AF register is the destination, wrap-around will occur but the flags will reflect the saturated result.

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 supports division. The divide function is achieved with additional shift circuitry not shown in Figure 2.2. 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 2.3 (on the next page).

2 Computational Units

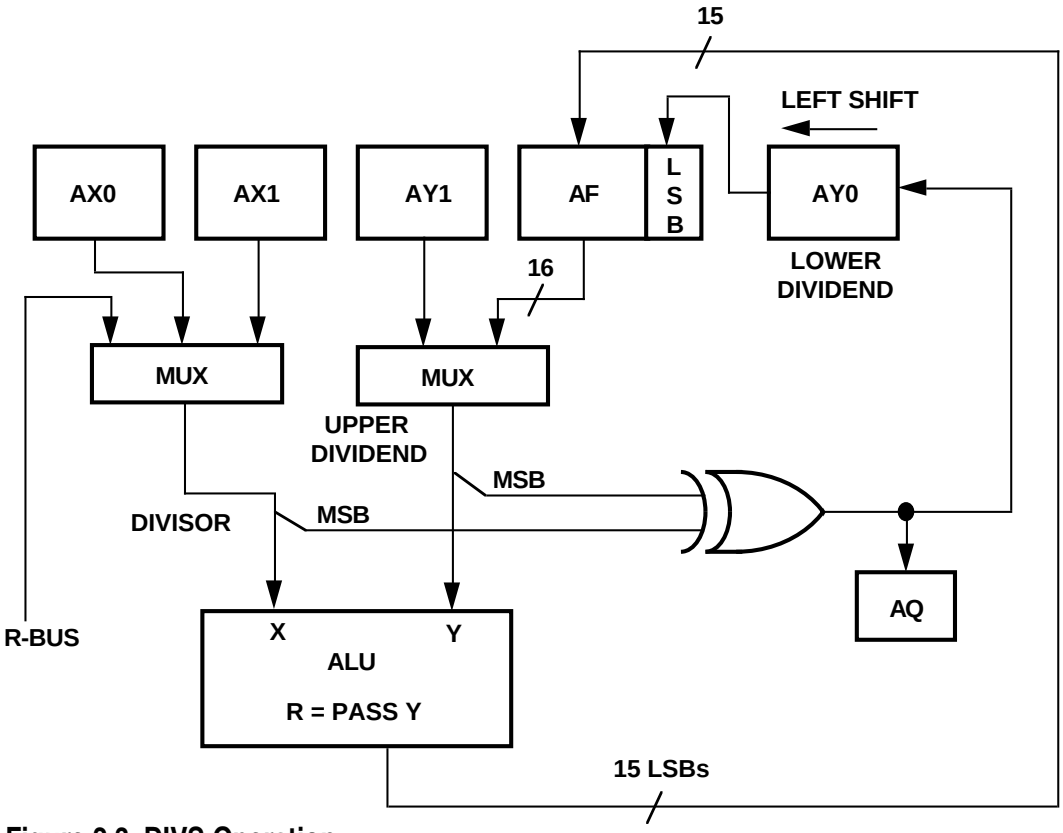


Figure 2.3 DIVS Operation

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.

The second division primitive is the "divide-quotient" (DIVQ) instruction 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 2.4.

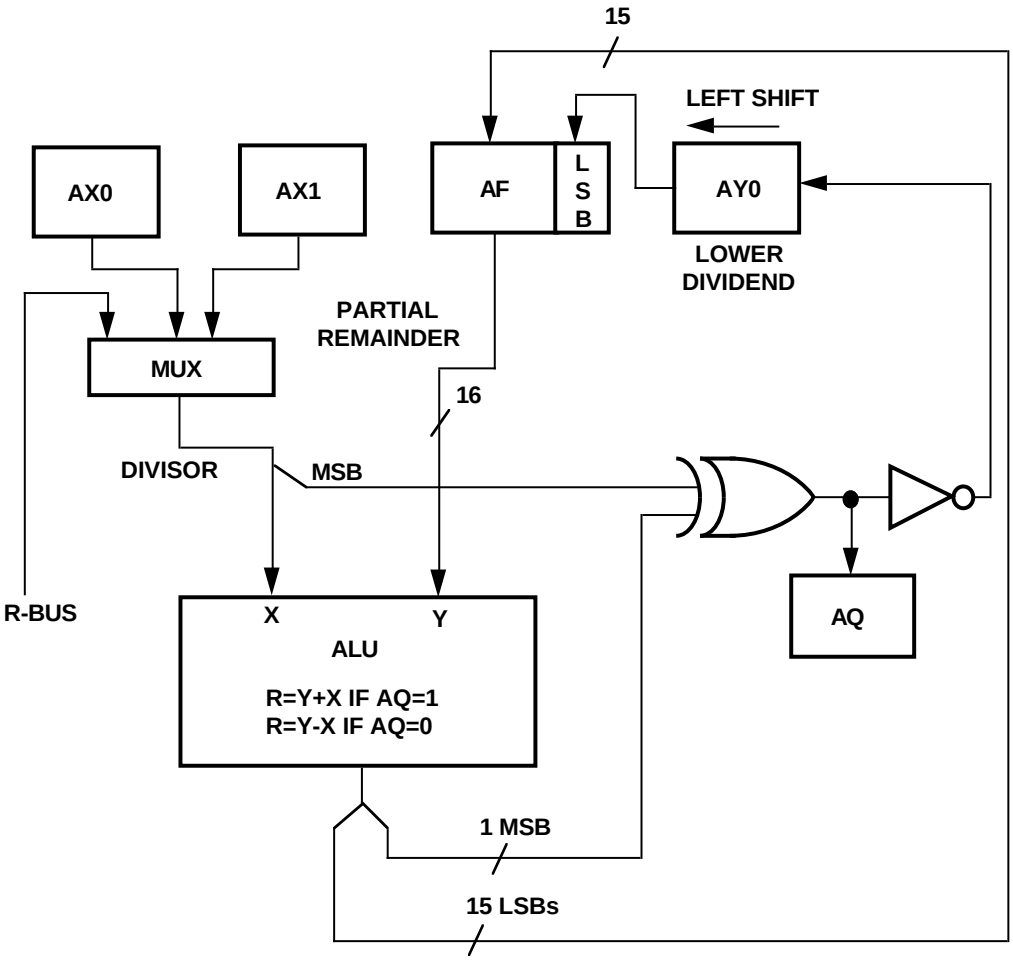


Figure 2.4 DIVQ Operation

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 handbook *Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 1.*

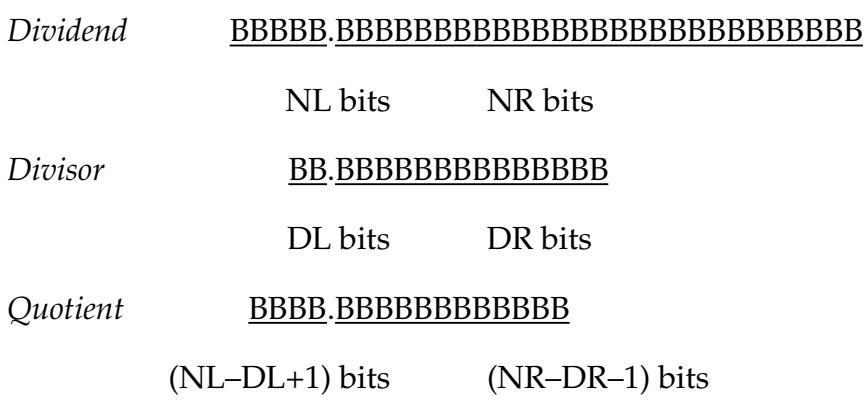


Figure 2.5 Quotient Format

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 the Program Control chapter.

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 2.6, on the following page, shows a block diagram of the multiplier/accumulator.

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/subtracter 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/subtracter is greater than 32 bits to allow for intermediate overflow in a series of multiply/accumulate operations. The multiply overflow (MV) status bit is set when the accumulator has overflowed beyond the 32-bit boundary, that is, when there are significant (non-sign) bits in the top nine bits of the MR register (based on twos-complement arithmetic).

2 Computational Units

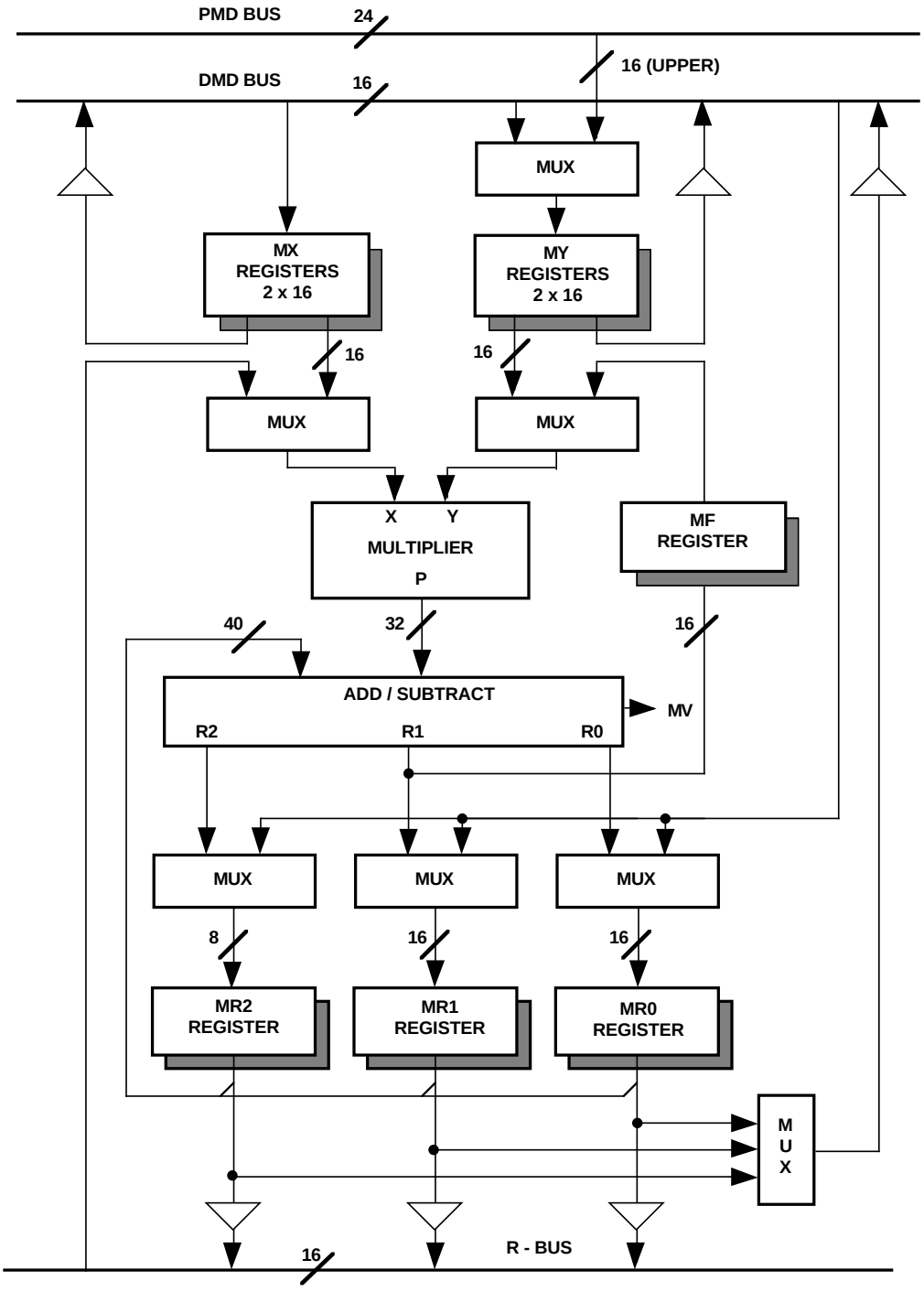


Figure 2.6 MAC Block Diagram

The input/output registers of the MAC 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 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/subtracter 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/subtracter 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 Chapter 15 "Instruction Set Reference" for an illustration of this same-cycle read and write.

2 Computational Units

The MAC contains a duplicate bank of registers, shown in Figure 2.6 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-2100 family provides two modes for the standard multiply/accumulate function: fractional mode for fractional numbers (1.15), and integer mode for integers (16.0).

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 2.7.

In the integer mode, the 32-bit P register is not shifted before being added to MR. Figure 2.8 shows the integer-mode result placement.

The mode is selected by bit 4 of the mode status register (MSTAT). If this bit is a 1, the integer mode is selected. Otherwise, the fractional mode is selected. In either mode, the multiplier output P is fed into a 40-bit adder/subtracter which adds or subtracts the new product with the current contents of the MR register to form the final 40-bit result R.

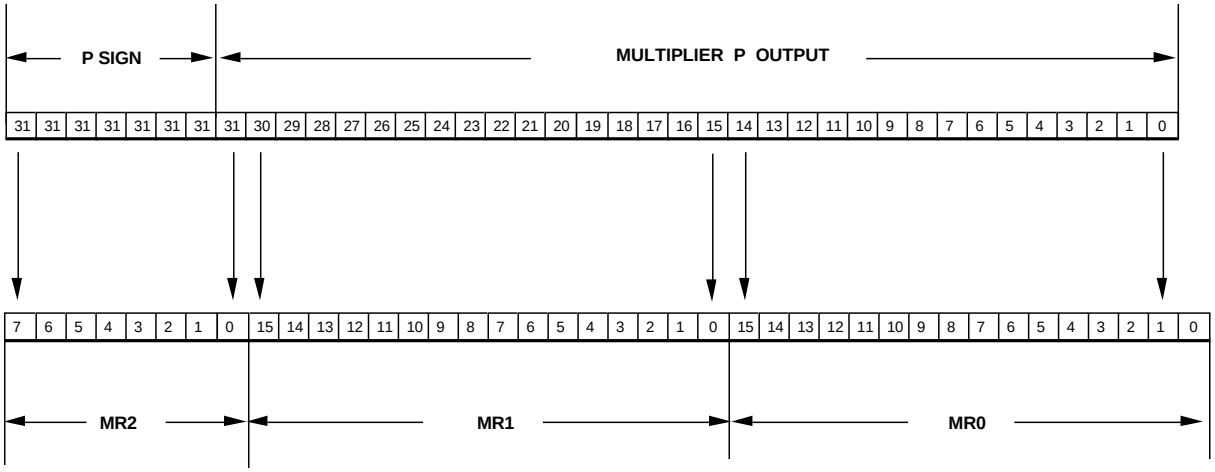


Figure 2.7 Fractional Multiplier Result Format

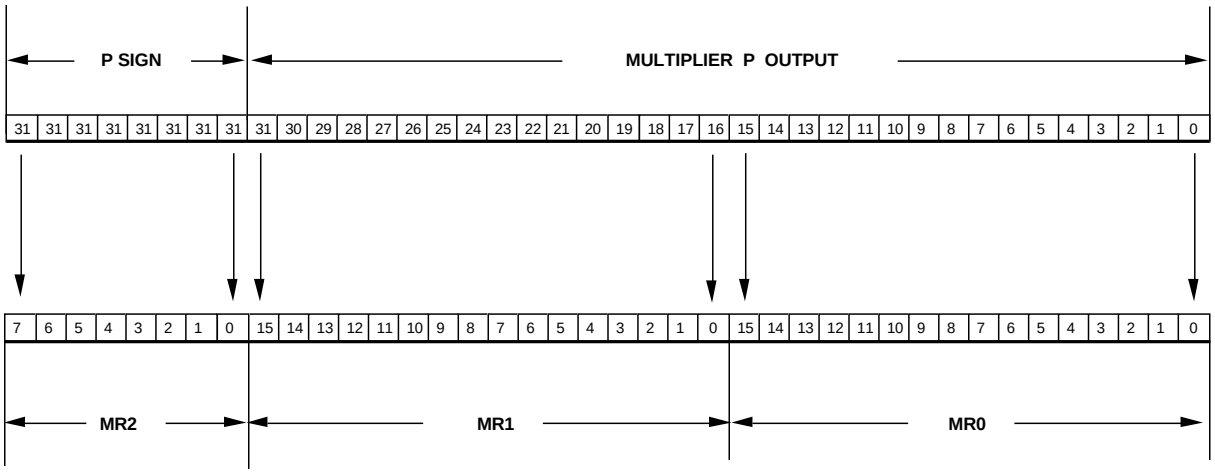


Figure 2.8 Integer Multiplier Result Format

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 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/subtracter 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 contents 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 Effect Of MAC Saturation Instruction

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 net positive bias since the midway value (when MR0=0x8000) is always rounded upward. The accumulator eliminates this bias by forcing bit 16 in the result output to zero when it detects this midway point. This has the effect of rounding odd MR1 values upward and even MR1 values downward, yielding a zero large-sample bias assuming uniformly distributed values.

Using x to represent any bit pattern (not all zeros), here are two examples of rounding. The first example is the typical rounding operation.

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 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.

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
	xxxxxxxx	xxxxxxxx01100111	0000000000000000
Since bit 16 = 1, force it to 0			
Rounded value:	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.

2.3.2.7 Biased Rounding (ADSP-217x, ADSP-218x, ADSP-21msp5x)

A mode is available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors to allow biased rounding in addition to the normal unbiased rounding. This mode is selected by the BIASRND bit (bit 12 of the SPORT0 Autobuffer Control register). When the BIASRND bit is set to 0, the normal unbiased rounding operations occur. When the BIASRND bit is set to 1, biased rounding occurs instead of the normal unbiased rounding. When operating in biased rounding mode all rounding operations with MR0 set to 0x8000 will round up, rather than only rounding odd MR1 values up. For example:

MR value before RND	*biased RND result*	*unbiased RND result*
00-0000-8000	00-0001-8000	00-0000-8000
00-0001-8000	00-0002-8000	00-0002-8000
00-0000-8001	00-0001-8001	00-0001-8001
00-0001-8001	00-0002-8001	00-0002-8001
00-0000-7FFF	00-0000-7FFF	00-0000-7FFF
00-0001-7FFF	00-0001-7FFF	00-0001-7FFF

This mode only has an effect when the MR0 register contains 0x8000; all other rounding operations work normally. This mode allows more efficient implementation of bit-specified algorithms that use biased rounding, for example the GSM speech compression routines. Unbiased rounding is preferred for most algorithms.

2 Computational Units

2.4 BARREL SHIFTER

The shifter 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 2.9 shows a block diagram of the shifter. The shifter can be divided into the following components: the shifter array, the OR/PASS logic, the exponent detector, and the exponent compare logic.

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 take 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.

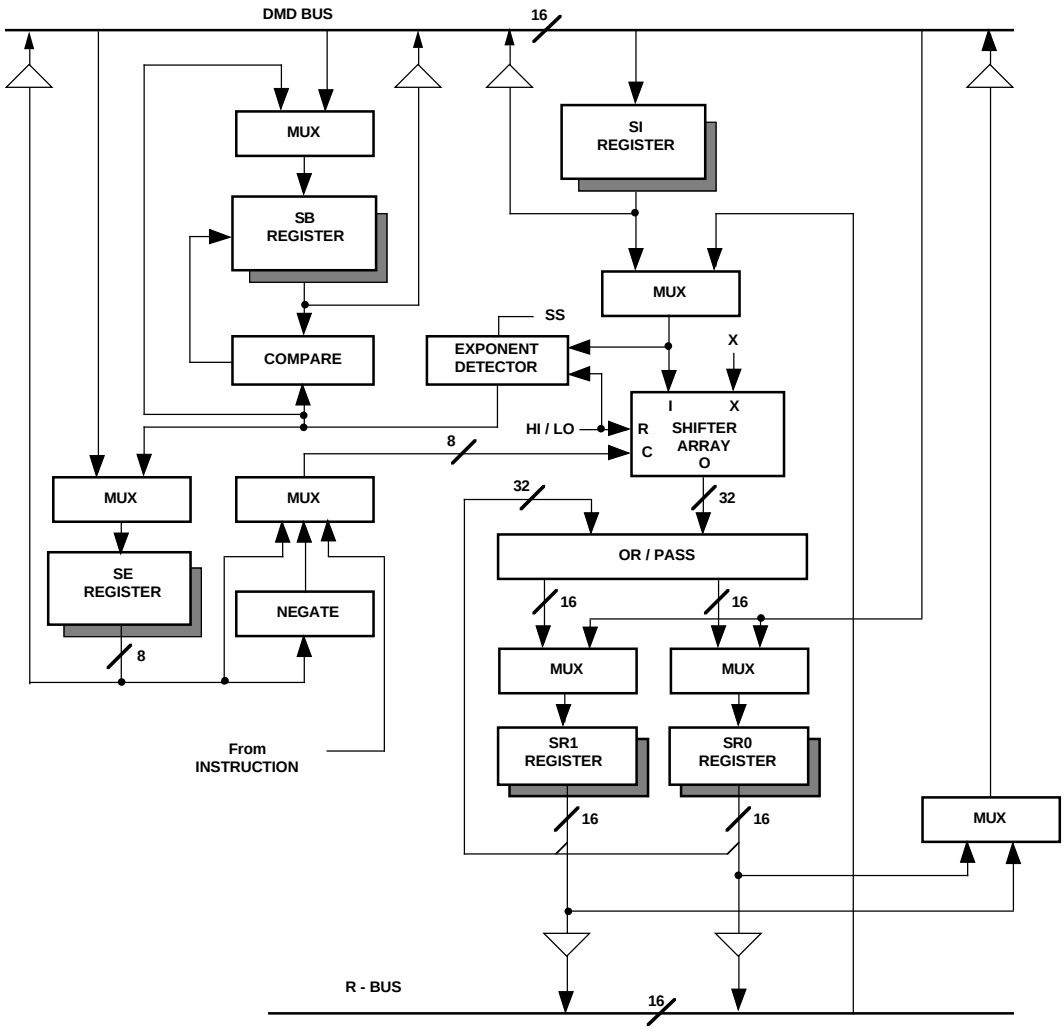


Figure 2.9 Shifter Block Diagram

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 Chapter 15, “Instruction Set Reference” for an illustration of this same-cycle read and write.

2 Computational Units

The shifter contains a duplicate bank of registers, shown in Figure 2.9 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 shows the shifter array output as a function of the control code and HI/LO signal.

The OR/PASS logic allows the shifted sections of a multiprecision number to be combined into a single quantity. In some shifter instructions, the shifted output may be logically ORed with the contents of the SR register; the shifter array is bitwise ORed with the current contents of the SR register before being loaded there. When the [SR OR] option is not used in the instruction, the shifter array output is passed through and loaded into the shifter result (SR) register unmodified.

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

ABCDEFGHIJKLMNPR represents the 16-bit input pattern

X stands for the extension bit

Table 2.4 Shifter Array Characteristic

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 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 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 shown in the following sections.

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

HI Mode

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

HIX Mode

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

LO Mode

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

Table 2.5 Exponent Detector Characteristics

2.4.2 Shifter Operations

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

- 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 [SR OR] and HI/LO modes to facilitate multiprecision operations. [SR OR] logically ORs the shift result with the current contents of SR. This option is used to join two 16-bit quantities into a 32-bit value in SR. When [SR OR] is not used, the shift value is passed through to SR directly. 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:

```
SI=0xB6A3;
SR=LSHIFT SI BY -5 (HI);
```

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:

```
SI=0xB6A3;
SR=LSHIFT SI BY 5 (LO);
```

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):

```
SI=0xB6A3;
SR=LSHIFT SI BY –5 (HI);

Input:       10110110 10100011

Shift value:  -5

SR:          00000101 10110101 00011000 00000000
```

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

```
SI=0xB6A3;
SR=ASHIFT SI BY –5 (HI);

Input:       10110110 10100011

Shift value:  -5

SR:          11111101 10110101 00011000 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.

2 Computational Units

Two examples of denormalizing a double-precision number are given 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 does not use the [SR OR] option.

Modifier = HI, No [SR OR], 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 [SR OR] option to avoid overwriting the previous half of the output value.

Modifier = LO, [SR 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 straight through to SR, but the second half is ORed to create a double-precision value in SR.

Modifier = LO, No [SR OR], Shift operation = Logical, SE = –3

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

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

Modifier = HI, [SR 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-point 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 it 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 also has the [SR OR] option. 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:

```
SE=EXP AR (HI);
```

Detects Exponent With 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 without the [SR OR] and the second half is normalized with [SR OR] 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, Modifier=HI, No [SR OR], SE = –3*

   First Input: `11110110 11010100`

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

4. *Normalize , Modifier=LO, [SR 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, Modifier=HI, No [SR OR], 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, Modifier=LO, [SR 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 whether or not 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.

Program Control 3

3.1 OVERVIEW

This chapter describes the program sequencer of the ADSP-2100 family processors. The program sequencer circuitry controls the flow of program execution. It contains an interrupt controller and status and condition logic.

3.2 PROGRAM SEQUENCER

The program sequencer generates a stream of instruction addresses and provides flexible control of program flow. It allows sequential instruction execution, zero-overhead looping, sophisticated interrupt servicing, and single-cycle branching with jumps and calls (both conditional and unconditional).

Figure 3.1, on the following page, shows a block diagram of the program sequencer. Each functional block of the sequencer is discussed is detail in this chapter.

This chapter discusses both program sequencer logic and the following instructions used to control program flow:

DO UNTIL
JUMP
CALL
RTS *(Return From Subroutine)*
RTI *(Return From Interrupt)*
IDLE

For a complete description of each instruction, refer to Chapter 15, Instruction Set Reference.

3 Program Control

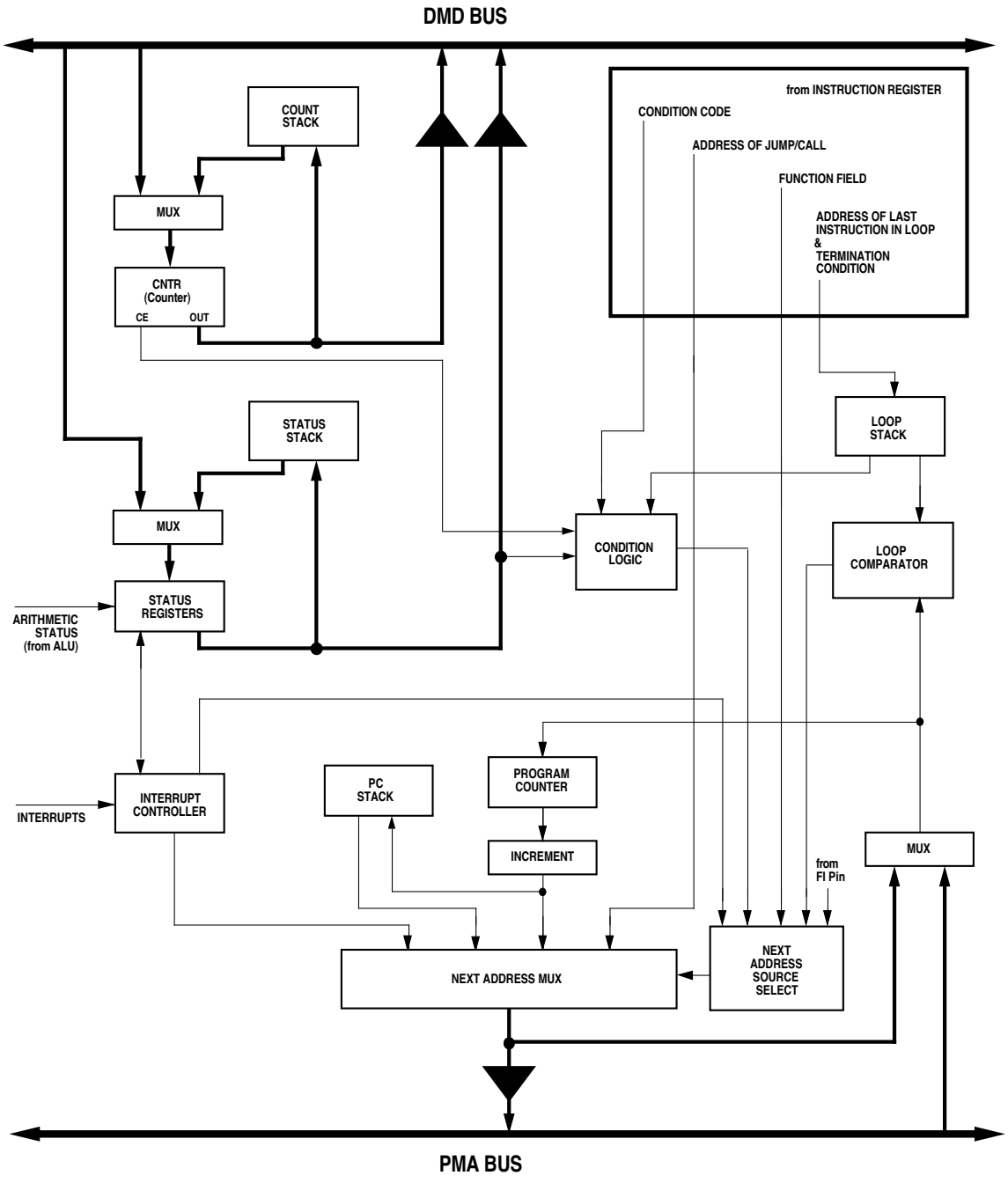


Figure 3.1 Program Sequencer Block Diagram

3.2.1 Next Address Select Logic

While the processor is executing an instruction, the program sequencer pre-fetches the next instruction. The sequencer's next address select logic generates a program memory address (for the pre-fetch) from one of four sources:

- PC incrementer
- PC stack
- instruction register
- interrupt controller

The next address circuit (shown in Figure 3.1) selects which of these sources is used, based on inputs from the instruction register, condition logic, loop comparator and interrupt controller. The next instruction address is then output on the PMA bus for the pre-fetch.

The PC incrementer is selected as the source of the next 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. The output of the PC incrementer is driven onto the PMA bus and is loaded back into the program counter to begin the next cycle.

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

The instruction register provides the next address when a direct jump is taken. The 14-bit jump address is embedded in the instruction word.

The interrupt controller provides the next program memory address when servicing an interrupt. Upon recognizing a valid interrupt, the processor jumps to the interrupt vector location corresponding to the active interrupt request.

Another possible source for the next address is one of the I4-I7 index registers of DAG2 (Data Address Generator 2), used when a register indirect jump is executed as in the following instruction:

```
JUMP (I4);
```

In this case the program counter (PC) is loaded from DAG2 via the PMA bus. (Data address generators are described in Chapter 4.)

3.2.2 Program Counter & PC 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 sequential instruction.

Associated with the PC is a 14-bit by 16-word stack that is pushed with the output of the incrementer when a CALL instruction is executed. The PC stack is also pushed when a 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 with the POP instruction.

A special instruction is provided for reading (and popping) or writing (and pushing) the top value of the PC stack. This instruction uses the pseudo register TOPPCSTACK, described at the end of this chapter.

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 directly from the PMA bus.

3.2.3 Loop Counter & Stack

The counter and count stack provide the program sequencer with a powerful looping mechanism. The counter is a 14-bit register with automatic post-decrement capability that controls 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 (CNTR register) is loaded with the desired loop count from the lower 14 bits of the DMD bus. The actual loop count *N* is loaded, as opposed to *N–1*. 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 of each processor cycle and the counter is decremented at the end; therefore CE is asserted when the counter reaches 1 so that the loop executes *N* times.

The counter may also be tested and automatically 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 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 indicates when the stack is empty.

The second exception is provided explicitly by the special purpose syntax OWRCNTR (overwrite counter). Writing a value to OWRCNTR overwrites the counter with the new value, and nothing is pushed onto the count stack. OWRCNTR cannot be read (i.e. used as a source register), and must not be written in the last instruction of a DO UNTIL loop.

3.2.4 Loop Comparator & Stack

The DO UNTIL instruction initiates a zero-overhead loop using the loop comparator and loop stack of the program sequencer.

On every processor cycle, the loop comparator compares the next address generated by the program sequencer to the address of the last instruction of the loop (which is embedded in the DO UNTIL instruction). The address of the first instruction in the loop is maintained on the 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 last instruction 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, an additional cycle is required for the initial loading of the counter. Table 3.1 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	
FOREVER	Always	

Table 3.1 DO UNTIL Termination Condition Logic

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 three possible results depending on the type of instruction at the end of the loop. Case 1 illustrates the most typical situation. Cases 2 and 3 are also allowed but involve greater program complexity for proper execution.

Case 1
If the last instruction in the loop is not a jump, call, return, or idle, the next address circuit will select the next address based on the termination condition stored on the top of the loop stack. If the condition is false, the top address on the PC stack is selected, causing a fetch of the first instruction 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 being used) are then popped.

(Note that conditional arithmetic instructions execute based on the condition explicitly stated in the instruction, whereas the loop sequencing is 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 do not take place: fetching the first instruction of the loop, falling out of the loop and popping the loop stack, PC stack, and counter stack, or decrementing the counter.

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

Case 3
If the last instruction in the loop is an IDLE, program flow is controlled by the IDLE instruction rather than the loop. When the IDLE instruction is executed, the processor enters a low-power wait-for-interrupt state. When the processor is interrupted, loop execution terminates and program execution continues with the first instruction following the loop.

Note: Caution is required when ending a loop with a JUMP, CALL, RETURN, or IDLE instruction, 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 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. A subroutine call poses this problem only when it is the last instruction in a loop; in such cases, the return causes program flow to transfer to the instruction just after the loop. Calls within a loop that are not the last instruction operate as in Case 1.

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.

3.3 PROGRAM CONTROL INSTRUCTIONS

The following sections describe the primary instructions used to control program flow.

3.3.1 JUMP Instruction

The 14-bit jump address is embedded in the JUMP instruction word. When a JUMP instruction is decoded, the jump address is input directly to the next address mux of the program sequencer. The address is driven onto the PMA bus and fed back to the PC for the next cycle. The following instruction, for example,

```
JUMP fir_start;
```

jumps to the address of the label `fir_start`.

3.3.1.1 Register Indirect JUMPs

In this case of register indirect jumps, the jump address is supplied by one of the I registers of DAG2 (I4, I5, I6, or I7). (Data address generators are described in Chapter 4.) The address is driven onto the PMA bus by DAG2, and is loaded into the PC on the next cycle. For example, the instruction

```
JUMP (I4);
```

will jump to the address contained in the I4 register.

3.3.2 CALL Instruction

The CALL instruction executes in a similar fashion as the JUMP instruction. The address of the subroutine is embedded in the CALL instruction word and, once extracted from the instruction register, is fed back the PC for the next cycle. In addition, the current value of the program counter is incremented and pushed onto the PC stack. Upon return from the subroutine, the PC stack is popped into the program counter and execution resumes with the instruction following the CALL.

3.3.3 DO UNTIL Loops

The most common form of a DO UNTIL loop uses the counter register (CNTR) as a loop iteration counter. When the counter is used to control loop iteration, CE (counter expired) must be used as the DO UNTIL termination condition. A simple example of this type of loop is as follows:

```
   L0=10;            {setup circular buffer length register}
   I0=^data_buffer;  {load pointer with first address of}
                     {circular buffer}
   M0=1;             {setup modify register for pointer increment}
   CNTR=10;          {load counter with circular buffer length}

   DO loop UNTIL CE; {repeat loop until counter expired}
      DM(I0,M0)=0;   {initialize/clear circular buffer}
      ...any instruction...
loop: ...any instruction...
```

When the

```
CNTR=10;
```

instruction is executed, prior to entering the loop, the counter is loaded via the DMD bus. Any previously existing count would be simultaneously pushed onto the count stack; this push operation is omitted if the counter is empty. The

```
DO loop UNTIL CE;
```

instruction itself only sets up the conditions for looping; no other operation occurs while the instruction is executed. This occurs only once, at the beginning of the first time through the loop.

Execution of the DO UNTIL instruction pushes the address of the instruction immediately following the DO UNTIL onto the PC stack (by pushing the incremented PC). On the same cycle, the loop stack is pushed with the address of the end-of-loop instruction and the termination condition.

As execution continues within the loop, the loop comparator checks each instruction's address against the address of the loop's last instruction. Until that address is reached, normal execution continues.

Each time the end of the loop is reached, the loop comparator determines that the currently executing instruction is the last in the loop. This affects the next address select logic of the program sequencer: instead of using the incremented PC for the next address, the loop termination condition is evaluated. If the termination condition is false, execution continues with the first instruction of the loop (the top of the PC stack is taken as the next address). Note that the PC and loop stacks are not popped, only read.

On the final pass through the loop, the termination condition is true. The PC stack is popped and execution continues with the instruction immediately following the last instruction of the loop. The loop stack and count stack are also popped on this cycle.

3.3.4 IDLE Instruction

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

3.3.4.1 Slow IDLE

An enhanced version of the IDLE intruction allows the processor's internal clock signal to be slowed, further reducing power consumption. The reduced clock frequency, a programmable fraction of the normal clock rate, is specified by a selectable divisor given in the IDLE instruction. The format of the instruction is

```
IDLE (n);
```

where n = 16, 32, 64, or 128. This instruction keeps the processor fully functional, but operating at the slower clock rate. While it is in this state, the processor's other internal clock signals, such as SCLK, CLKOUT, and timer clock, are reduced by the same ratio. The default form of the instruction, when no clock divisor is given, is the standard IDLE instruction.

When the IDLE (n) instruction is used, it effectively slows down the processor's internal clock and thus its response time to incoming interrupts. The one-cycle interrupt response time of the standard idle state is increased by *n*, the clock divisor. When an enabled interrupt is received, the processor will remain in the idle state for up to a maximum of *n* processor cycles before resuming normal operation (*n = 16, 32, 64, or 128*).

When the IDLE (n) instruction is used in systems that have an externally generated serial clock (SCLK), the serial clock rate may be faster than the processor's reduced internal clock rate. Under these conditions, interrupts must not be generated at a faster rate than can be serviced, due to the additional time the processor takes to come out of the idle state (a maximum of *n* processor cycles).

3.4 INTERRUPTS

The program sequencer's interrupt controller responds to interrupts by shifting control to the instruction located at the appropriate interrupt vector address. Tables 3.2–3.7 show the interrupts and associated vector addresses for each processor of the ADSP-2100 family. (Note that SPORT1 can be configured as either a serial port or as a collection of control pins including two external interrupt inputs, $\overline{\text{IRQ0}}$ and $\overline{\text{IRQ1}}$. See Chapter 5, "Serial Ports," for more information about the configuration of SPORT1.)

The interrupt vector locations are spaced four program memory locations apart—this allows short interrupt service routines to be coded in place, with no jump to the service routine required. For interrupt service routines with more than four instructions, however, program control must be transferred to the service routine by means of a jump instruction placed at the interrupt vector location.

After an interrupt has been serviced, an RTI (Return From Interrupt) instruction returns control to the main program by popping the top value on the PC stack into the PC; the status stack is also popped to restore the previous processor state.

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

3 Program Control

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.

The ADSP-2100 family processors include a secondary register set which can be used to provide a fresh set of ALU, MAC, and Shifter registers during interrupt servicing. This feature allows single-cycle context switching. Use of the secondary registers is described in the "Mode Status Register (MSTAT)" section of this chapter.

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup	0x0000
$\overline{\text{IRQ2}}$	0x0004 *(highest priority)*
SPORT0 Transmit	0x0008
SPORT0 Receive	0x000C
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0010
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0014
Timer	0x0018 *(lowest priority)*

Table 3.2 ADSP-2101/2115 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup	0x0000
$\overline{\text{IRQ2}}$	0x0004 *(highest priority)*
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0010
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0014
Timer	0x0018 *(lowest priority)*

Table 3.3 ADSP-2105 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup	0x0000
$\overline{\text{IRQ2}}$	0x0004 *(highest priority)*
HIP Write (from Host)	0x0008
HIP Read (to Host)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0018
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x001C
Timer	0x0020 *(lowest priority)*

Table 3.4 ADSP-2111 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup (or powerup w/PUCR=1)	0x0000 *(highest priority)*
Powerdown (non-maskable)	0x002C
$\overline{\text{IRQ2}}$	0x0004
HIP Write (from Host)	0x0008
HIP Read (to Host)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
Software Interrupt 1	0x0018
Software Interrupt 2	0x001C
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0020
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0024
Timer	0x0028 *(lowest priority)*

Table 3.5 ADSP-2171 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup (or powerup w/PUCR=1)	0x0000 *(highest priority)*
Powerdown (non-maskable)	0x002C
$\overline{\text{IRQ2}}$	0x0004
$\overline{\text{IRQL1}}$ (level-sensitive)	0x0008
$\overline{\text{IRQL0}}$ (level-sensitive)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
$\overline{\text{IRQE}}$ (edge-sensitive)	0x0018
Byte DMA Interrupt	0x001C
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0020
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0024
Timer	0x0028 *(lowest priority)*

Table 3.6 ADSP-2181 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup (or powerup w/PUCR=1)	0x0000 *(highest priority)*
Powerdown (non-maskable)	0x002C
$\overline{\text{IRQ2}}$	0x0004
HIP Write (from Host)	0x0008
HIP Read (to Host)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
Analog (DAC) Transmit	0x0018
Analog (ADC) Receive	0x001C
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0020
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0024
Timer	0x0028 *(lowest priority)*

Table 3.7 ADSP-21msp58/59 Interrupts & Interrupt Vector Addresses

3.4.1 Interrupt Servicing Sequence

When an interrupt request occurs, it is latched while the processor finishes executing the current instruction. The interrupt request is then compared with the interrupt mask register, IMASK, by the interrupt controller.

If the interrupt is not masked, the program sequencer pushes the current value of the program counter (which contains the address of the next instruction) onto the PC stack—this allows execution to continue, after the interrupt is serviced, with the next instruction of the main program. The program sequencer also pushes the current values of the ASTAT, MSTAT, and IMASK registers onto the status stack. ASTAT, MSTAT and IMASK are stored in this order, with the MSB of ASTAT first, and so on. When IMASK is pushed, it is automatically reloaded with a new value that determines whether or not interrupt nesting is allowed (based on the value of the interrupt nesting enable bit in ICNTL).

The processor then executes a NOP while simultaneously fetching the instruction located at the interrupt vector address. Upon return from the interrupt service routine, the PC and status stacks are popped and execution resumes with the next instruction of the main program.

3.4.2 Configuring Interrupts

The following registers are used to configure interrupts:

- ICNTL—Determines whether interrupts can be nested and configures the external interrupts $\overline{\text{IRQ2}}$, $\overline{\text{IRQ1}}$, $\overline{\text{IRQ0}}$ as edge-sensitive or level-sensitive
- IMASK—Enables or disables (i.e. masks) each individual interrupt (both external and internal).
- IFC—Forces an interrupt or clears a pending edge-sensitive interrupt.

The $\overline{\text{IRQ2}}$, $\overline{\text{IRQ1}}$, $\overline{\text{IRQ0}}$ interrupts may be either edge-sensitive or level-sensitive, as selected in the ICNTL register. The ADSP-2181 has three additional interrupt pins: $\overline{\text{IRQE}}$, $\overline{\text{IRQL1}}$, and $\overline{\text{IRQL2}}$. The $\overline{\text{IRQE}}$ input is edge-sensitive, while the $\overline{\text{IRQL1}}$ and $\overline{\text{IRQL2}}$ inputs are level-sensitive.

For edge-sensitive $\overline{\text{IRQx}}$ interrupts, an interrupt request is latched internally whenever a falling edge (high-to-low transition) occurs at the input pin. The latch remains set until the interrupt is serviced; it is then automatically cleared. A pending edge-sensitive interrupt can also be cleared in software by setting the corresponding clear bit in the IFC register.

Edge-sensitive interrupt inputs generally require less external hardware than level-sensitive inputs, and allow signals such as sampling-rate clocks to be used as interrupts.

A level-sensitive interrupt must remain asserted until the interrupt is serviced. The interrupting device must then deassert the interrupt request so that the interrupt is not serviced again. Level-sensitive inputs, however, allow many interrupt sources to use the same input by combining them logically to produce a single interrupt request. Level-sensitive interrupts are not latched.

Your program can also determine whether or not interrupts can be nested. In non-nesting mode, all interrupt requests are automatically masked out when an interrupt service routine is entered. In nesting mode, the processor allows higher-priority interrupts to be recognized and serviced.

There are two levels of masking for the Host Interface Port (HIP) interrupts of the ADSP-2111, ADSP-2171, and ADSP-21msp58/59. The memory-mapped HMASK register configures masking out the generation of individual read or write interrupts for each HIP data register. The IMASK register can be set to mask or enable the servicing of all HIP read interrupts or all HIP write interrupts. Both IMASK and HMASK must be set for HDR interrupts. See Chapter 7, "Host Interface Port," for details.

3.4.2.1 Interrupt Control Register (ICNTL)

ICNTL is a 5-bit register that configures the external interrupt requests ($\overline{\text{IRQx}}$) of each processor. All bits in ICNTL are undefined after a processor reset. The bit definitions for each processor's ICNTL register are given in Appendix E, "Control/Status Registers."

ICNTL contains an $\overline{\text{IRQx}}$ sensitivity bit for each external interrupt. The sensitivity bits determine whether a given interrupt input is edge- or level-sensitive (0 = level-sensitive, 1 = edge-sensitive). There are no sensitivity bits for internally generated interrupts.

The interrupt nesting enable bit (bit 4) in ICNTL determines whether nesting of interrupt service routines is allowed.

When the value of ICNTL is changed, there is a one cycle latency before the change in interrupt configuration.

3.4.2.2 Interrupt Mask Register (IMASK)

Each bit of the IMASK register enables or disables the servicing of an individual interrupt. Specific bit definitions for each processor's IMASK register are given in Appendix E, "Control/Status Registers." The mask bits are positive sense: 0=masked, 1=enabled. IMASK is set to zero upon a processor reset.

On the ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors, all interrupts are automatically disabled for one instruction cycle following the execution of an instruction that modifies IMASK. This does not affect serial port autobuffering or DMA transfers.

If an edge-sensitive interrupt request signal occurs when the interrupt is masked, the request is latched but not serviced; the interrupt can then be recognized in software and serviced later.

The contents of IMASK 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 the interrupt nesting enable bit (bit 4) of ICNTL; it may be altered, though, 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 shown graphically, for the ADSP-2101, in Table 3.8.

The interrupt nesting enable bit (in ICNTL) determines the state of IMASK upon entering the interrupt, as shown in Table 3.8.

ICNTL Interrupt Nesting Enable bit = 0 (nesting disabled)

Interrupt level serviced	*IMASK contents before (pushed on stack)*	*IMASK contents entering interrupt service routine*
0 (low)	`ijklmn`	`000000`
1	`ijklmn`	`000000`
2	`ijklmn`	`000000`
3	`ijklmn`	`000000`
4	`ijklmn`	`000000`
5 (high)	`ijklmn`	`000000`

ICNTL Interrupt Nesting Enable bit = 1 (nesting enabled)

Interrupt level serviced	*IMASK contents before (pushed on stack)*	*IMASK contents entering interrupt service routine*
0 (low)	`ijklmn`	`ijklm0`
1	`ijklmn`	`ijkl00`
2	`ijklmn`	`ijk000`
3	`ijklmn`	`ij0000`
4	`ijklmn`	`i00000`
5 (high)	`ijklmn`	`000000`

("ijklmn" represents any pattern of ones and zeroes)

Table 3.8 IMASK Entering Interrupt Service Routines (ADSP-2101 example)

3.4.2.3 Global Enable/Disable for Interrupts

Global interrupt enable and disable instructions are available on the ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors:

```
ENA INTS;
DIS INTS;
```

Interrupts are enabled by default after reset. The `DIS INTS` instruction causes all interrupts (including powerdown) to be masked out regardless of the contents of IMASK. The `ENA INTS` instruction allows all unmasked interrupts to be serviced again.

Disabling interrupts does not affect serial port autobuffering.

3.4.2.4 Interrupt Force & Clear Register (IFC)

IFC is a write-only register that allows the forcing and clearing of edge-sensitive interrupts in software. An interrupt is forced or cleared under program control by setting the force or clear bit corresponding to the desired interrupt. After the force or clear bit is set, there is one cycle of latency before the interrupt is actually forced or cleared (except for the timer interrupt on the ADSP-2101/2105/2111/2115 processors).

Edge-sensitive interrupts can be forced by setting the appropriate force bit in IFC. This causes the interrupt to be serviced once, unless masked. An external interrupt must be edge-sensitive (as determined by ICNTL) to be forced. The timer, SPORT, and analog ADC/DAC interrupts also behave like edge-sensitive interrupts and can be masked, cleared and forced.

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

Specific bit definitions for each processor's IFC register are given in Appendix E, "Control/Status Registers." The IFC registers of the ADSP-2111, ADSP-2171, and ADSP-21msp58 processors do not include force/clear bits for Host Interface Port interrupts; HIP interrupts cannot be forced or cleared in software.

3.4.3 Interrupt Latency

For the timer, $\overline{\text{IRQ}}$x, SPORT, HIP, and analog interface interrupts, the latency from when an interrupt occurs to when the first instruction of the service routine is executed is at least three full cycles. This is shown in Figure 3.2. Two cycles are required to synchronize the interrupt internally, assuming that setup and hold times are met (for the $\overline{\text{IRQ}}$x input pins).

Since interrupts are only serviced on instruction boundaries, the instruction(s) executed during these two cycles must be fully completed, including any extra cycles inserted due to Bus Request/Bus Grant or memory wait states, before execution continues.

The third cycle of latency is needed to fetch the first instruction stored at the interrupt vector location. During this cycle, the processor executes a NOP instead of the instruction that would normally have been executed. On the next cycle, execution continues at the first instruction of the interrupt service routine. The address of the aborted instruction is pushed onto the PC stack; it will be fetched when the interrupt service routine is completed.

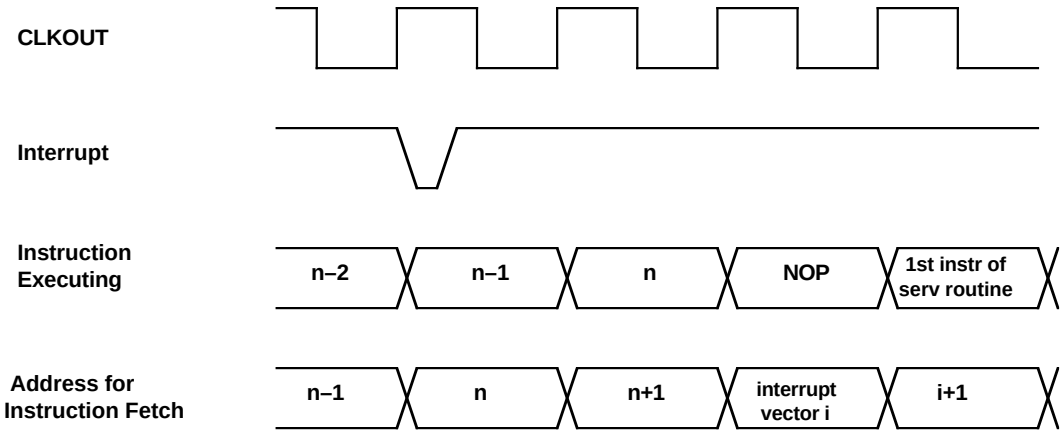


Figure 3.2 Interrupt Latency (Timer, $\overline{IRQx}$, SPORT, HIP, & Analog Interrupts)

(Note that this latency for the timer interrupt only applies for the ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors. See the next section for a description of timer interrupt latency on the ADSP-2101, ADSP-2105, ADSP-2115, ADSP-2111.)

For a pending interrupt that is masked, the latency from execution of the instruction that unmasks the interrupt (in IMASK) to the first instruction of the service routine is one cycle. This one-cycle latency is similar to that shown in Figure 3.3 for the timer interrupt of the ADSP-2101/2105/2111/2115, with the "n" instruction executing being the instruction that writes to IMASK (to unmask the interrupt).

3.4.3.1 Timer Interrupt Latency on ADSP-2101, ADSP-2105, ADSP-2115, ADSP-2111

For the timer interrupt on these processors, the latency from when the interrupt occurs to when the first instruction of the service routine is executed is only one cycle. This is shown in Figure 3.3. The single cycle of latency is needed to fetch the instruction stored at the interrupt vector location.

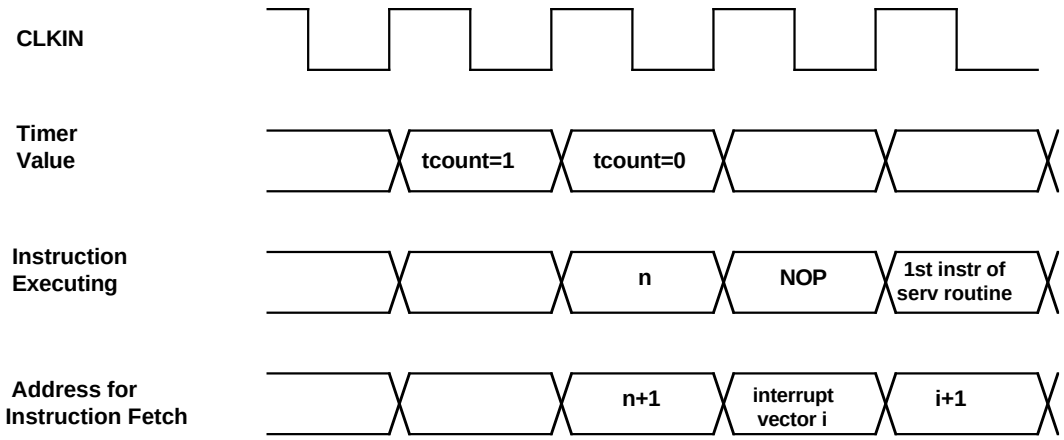


Figure 3.3 Timer Interrupt Latency for ADSP-2101, ADSP-2105, ADSP-2115, ADSP-2111

3 Program Control

3.5 STATUS REGISTERS & STATUS STACK

Processor status and mode bits are maintained in internal registers which can be independently read and written over the DMD bus. These registers are:

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

The interrupt-configuring status registers are described in the previous section. ASTAT, SSTAT, and MSTAT are discussed in the following sections.

The current ASTAT, MSTAT, and IMASK values are pushed onto the status stack when the processor responds to an interrupt; they are popped upon return from the interrupt service routine (with the RTI instruction). The depth of the stack varies from processor to processor. In each case, sufficient stack depth is provided to accommodate nesting of all interrupts.

3.5.1 Arithmetic Status Register (ASTAT)

ASTAT is eight bits wide and holds the status information generated by the computational blocks of the processor. The individual bits of ASTAT are defined as shown in Figure 3.4. The bits which express a particular condition (AZ, AN, AV, AC, MV) are all positive sense (1=true, 0=false).

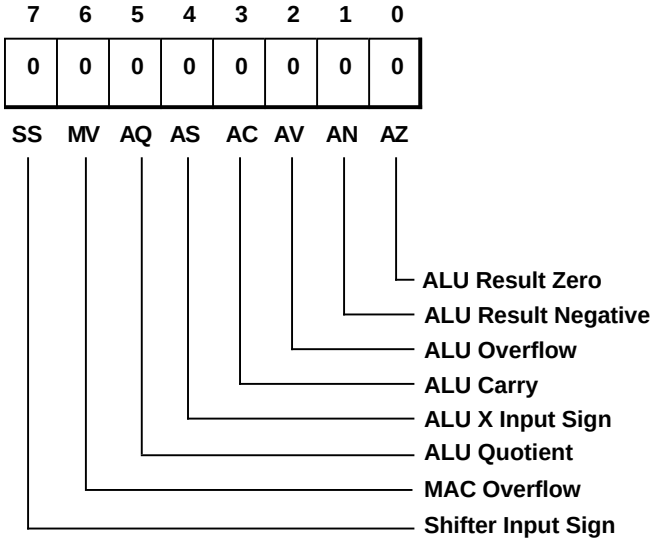


Figure 3.4 ASTAT Register

Each of the bits is automatically updated when a new status is generated by an arithmetic instruction. 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 (SAT MR)
SS	Shifter EXP operation

Arithmetic status is latched into ASTAT at the end of the cycle in which it was generated, and 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.

3.5.2 Stack Status Register (SSTAT)

The SSTAT register is eight bits wide and holds information about the four processor stacks. The individual bits of SSTAT are defined as shown in Figure 3.5. All of the bits are positive sense (1=true, 0=false).

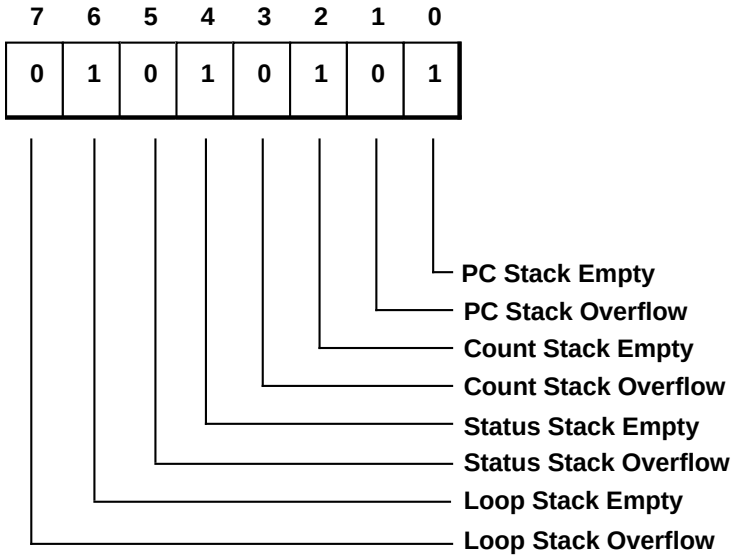


Figure 3.5 SSTAT Register (Read-Only)

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 that have occurred since the last processor 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 total depth of the stack. When this occurs, the values most recently pushed will be missing from the stack—older stack values are considered more important than new.

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. In this situation, then, it is possible to have both the stack empty and stack overflow bits set for a given stack.

Assume, for example, that the four-location count stack is overflowed by five successive pushes. Five successive pops will restore the stack empty condition, but will not clear the overflow condition. The processor must be reset to clear the stack overflow status.

3.5.3 Mode Status Register (MSTAT)

The MSTAT register determines the operating mode of the processor. The individual bits of MSTAT are defined as shown in Figure 3.6.

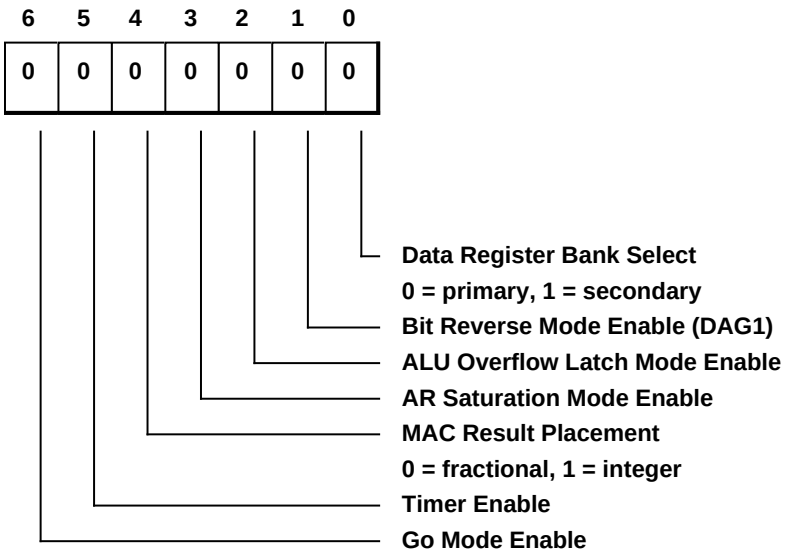


Figure 3.6 MSTAT Register

MSTAT can be modified by writing a new value to it with a MOVE instruction. Unlike the other status registers, MSTAT can also be altered with the Mode Control instruction (ENA, DIS). The Mode Control instruction provides a high-level, self-documenting method of configuring the processors' operating modes. Refer to the description of the Mode Control instruction in Chapter 15, "Instruction Set Reference," for further details.

To enable the bit reverse mode, for example, the following instruction could be used:

```
ENA BIT_REV;
```

The bit-reverse mode, when enabled, bitwise reverses all addresses generated by data address generator 1 (DAG1). This is useful for reordering the input or output data of an FFT algorithm.

The ADSP-2100 family processors include a secondary register set which can be used to provide a fresh set of ALU, MAC, and Shifter registers at any time, for example during execution of a subroutine. The data register bank select bit of MSTAT determines which set of data registers is active (0=primary, 1=secondary). The secondary register set duplicates all of the input and result registers of the computation units, ALU, MAC, and Shifter:

AX0	MX0	SI
AX1	MX1	SE
AY0	MY0	SB
AY1	MY1	SR1
AF	MF	SR0
AR	MR0	
	MR1	
	MR2	

The following mode control instruction, for example, switches from the processor's primary register set to its secondary register set:

```
ENA SEC_REG;
```

while the following instruction switches back to the primary register set:

```
DIS SEC_REG;
```

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

AR saturation mode, when enabled, causes AR to be saturated to the maximum positive (0x7FFF) or negative (0x8000) values whenever an ALU overflow occurs.

The MAC result placement mode determines whether the multiplier operates in integer or fractional format. This mode is discussed in Chapter 2, “Computational Units.”

Setting the timer enable bit causes the timer to begin decrementing. Clearing this bit halts the timer.

Enabling GO mode allows the processor to continue executing instructions from internal program memory during a bus grant. The processor will halt, waiting for the buses to be released, only when an access of external memory is required. When GO mode is disabled, the processor always halts during bus grant.

3.6 CONDITIONAL INSTRUCTIONS

The condition logic circuit of the program sequencer determines whether a conditional instruction is executed, for example a jump, call, or arithmetic operation. It also controls 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 status conditions and corresponding assembly language syntax are listed in Table 3.9. These status conditions are used with the *IF condition* clause available on some instructions. In addition, the status of the FI pin (Flag In) can also be used as a condition for JUMP and CALL instructions.

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	—
FLAG_IN*	FI pin	Last sample of FI pin = 1
NOT FLAG_IN*	Not FI pin	Last sample of FI pin = 0

* Only available on JUMP and CALL instructions.

Table 3.9 IF Condition Logic

3.7 TOPPCSTACK

A special version of the Register-to-Register Move instruction, Type 17, is provided for reading (and popping) or writing (and pushing) the top value of the PC stack. The normal POP PC instruction does not save the value popped from the stack, so to save this value into a register you must use the following special instruction:

```
reg = TOPPCSTACK;     {pop PC stack into reg}
                      {“toppcstack” may also be
lowercase}
```

The PC stack is also popped by this instruction, after a one-cycle delay. A NOP should usually be placed after the special instruction, to allow the pop to occur properly:

```
reg = TOPPCSTACK;
NOP;                  {allow pop to occur correctly}
```

3 Program Control

There is no standard PUSH PC stack instruction. To push a specific value onto the PC stack, therefore, use the following special instruction:

```
TOPPCSTACK = reg;     {push reg contents onto PC stack}
```

The stack is pushed immediately, in the same cycle.

Examples:

```
AX0 = TOPPCSTACK;     {pop PC stack into AX0}
NOP;

TOPPCSTACK = I7;      {push contents of I7 onto PC stack}
```

Only the following registers may be used in the special TOPPCSTACK instructions:

ALU, MAC, & Shifter <u>*Registers*</u>	*DAG* <u>*Registers*</u>	
AX0	I0	I4
AX1	I1	I5
MX0	I2	I6
MX1	I3	I7
AY0	M0	M4
AY1	M1	M5
MY0	M2	M6
MY1	M3	M7
AR	L0	L4
MR0	L1	L5
MR1	L2	L6
MR	L3	L7
SI		
SE		
SR0		
SR1		

The Type 17 Register Move instruction is described in Chapter 15, Instruction Set Reference. *Note that TOPPCSTACK may not be used as a register in any other instruction type!*

3.7.1 TOPPCSTACK Restrictions

There are several restrictions on the use of the special TOPPCSTACK instructions, as described below.

1.) The pop and read TOPPCSTACK instruction may not be placed directly before an RTI instruction (return from interrupt). A NOP must be inserted in between:

```
reg = TOPPCSTACK;
NOP;              {allow pop to occur correctly}
RTI;              {another pop happens automatically}
```

2.) The pop and read TOPPCSTACK instruction may not be the last or next-to-last instruction in a Do Until loop. Neither instruction 1 nor instruction 2 may be the pop/read TOPPCSTACK instruction in the following code:

```
DO loop UNTIL CE;

      AX0=DM(I5,M5);
      ...
      instruction 2;
loop: instruction 1;
```

3.) There must be an equal number of pushes and pops within any Do Until loop, including any normal POP PC instructions as well as the special TOPPCSTACK pop/read and push/write instructions.

4.) Several restrictions exist in relation to the RTS (return from subroutine), RTI (return from interrupt routine), and POP PC instructions. If instruction 3 in the following sequence is an RTS, RTI, or POP PC,

```
instruction 1;
instruction 2;
```

```
instruction 3; {if this is an RTS, RTI, or POP PC … }
```

then the following restrictions must be observed:

- instruction 2 may not be either the pop/read or push/write TOPPCSTACK instruction.
- If instruction 3 is also the last instruction of a Do Until loop, then instruction 1 may not be the push/write TOPPCSTACK instruction.

Data Transfer 4

4.1 OVERVIEW

This chapter describes the processor units that control the movement of data to and from the processor, and from one data bus to the other within 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.

4.2 DATA ADDRESS GENERATORS (DAGS)

Every device in the ADSP-2100 family 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 generates only data memory addresses, but provides an optional bit-reversal capability, DAG2 can generate both data memory and program memory addresses, but has no bit-reversal capability.

While the following discussion explains the internal workings of the DAGs, bear in mind that the ADSP-2100 Family Development Software (assembler and linker) provides a direct method for declaring data buffers as circular or linear and for managing the placement of the buffer in memory. Only the initializing of DAG registers must be explicitly programmed: see “Indirect Addressing” and “Modulo Addressing (Circular Buffers)” below.

4.2.1 DAG Registers

Figure 4.1, 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.

4 Data Transfer

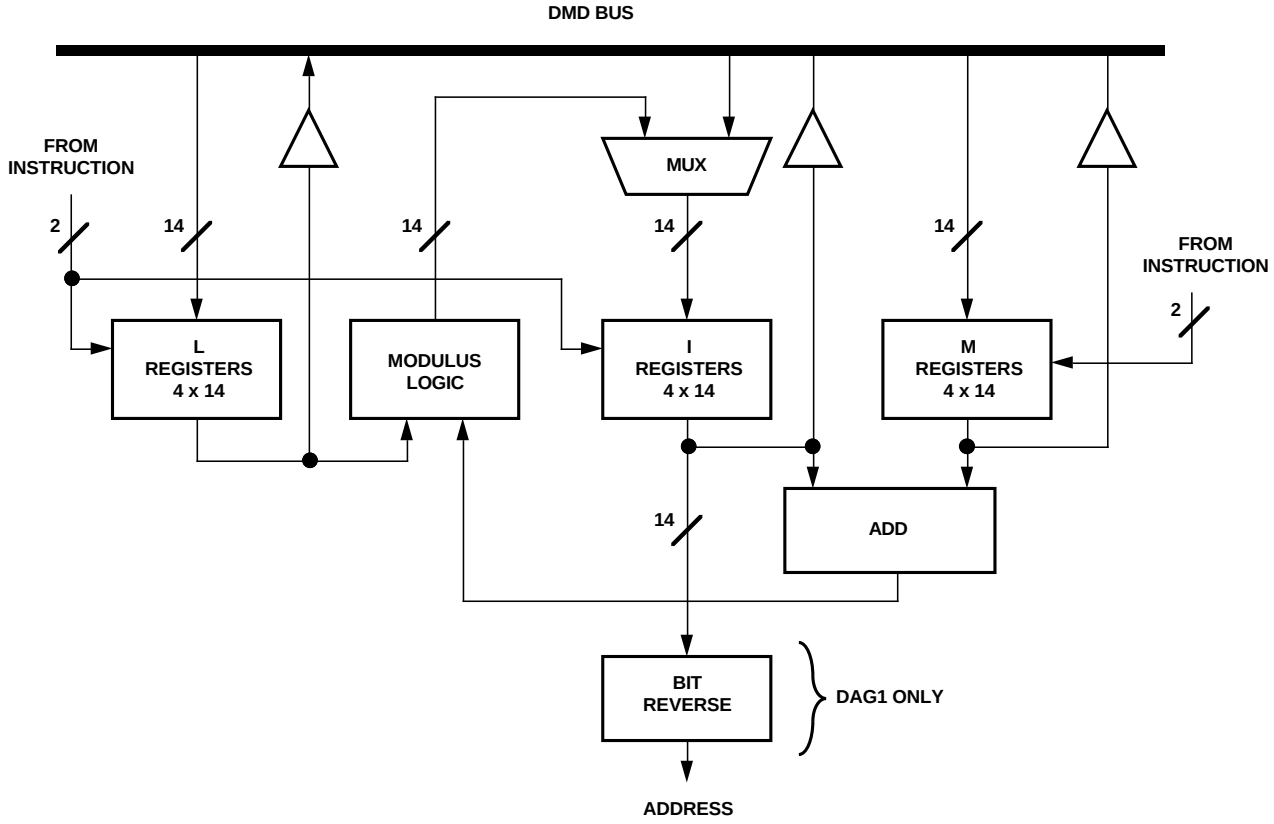


Figure 4.1 Data Address Generator Block Diagram

The I registers (I0-I3 in DAG1, I4-I7 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 appropriate mode bit in the mode status register (MSTAT) as discussed below or by using the ENA BIT_REV instruction. Bit-reversal facilitates FFT addressing.

The data address generators employ a post-modify scheme; after an indirect data access, the specified M register (M0-M3 in DAG1, M4-M7 in DAG2) is added to the specified I register to generate the updated 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-M3 set in any combination, but not by those in DAG2 (M4-M7). 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 corresponding to an I register (for example, L0 would correspond to I0) determines which addressing scheme is used for that I register*. 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.

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 and the I register 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.

4.2.2 Indirect Addressing

The ADSP-2100 family processors allow two addressing modes for data memory fetches: direct and register indirect. Indirect addressing is accomplished by loading an address into an I (index) register and specifying one of the available M (modify) registers.

The L registers are provided to facilitate wraparound addressing of circular data buffers. A circular buffer is only implemented when an L register is set to a non-zero value. For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.

Do not assume that the L registers are automatically initialized or may be ignored; the I, M, and L registers contain random values following processor reset. Your program must initialize the L registers corresponding to any I registers it uses.

4.2.2.1 Initialize L Registers To 0 For Non-Circular Addressing

Setting an L register to a non-zero value activates the processor's circular addressing modulus logic. For linear indirect addressing you must set the appropriate L register to zero to disable the modulus logic.

Here is a simple example of linear indirect addressing:

```
I3=0x3800;
M2=0;
L3=0;
AX0=DM(I3,M2);
```

Here is an example which uses a memory variable to store an address pointer:

```
.VAR/DM/RAM  addr_ptr;  {variable holds address to be
accessed}
I3=DM(addr_ptr);        {I3 loaded using direct addressing}
L3=0;                   {disable circular addressing}
M1=0;                   {no post-modify of I3}
AX0=DM(I3,M1);          {AX0 loaded using indirect addressing}
```

4.2.3 Modulo Addressing (Circular Buffers)

The modulus logic implements automatic modulo addressing for accessing circular data 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 according to 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

The inputs are subject to the condition:

$|M| < L$

This condition insures that the next address cannot wrap around the buffer more than once in one operation.

4.2.4 Calculating The Base Address

The base address of a circular buffer of length L is 2^n or a multiple of 2^n, where *n* satisfies the condition:

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

In other words, the base address is L "rounded" upwards to the closest power of 2 (or its multiple). This rule implies that a certain number of low-order bits of the base address must be zeroes.

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

4.2.4.1 Circular Buffer Base Address Example 1

For example, let us assume that the buffer length is eight. The length of the buffer must be less than or equal to some value 2^n; *n* therefore, must be three or greater. The left side of the inequality rule 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 satisfies both inequalities is three. Valid base addresses are multiples of 2^n, so in this example valid base addresses are multiples of eight: 0x0008, 0x0010, 0x0018, and so on.

4.2.4.2 Circular Buffer Base Address Example 2

As a second example, assume a buffer length of seven. The inequality again yields the same value for *n*, namely, three. With a buffer length of seven, therefore, the valid base addresses are multiples of eight: 0x0008, 0x0010, 0x0018, and so on.

4.2.4.3 Circular Buffer Operation Example 1

Suppose that I0 = 5, M0 = 1, L0 = 3, and the base address = 4. The next address is calculated as:

(I0 + M0 - B) mod L0 +B = (5 + 1 - 4) mod 3 + 4 = 6

The successive address calculations using I0 for indirect addressing produce the sequence: 5, 6, 4, 5, 6, 4, 5 …. For M0 = –1 (0x3FFF), I0 would produce the sequence: 5, 4, 6, 5, 4, 6, 5, 4 ….

4.2.4.4 Circular Buffer Operation Example 2

Assume that I0 = 9, M0 = 3, L0 = 5, and the base address = 8. The five-word buffer resides at locations 8 through 12 inclusive. The next address is calculated as:

(I0 + M0 – B) mod L0 + B = (9 + 3 – 8) mod 5 + 8 = 12

The successive address calculations using I0 for indirect addressing produce the sequence: 9, 12, 10, 8, 11, 9 ... This example highlights the fact that the address sequence does not have to result in a "direct hit" of the buffer boundary.

4.2.5 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 address lines ($ADDR_N$)
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 wide. You must determine the first address and also initialize the M register to be used with a value calculated to modify the I register bit-reversed output to the desired range. This value is:

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

where N is the number of bits you wish to output reversed. For a complete example of this, refer to Section 6.6.5.2 "Modified Butterfly" in Chapter 6, One-Dimensional FFTs, of the applications handbook *Digital Signal Processing Applications Using the ADSP-2100 Family (Volume 1)*.

4.3 PROGRAMMING DATA ACCESSES

The ADSP-2100 Family Development Software supports the declaration and use of a simple data structure: one-dimensional arrays, or buffers. The array may contain a single value (a variable) or multiple values (an array). 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 in assembly language. Complete syntax for all assembler directives is given in the *ADSP-2100 Family Assembler Tools Manual*.

4.3.1 Variables & Arrays

Arrays are the basic data structure of the ADSP-21xx. In our literature, the word "array" and the expression "data buffer" (as well as "variable") 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}
L0=0;                  {set L register to zero}
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 assembler directive reads the values from the file *filename.dat* into the array at link time. This feature is supported only in the simulator — 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 is a simple single-word variable, and is declared in this way:

```
.VAR/DM coefficient;
```

4.3.2 Circular Buffers

A common requirement in DSP is the circular buffer. This is directly implemented by the processors' data address generators (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 assemblers's % 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 first address of buffer}
M0=1;               {increment by 1 location each time}
```

Now a statement like

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

placed in a loop, cycles continuously through *coefficients* and wraps around automatically.

4.4 PMD-DMD BUS EXCHANGE

The PMD-DMD bus exchange unit couples the program memory data bus and the data memory data bus, allowing them to transfer data between them 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.

4.4.1 PMD-DMD Block Diagram Discussion

Figure 4.2 shows a block diagram of the PMD-DMD bus exchange. There are two types of connections provided by this circuitry.

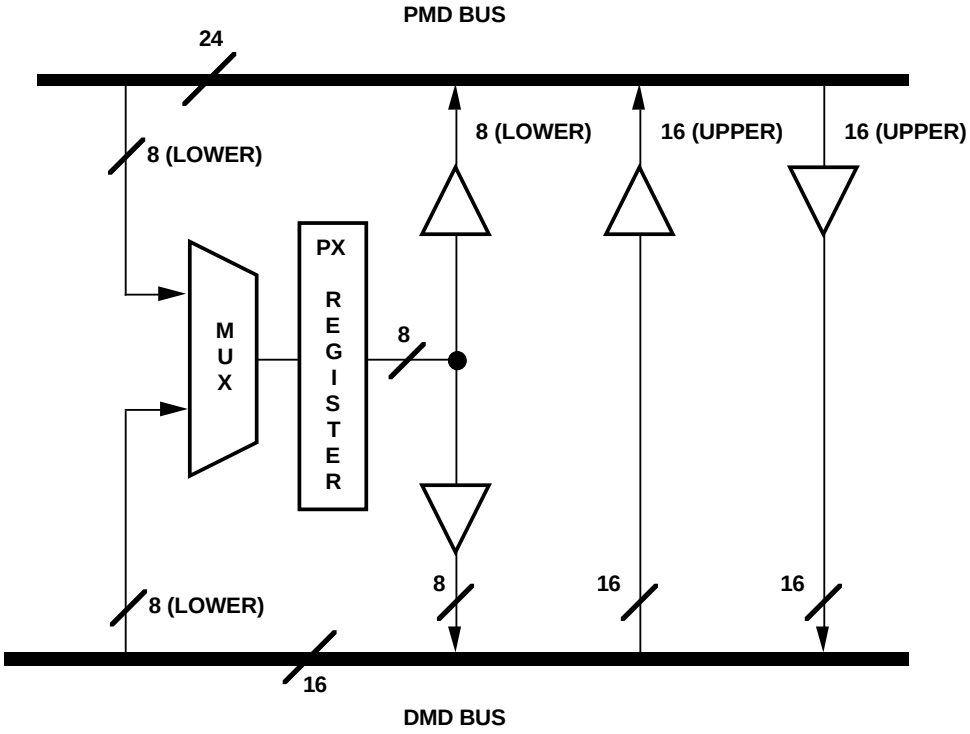


Figure 4.2 PMD–DMD Bus Exchange

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. For example:

   AX0 = PM(I4,M4);

   In this example, the upper 16 bits of a 24-bit program memory word are loaded into AX0 and the lower 8 bits are automatically loaded into PX.

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

   PM(I4,M4) = AX0;

   In this example, the 16 bits of AX0 are stored into the upper 16 bits of a 24-bit program memory word. The 8 bits of PX are automatically

stored to the 8 lower bits of the memory word.

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.

   PX = AX0;

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.

   AX0 = PX;

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.

Serial Ports 5

5.1 OVERVIEW

Synchronous serial ports, or SPORTs, support a variety of serial data communications protocols and can provide a direct interconnection between processors in a multiprocessor system.

These ADSP-2100 family processors contain serial ports:

Processor	*Number of Serial Ports*
ADSP-2101	2
ADSP-2105	1
ADSP-2115	2
ADSP-2111	2
ADSP-2171	2
ADSP-2181	2
ADSP-21msp58/59	2

The serial ports, designated SPORT0 and SPORT1, have some differences that are described in this chapter. On the ADSP-2105, only SPORT1 is provided.

5.2 BASIC SPORT DESCRIPTION

Each SPORT has a five-pin interface:

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

Table 5.1 SPORT External Interface

5 Serial Ports

A SPORT receives serial data on its DR input and transmits serial data on its DT output. It can receive and transmit simultaneously, for full duplex operation. The data bits are synchronous to the serial clock SCLK, which is an output if the processor generates this clock or an input if the clock is generated externally. Frame synchronization signals RFS and TFS are used to indicate the start of a serial data word or stream of serial words.

Figure 5.1, shows a simplified block diagram of a single SPORT. Data to be transmitted is written from an internal processor register to the SPORT's TX register via the DMD bus. This data is optionally compressed in hardware, then automatically transferred to the transmit shift register. The bits in the shift register are shifted out on the SPORT's DT pin, MSB first, synchronous to the serial clock. The receive portion of the SPORT accepts data from the DR pin, synchronous to the serial clock. When an entire word is received, the data is optionally expanded, then automatically transferred to the SPORT's RX register, where it is available to the processor.

The following is a list of SPORT characteristics. Many of the SPORT characteristics are configurable to allow flexibility in serial communication.

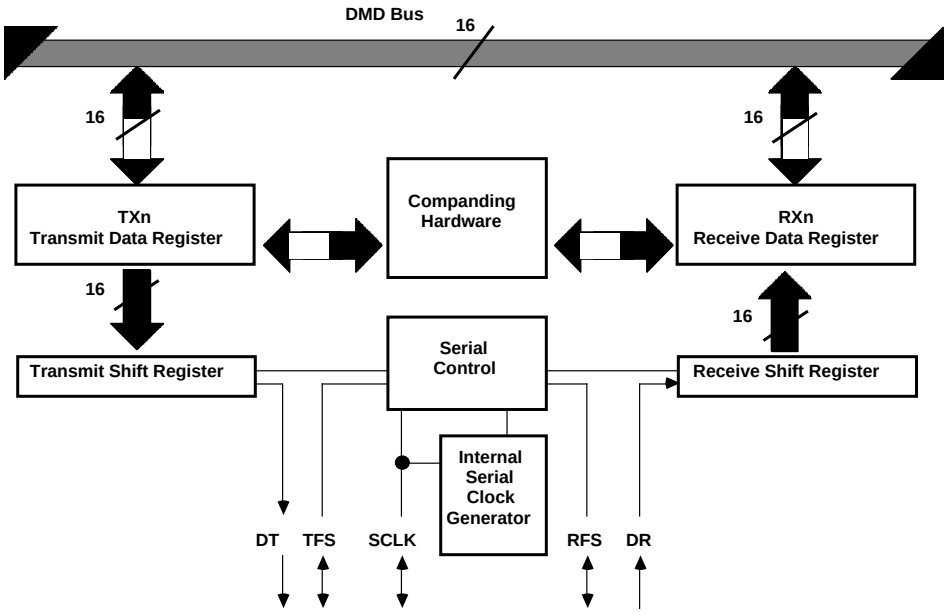


Figure 5.1 Serial Port Block Diagram

- Bidirectional: each SPORT has independent transmit and receive sections.
- Double-buffered: each SPORT section (both receive and transmit) has a data register for transferring data words to and from other parts of the processor and a register for shifting data in or out. The double-buffering provides additional time to service the SPORT.
- Clocking: each SPORT can use an external serial clock or generate its own in a wide range of frequencies down to 0 Hz. See Section 5.5.
- Word length: each SPORT supports serial data word lengths from three to sixteen bits. See Section 5.6.
- Framing: each SPORT section (receive and transmit) can operate with or without frame synchronization signals for each data word; with internally-generated or externally-generated frame signals; with active high or active low frame signals; with either of two pulse widths and frame signal timing. See Section 5.7.
- Companding in hardware: each SPORT can perform A-law and μ-law companding according to CCITT recommendation G.711. See Section 5.10.
- Autobuffering with single-cycle overhead: using the DAGs, each SPORT can automatically receive and/or transmit an entire circular buffer of data with an overhead of only one cycle per data word. Transfers between the SPORT and the circular buffer are automatic in this mode and do not require additional programming. See Section 5.11.
- Interrupts: each SPORT section (receive and transmit) generates an interrupt upon completing a data word transfer, or after transferring an entire buffer if autobuffering is used. See Section 5.13.
- Multichannel capability: SPORT0 can receive and transmit data selectively from channels of a serial bitstream that is time-division multiplexed into 24 or 32 channels. This is especially useful for T1 interfaces or as a network communication scheme for multiple processors. See Section 5.12. **Note:** The ADSP-2105 has only one serial port (SPORT1) and does not support multichannel operation.
- Alternate configuration: SPORT1 can be configured as two external

5 Serial Ports

interrupt inputs, IRQ0 and 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. See Section 5.4.

5.2.1 Interrupts

Each SPORT has a receive interrupt and a transmit interrupt. The priority of these interrupts is shown in Table 5.2.

Highest	SPORT0 Transmit *(on 2-SPORT processors)*
	SPORT0 Receive *(on 2-SPORT processors)*
	SPORT1 Transmit
Lowest	SPORT1 Receive

Table 5.2 SPORT Interrupt Priorities

For complete details about how interrupts are handled, see the "Interrupts" section in Chapter 3, "Program Control."

5.2.2 SPORT Operation

Writing to a SPORT's TX register readies the SPORT for transmission; the TFS signal initiates the transmission of serial data. Once transmission has begun, each value written to the TX register is transferred to the internal transmit shift register and subsequently the bits are sent, MSB first. Each bit is shifted out on the rising edge of SCLK.

After the first bit (MSB) of a word has been transferred, the SPORT generates the transmit interrupt. The TX register is now available for the next data word, even though the transmission of the first word is ongoing.

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 RX register and the receive interrupt for that SPORT is generated.

Interrupts are generated differently if autobuffering is enabled; see "Autobuffering" later in this chapter.

5.3 SPORT PROGRAMMING

To the programmer, the SPORT can be viewed as two functional sections.

The configuration section is a block of control registers (mapped to data memory) that the program must initialize before using the SPORTs. The data section is a register file used to transmit and receive values through the SPORT.

5.3.1 SPORT Configuration

SPORT configuration is accomplished by setting bit and field values in configuration registers. These registers are memory mapped in data memory space. SPORT0 configuration registers occupy locations 0x3FF3 to 0x3FFA; SPORT1 configuration registers occupy locations 0x3FEF to 0x3FF2. The contents of these registers are summarized in Table 5.3 and in the register summary in Appendix E. The effects of the various settings are described at length in the sections that follow.

Address	*Contents*
0x3FFA	SPORT0* multichannel receive word enables (31-16)
0x3FF9	SPORT0* multichannel receive word enables (15-0)
0x3FF8	SPORT0* multichannel transmit word enables (31-16)
0x3FF7	SPORT0* multichannel transmit word enables (15-0)
0x3FF6	SPORT0* control register Multichannel mode controls Serial clock source Frame synchronization controls Companding mode Serial word length
0x3FF5	SPORT0* serial clock divide modulus (determines frequency)
0x3FF4	SPORT0* receive frame sync divide modulus (determines frequency)
0x3FF3	SPORT0* autobuffer control register
0x3FF2	SPORT1 control register Flag output value Serial clock source Frame synchronization controls Companding mode Serial word length
0x3FF1	SPORT1 serial clock divide modulus (determines frequency)
0x3FF0	SPORT1 receive frame sync divide modulus (determines frequency)
0x3FEF	SPORT1 autobuffer control register (not on ADSP-21msp58/59)

*SPORT0 configuration registers are defined only on processors that have both SPORT0 and SPORT1

Table 5.3 SPORT Configuration Registers

There are two ways to initialize or to change values in SPORT

configuration registers: write a register to an immediate address (instruction type 3) or write immediate data to an indirect address (instruction type 2). With either method, it is important to configure the serial port before enabling it.

The first method of programming configuration registers requires no setup of DAG registers but does require two instructions to perform the write. For example:

```
AX0 = 0x6B27;
DM(0x3FF2) = AX0;          {the contents of AX0 are written}
                           {to the address 0x3FF2}
AX0 = 0;
DM(0x3FF3) = AX0;          {the contents of AX0 are written}
                           {to address 0x3FF3}
```

In the second method, the DAG (I) index register must contain the data memory address of the configuration register to be written. The modify (M) register, which updates the I register after the write, must also contain a valid value. And the length (L) register that has the same number as the I register must be initialized to zero so that the circular buffer capability is not active. For example:

```
I0 = 0x3FF2;
M0 = 1;
L0 = 0;
DM(I0,M0) = 0x6B27;        {the constant 0x6B27 is written to
}
                           {address pointed to by I0; pointer
}
                           {then modified by M0}
DM(I0,M0) = 0;             {address 0x3FF3 is set to 0}
```

Either method works. The second method requires only one cycle to configure the registers once the I, M and L registers are initialized. This method is, however, more prone to error because the registers are written indirectly. You must make sure that the I register contains the intended value before the write.

5.3.2 Receiving And Transmitting Data

Each SPORT has a receive register and a transmit register. These registers are not memory mapped, but are identified by assembler mnemonics. The transmit registers are named TX0 and TX1, for SPORT0 and SPORT1 respectively. Receive registers are named RX0 and RX1 for SPORT0 and SPORT1 respectively. These registers can be accessed at any time during program execution using a data memory access with immediate address,

load of a non-data register with immediate data or register-to-register move (instruction types 3, 7 and 17). For example, the following instruction would ready SPORT1 to transmit a serial value, assuming SPORT1 is configured and enabled:

```
TX1 = AX0;          {the contents of AX0 are transmitted}
                    {on SPORT1}
```

The following instruction would access a serial value received on SPORT0:

```
AY0 = RX0;          {the contents of SPORT0 receive register}
                    {is transferred to AY0}
```

Because the SPORTs are interrupt driven, these instructions would typically be executed within a interrupt service routine in response to a SPORT interrupt.

5.4 SPORT ENABLE

SPORTs are enabled through bits in the system control register. This register is mapped to data memory address 0x3FFF. Bit 12 enables SPORT0 if it is a 1, and bit 11 enables SPORT1 if it is a 1. Both of these bits

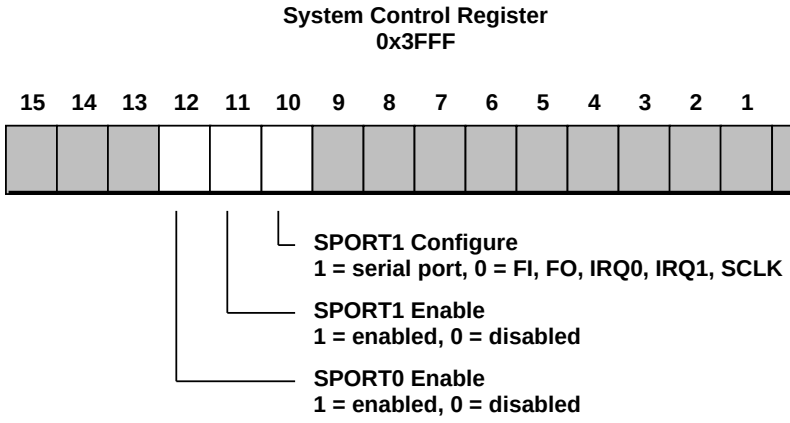


Figure 5.2 SPORT Enables In The System Control Register

are cleared at reset, disabling both SPORTs.

Bit 10 of the system control register determines the configuration of SPORT1, either as a serial port or as interrupts and flags, according to Table 5.4 on the next page. If bit 10 is a 1, SPORT1 operates as a serial port; if it is a 0, the alternate functions are in effect (and bit 11 is ignored). At reset, bit 10 is a 1, so SPORT1 functions as a serial port.

Pin Name	*Alternate Name*	*Alternate Function*
RFS1	IRQ0	External interrupt 0
TFS1	IRQ1	External interrupt 1
DR1	FI	Flag input
DT1	FO	Flag output
SCLK1	Same	Same

Table 5.4 SPORT1 Alternate Configuration

5.5 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, bit 14 in either the SPORT0 or SPORT1 control register, determines the SCLK source for the SPORT. If this bit is a 1, the processor generates the SCLK signal; if it is a 0, the processor expects to receive an external clock signal on SCLK. At reset, ISCLK is cleared, so both serial ports are in the external clock mode. When ISCLK is set, internal generation of the SCLK signal begins on the next instruction cycle, whether or not the corresponding SPORT is enabled.

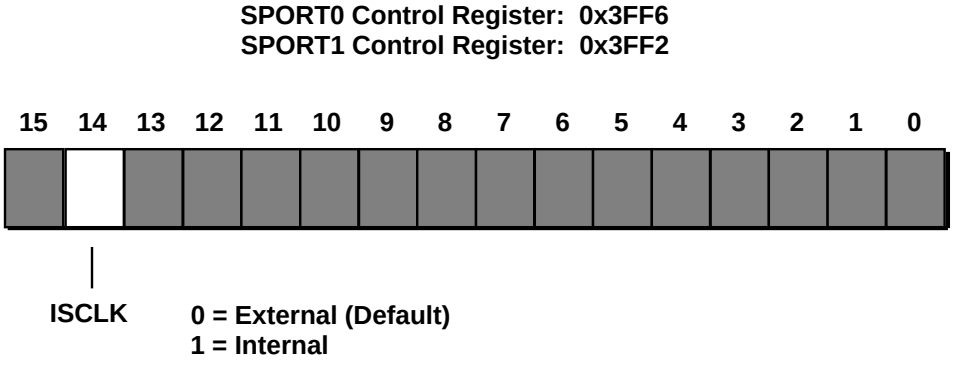


Figure 5.3 ISCLK Bit In SPORT Control Register

External serial clock frequencies may be as high as the processor's cycle rate, up to a maximum of 13.824 MHz; internal clock frequencies may be as high as one-half the processor's clock rate. The frequency of an internally generated clock is a function of the processor clock frequency (as seen at the CLKOUT pin) and the value of the 16-bit serial clock divide modulus register SCLKDIV (0x3FF5 for SPORT0 and 0x3FF1 for SPORT1).

$$\text{SCLK frequency} = \frac{\text{CLKOUT frequency}}{2 \times (\text{SCLKDIV} + 1)}$$

Table 5.5 shows how some common SCLK frequencies correspond to values of SCLKDIV.

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

(Assumes CLKOUT frequency of 12.288 MHz)

Table 5.5 Common Serial Clock Frequencies (Internally Generated)

If the value of SCLKDIV is changed while the internal serial clock is enabled, the change in SCLK frequency takes effect at the start of the next rising edge of SCLK.

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 signal as described above.

5.6 WORD LENGTH

Each SPORT independently handles words of 3 to 16 bits. The data is right-justified in the SPORT data registers if it is fewer than 16 bits long. The serial word length (SLEN) field in each SPORT control register determines the word length according to this formula:

$$\text{Serial Word Length} = \text{SLEN} + 1$$

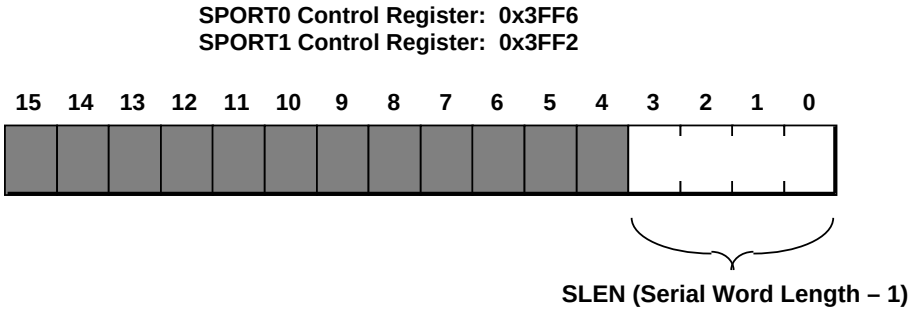


Figure 5.4 SLEN Field In SPORT Control Register

For example, if you are using 8-bit serial words, set SLEN to 7 (0111 binary). The SLEN field is bits 3-0 in the SPORT control register (0x3FF6 for SPORT0 and 0x3FF2 for SPORT1). See Figure 5.4 on the next page.

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

5.7 WORD FRAMING OPTIONS

Framing signals identify the beginning of each serial word transfer. The SPORTs have many ways of handling framing signals. Transmit and receive framing are independent of each other. All frame sync signals are sampled on the falling edge of the serial clock (SCLK).

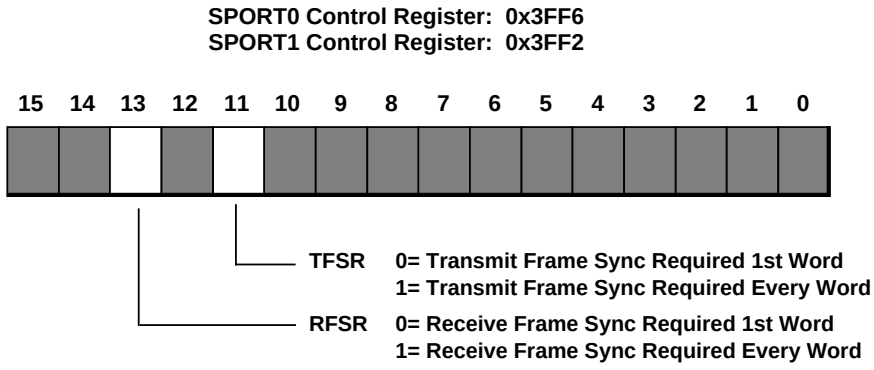


Figure 5.5 TFSR And RFSR Bits In SPORT Control Register

5.7.1 Frame Synchronization

Word framing signals are optional. If the receive frame sync required (RFSR) or transmit frame sync required (TFSR) bit in the SPORT control register is a 0, a frame sync 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 a 1, a frame sync signal is required at the start of every data word.

The RFSR bit is bit 13 in the SPORT control register (0x3FF6 for SPORT0 and 0x3FF2 for SPORT1), and the TFSR bit is bit 11. These bits are both cleared at reset, so that communication in both directions on both serial ports is unframed.

See "Configuration Examples" later in this chapter for examples of frame sync timing.

5.7.2 Frame Sync Signal Source

The processor can generate frame synchronization signals internally or receive them from an external source. The sources for transmit frame syncs and receive frames syncs can be set independently. If the internal receive frame sync (IRFS) bit or internal transmit frame sync (ITFS) bit in the SPORT control register is a 0, the processor expects to receive a signal

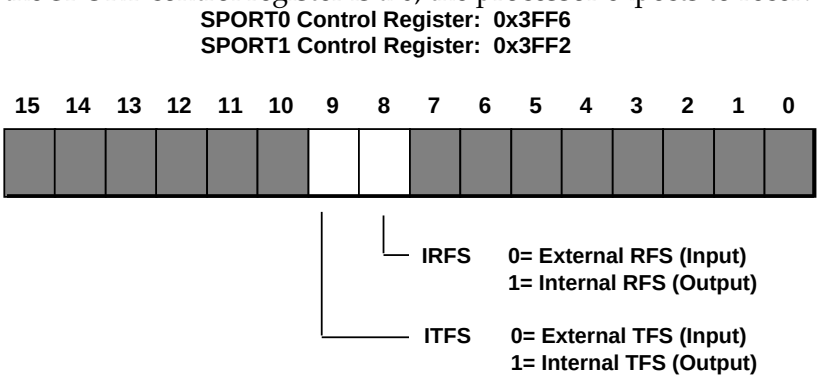


Figure 5.6 ITFS And IRFS Bits In SPORT Control Register

on its frame sync pin (RFS or TFS). If the IRFS or ITFS bit is a 1, the processor generates its own frame sync signal and drives the RFS or TFS pin as an output.

The IRFS bit is bit 8 in the SPORT control register (0x3FF6 for SPORT0 and 0x3FF2 for SPORT1), and the ITFS bit is bit 9. Both of these bits are cleared at reset, that is, both serial ports require externally generated frame sync signals for both transmitting and receiving data.

If frame sync signals are generated externally, then RFS and TFS are inputs, and the external source controls data transmission and reception. The SPORT will wait for a transmit frame sync before transmitting data and for a receive frame sync before receiving data. If frame sync signals are generated internally, however, then RFS and TFS are outputs, and the processor controls the timing of data operations.

The SPORT outputs an internally generated transmit framing signal after data is loaded into the transmit (TX0 or TX1) register, 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; see "Normal and Alternate Framing Modes," the next section). The occurrence of the transmit frame sync is a result of the availability of data in the transmit register.

With an internally generated receive framing signal, the processor controls the timing of the receive data. The external data source must provide data to the serial port synchronized to the receive framing signal (the timing depends on the framing mode being used; see "Normal and Alternate Framing Modes," the next section). The processor generates RFS periodically on a multiple of SCLK cycles, based on the value of the 16-bit receive frame sync divide modulus register, RFSDIV (0x3FF4 for SPORT0 and 0x3FF0 for SPORT1):

Number of SCLK cycles between RFS assertions = RFSDIV + 1

For example, to allow 256 SCLK cycles between RFS assertions, set RFSDIV to 255 (0xFF).

Values of RFSDIV+1 that are less than the word length are not recommended.

Note that frame sync signals may be generated internally even when

SCLK is supplied externally. This provides a way to divide external clocks for any purpose.

You can also use one frame sync to generate a single signal for both transmit and receive data. For example, an internally generated RFS (output) could be connected to an externally generated TFS (input) on the same SPORT for simultaneous transmit and receive operations. This interconnection is especially useful for combo codec interfaces.

5.7.3 Normal And Alternate Framing Modes

In the normal framing mode, the framing signal is checked at the falling edge of SCLK. If the framing signal is asserted, received data is latched on the *next falling* edge of SCLK and transmitted data is driven on the *next rising* edge of SCLK. 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.

In the alternate framing mode, the framing signal should be asserted in the same SCLK cycle as the first bit of a word. Received data bits are latched on the falling edge of SCLK and transmitted bits are driven on the rising edge of SCLK, but the framing signal is checked only on the first bit. Internally generated frame sync signals remain asserted for the length of the serial word. Externally generated frame sync signals are only checked during the first bit time.

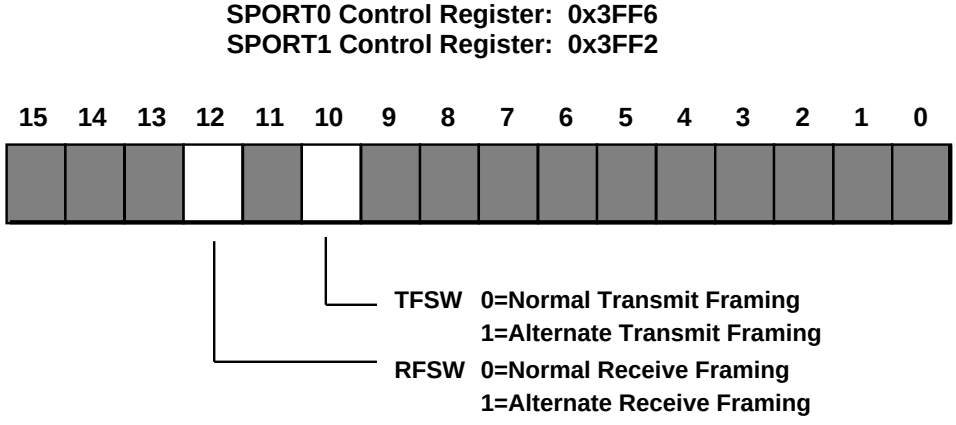


Figure 5.7 TFSW And RFSW Bits In SPORT Control Register

Framing modes for receiving and transmitting data are independent. If the receive frame sync width (RFSW) bit or transmit frame sync width (TFSW) bit in the SPORT control register is a 0, normal framing is enabled. If the RFSW or TFSW bit is a 1, alternate framing is used. The RFSW bit is bit 12 in the SPORT control register (0x3FF6 for SPORT0 and 0x3FF2 for SPORT1), and the TFSW bit is bit 10. These bits are both cleared at reset, so that normal framing in both directions is enabled.

For examples of normal and alternate framing, see "Configuration Examples" later in this chapter.

5.7.4 Active High Or Active Low

Framing sync signals for receiving and transmitting data can be either active high or active low and are configured independently. If the invert RFS (INVRFS) bit or invert TFS (INVTFS) bit in the SPORT control register is a 0, the corresponding frame sync signal is active high. If the INVRFS or INVTFS bit is a 1, the frame sync signal is active low. These controls apply

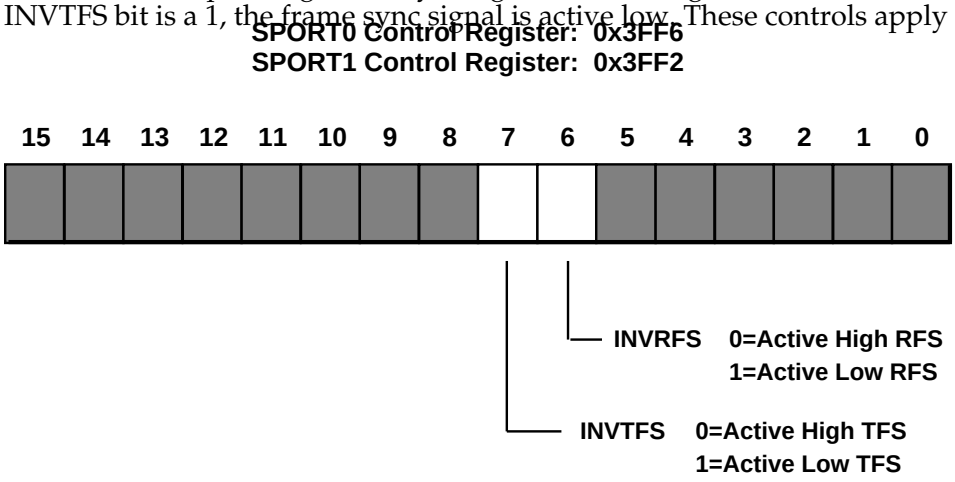


Figure 5.8 INVTFS And INVRFS Bits In SPORT Control Register

regardless of the source of frame sync signals; they either control the polarity of internally generated signals or determine how externally generated signals are interpreted.

The INVRFS bit is bit 6 in the SPORT control register (0x3FF6 for SPORT0 and 0x3FF2 for SPORT1), and the INVTFS bit is bit 7. These bits are both cleared at reset, so that frame sync signals are active high.

5.8 CONFIGURATION EXAMPLE

The example code that follows illustrates how to configure the SPORTs. This example configures both SPORT0 and SPORT1. SPORT0 is configured for an internally generated serial clock (SCLK), internally generated frame synchronization, and μ-law companded 8-bit data. This is a typical setup for communication with a combo codec. SPORT1 is configured for an externally generated serial clock, externally generated frame synchronization, non-companded 16-bit data and autobuffering. This setup could be used to transfer data between processors in a multiprocessor system.

Only the needed memory mapped registers are initialized. Notice that the SPORTs are configured before they are enabled and that any extraneous latched interrupts are cleared before interrupts are enabled.

```
      {——   SPORT INITIALIZATION CODE   ——}

      {SPORT1 inits }

      AX0 = 0x0017;
      DM(0x3FEF) = AX0;       {enable SPORT1 autobuffering}
                              {TX autobuffer uses I0 and M0}
                              {RX autobuffer uses I1 and M1}

      AX0 = 0x280F;
      DM(0x3FF2) = AX0;       {external serial clock, RFS and
TFS}
                              {RFS and TFS are required, normal}
                              {framing, no companding and 16
bits}
```

(continued on next page)

(continued from previous page)

```
      {SPORT0 inits}

      {Assumes a CLKIN of 12.288 MHz. Internally generated}
      {SCLK will be 2.048 MHz, and framing sync of 8 kHz}

      AX0 = 255;
      DM(0x3FF4) = AX0;       {RFSDIV = 256, 256 SCLKs between}
                              {frame syncs: 8 kHz framing}

      AX0 = 2;
      DM(0x3FF5) = AX0;       {SCLK = 2.048 MHz}

      AX0 = 0x6B27;
      DM(0x3FF6) = AX0;    {internal SCLK, RFS and TFS}
                           {normal framing, mu-law companding}
                           {8 bit words}

      {SPORT ENABLE}

      IFC = 0x1E;          {clear any extraneous SPORT
interrupts}
      ICNTL = 0;           {interrupt nesting disabled}

      AX0 = 0x1C1F;        {both SPORTs enabled, BWAIT and}
      DM(0x3FFF) = AX0;    {PWAIT left as default}

      IMASK = 0x1E;        {SPORT interrupts are enabled}

      {——   END SPORT INITIALIZATIONS   ——}
```

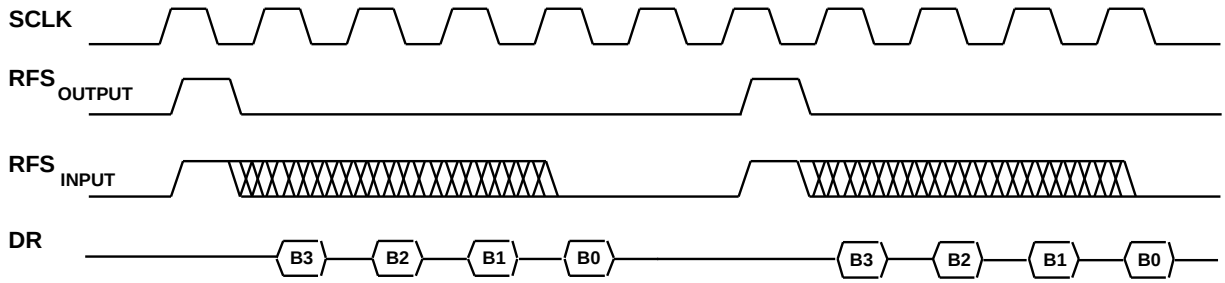
Figure 5.9 Example SPORT Configuration Code

5.9 TIMING EXAMPLES

This section contains examples of some combinations of the various framing options. The timing diagrams show relationships between signals, but are not scaled to show the actual timing parameters of the processor. Consult the data sheet for actual timing parameters and values.

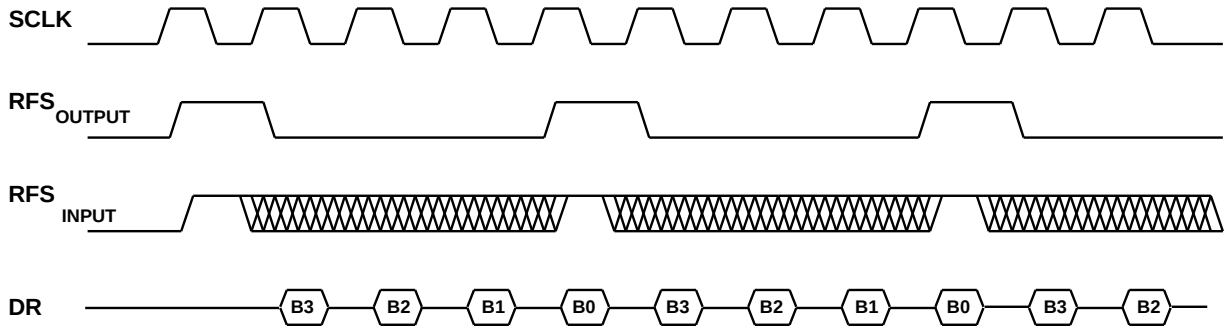
The examples assume a word length of four bits, that is, SLEN = 3. Framing signals are active high, that is, INVRFS = 0 and INVTFS = 0.

The value of the SPORT control register (0x3FF6 for SPORT0 and 0x3FF2 for SPORT1) is shown for each example. In these binary values, 1= high, 0



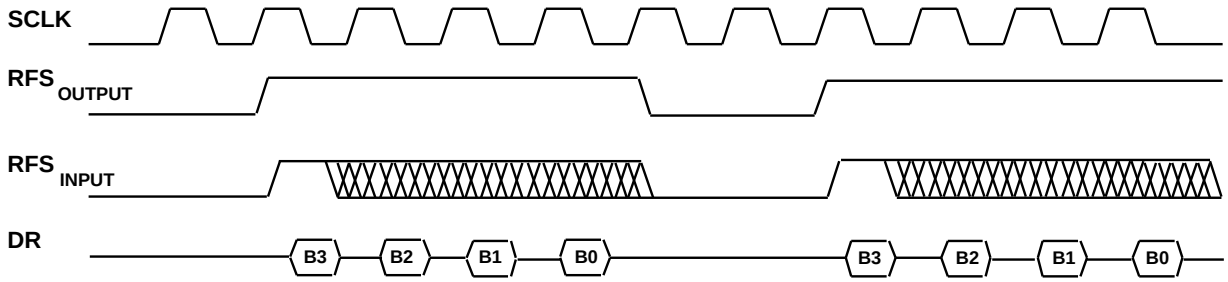
SPORT Control Register:
Internal Frame Sync 0X**10** XXX**1** X**0**XX 0011
External Frame Sync 0X**10** XXX**0** X**0**XX 0011
Both Internal Framing Option and External Framing Option Shown

Figure 5.10 SPORT Receive, Normal Framing



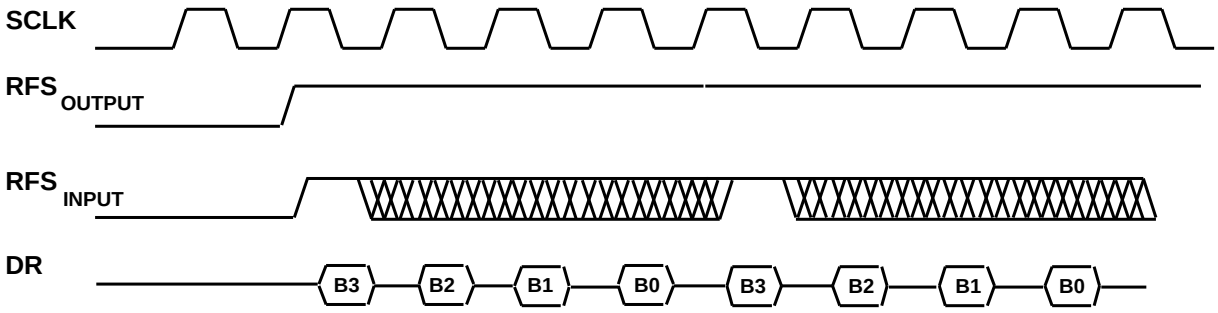
SPORT Control Register:
Internal Frame Sync 0X**10** XXX**1** X**0**XX 0011
External Frame Sync 0X**10** XXX**0** X**0**XX 0011
Both Internal Framing Option and External Framing Option Shown

Figure 5.11 SPORT Continuous Receive, Normal Framing



SPORT Control Register:
Internal Frame Sync 0X**11** XXX**1** X**0**XX 0011
External Frame Sync 0X**11** XXX**0** X**0**XX 0011
Both Internal Framing Option and External Framing Option Shown

Figure 5.12 SPORT Receive, Alternate Framing

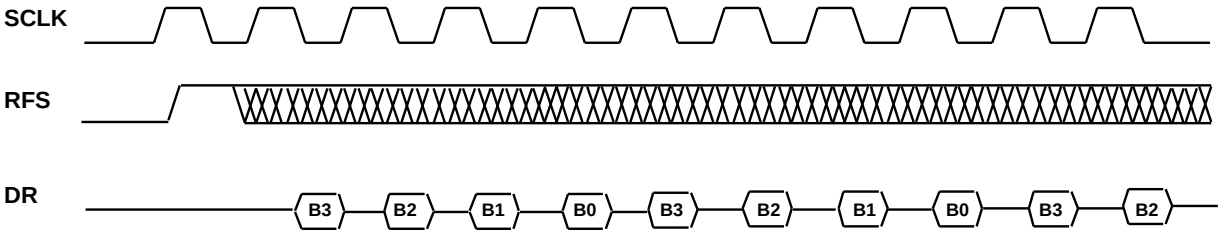


SPORT Control Register:
Internal Frame Sync 0X**11** XXX**1** X**0**XX 0011
External Frame Sync 0X**11** XXX**0** X**0**XX 0011
Both Internal Framing Option and External Framing Option Shown

Figure 5.13 SPORT Continuous Receive, Alternate Framing

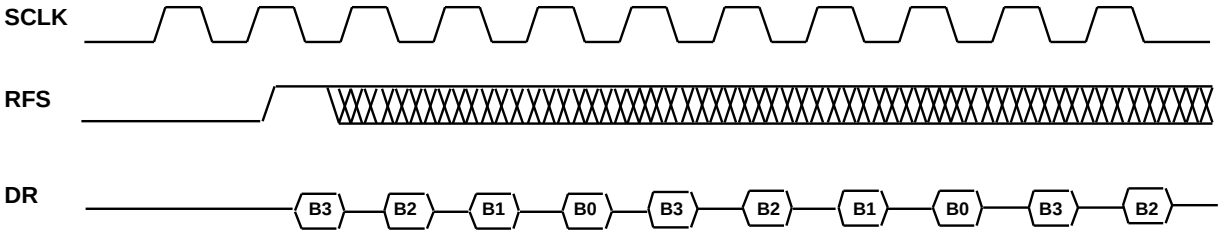
= low, and X can be either. The **underlined** bit values are the bits which set the modes illustrated in the example.

Figures 5.10 to 5.15 show framing for receiving data. In Figures 5.10 and 5.11, the normal framing mode is shown for noncontinuous data (any number of SCLK cycles between words) and continuous data (no SCLK cycles between words). Figures 5.12 and 5.13 show noncontinuous and



SPORT Control Register:
Internal Frame Sync 0X**<u>00</u>** XXX**<u>1</u>** X**<u>0</u>**XX 0011
External Frame Sync 0X**<u>00</u>** XXX**<u>0</u>** X**<u>0</u>**XX 0011

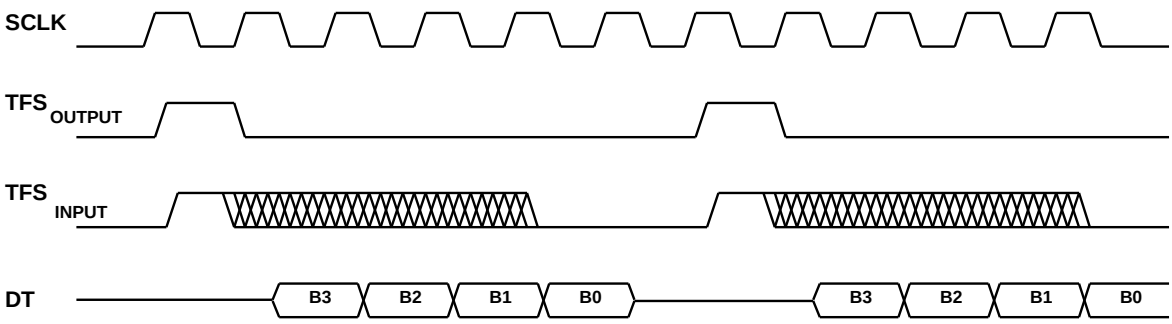
Figure 5.14 SPORT Receive, Unframed Mode, Normal Framing



SPORT Control Register:
Internal Frame Sync 0X**<u>01</u>** XXX**<u>1</u>** X**<u>0</u>**XX 0011
External Frame Sync 0X**<u>01</u>** XXX**<u>0</u>** X**<u>0</u>**XX 0011

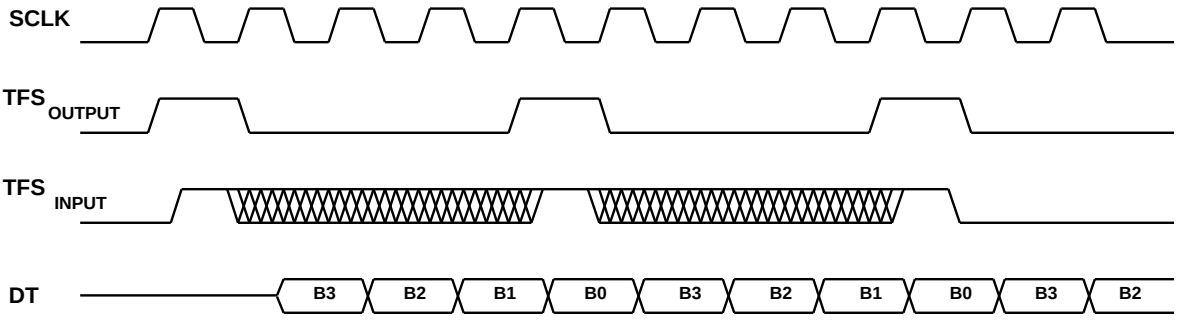
Figure 5.15 SPORT Receive, Unframed Mode, Alternate Framing

continuous receiving in the alternate framing mode. In these four figures, both the input timing requirement for an externally generated frame sync and the output timing characteristic of an internally generated frame sync are shown. Note that the output meets the input timing requirement; thus, on processors with two SPORTs, one SPORT could provide RFS for the other.



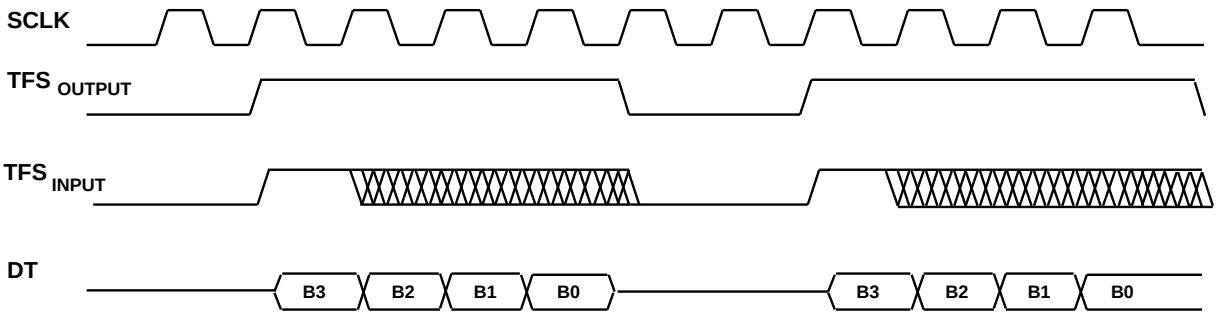
SPORT Control Register:
Internal Frame Sync 0XXX **101**X **0**XXX 0011
External Frame Sync 0XXX **100**X **0**XXX 0011
Both Internal Framing Option and External Framing Option Shown

Figure 5.16 SPORT Transmit, Normal Framing



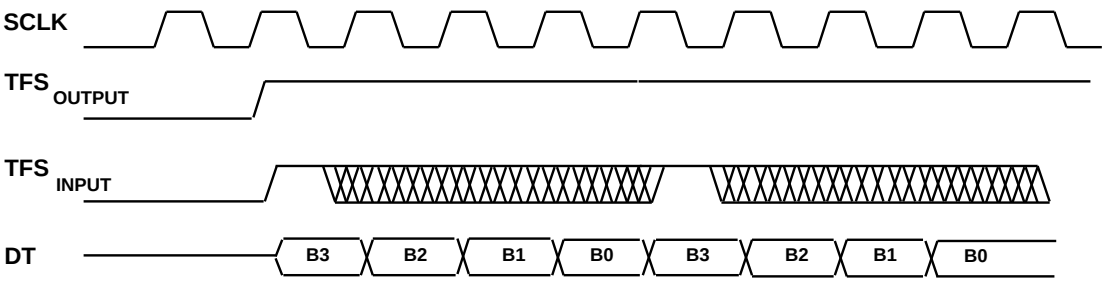
SPORT Control Register:
Internal Frame Sync 0XXX **101**X **0**XXX 0011
External Frame Sync 0XXX **100**X **0**XXX 0011
Both Internal Framing Option and External Framing Option Shown

Figure 5.17 SPORT Continuous Transmit, Normal Framing



SPORT Control Register:
Internal Frame Sync 0XXX **111**X **0**XXX 0011
External Frame Sync 0XXX **110**X **0**XXX 0011
Both Internal Framing Option and External Framing Option Shown
Note: There is an asynchronous delay between TFS input and DT. See the appropriate data sheet for specifications.

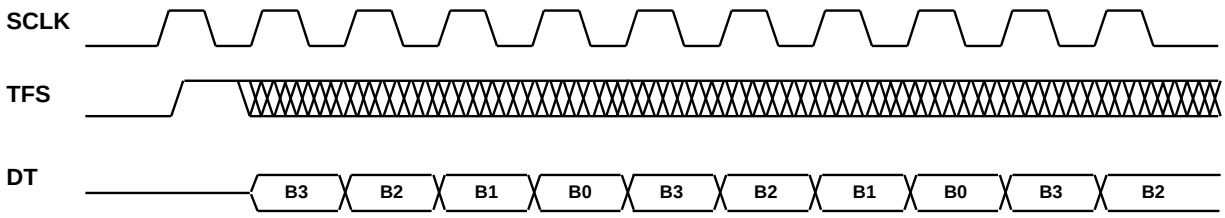
Figure 5.18 SPORT Transmit, Alternate Framing



SPORT Control Register:
Internal Frame Sync 0XXX **111**X **0**XXX 0011
External Frame Sync 0XXX **110**X **0**XXX 0011
Both Internal Framing Option and External Framing Option Shown
Note: There is an asynchronous delay between TFS input and DT. See the appropriate data sheet for specifications.

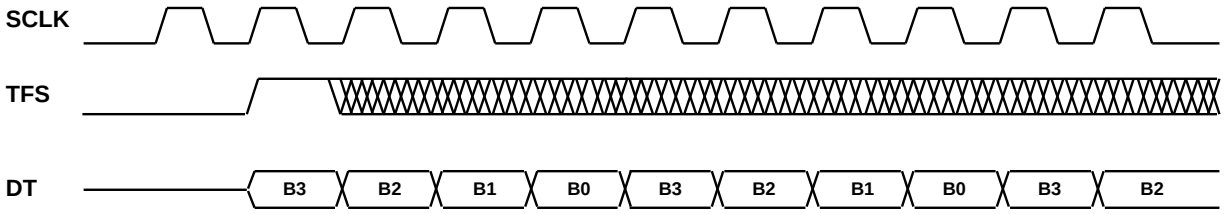
Figure 5.19 SPORT Continuous Transmit, Alternate Framing

Figures 5.14 and 5.15 show the receive operation with normal framing and alternate framing, respectively, in the unframed mode. There is a single the frame sync signal that occurs only at the start of the first word, either one SCLK before the first bit (normal) or at the same time as the first bit (alternate). This mode is appropriate for multiword bursts (continuous reception).



SPORT Control Register:
Internal Frame Sync 0XXX **001**X **0**XXX 0011
External Frame Sync 0XXX **000**X **0**XXX 0011

Figure 5.20 SPORT Transmit, Unframed Mode, Normal Framing



SPORT Control Register:
Internal Frame Sync 0XXX **011**X **0**XXX 0011
External Frame Sync 0XXX **010**X **0**XXX 0011
Note: There is an asynchronous delay between TFS input and DT. See the appropriate data sheet for specifications.

Figure 5.21 SPORT Transmit, Unframed Mode, Alternate Framing

Figures 5.16 to 5.21 show framing for transmitting data and are very similar to Figures 5.10 to 5.15. In Figures 5.16 and 5.17, the normal framing mode is shown for noncontinuous data and continuous data. Figures 5.18 and 5.19 show noncontinuous and continuous transmission in the alternate framing mode. As with receive timing, the TFS output meets the TFS input timing requirement.

Figures 5.20 and 5.21 show the transmit operation with normal framing and alternate framing, respectively, in the unframed mode. There is a single the frame sync signal that occurs only at the start of the first word, either one SCLK before the first bit (normal) or at the same time as the first bit (alternate).

5.10 COMPANDING AND DATA FORMAT

Companding (a contraction of COMpressing and exPANDing) is the process of logarithmically encoding and decoding 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.

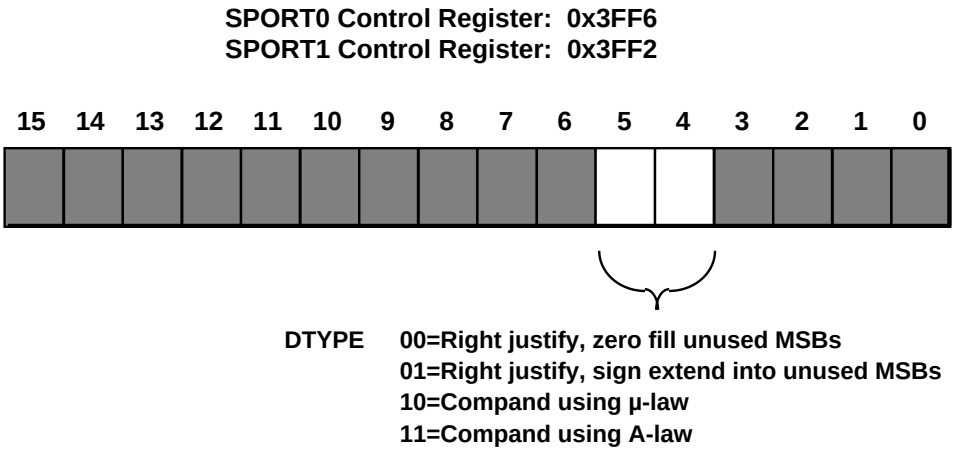


Figure 5.22 DTYPE Field In SPORT Control Register

5 Serial Ports

The ADSP-2100 family of processors supports both of the widely used algorithms for companding: A-law and μ-law. The processor compands data according to the CCITT G.711 recommendation. The type of companding can be selected independently for each SPORT.

If companding is not enabled, there are two formats available for received data words of fewer than 16 bits: one that fills unused MSBs with zeros, and another that sign-extends the MSB into the unused bits.

The type of companding, as well as the non-companding data format, are controlled by the DTYPE field (bits 5-4) in the SPORT control register (0x3FF6 for SPORT0 and 0x3FF2 for SPORT1) as shown in Figure 5.22.

When companding is enabled, valid data in the RX0 or RX1 register is the right-justified, sign-extended, expanded value of the eight LSBs received. Likewise, a write to TX0 or TX1 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 magnitude of the 16-bit value is greater than the 13-bit A-law or 14-bit μ-law maximum, the value is automatically compressed to the maximum positive or negative value.

5.10.1 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. After the MSB 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.

5.10.2 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 latency prior to receiving the interrupt.

5.10.3 Companding Internal Data

Because the values in the RX and TX registers are actually companded “in place” it is possible to use the companding hardware internally, without any transmission or reception at all and without enabling the serial port. This operation can be used for debugging or data conversion and requires a single cycle of overhead.

To compress data, enable companding and then:

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 8-bit compressed data)

The code might look like this:

```
TX0 = AX0;     {linear data written to transmit register}
NOP;           {any instruction}
AX1 = TX0;     {compressed data transferred to AX1}
```

Use the same procedure to expand data, but use RXn instead of TXn.

```
RX0 = AX0;     {compressed data written to receive register}
NOP;           {any instruction}
AX1 = RX0;     {expanded - linear value transferred to AX1}
```

5.11 AutoBuffering

In normal operation, a SPORT generates an interrupt when it has received or has started to transmit a data word. Autobuffering provides a mechanism for receiving or transmitting an entire block of serial data before an interrupt is generated. Service routines can operate on the entire block of data, rather than on a single word, reducing overhead significantly. Autobuffering is available on both SPORT0 and SPORT1, except on the ADSP-21msp58/59 which autobuffers only on SPORT0.

Autobuffering uses the circular buffer addressing capability of the DAGs. With autobuffering enabled, each serial data word is transferred (or if multichannel operation is enabled, each active word is 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.

The autobuffer transfer cannot be duplicated by any instruction. However, an equivalent assembly language instruction would be:

```
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 processor 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 around, the SPORT interrupt is generated. The receive interrupt occurs after the complete buffer has been received. The transmit interrupt occurs when the last word is loaded into TXn, prior to transmission.

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 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; in the case that two or more are needed in the same cycle, they have the following priority, which is the same as the SPORT interrupt priority:

Highest	SPORT0 Transmit
	SPORT0 Receive
	SPORT1 Transmit
Lowest	SPORT1 Receive

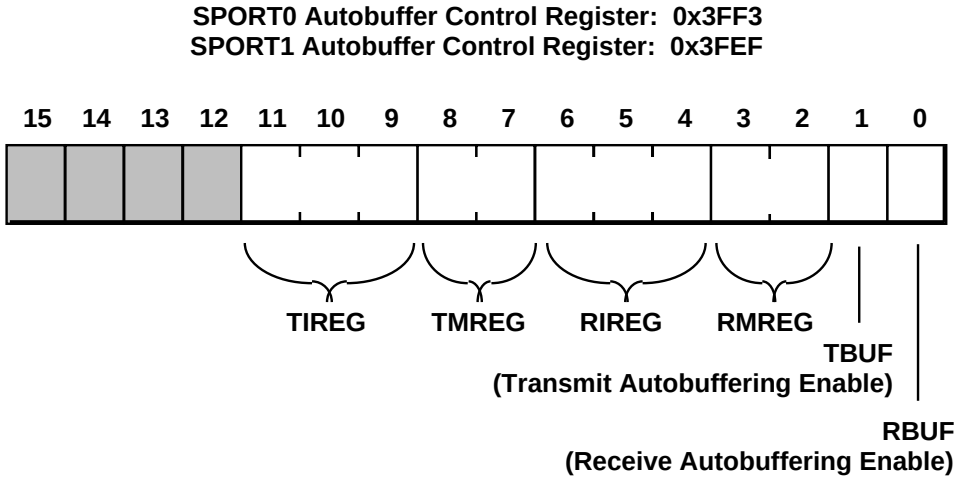


Figure 5.23 SPORT Autobuffer Control Register

In the worst case that all four autobuffer transfers are required at about the same time, interrupt latency would increase by the time it takes for all the transfers to occur, which is affected by wait states and bus request.

5.11.1 Autobuffering Control Register

In autobuffering mode, an interrupt is generated when the modification of a specified I register (in the DAG) by the value in the specified M register (in the DAG) causes a modulus overflow (pointer wraparound). This means that the end of the buffer has been detected.

The autobuffering mode is enabled separately for receiving and transmitting by bits in the SPORT's autobuffer control register (0x3FF3 for SPORT0 or 0x3FEF for SPORT1), shown in Figure 5.23.

The I and M registers used for autobuffering are identified by fields in the autobuffer control register. TIREG and TMREG are binary values that indicate the numbers of the I and M registers, respectively, associated with the transmit buffer. The rules governing the pairing of I and M registers are the same as for other DAG operations: the I and M registers must be in the same DAG, numbered either 0-3 for DAG1 or 4-7 for DAG2. Consequently, three bits identify the I register, but only two bits are necessary to indicate the M register because the third bit (MSB) of the M register number must be the same as for the I register.

Likewise, RIREG and RMREG indicate the numbers of the I and M registers, respectively, associated with the receive buffer.

The TBUF and RBUF bits enable transmit autobuffering and receive autobuffering, respectively. These bits are cleared to zeros at reset and after a reboot. Consequently, autobuffering in progress cannot continue through a reboot operation; you must re-enable autobuffering after a reboot.

5.11.2 Autobuffering Example

The code shown below is an example that sets up SPORT1 for autobuffering operation. The code assumes that the processor is driven with a clock frequency of 12.288 MHz. The SPORT will automatically transmit values from the circular buffer named *tx_buffer*. It will receive values as they are sent to the SPORT and automatically transfer the data into the buffer named *rx_buffer*. A transmit interrupt will be generated once all of the *tx_buffer* values have been transferred to TX1, but before the

last value has been loaded into the transmit shift register. A receive interrupt will be generated once the *rx_buffer* has been completely filled.

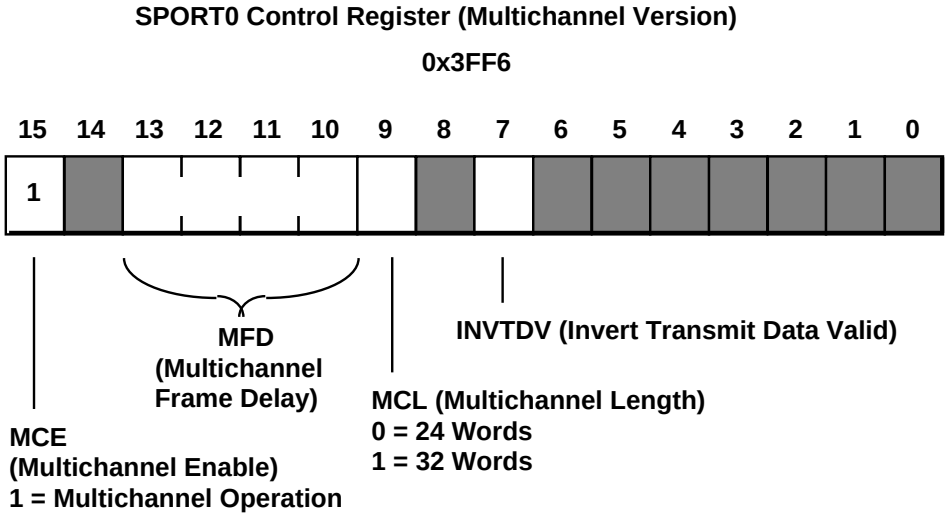
```
.MODULE/RAM    code_to_init_AB_SPORT1;

{——  Initialization code for autobuffer  ——}

.VAR/DM/CIRC   tx_buffer[10];
.VAR/DM/CIRC   rx_buffer[10];
.ENTRY         sport1_inits;

{set up I,M, and L registers}


sport1_inits:  I0 = ^tx_buffer; {I0 contains address of
tx_buffer}
               M0 = 1;          {fill every location}
               L0 = %tx_buffer; {L0 set to length of tx_buffer}

               I1 = ^rx_buffer; {I1 points to rx_buffer}
               L1 = %rx_buffer; {L1 set to length of rx_buffer}

{set up SPORT1 for autobuffering}

               AX0 = 0x0013;     {TX uses I0, M0; RX uses I1, M0}
               DM(0x3FEF) = AX0; {autobuffering enabled}

{set up SPORT1 for 8 kHz sampling and 2.048 MHz SCLK}

               AX0 = 255;       {set RFSDIV to 255 for 8 kHz}
               DM(0x3FF0) = AX0;

               AX0 = 2;     {set SCLKDIV to 2 for 2.048 MHz SCLK}
               DM(0x3FF5) = AX0;

{set up SPORT1 for normal required framing, internal SCLK}
{internal generated framing}

               AX0 = 0x6B27;     {normal framing, 8 bit mu-law}
               DM(0x3FF2) = AX0; {internal clock, framing}

{set up interrupts}

               IFC = 6;    {clear any extraneous SPORT
interrupts}
               ICNTL = 0;  {interrupt nesting disabled}
```

```
                IMASK = 6;  {enable SPORT1 interrupts}

{enable SPORT1}

                AX0 = 0x0C1F;         {enable SPORT1 leave PWAIT,}
                DM(0x3FFF) = AX0;     {BWAIT as default}

{Place first transfer value into TX1}

                AX0 = DM(I0,M0);
                TX1 = AX0;
                RTS;
.ENDMOD;
```

Figure 5.24 Autobuffering Example Configuration Code

5.12 MULTICHANNEL FUNCTION

SPORT0 supports a multichannel function. In the multichannel mode of operation, serial data is time-division multiplexed. Each subsequent word belongs to the next consecutive channel so that, for example, a 24-word block of data contains one word for each of 24 channels. SPORT0 supports 32 or 24 channels and can automatically select words for particular channels while ignoring the others.

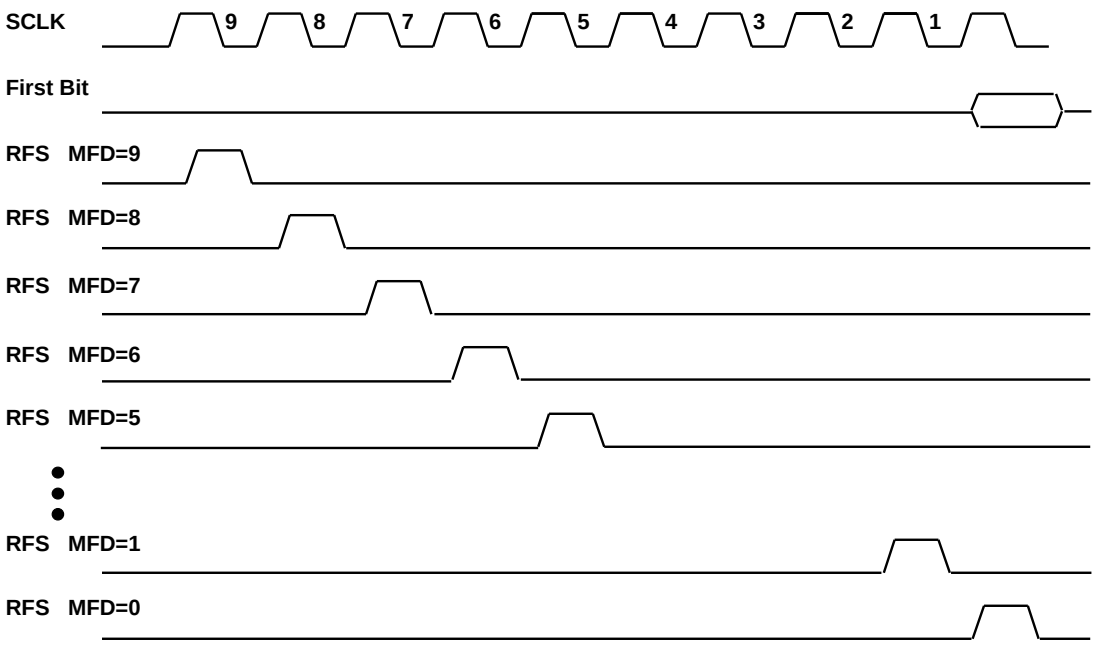


In single-channel mode, receive and transmit framing identifies the start of

a single word or continuous stream, with independent receive and transmit operation. In the multichannel mode, the receive frame sync signal (RFS0) identifies the start of a 24- or 32-word block of serial data with the receiver and transmitter operating in parallel. TFS0 has an alternate function, described below. **Note:** The ADSP-2105 has only one serial port (SPORT1) and does not support multichannel operation.

5.12.1 Multichannel Setup

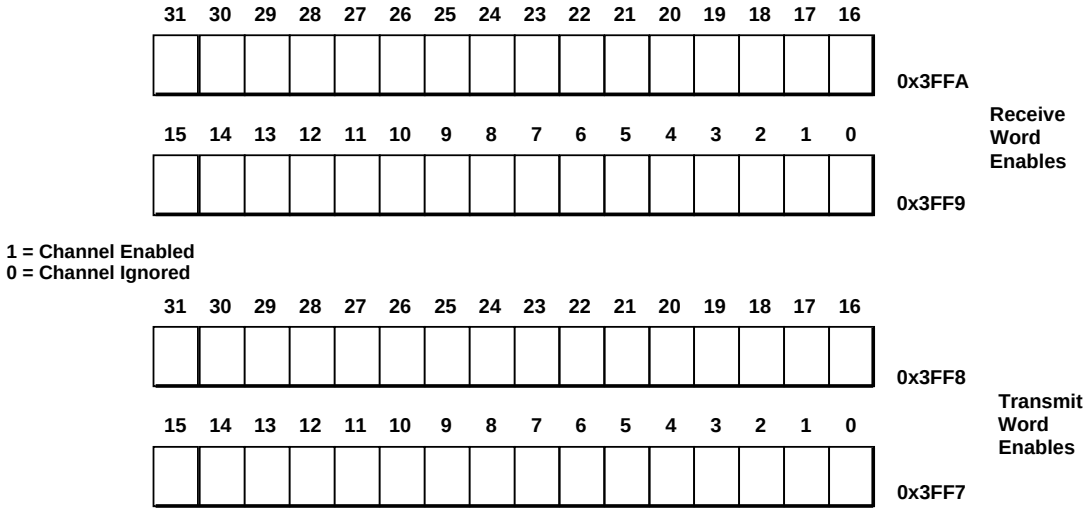
Multichannel operation is enabled by bit 15 in SPORT0's control register (0x3FF6). When this bit is a 1, multichannel mode is enabled, and some



control bits in the SPORT0 control register are redefined. Bits affected by multichannel mode are shown in Figure 5.25. At reset, bit 15 is cleared, disabling multichannel mode and enabling normal operation.

Figure 5.25 SPORT0 Control Register With Multichannel Mode Enabled

The state of the multichannel length bit MCL, bit 9, determines whether there are 24 or 32 channels, i.e. whether the block length is 24 or 32 words. A 0 selects 24-word blocks; a 1, 32-word blocks. In multichannel mode, the



word length is still set by the SLEN field in the SPORT control register and can be 3 to 16 bits.

The multichannel frame delay (MFD) is a 4-bit field specifying (in binary) the number of serial clock cycles between the frame sync signal and the first data bit. This allows the processor to work with different types of T1 interface devices. Figure 5.26 shows a variety of delays.

Figure 5.26 SPORT Multichannel Frame Delay Examples

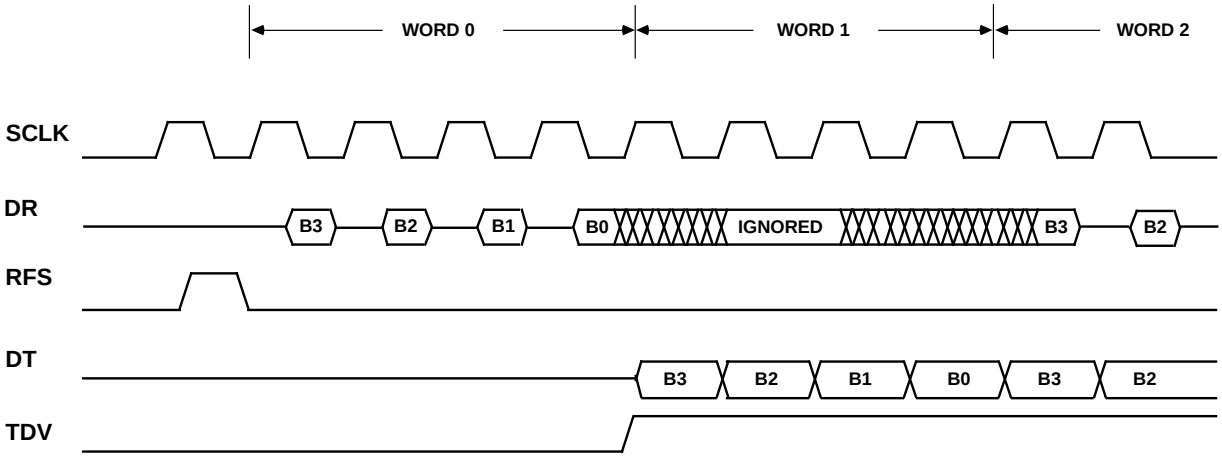
The memory-mapped receive enable register and transmit enable register are each 32 bits wide and made up of two contiguous sixteen-bit registers, as shown in Figure 5.27, which can be found on the next page. Each bit corresponds to a channel; setting the bit enables that channel so that the processor will select its word from the 24- or 32-word block. For example, setting bit 0 selects word 0, bit 12 selects word 12, and so on.

Figure 5.27 SPORT0 Multichannel Word Enable Registers

5.12.2 Multichannel Operation

Received words for channels that are not enabled are ignored; that is, no interrupts are generated for these words, no autobuffering occurs and no data is written to the RX0 register. Likewise, there are no interrupts and no autobuffering for transmit words that are not enabled. During transmit word time slots for channels that are not enabled, the data transmit (DT) pin is tristated.

Most aspects of SPORT0 operate normally in the multichannel mode. Specifically, word length (SLEN), internal or external framing (IRFS),



frame signal inversion (INVRFS), companding (DTYPE) and autobuffering are unchanged in the multichannel mode. **Note:** It is important that RFS does not occur more than once per frame in multichannel mode.

Instead of providing frame synchronization, the TFS0 signal functions as a transmit data valid (TDV) signal in multichannel mode. TDV is asserted while the transmitter is active. TDV can be active high or low, and its polarity is controlled by the INVTFS bit, renamed INVTDV in this context. If INVTDV is a 1, TDV is active low; otherwise it is active high. TDV can be used to enable additional buffer logic, if required.

Figure 5.28 shows the start of a multichannel transfer. As in earlier examples, word length is four bits (SLEN=3) and frame sync signals are active high. Multichannel frame delay (MFD) is one SCLK cycle. For the purpose of illustration, words 0 and 2 are selected for receiving and words 1 and 2 are selected for transmission.

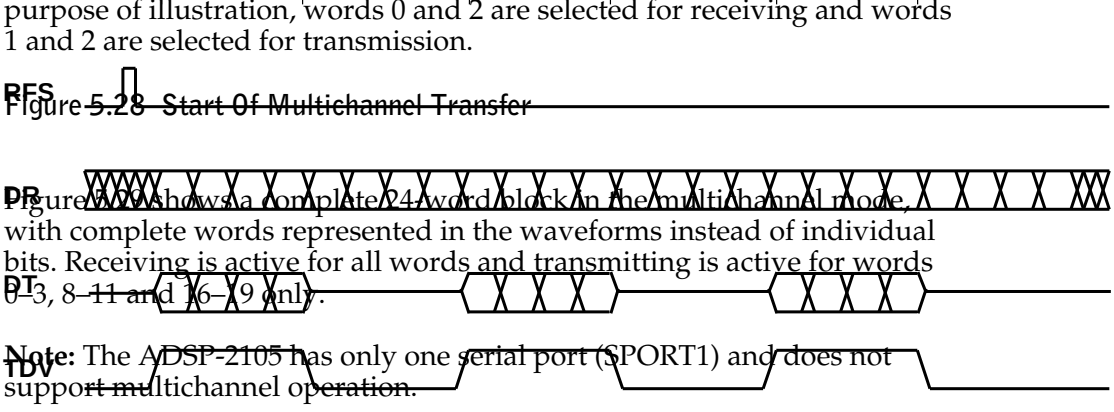


Figure 5.28 Start Of Multichannel Transfer

Figure 5.29 shows a complete 24-word block 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.

Note: The ADSP-2105 has only one serial port (SPORT1) and does not support multichannel operation.

Figure 5.29 Complete Multichannel Example

5.13 SPORT TIMING CONSIDERATIONS

The SPORTs support full duplex operation and are normally interrupt driven. That is, whenever a SPORT transaction has completed, the processor generates an internal interrupt. Under most operating conditions, the actual timing of the SPORT interrupts is not critical. In some sophisticated DSP systems, however, it is important to know the timing of the interrupt relative to the operation of the serial port.

5.13.1 Companding Delay

Use of the companding circuit introduces latency in two ways. First, compressing or expanding a data value takes a single processor cycle. Second, SPORT0 has priority over SPORT1 if both require an expansion or compression operation in the same cycle; in this case, SPORT1 must wait one processor cycle. See the section on companding earlier in this chapter for more details on companding.

5.13.2 Clock Synchronization Delay

Some SPORT timings depend on the processor clock. Other timings depend on the serial clock (SCLK0 or SCLK1). These clocks are asynchronous. There is a delay associated with synchronizing the serial clock to the processor clock whether the serial clock is internally or externally generated. This delay is different for the transmit and receive interrupts, as explained in the following sections.

5.13.2.1 Startup Timing

When a serial port is enabled by a write to the System Control Register, it takes two SCLK cycles before it is actually enabled. On the next (third)

SCLK cycle, the serial port becomes active, looking for a frame sync.

5.13.3 Internally Generarated Frame Sync Timing

When internally generated frame syncs are used, all that is necessary to transmit data, from the programmer's point of view, is to move the data into the appropriate TX register with an instruction such as:

```
TX0 = AX0;
```

Once data is written into the TX register, the processor generates a frame sync after a synchronization delay. This delay in turn affects the timing of the serial port transmit interrupt. The latency depends on five factors: the frequency of the serial clock, whether or not companding is enabled, whether or not there is contention for the companding circuit, whether the current word has finished transmitting and the logic level of the SCLK

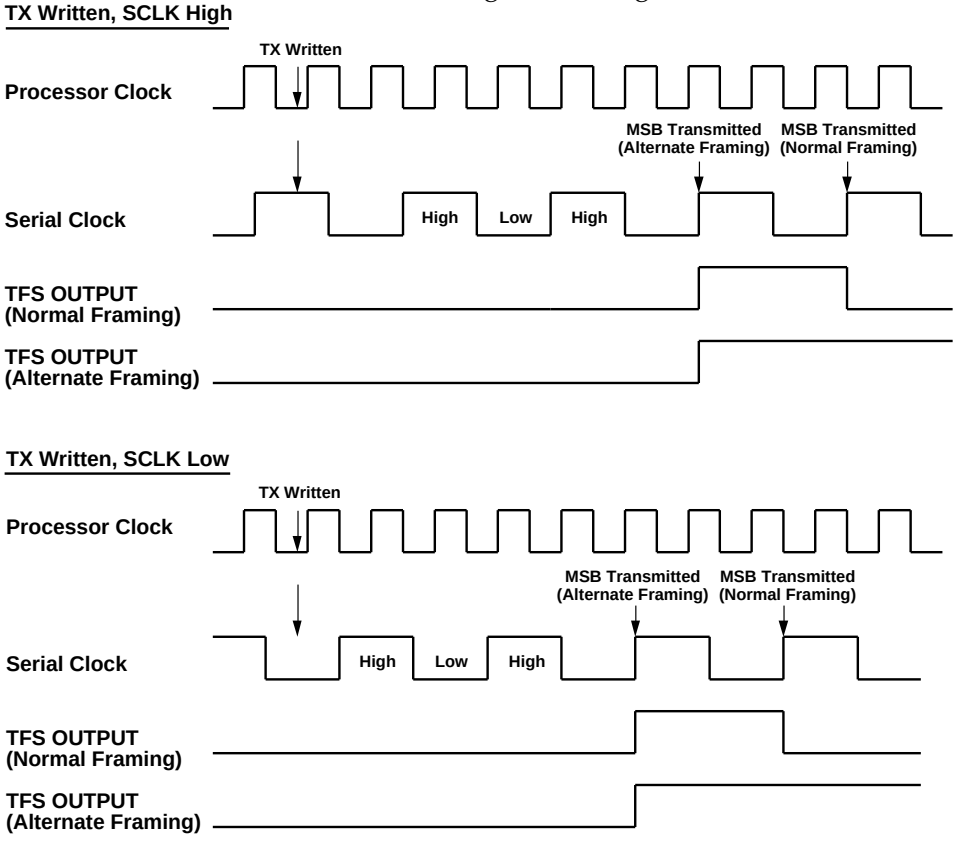


Figure 5.30 Clock Synchronization

when the data value was loaded into the transmit register.

(Note that if the transmit frame sync is generated externally, data starts transmitting when a frame sync signal is received.)

After the TX register is loaded, it takes three complete phases of the serial clock, HIGH, LOW and HIGH, in that order, to ensure synchronization (see Figure 5.30). Once synchronization has been ensured and a frame sync generated, the most significant bit of the transmit word is shifted out on the same rising edge as the frame sync if alternate framing is used and on the rising edge of the next serial clock if normal framing is used. Therefore, the worst-case synchronization delay is two SCLK cycles.

There is additional delay if the previous data transmission has not

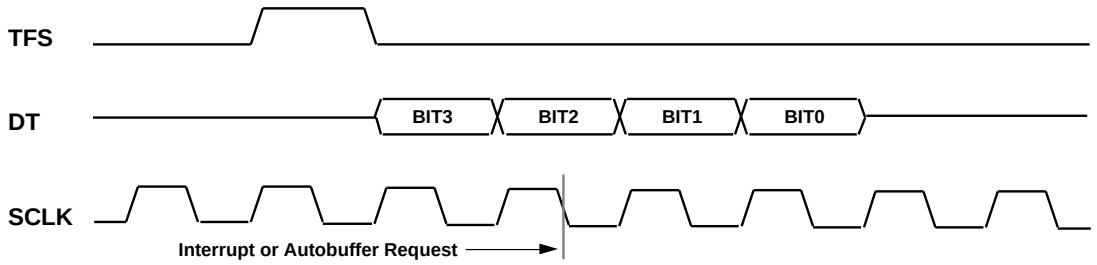


Figure 5.31 SPORT Interrupt or Autobuffer Timing, Transmit 4-Bit Words (No Companding)

completed; the TX register cannot be loaded into the transmit shift register until the previous transmission is complete.

5.13.4 Transmit Interrupt Timing

Once the MSB has been transmitted, the subsequent bits are transmitted on the rising edges of the SCLK. The transmit interrupt (or autobuffer request) is generated internally on the falling edge of SCLK during the transmission of the second bit (see Figure 5.31 below). This timing gives the program time to load the TX register with the next data for continuous data transmission.

The transmit interrupt, like any other interrupt, must be synchronized to the processor clock. Servicing is subject to the same latencies as other interrupts.

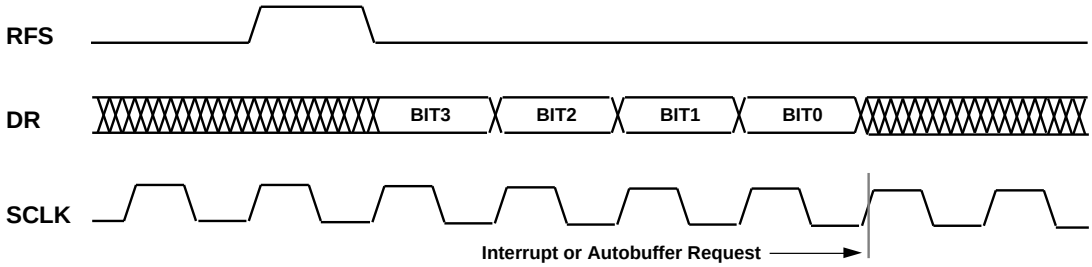


Figure 5.32 SPORT Interrupt or Autobuffer Timing, Receive 4-Bit Words (No Companding)

The transmit interrupt essentially means that it is all right to write a value to the TX register.

5.13.5 Receive Interrupt Timing

The receiver portion of the SPORT latches data on the DR pin on the falling edges of SCLK.

Receive interrupt timing differs from transmit interrupt timing. The receive interrupt or autobuffer request occurs only after an entire word is

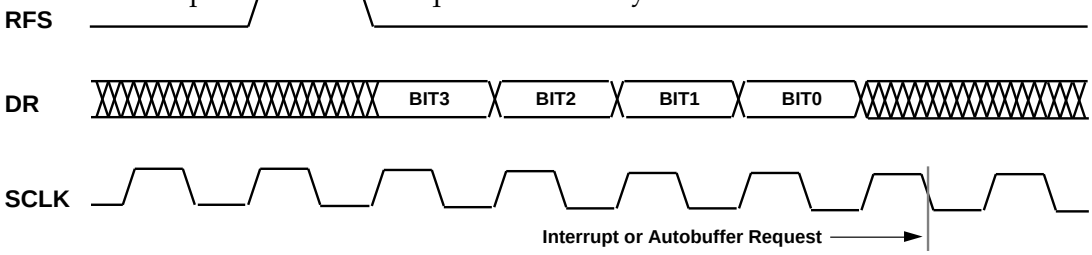


FIgure 5.33 SPORT Interrupt or Autobuffer Timing, Receive 4-Bit Words (Companding Enabled)

received. The interrupt request occurs on the rising edge of SCLK after a word is received (see Figure 5.32) and indicates that new data in the RX register can be read.

Companding causes a delay in the same manner as for transmitting. However, the latency is transparent, as the receive interrupt is generated after the expansion has taken place.

The LSB is received on the falling edge of SCLK. One processor cycle elapses to allow synchronization to the processor clock. One processor

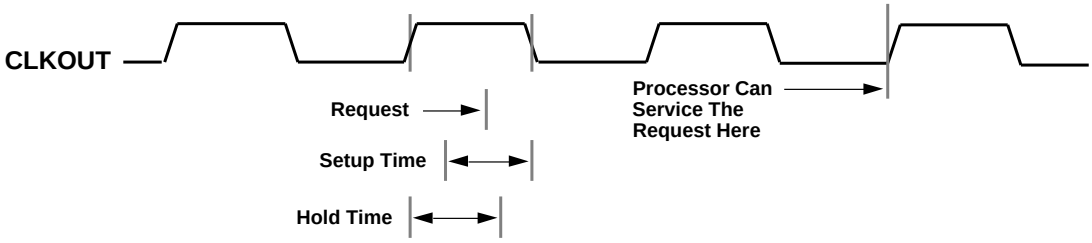


Figure 5.34 Synchronization of Autobuffer or Interrupt Request to Processor Clock

cycle later, the SPORT attempts to expand the data if companding is enabled and the other serial port is not using the companding circuitry. Companding latencies as discussed above occur prior to generation of a receive interrupt. Servicing the receive interrupt is subject to the same latencies as other interrupts.

5.13.6 Interrupt & Autobuffer Synchronization

The serial ports are treated as an asynchronous system to the processor, even if the processor is providing the serial clock. Internal to the processor is a circuit which synchronizes the autobuffer or interrupt requests to the processor clock. Figure 5.34 shows the synchronization delay for the serial ports, assuming the setup and hold times are met for the current processor cycle. The setup and hold times for the serial port requests are the same as shown on the data sheet for the IRQ2 signal. If the setup and hold times are not met, there is an additional processor cycle of delay added.

As shown in Figure 5.34, there is a two-processor-cycle delay before the autobuffer or interrupt request is acted on by the processor. The same

latencies exist for all external interrupts. The processor can only service interrupt or autobuffer requests on instruction cycle boundaries, so there may be additional latency cycles added due to the completion of an instruction.

5.13.7 Instruction Completion Latencies

There are several situations which can cause an instruction to take more than one processor cycle. Any of the following can delay the processor's ability to service a pending interrupt or autobuffer request:

- External memory wait states
- Bus request when an external access is required (in go-mode)
- Bus request with go-mode disabled
- Multiple external accesses required for a single instruction

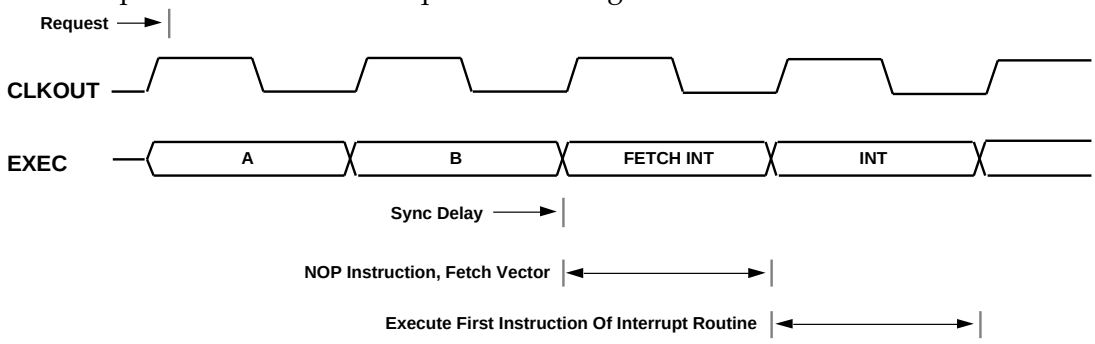


Figure 5.35 Interrupt Service Example

- A pending higher priority autobuffer or interrupt request
- Interrupt being masked

On instruction cycle boundaries the processor will service multiple pending interrupt or autobuffer requests in the following priority order:

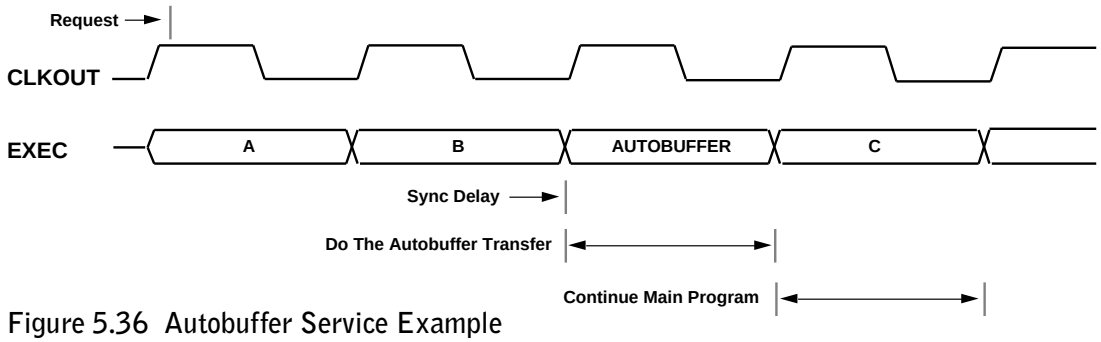


Figure 5.36 Autobuffer Service Example

- SPORT0 transmit autobuffer—highest priority *(not on ADSP-2105)*
- SPORT0 receive autobuffer *(not on ADSP-2105)*
- SPORT1 transmit autobuffer
- SPORT1 receive autobuffer
- Unmasked pending interrupts in priority order

5.13.8 Interrupt & Autobuffer Service Example

Figure 5.35 shows the execution of a serial port interrupt based on a request that meets the setup and hold time requirements. This example is the same for a receive or a transmit interrupt request.

An additional latency cycle is consumed due to the fetching of the first instruction of the interrupt routine. The interrupt can only be serviced on an instruction cycle boundary. The above example (in Figure 5.35) assumes all instructions are completed in one processor cycle. Figure 5.36 shows the result of an autobuffer request that meets the setup and hold requirements.

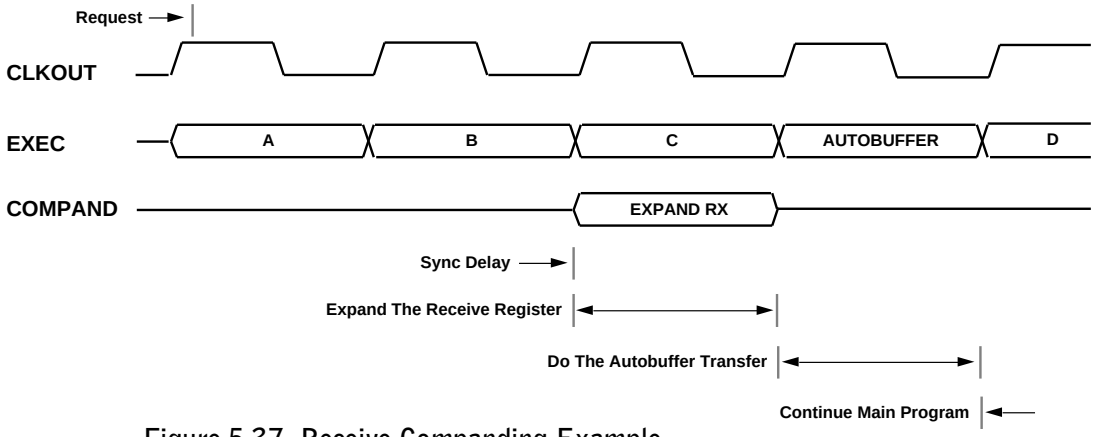


Figure 5.37 Receive Companding Example

Autobuffering only consumes the cycles necessary to perform the data transfer; no additional cycles are lost fetching instructions. The above diagram assumes that all instructions and data transfers occur in one

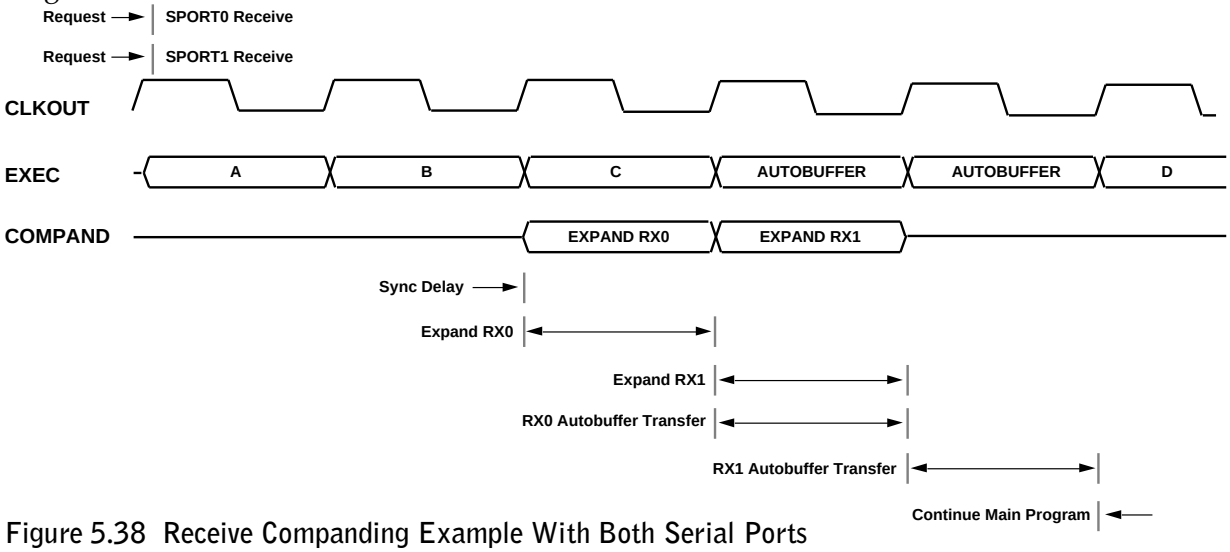


Figure 5.38 Receive Companding Example With Both Serial Ports

processor cycle.

5.13.9 Receive Companding Latency

In addition to the cycles used for synchronization, there are some additional delays possible due to receive companding. The synchronized request is used by the processor to decide when to write the receive

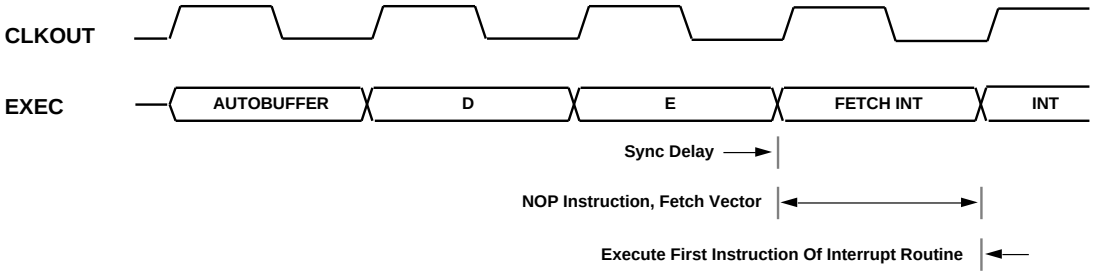


Figure 5.39 Autobuffering Interrupt Example

register with the expanded value. This can only occur on instruction cycle boundaries and only one receive register can be expanded at a time. On the ADSP-2100 family processors that have two serial ports (i.e. all except the ADSP-2105), there is also a possibility of a delay due to the availability of the companding circuitry. SPORT0 has the higher priority. When companding is enabled, the autobuffer or interrupt request does not occur until the register has been expanded. The next two diagrams show examples of autobuffering with companding and the latencies involved.

The following diagram shows the latency when there are two pending

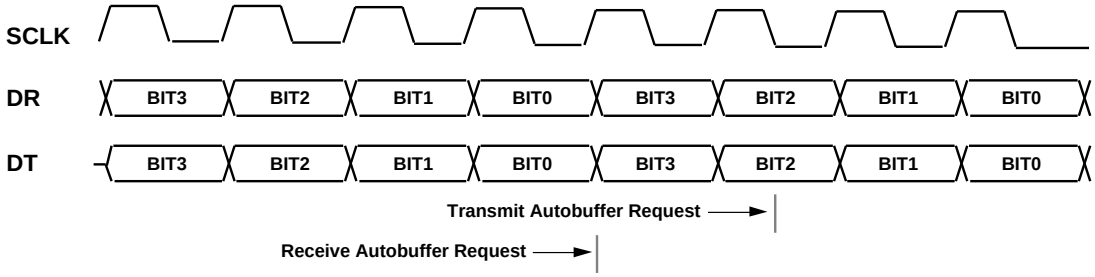


Figure 5.40 Using One Index Register for Transmit and Receive Autobuffer

receive autobuffer requests with companding enabled.

5.13.10 Interrupts With Autobuffering Enabled

When autobuffering is enabled, SPORT interrupts occur when the address modification done during the autobuffer operation causes a modulus wraparound. The synchronization delay applies to this type of interrupt as well. An example is shown below in Figure 5.39:

5.13.11 Unusual Complications

In most cases the serial port companding, autobuffer, and interrupt latencies are transparent to your application program. When trying to use the same I register for more than one autobuffer channel, it becomes important to make sure that the latencies do not effect the correct order of operations. For example, if the serial port data is continuous, and the receiver and transmitter are working with the same frame signal, the order of the transmit and receive autobuffer or interrupt operations may be affected by the latencies shown below in Figure 5.40.

If the processor is free to handle the autobuffer requests in the order they are generated, the receive autobuffer happens first and is then followed by the transmit autobuffer. The order of these operations may change if the processor is not available to handle the requests due to any of the previously mentioned latencies. In this case there are 1½ serial clock cycles between the requests. If the processor is subject to bus requests, wait states, or other latencies which are longer than 1½ serial clock cycles, both autobuffer operations may be held off. Since the transmit autobuffer has a higher priority, it's request will occur first. Because of the priority of the autobuffer requests the use of a single I register more difficult or even impossible in some cases. As long as there are no possible latency cases longer than the difference in the timing of the requests, it is quite possible to use a single I register for serial port autobuffering.

Timer 6

6.1 OVERVIEW

The 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 timer allows the 16-bit counter to generate periodic interrupts over a wide range of periods. Given a processor cycle time of 80 ns, the timer can generate interrupts with periods of 80 ns up to 5.24 ms 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 3, "Program Control."

6.2 TIMER ARCHITECTURE

The 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 Chapter 15, Instruction Set Reference. The timer registers, which are memory-mapped, are shown in Figure 6.1 (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.

6 Timer

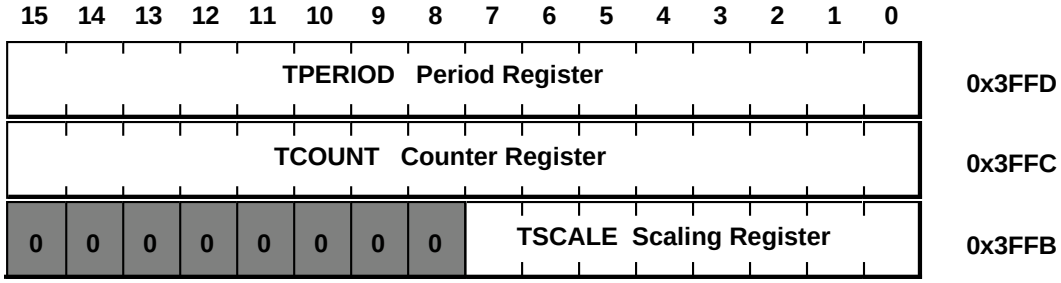


Figure 6.1 Timer Registers

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 6.2 shows the timer block diagram.

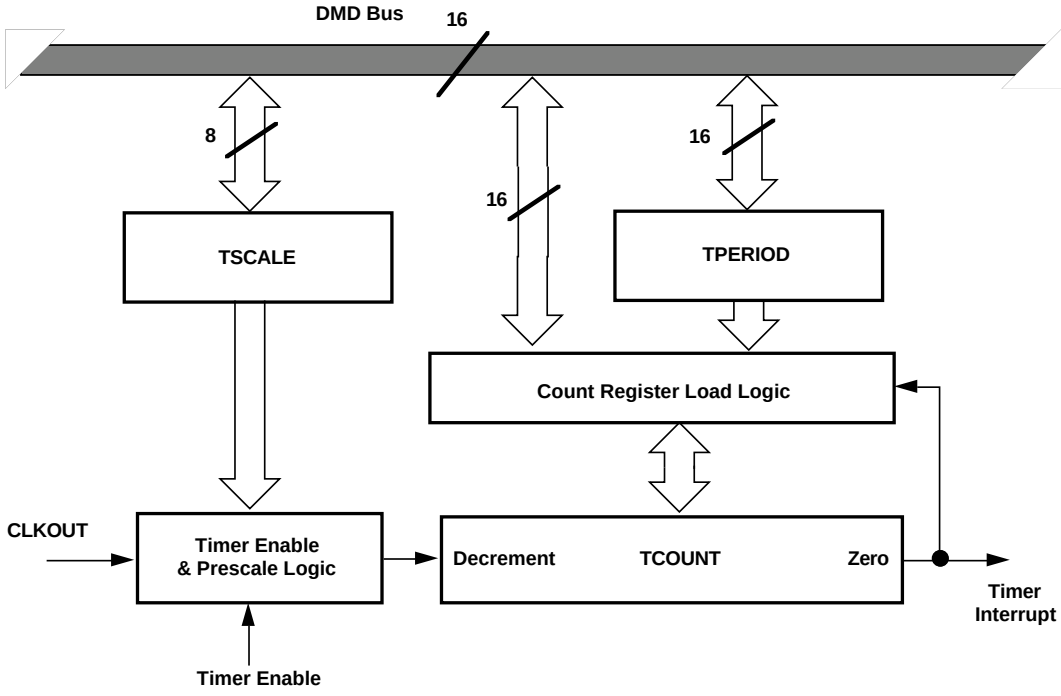


Figure 6.2 Timer Block Diagram

6.3 RESOLUTION

TSCALE provides the capability to program longer time intervals between interrupts, extending the range of the 16-bit TCOUNT register. Table 6.1 shows the range and the relationship between period length and resolution for TPERIOD = maximum.

Cycle Time = 80 ns

TSCALE	*Interrupt Every…*	*Resolution*
0	5.24 ms	80 ns
255	1.34 s	20 μs

Table 6.1 Timer Range And Resolution

6.4 TIMER OPERATION

Table 6.2 shows the effect of operating the timer with TPERIOD = 5, TSCALE = 1 and TCOUNT = 5. After the timer is enabled (cycle n–1) the counter begins. Because TSCALE is 1, TCOUNT is decremented on every other cycle. 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 6.2 Example Of Timer Operation

6 Timer

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.

Host Interface Port 7

7.1 OVERVIEW

The host interface port (HIP) of the ADSP-2111, ADSP-2171, and ADSP-21msp58/59 is a parallel I/O port that allows these processors to be used as memory-mapped peripherals of a host computer (i.e. slave DSP processors). Examples of host computers include the Intel 8051, Motorola 68000 family, and even other ADSP-21xx processors.

The host interface port can be thought of as an area of dual-ported memory, or mailbox registers, that allow communication between the host and the processor core of the ADSP-21xx. The host addresses the HIP as a segment of 8- or 16-bit words of memory. To the processor core, the HIP is a group of eight data-memory-mapped registers.

Any number of ADSP-21xx processors can be used in parallel as memory-mapped peripherals. Assigning a different address location to each one allows the host to control them all.

The operating speed of the HIP is similar to that of the processor data bus. A read or write operation can occur within a single instruction cycle. Because the HIP is normally connected with devices that are much slower (the 68000, for example, can take four cycles to perform a bus operation), the data transfer rate is usually limited by the host computer.

The host interface port is completely asynchronous to the rest of the ADSP-21xx's operations. The host can write data to or read data from the HIP while the ADSP-21xx is operating at full speed. The HIP can be configured for operation on an 8-bit or 16-bit data bus and for either a multiplexed address/data bus or separate address and data buses.

The ADSP-2111, ADSP-2171, and ADSP-21msp58/59 support two types of booting operations. One method boots from external memory (usually EPROM) using the boot memory interface described in the "Memory Interface" chapter. The other method uses the HIP to boot load a program from the host computer. HIP booting is described at the end of this chapter.

7 Host Interface Port

7.2 HIP PIN SUMMARY

The HIP consists of 27 pins. As shown in Table 7.1, 16 of these are data pins and 11 are control pins. Some of the control pins have dual functions, allowing the processor to support different bus protocols.

Pin Name	*Number of Pins*	*Direction*	*Function*
HSEL	1	Input	HIP Select
HACK	1	Output	HIP Acknowledge
HSIZE	1	Input	HIP 8/16 Bit Host 0=16-bit; 1=8-bit
BMODE	1	Input	HIP Boot Mode Select 0=normal (EPROM); 1=HIP
HMD0	1	Input	HIP Bus Strobe Select 0=RD, WR; 1=RW, DS
HRD/HRW *	1	Input	HIP **Read Strobe**/ Read/Write Select
HWR/HDS *	1	Input	HIP **Write Strobe**/ Host Data Strobe
HMD1	1	Input	HIP Address/Data Mode 0=separate; 1=multiplexed
HD15-0/HAD15-0 **	16	Bidirectional	HIP **Data**/Address & Data
HA2 /ALE **	1	Input	HIP **Host Address 2**/ Address Latch Enable
HA1-0/no function **	2	Input	**Host Addresses 1 & 0**
TOTAL	**27**		

* HMD0 selects function
** HMD1 selects function

Table 7.1 Host Interface Port Pins

HSEL is a host select which allows the host to enable or disable the HIP for host data transfers.

HACK is a host acknowledge output for hosts that require an acknowledge for handshaking.

HSIZE configures the bus size; the HIP can function in both 8-bit and 16-bit modes. If the HIP is configured for an 8-bit host (HSIZE=1), data is read from and written to the lower eight bits of a HIP data register and the upper eight bits are zero-filled (on host writes) or tristated (on host reads).

BMODE determines whether booting occurs through the HIP or through the memory interface pins.

HMD0 and HMD1 are mode pins that configure the address, data and strobe pins, as shown in Table 7.2. HMD0 configures the bus strobes, selecting either separate read and write strobes or a single read/write select and a host data strobe. HMD1 configures the bus protocol, selecting either separate address (3-bit) and data (16-bit) buses or a multiplexed 16-bit address/data bus with address latch enable. The timings of each of the four bus protocols are described later in this chapter.

	HMD1=0		HMD1=1	
HMD0=0	HRD	HIP Read Strobe	HRD	HIP Read Strobe
	HWR	HIP Write Strobe	HWR	HIP Write Strobe
	HD15-0	HIP Data	HAD15-0	HIP Address/Data
	HA2-0	HIP Address	ALE	HIP Address Latch Enable
HMD0=1	HRW	HIP Read/Write Select	HRW	HIP Read/Write Select
	HDS	HIP Data Strobe	HDS	HIP Data Strobe
	HD15-0	HIP Data	HAD15-0	HIP Address/Data
	HA2-0	HIP Address	ALE	HIP Address Latch Enable

Table 7.2 HIP Configuration Modes

7 Host Interface Port

The functions of the following pins are determined by HMD0 and HMD1 as described above:

HD15-0/HAD15-0 are either a data bus or a multiplexed address/data bus. (Only the 3 least significant address bits are used.)

HRD/HRW is either a read strobe or a read/write select (1=read, 0=write).

HWR/HDS is either a write strobe or a data strobe.

HA2/ALE is either the most significant host address bit or an address latch enable.

HA1-0 are either the two least significant host address bits or are unused.

7.3 HIP FUNCTIONAL DESCRIPTION

The HIP consists of three functional blocks, shown in Figure 7.1: a host control interface block (HCI), a block of six data registers (HDR5-0) and a block of two status registers (HSR7-6). The HIP also includes an associated HMASK register for masking interrupts generated by the HIP. The HCI provides the control for reading and writing the host registers. The two status registers provide status information to both the host and the ADSP-21xx core.

The HIP data registers HDR5-0 are memory-mapped into internal data memory at locations 0x3FE0 (HDR0) to 0x3FE5 (HDR5). These registers can be thought of as a block of dual-ported memory. None of the HDRs are dedicated to either direction; they can be read or written by either the host or the ADSP-21xx. When the host reads an HDR register, a maskable HIP read interrupt is generated. When the host writes an HDR, a maskable HIP write interrupt is generated.

The read/write status of the HDRs is also stored in the HSR registers. These status registers can be used to poll HDR status. Thus, data transfers through the HIP can be managed by using either interrupts or a polling scheme, described later in this chapter.

Host Interface Port 7

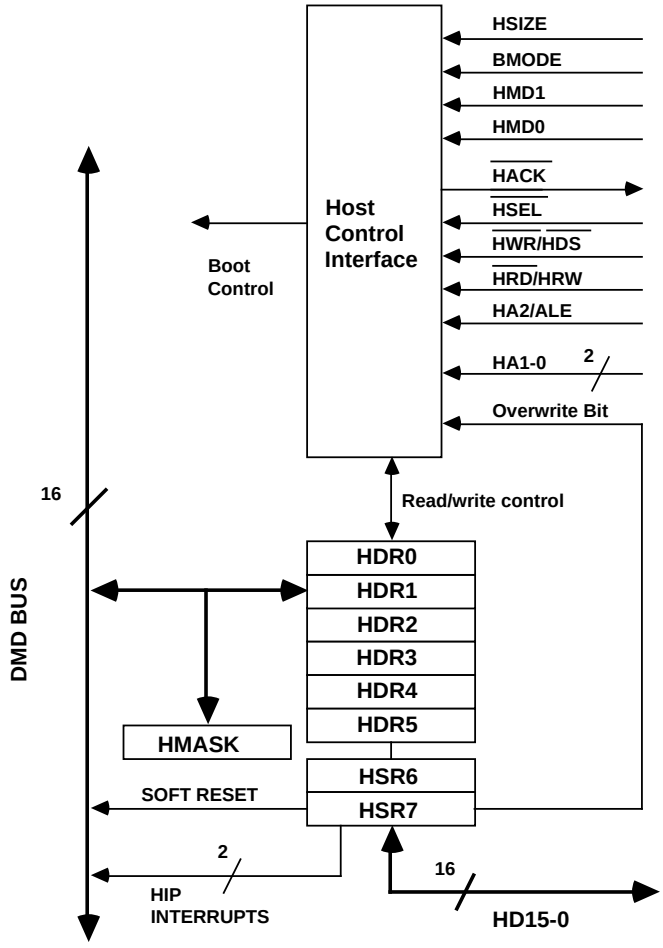


Figure 7.1 HIP Block Diagram

The HSR registers are shown in Figure 7.2, which can be found on the following page. Status information in HSR6 and HSR7 shows which HDRs have been written. The lower byte of HSR6 shows which HDRs have been written by the host computer. The upper byte of the HSR6 shows which HDRs have been written by the ADSP-21xx. When an HDR register is read, the corresponding HSR bit is cleared.

7 Host Interface Port

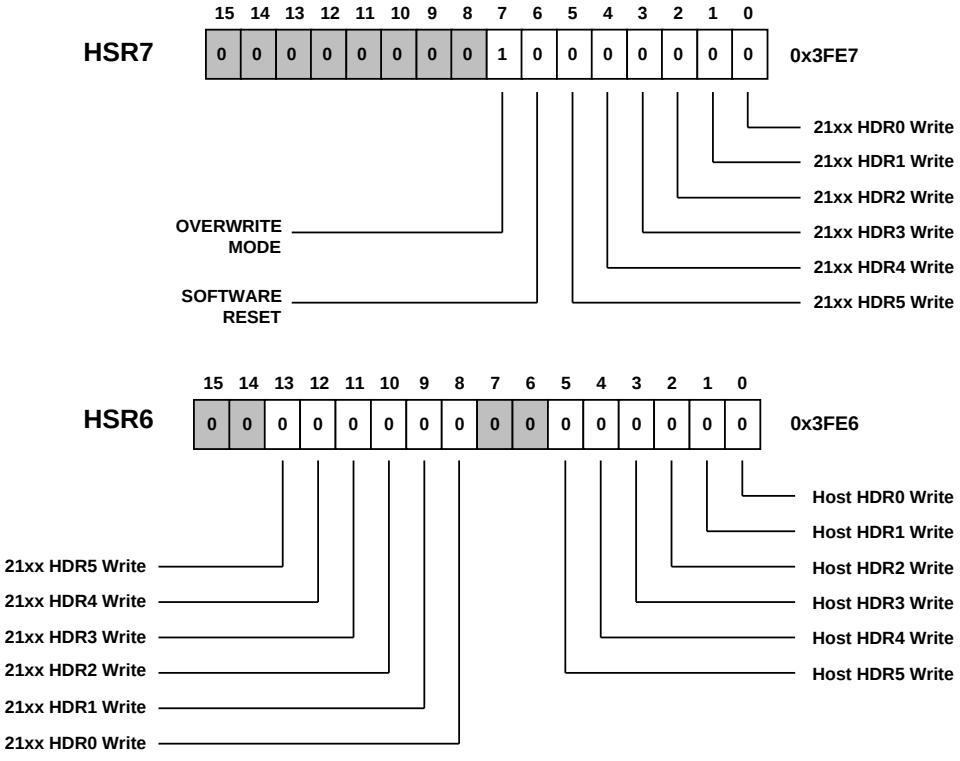


Figure 7.2 HIP Status Registers

The lower six bits of HSR7 are copied from the upper byte of HSR6 so that 8-bit hosts can read both sets of status. Bits 7 and 6 of HSR7 control the overwrite mode and software reset, respectively; these functions are described later in this chapter. The upper byte of HSR7 is reserved. All reserved bits and the software reset bit read as zeros. The overwrite bit is the only bit in the HSRs that can be both written and read. At reset, all HSR bits are zeros except for the overwrite bit, which is a one.

7.4 HIP OPERATION

The ADSP-21xx core can place a data value into one of the HDRs for retrieval by the host computer. Similarly, the host computer can place a data value into one of the HDRs for retrieval by the ADSP-21xx. To the host computer, the HDRs function as a section of memory. To the ADSP-21xx, the HDRs are memory-mapped registers, part of the internal data memory space.

Because the HIP typically communicates with a host computer that has both a slower instruction rate and a multicycle bus cycle, the host computer is usually the limiting factor in the speed of HIP transfers. During a transfer, the ADSP-21xx executes instructions normally, independent of HIP operation. This is true even during a multicycle transfer from the host.

For host computers that require handshaking, the ADSP-21xx returns HACK in the same cycle as the host access, except in overwrite mode. In overwrite mode, the ADSP-21xx can extend a host access by not asserting the HACK acknowledge until the cycle is complete. The user can enable and disable overwrite mode by setting and clearing a bit in HSR7. Overwrite mode is described in more detail later in this chapter.

The HDRs are not initialized during either hardware or software reset. The host can write information to the HDRs before a reset, and the ADSP-21xx can read this information after the reset is finished. During reset, however, HIP transfers cannot occur; the HACK pin is deasserted and the data pins are tristated.

Because a host computer that requires handshaking must wait for an acknowledgement from the ADSP-21xx, it is possible to cause such a host to hang. If, when the host has initiated a transfer, but has not yet received an acknowledgement, the ADSP-21xx is reset, then the acknowledgement can not be generated, thus causing the host to wait indefinitely.

There is no hardware in the HIP to prevent the host from writing a register that the ADSP-21xx core is reading (or vice versa). If the host and the ADSP-21xx try to write the same register at the same time, the host takes precedence. Simultaneous writes should be avoided, however: since the ADSP-21xx and the host operate asynchronously, simultaneous writes can cause unpredictable results.

7.4.1 Polled Operation

Polling is one method of transferring data between the host and the ADSP-21xx. Every time the host writes to an HDR, a bit is automatically set in the lower byte of HSR6. This bit remains set until the ADSP-21xx reads the HDR. Similarly, when the ADSP-21xx writes to an HDR, a bit in the upper byte of HSR6 (and the lower byte of HSR7) is set. This bit is cleared automatically when the host reads the HDR.

For example, the ADSP-21xx can wait in a loop reading an HSR bit to see if the host has written new data. When the ADSP-21xx sees that the bit is set, it conditionally jumps out of the loop, processes the new data, then returns to the loop. When transferring data to the host, the ADSP-21xx waits for the host to read the last data written so that new data can be transferred. The host polls the HSR bits to see when the new data is available.

7.4.1.1 HIP Status Synchronization

Processes running on the ADSP-21xx are asynchronous to processes running on the host. Values in the shared status registers (HSR6, HSR7) can therefore change at any time, and reading a changing value could give unpredictable results. The ADSP-21xx HIP, however, includes synchronization circuitry which guarantees that the HIP status is constant during a read by either the ADSP-21xx core or the host. This synchronization is illustrated in Figures 7.3 and 7.4. The status registers are updated by the ADSP-21xx and thus are synchronous with the ADSP-21xx processor clock, but host accesses are asynchronous with respect to the ADSP-21xx clock.

When the host reads HSR6 or HSR7 to obtain status information, there is a one-cycle synchronization delay before the current (i.e. updated) status is available. *To obtain the correct, current status, therefore, the host must perform*

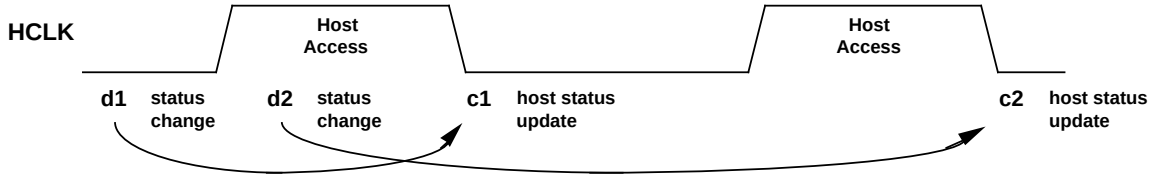


Figure 7.3 Host Status Synchronization

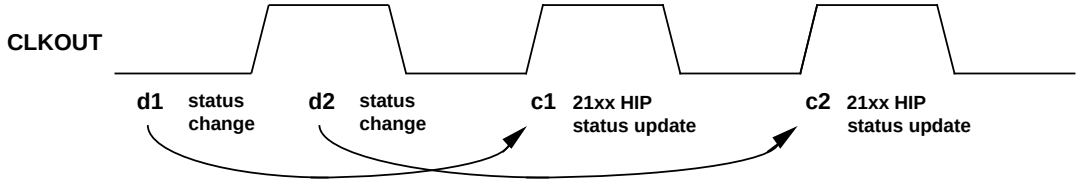


Figure 7.4 ADSP-21xx HIP Status Synchronization

two consecutive reads—the second read will generate the correct status information (the first read generates the previous status).

In Figure 7.3, host status synchronization is based on a pseudo-clock HCLK, internal to the ADSP-21xx, which is a logical combination of HRD, HWR and HSEL. The first event shown in the figure is a status change at d1. The host status will then be updated after the HCLK low, HCLK high, HCLK low sequence at point c1. A status change at d2 would wait for the HCLK low, HCLK high, HCLK low sequence, and then host status would be updated at point c2.

Status synchronization for the ADSP-21xx requires one full CLKOUT cycle (starting at the rising edge) after a status change. As shown in Figure 7.4, a status change at point d1 would cause a 21xx HIP status update at c1. A status change at d2 would cause a 21xx HIP status update at c2.

7.4.2 Interrupt-Driven Operation

Using an interrupt-driven protocol frees the host and the ADSP-21xx from polling the HSR(s) to see when data is ready to be read. For interrupt-driven transfers to the ADSP-21xx, the host writes data into an HDR, and the HIP automatically generates an internal interrupt. The interrupt is serviced like any other interrupt.

For transfers to the host, the ADSP-21xx writes data to an HDR, then sets a flag output, which is connected to a host interrupt input, to signal the host that new data is ready to be transferred. Flag outputs are discussed in detail in Chapter 9, "System Interface." If the ADSP-21xx passes data to the host through only one HDR, then that HDR can be read directly by the host when it receives the interrupt. If more than one HDR is used to pass data, then the host must read the appropriate HSR(s) to determine which HDR was written by the ADSP-21xx.

7.4.3 HDR Overwrite Mode

In most cases, the ADSP-21xx reads host data sent through the HIP faster than the host can send them. However, if the host is sufficiently fast, if the ADSP-21xx is busy, or if the ADSP-21xx is driven by a slow clock, there may be a delay in servicing a host write interrupt. If the host computer uses a handshaking protocol requiring the ADSP-21xx to assert HACK to complete a host transfer, the ADSP-21xx can optionally hold off the next host write until it has processed the current one.

If the HDR overwrite bit (bit 7 in HSR7) is cleared, and if the host tries to write to a register before it has been read by the ADSP-21xx, HACK is not asserted until the ADSP-21xx has read the previously written data. The host processor must wait for HACK to be asserted. As described earlier, however, there is a delay from when the host writes data to when the status is synchronized to the ADSP-21xx. During this interval, it is possible for the host to write an HDR a second time even when the overwrite bit is cleared.

If the HDR overwrite bit is set, the previous value in the HDR is overwritten and HACK is returned immediately. If the ADSP-21xx is reading the register that is being overwritten, the result is unpredictable.

After reset, the HDR overwrite bit is set. If the host does not require an acknowledge (HACK is not used), the HDR overwrite bit should be always be set, because there is no way for the ADSP-21xx to prevent overwrite.

7.4.4 Software Reset

Writing a 1 to bit 6 of HSR7 causes software reset of the ADSP-21xx. If the ADSP-21xx writes the software reset bit, the reset happens immediately. Otherwise, the reset happens as soon as the write is synchronized to the ADSP-21xx system clock. The internal software reset signal is held for five ADSP-21xx clock cycles and then released.

7.5 HIP INTERRUPTS

HIP interrupts can be masked using either the IMASK register or the HMASK register. Bits in the IMASK register enable or disable all HIP read interrupts or all HIP write interrupts. The HMASK register, on the other hand, has bits for masking the generation of read and write interrupts for individual HDRs. In order for a read or write of an HDR to cause an interrupt, the HIP read or write interrupt must be enabled in IMASK, and the read or write to the particular HDR must be enabled in HMASK. HMASK is mapped to memory location 0x3FE8. IMASK is described in Chapter 3, “Program Control.”

A host write interrupt is generated whenever the host completes a write to an HDR. A host read interrupt is generated when an HDR is ready to receive data from the ADSP-21xx—this occurs when the host has read the previous data, and also after reset, before the ADSP-21xx has written any data to the HDR. HMASK, however masks all HIP interrupts at reset. The read interrupt allows the ADSP-21xx to transfer data to the host at a high rate without tying up the ADSP-21xx with polling overhead.

HMASK allows reads and writes of some HDRs to not generate interrupts. For example, a system might use HDR2 and HDR1 for data values and HDR0 for a command value. Host write interrupts from HDR2 and HDR1 would be masked off, but the write interrupt from HDR0 would be unmasked, so that when the host wrote a command value, the ADSP-21xx would process the command. In this way, the overhead of servicing

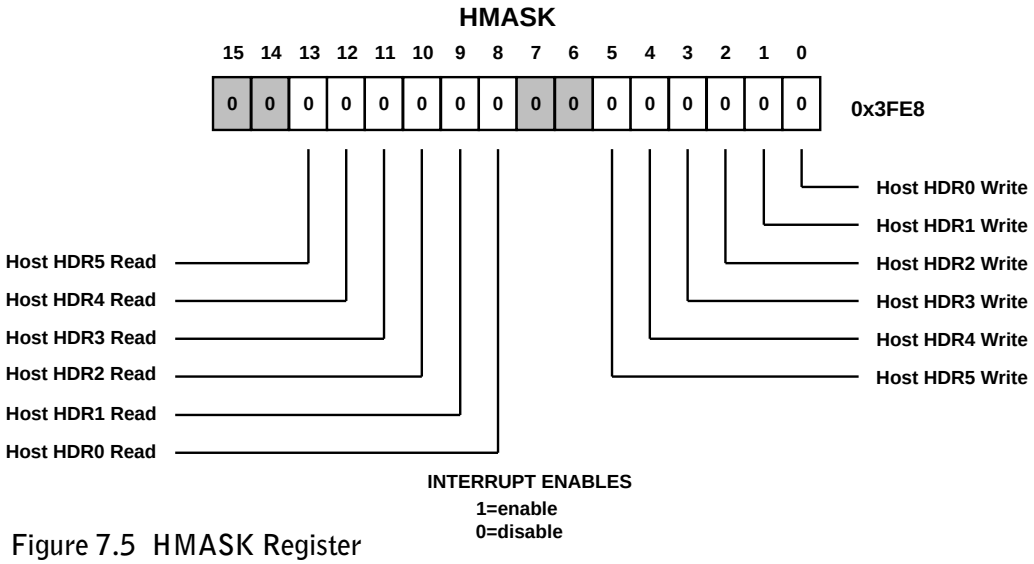


Figure 7.5 HMASK Register

interrupts when the host writes data values is avoided.

The HMASK register is organized in the same way as HSR6; the mask bit is in the same location as the status bit for the corresponding register. The lower byte of HMASK masks host write interrupts and the upper byte masks host read interrupts. The bits are all positive sense (0=masked, 1=enabled).

HMASK is mapped to the internal data memory space at location 0x3FE8. At reset, the HMASK register is all zeros, which means that all HIP interrupts are masked.

HIP read and write interrupts are not cleared by servicing such an interrupt. Reading the HDR clears a write interrupt, and writing the HDR clears a read interrupt. The logical combination of all read and write interrupt requests generates a HIP interrupt. Pending interrupt requests remain until all HIP interrupts are cleared by either reading or writing the appropriate HIP data register. If the ADSP-21xx is reading registers that the host might be writing, it is not certain that an interrupt will be generated. To ensure that all host writes generate interrupts, you must make sure that the ADSP-21xx is not reading the HDRs that the host is writing. While servicing the interrupt, the status register can be read to determine which operation generated the interrupt and whether multiple interrupt requests need to be serviced.

HIP interrupts cannot be forced or cleared by software, as other interrupts can. The HIP write interrupt vector is location 0x0008. The HIP read interrupt vector is location 0x000C.

7 Host Interface Port

7.6 HOST INTERFACE TIMING

The following diagrams show the timings of HIP signals in the various modes determined by HMD0 and HMD1. HMD0 configures the bus strobes, selecting either separate read and write strobes or a single read/write select and a host data strobe. HMD1 configures the bus protocol, selecting either separate address (3-bit) and data (16-bit) buses or a multiplexed 16-bit address/data bus with address latch enable. The HSIZE pin can be changed on a cycle-by-cycle basis; although not shown in the following diagrams, it has the same timing as the HRD/HRW signal.

Figure 7.6 shows the HIP timing when both HMD0=0 and HMD1=0.

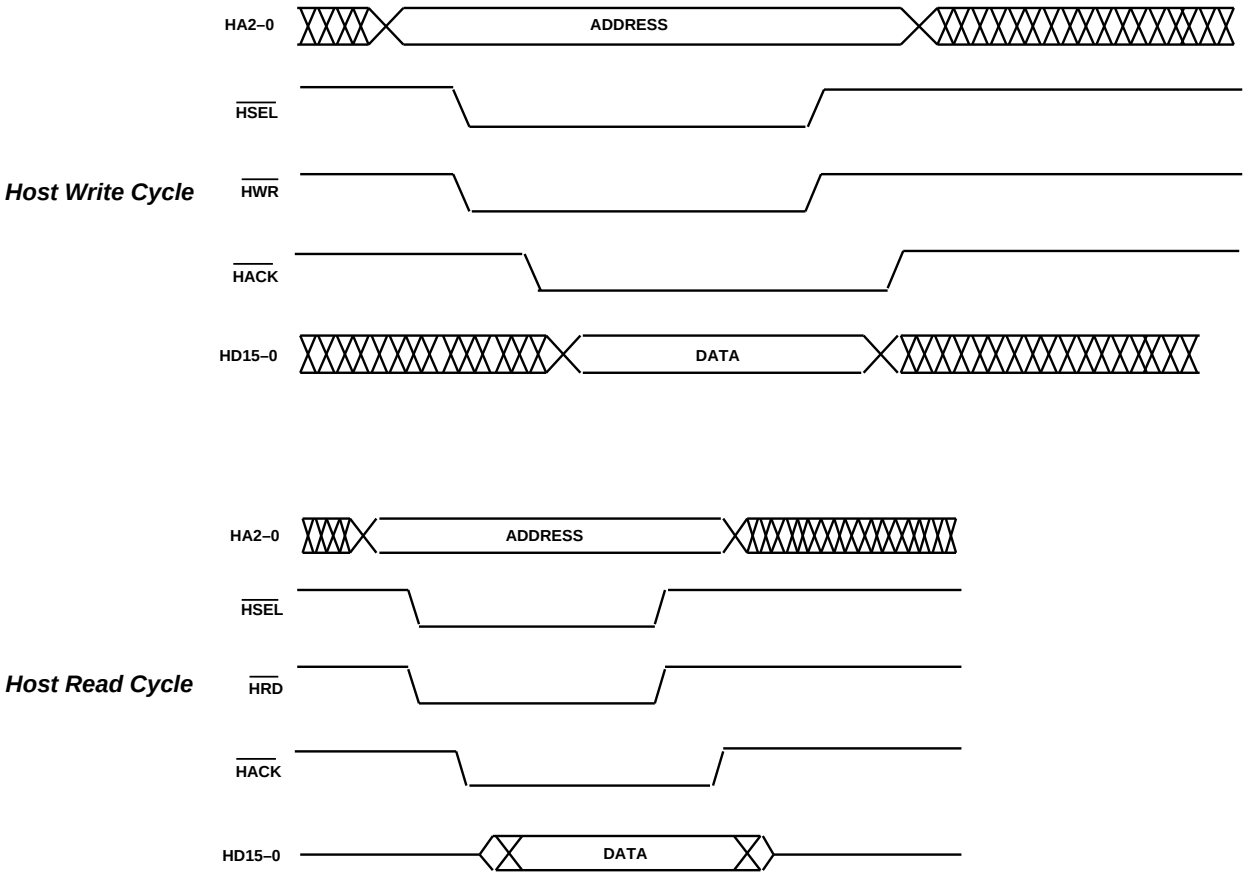


Figure 7.6 HIP Timing: Separate Strobes, Separate Buses

HMD0 selects separate read and write strobes, and HMD1 selects separate address and data buses. The timing for the read cycle and the write cycle is as follows:

1. The host asserts the address.
2. The host asserts (HRD or HWR) and HSEL.
3. The ADSP-21xx returns HACK (and, for a read cycle, the data).
4. For a write cycle, the host asserts the data.
5. The host deasserts (HRD or HWR) and HSEL.
6. The host deasserts the address (and, for a write cycle, the data).
7. The ADSP-21xx deasserts HACK (and, for a read cycle, the data).

Figure 7.7 shows the HIP timing when HMD0=1 and HMD1=0. HMD0

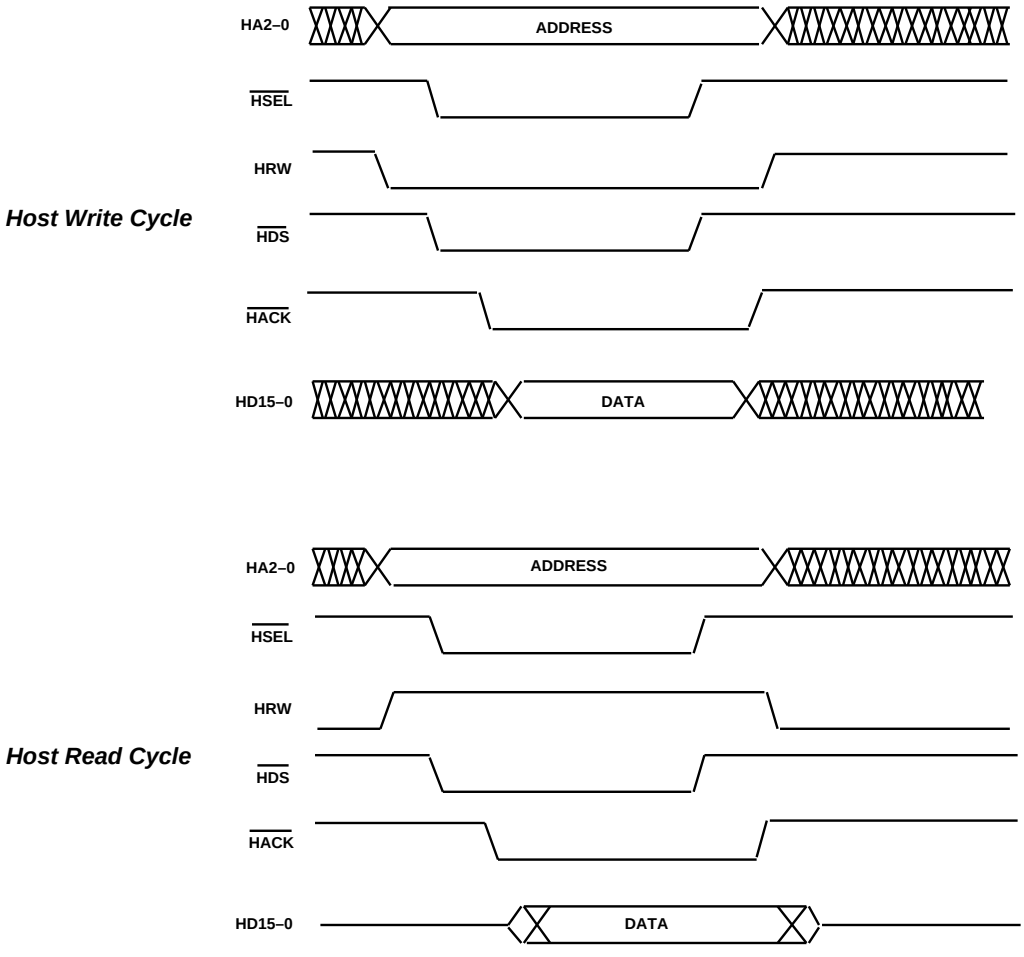


Figure 7.7 HIP Timing: Multiplexed R/W Strobe, Separate Buses

selects a multiplexed read/write select with data strobe, and HMD1 selects separate address and data buses. The timing for the read cycle and the write cycle is as follows:

1. The host asserts HRW and the address.
2. The host asserts HDS and HSEL.
3. The ADSP-21xx returns HACK (and, for a read cycle, the data).
4. For a write cycle, the host asserts the data.
5. The host deasserts HDS and HSEL.
6. The host deasserts HRW and the address (and, for a write cycle, the data).
7. The ADSP-21xx deasserts HACK (and, for a read cycle, the data).

Figure 7.8 shows the HIP timing when HMD0=0 and HMD1=1. HMD0 selects separate read and write strobes, and HMD1 selects multiplexed address and data buses. HD0-HD2 are used for the address. The timing

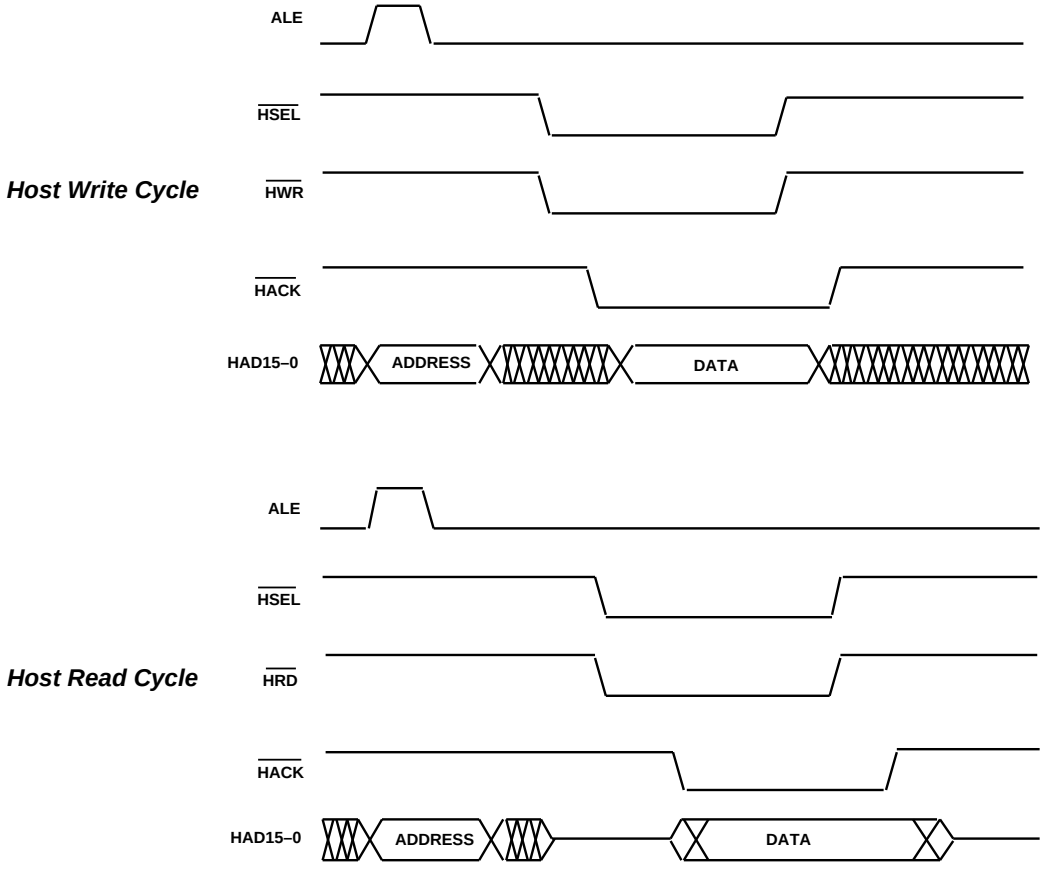


Figure 7.8 HIP Timing: Separate Strobes, Multiplexed Buses

for the read cycle and the write cycle is as follows:

1. The host asserts ALE.
2. The host drives the address.
3. The host deasserts ALE.
4. The host stops driving the address.
5. The host asserts (HRD or HWR) and HSEL.
6. The ADSP-21xx returns HACK (and, for a read cycle, the data).
7. For a write cycle, the host asserts the data.
8. The host deasserts (HRD or HWR) and HSEL.
9. For a write cycle, the host deasserts the data.
10. The ADSP-21xx deasserts HACK (and, for a read cycle, the data).

Figure 7.9 shows the HIP timing when HMD0=1 and HMD1=1. HMD0 selects a multiplexed read/write select with data strobe, and HMD1 selects multiplexed address and data buses. HD0-HD2 are used for the address. The timing for the read cycle and the write cycle is as follows:

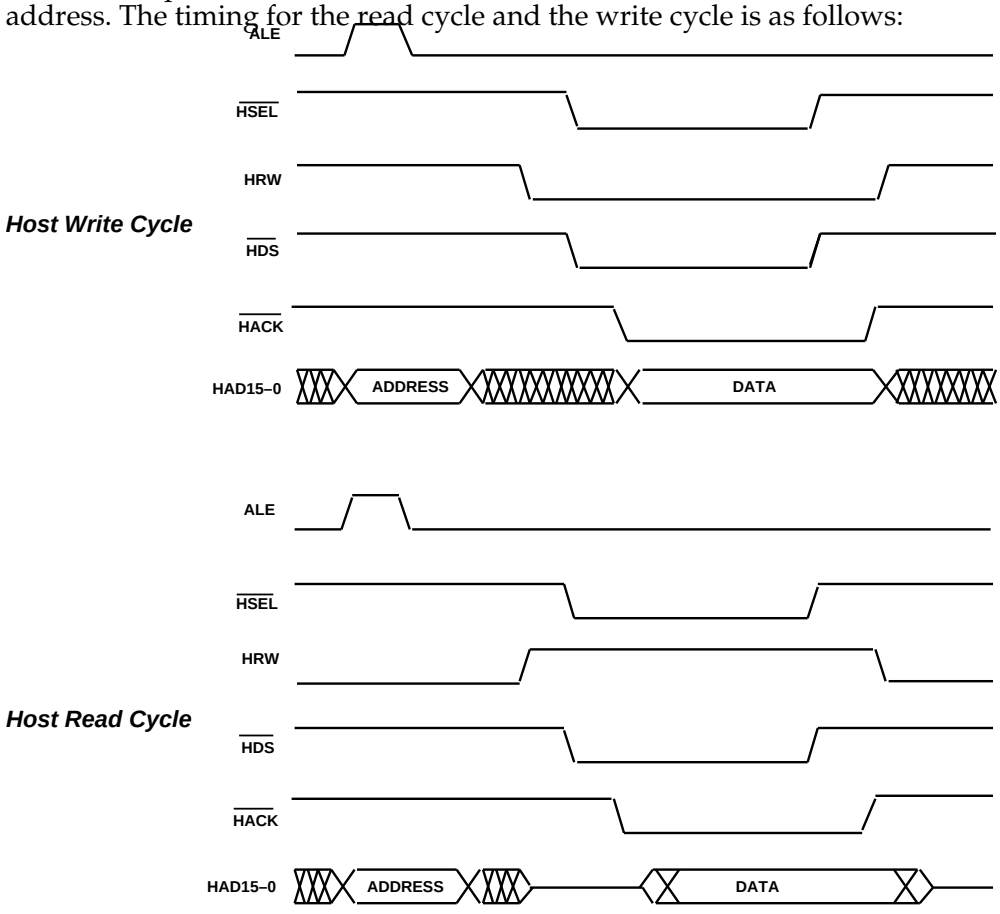


Figure 7.9 HIP Timing: Multiplexed R/W Strobe, Multiplexed Buses

1. The host asserts ALE.
2. The host drives the address.
3. The host deasserts ALE.
4. The host stops driving the address.
5. The host asserts HRW.
6. The host asserts HDS and HSEL.
7. The ADSP-21xx returns HACK (and, for a read cycle, the data).
8. For a write cycle, the host asserts the data.
9. The host deasserts HDS and HSEL.
10. The host deasserts HRW (and, for a write cycle, the data).
11. The ADSP-21xx deasserts HACK (and, for a read cycle, the data).

7.7 BOOT LOADING THROUGH THE HIP

The entire internal program RAM of the ADSP-21xx, or any portion of it, can be loaded using a boot sequence. Upon hardware or software reset, the boot sequence occurs if the MMAP pin is 0. If the MMAP pin is 1, the boot sequence does not occur.

The ADSP-21xx can boot in either of two ways: from external memory (usually EPROM), through the boot memory interface, or from a host processor, through the HIP. The BMODE pin selects which type of booting occurs.

When BMODE=0, booting occurs through the memory interface. This process is described in Chapter 10, “Memory Interface.” When the BMODE=1, booting occurs through the HIP.

To generate a file for HIP booting, use the HIP Splitter utility program of the ADSP-2100 Family Development Software. (This utility produces HIP boot files while the PROM Splitter utility produces files for EPROM booting.)

The BMS signal is asserted when booting through the HIP just as when booting through the memory interface; in this case, it serves as an indication that the boot sequence is occurring. Boot memory wait states have no effect when booting through the HIP.

Booting through the HIP occurs in the following sequence:

1. After reset, the host writes the length of the boot sequence to HDR3.

2. The host waits at least two ADSP-21xx processor cycles.

3. Starting with the instruction which is to be loaded into the highest address of internal program memory, the host writes an instruction into HDR0, HDR2 and HDR1 (in that order), one byte each. The upper byte goes into HDR0, the lower byte goes into HDR2 and the middle byte goes into HDR1.

4. The address of the instruction is decremented, and Step 3 is repeated. This continues until the last instruction has been loaded into the HIP.

The ADSP-21xx reads the length of the boot load first, then bytes are loaded from the highest address downwards. This results in shorter booting times for shorter loads.

The number of instructions booted must be a multiple of eight. The boot length value is given as:

$$\text{length} = (\text{number of 24-bit program memory words} \div 8) - 1$$

That is, a length of 0 causes the HIP to load eight 24-bit words.

In most cases, no handshaking is necessary, and the host can transfer data at the maximum rate it is capable of. If the host operates faster than the ADSP-21xx, wait states or NOPs must be added to the host cycle to slow it down to one write every ADSP-21xx clock cycle.

7 Host Interface Port

The following example shows the data that a host would write to the HIP for a 1000-instruction boot:

Data	*Location*
Page Length (124 decimal)	HDR3
Upper Byte of Instruction at 999	HDR0
Lower Byte of Instruction at 999	HDR2
Middle Byte of Instruction at 999	HDR1
Upper Byte of Instruction at 998	HDR0
Lower Byte of Instruction at 998	HDR2
Middle Byte of Instruction at 998	HDR1
Upper Byte of Instruction at 997	HDR0
Lower Byte of Instruction at 997	HDR2
Middle Byte of Instruction at 997	HDR1
•	•
•	•
•	•
Upper Byte of Instruction at 0	HDR0
Lower Byte of Instruction at 0	HDR2
Middle Byte of Instruction at 0	HDR1

A 16-bit host boots the ADSP-21xx at the same rate as an 8-bit host. Either type of host must write the same data to the same the HDRs in the same sequence (HDR0, HDR2, HDR1). If a 16-bit host writes 16-bit data, the upper byte of the data must be 0x00. The following example, loading the instruction 0xABCDEF, illustrates this:

	8-Bit Host	*16-Bit Host*
1st Write (to HDR0)	`0xAB`	`0x00AB`
2nd Write (to HDR2)	`0xEF`	`0x00EF`
3rd Write (to HDR1)	`0xCD`	`0x00CD`

Analog Interface 8

8.1 OVERVIEW

The ADSP-21msp58 and ADSP-21msp59 processors include an analog signal interface consisting of a 16-bit sigma-delta A/D converter, a 16-bit sigma-delta D/A converter, and a set of memory-mapped control and data registers. The analog interface offers the following features:

- linear-coded 16-bit sigma-delta ADC
- linear-coded 16-bit sigma-delta DAC
- on-chip anti-aliasing and anti-imaging filters
- 8 kHz sampling frequency
- programmable gain for DAC and ADC
- on-chip voltage reference

The analog interface provides a complete analog front end for high performance voiceband DSP applications. The ADC and DAC operate at a fixed sampling rate of 8 kHz. The inclusion of on-chip anti-aliasing and anti-imaging filters, 16-bit sigma-delta converters, and programmable gain amplifiers ensures a highly integrated solution to voiceband analog processing requirements. Sigma-delta conversion technology eliminates the need for complex off-chip anti-aliasing filters and sample-and-hold circuitry.

The ADSP-21msp58 and ADSP-21msp59 contain the same analog interface—they differ only in the amount of on-chip memory. Refer to the *ADSP-21msp58/59 Data Sheet* for detailed analog performance specifications.

The analog interface of the ADSP-21msp58/59 is operated by using several data-memory-mapped control and data registers. The ADC and DAC I/O can be transmitted and received via individual memory-mapped registers, or the data can be autobuffered directly into the processor's data memory. This autobuffering is similar to serial port autobuffering, as described in Chapter 5.

8 Analog Interface

Two ADSP-21msp58/59 interrupts are dedicated to the ADC and DAC converters. One interrupt is used for the ADC and the other interrupt is used for the DAC. Interrupts occur at the sample rate or when the autobuffer transfer is complete.

A block diagram of the analog interface is shown in Figure 8.1, and pin definitions are given in Table 8.1.

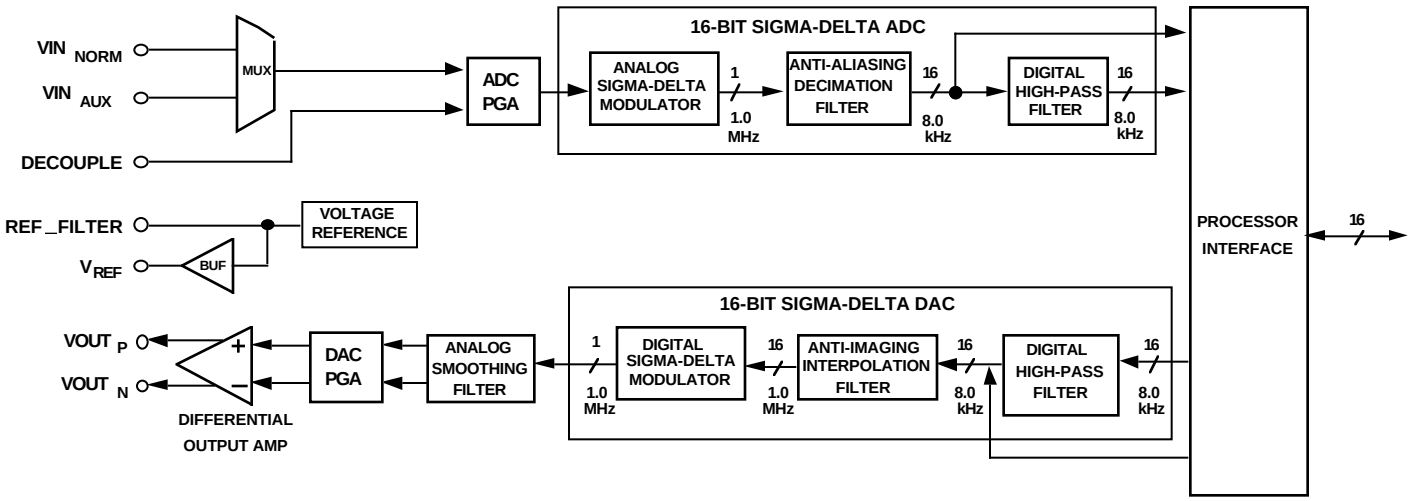


Figure 8.1 Analog Interface Block Diagram (ADSP-21msp58/59)

8.2 A/D CONVERSION

The A/D conversion circuitry of the ADSP-21msp58/59's analog interface consists of an input multiplexer, a programmable gain amplifier (PGA), and a sigma-delta analog-to-digital converter (ADC).

8.2.1 Analog Input

The analog input is internally biased by an on-chip voltage reference to allow operation of the ADSP-21msp58/59 with a single +5V power supply. The analog inputs should be ac-coupled.

An analog multiplexer selects either the NORM or AUX input. The input multiplexer is configured by bit 1 (IMS) of the ADSP-21msp58/59's analog control register (which is memory-mapped at address 0x3FEE in data memory). The multiplexer setting should not be changed while an input signal is being processed.

Analog Interface 8

Pin Name	*I/O*	*Function*
VIN_{NORM}	I	Input terminal of the NORM channel of the ADC.
VIN_{AUX}	I	Input terminal of the AUX channel of the ADC.
Decouple	I	Ground reference of the NORM and AUX channels for the ADC.
$VOUT_P$	O	Non-inverting output terminal of the differential output amplifier from the DAC.
$VOUT_N$	O	Inverting output terminal of the differential output amplifier from the DAC.
V_{REF}	O	Buffered output voltage reference.
REF_FILTER	O	Voltage reference external bypass filter node.
V_{CC}		Analog supply voltage.
GND_A		Analog ground.

Table 8.1 Analog Interface Pin Definitions

The ADC PGA may be used to increase the signal level by +6 dB, +20 dB, or +26 dB. This selection is configured by bits 9 and 0 (IG1, IG0) of the analog control register. Input signal level to the sigma-delta modulator should not exceed the V_{INMAX} specification listed in the *ADSP-21msp58/59 Data Sheet*. Refer to "Analog Input" in the "Design Considerations" section of this chapter for more information.

An offset may be added to the input of the ADC in order to move the ADC's idle tones out of the 4.0 kHz speech band range. This is selected by bit 10 of the analog control register. The added offset must be removed by the ADC's high pass filter; therefore the high pass filter must be inserted (not bypassed) when the offset is added.

8.2.2 ADC

The analog interface's ADC consists of a 4th-order analog sigma-delta modulator, an anti-aliasing decimation filter, and a digital high pass filter. The sigma-delta modulator noise-shapes the signal and produces 1-bit samples at a 1.0 MHz rate. This bit stream, which represents the analog input signal, is fed to the anti-aliasing decimation filter.

8 Analog Interface

8.2.2.1 Decimation Filter

The ADC's anti-aliasing decimation filter contains two stages. The first stage is a sinc^4 digital filter that increases resolution to 16 bits and reduces the sample rate to 40 kHz. The second stage is an IIR low pass filter.

The IIR low pass filter is a 10th-order elliptic filter with a passband edge at 3.7 kHz and a stopband attenuation of 65 dB at 4 kHz. This filter has the following specifications:

Filter type:	10th-order low pass elliptic IIR
Sample frequency:	40.0 kHz
Passband cutoff*:	3.70 kHz
Passband ripple:	±0.2 dB
Stopband cutoff:	4.0 kHz
Stopband ripple:	–65.00 dB

* The passband cutoff frequency is defined to be the last point in the passband that meets the passband ripple specification.

(Note that these specifications apply only to this filter, and not to the entire ADC. The specifications can be used to perform further analysis of the exact characteristics of the filter, for example using a digital filter design software package.)

Figure 8.2 shows the frequency response of the IIR low pass filter.

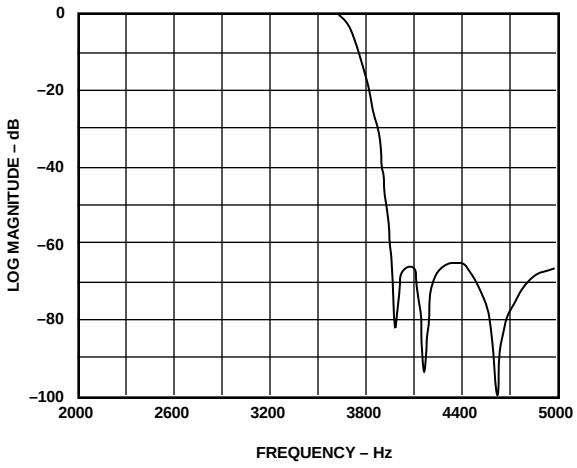


Figure 8.2 IIR Low Pass Filter Frequency Response

8.2.2.2 High Pass Filter

The ADC's digital high pass filter removes frequency components at the low end of the spectrum; it attenuates signal energy below the passband of the converter. The ADC's high pass filter can be bypassed by setting bit 7 (ADBY) of the ADSP-21msp58/59's analog control register.

The high pass filter is a 4th-order elliptic filter with a passband cutoff at 150 Hz. Stopband attenuation is 25 dB. This filter has the following specifications:

Filter type:	4th-order high pass elliptic IIR
Sample frequency:	8.0 kHz
Passband cutoff:	150.0 Hz
Passband ripple:	±0.2 dB
Stopband cutoff:	100.0 Hz
Stopband ripple:	–25.00 dB

(Note that these specifications apply only to this filter, and not to the entire ADC. The specifications can be used to perform further analysis of the exact characteristics of the filter, for example using a digital filter design software package.)

Figure 8.3 shows the frequency response of the high pass filter.

Passband ripple is ±0.2 dB for the combined effects of the ADC's digital filters (i.e. high pass filter and IIR low pass of the decimation filter) in the 300–3400 Hz passband.

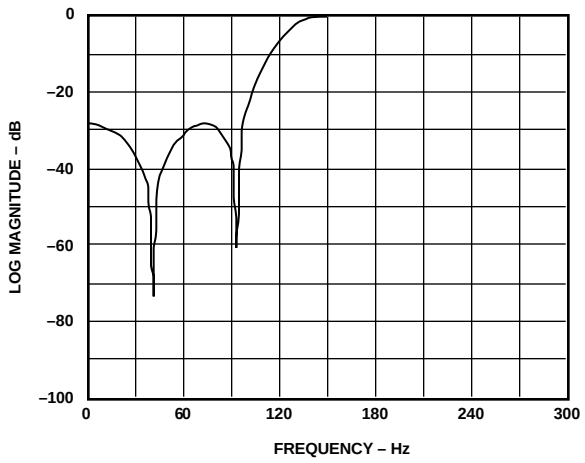


Figure 8.3 High Pass Filter Frequency Response

8 Analog Interface

8.3 D/A CONVERSION

The D/A conversion circuitry of the ADSP-21msp58/59's analog interface consists of a sigma-delta digital-to-analog converter (DAC), an analog smoothing filter, a programmable gain amplifier, and a differential output amplifier.

8.3.1 DAC

The analog interface's DAC implements digital filters and a sigma-delta modulator with the same characteristics as the filters and modulator of the ADC. The DAC consists of a digital high pass filter, an anti-imaging interpolation filter, and a digital sigma-delta modulator.

The DAC receives 16-bit data values from the ADSP-21msp58/59's DAC Transmit data register (which is memory-mapped at address 0x3FEC in data memory). The data stream is filtered first by the DAC's high pass filter and then by the anti-imaging interpolation filter. These filters have the same characteristics as the ADC's anti-aliasing decimation filter and digital high pass filter.

The output of the interpolation filter is fed to the DAC's digital sigma-delta modulator, which converts the 16-bit data to 1-bit samples at a 1.0 MHz rate. The modulator noise-shapes the signal such that errors inherent to the process are minimized in the passband of the converter.

The bit stream output of the sigma-delta modulator is fed to the DAC's analog smoothing filter where it is converted to an analog voltage.

8.3.1.1 High Pass Filter

The DAC's digital high pass filter has the same characteristics as the high pass filter of the ADC. The high pass filter removes frequency components at the low end of the spectrum; it attenuates signal energy below the passband of the converter. The DAC's high pass filter can be bypassed by setting bit 8 (DABY) of the ADSP-21msp58/59's analog control register.

The high pass filter is a 4th-order elliptic filter with a passband cutoff at 150 Hz. Stopband attenuation is 25 dB. This filter has the following specifications:

Filter type:	4th-order high pass elliptic IIR
Sample frequency:	8.0 kHz
Passband cutoff:	150.0 Hz
Passband ripple:	±0.2 dB
Stopband cutoff:	100.0 Hz
Stopband ripple:	–25.00 dB

(Note that these specifications apply only to this filter, and not to the entire DAC. The specifications can be used to perform further analysis of the exact characteristics of the filter, for example using a digital filter design software package.)

Figure 8.3 shows the frequency response of the high pass filter.

8.3.1.2 Interpolation Filter

The DAC's anti-imaging interpolation filter contains two stages. The first stage is is an IIR low pass filter that interpolates the data rate from 8 kHz to 40 kHz and removes images produced by the interpolation process. The output of this stage is then interpolated to 1.0 MHz and fed to the second stage, a sinc^4 digital filter that attenuates images produced by the 40 kHz to 1.0 MHz interpolation process.

The IIR low pass filter is a 10th-order elliptic filter with a passband edge at 3.70 kHz and a stopband attenuation of 65 dB at 4 kHz. This filter has the following specifications:

Filter type:	10th-order low pass elliptic IIR
Sample frequency:	40.0 kHz
Passband cutoff*:	3.70 kHz
Passband ripple:	±0.2 dB
Stopband cutoff:	4.0 kHz
Stopband ripple:	–65.00 dB

* The passband cutoff frequency is defined to be the last point in the passband that meets the passband ripple specification. (Note that these specifications apply only to this filter, and not to the entire DAC. The specifications can be used to perform further analysis of the exact characteristics of the filter, for example using a digital filter design software package.)

8 Analog Interface

Figure 8.2 shows the frequency response of the IIR low pass filter.

Passband ripple is ±0.2 dB for the combined effects of the DAC's digital filters (i.e. high pass filter and IIR low pass of the interpolation filter) in the 300–3400 Hz passband.

8.3.1.3 Analog Smoothing Filter & Programmable Gain Amplifier

The DAC's programmable gain amplifier (PGA) can be used to adjust the output signal level by –15 dB to +6 dB. This gain is selected by bits 2-4 (OG0, OG1, OG2) of the of the ADSP-21msp58/59's analog control register.

The DAC's analog smoothing filter consists of a 2nd-order Sallen-Key continuous-time filter and a 3rd-order switched capacitor filter. The Sallen-Key filter has a 3 dB point at approximately 25 kHz.

8.3.2 Differential Output Amplifier

The ADSP-21msp58/59's analog output signal ($VOUT_P$ – $VOUT_N$) is produced by a differential amplifier. The differential amplifier meets specifications for loads greater than 2 kΩ ($R_L \geq$ 2 kΩ) and has a maximum differential output voltage swing of ±3.156 V peak-to-peak (3.17 dBm0). The DAC will drive loads smaller than 2 kΩ, but with degraded performance.

The output signal is dc-biased to the on-chip voltage reference (V_{REF}) and can be ac-coupled directly to a load or dc-coupled to an external amplifier. Refer to "Analog Output" in the "Design Considerations" section of this chapter for more information.

The $VOUT_P$ – $VOUT_N$ outputs must be used as a differential signal, otherwise performance will be severely degraded. Do not use either pin as a single-ended output.

8.4 OPERATING THE ANALOG INTERFACE

The analog interface of the ADSP-21msp58/59 is operated with the use of several memory-mapped control and data registers. The ADC and DAC I/O data can be received and transmitted in two memory-mapped data registers. The data can also be autobuffered into (and from) on-chip memory where data is automatically transferred to or from the data registers. In both cases, the I/O processing is interrupt-driven: two ADSP-21msp58/59 interrupts are dedicated to the analog interface, one for ADC receive data and one for DAC transmit data. (**Note:** Autobuffering with SPORT1 is not available on the ADSP-21msp5x processors because this autobuffering channel is used for the analog interface.)

The ADSP-21msp58/59 must have an input clock frequency of 13 MHz. At this frequency, analog-to-digital and digital-to-analog converted data is transmitted at an 8 kHz rate with a single 16-bit word transmitted every 125 μs.

8.4.1 Memory-Mapped Control Registers

Two memory-mapped control registers are used to configure the ADSP-21msp58/59's analog interface: the analog control register and analog autobuffer/powerdown register.

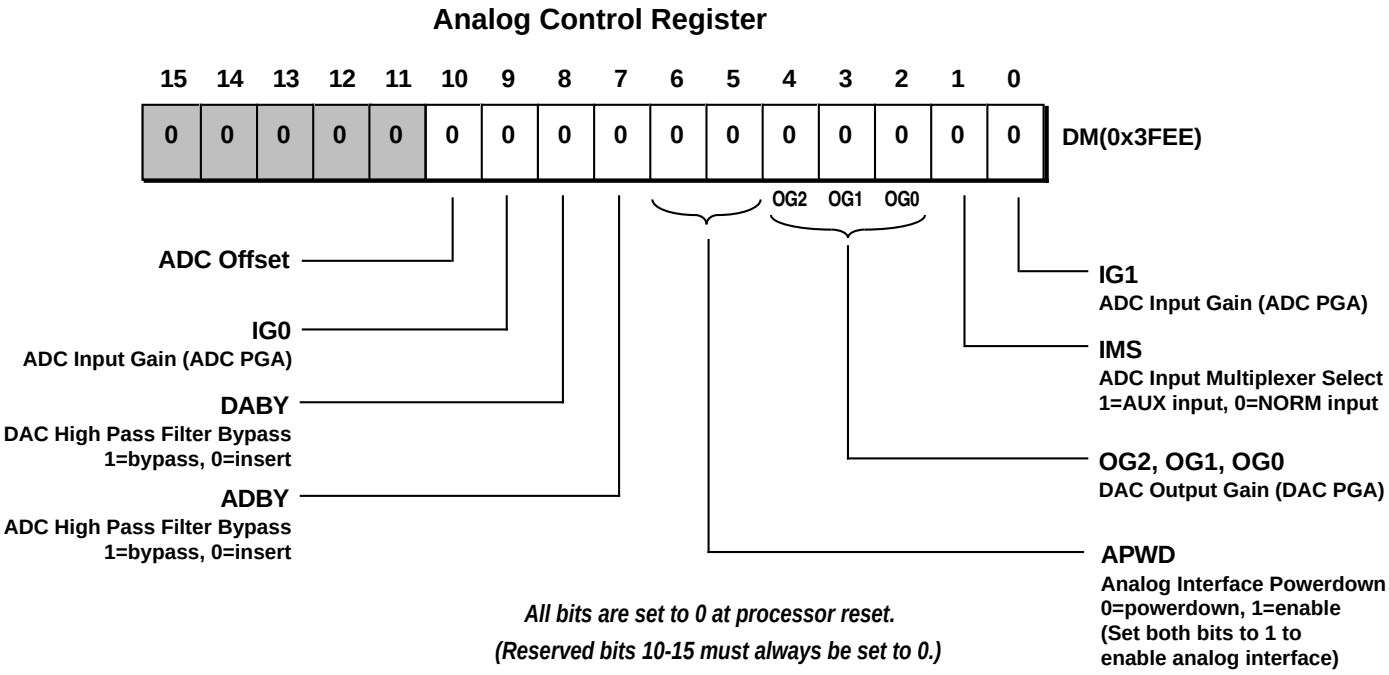
8.4.1.1 Analog Control Register

The analog control register (located at address 0x3FEE in data memory) is shown in Figure 8.4. This register configures the ADC input multiplexer, ADC input gain PGA, ADC high pass filter, DAC high pass filter, and DAC output gain PGA.

The analog control register also contains the APWD bits (bits 5, 6) which must both be set to ones to enable and start up the analog interface—*always enable and disable the analog interface using both bits 5 and 6.* The DAC and ADC begin transmitting data after these bits are set. Clearing the APWD bits disables the entire analog interface by putting it in a powerdown state. The APWD bits must be cleared (to zeros) at least three processor cycles before putting the processor in powerdown. See "Powerdown" in Chapter 9, System Interface.

The analog control register is cleared (to 0x0000) by the processor's $\overline{\text{RESET}}$ signal. Note that bits 10-15 of this register are reserved and must always be set to zero.

8 Analog Interface



IG1, IG0
ADC Input Gain (ADC PGA)

Gain	IG1	IG0
0 dB	0	0
+6 dB	0	1
+20 dB	1	0
+26 dB	1	1

OG2, OG1, OG0
DAC Output Gain (DAC PGA)

Gain	OG2	OG1	OG0
+6 dB	0	0	0
+3 dB	0	0	1
0 dB	0	1	0
–3 dB	0	1	1
–6 dB	1	0	0
–9 dB	1	0	1
–12 dB	1	1	0
–15 dB	1	1	1

Figure 8.4 Analog Control Register

8.4.1.2 Analog Autobuffer/Powerdown Register

The analog autobuffer/powerdown register (located at address 0x3FEF in data memory) is shown in Figure 8.5. This register enables or disables autobuffering of ADC receive data and/or DAC transmit data—autobuffering is enabled by writing ones to the ARBUF (bit 0) and/or ATBUF (bit 1) bits. When autobuffering is enabled, I (index) and M (modify) registers are selected in bits 2–11 for the receive and/or transmit data buffers. See "Autobuffering" in the Serial Ports chapter for details on autobuffering.

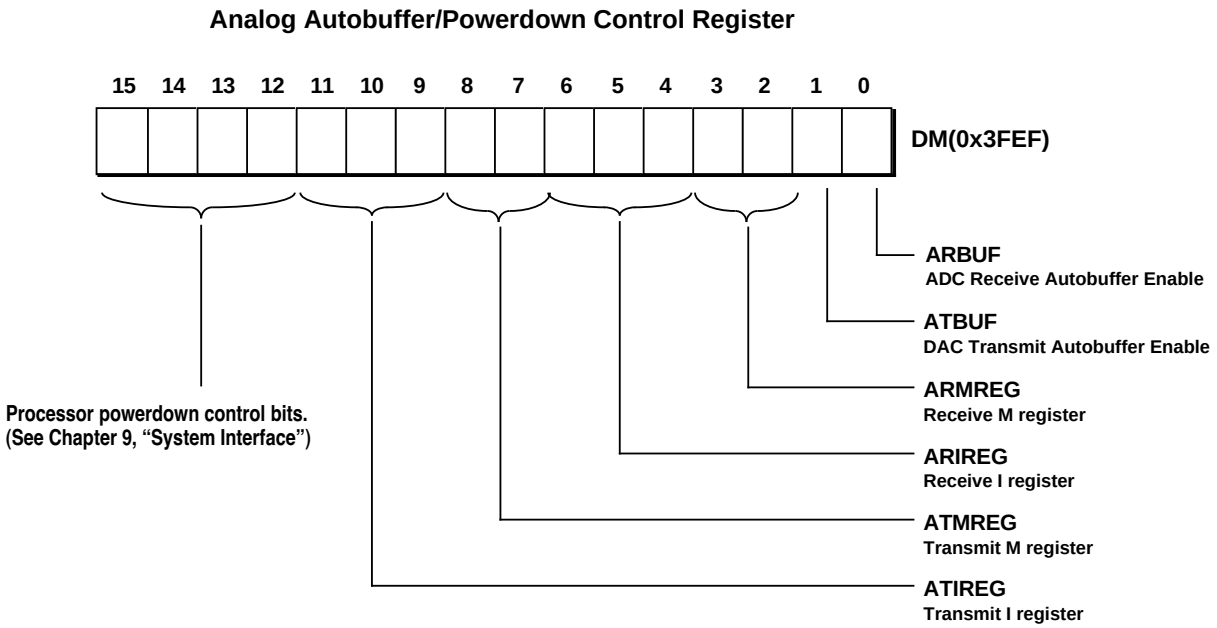


Figure 8.5 Analog Autobuffer/Powerdown Control Register

Bits 12–15 of the analog autobuffer/powerdown register control the ADSP-21msp58/59's processor powerdown function, *not* powerdown of the analog interface—powerdown of the analog interface only is controlled by the APWD bits (bits 5, 6) of the analog control register. The ADSP-21msp58/59's powerdown function is described in the "Powerdown" section of Chapter 9, System Interface.

8.4.2 Memory-Mapped Data Registers

There are two memory-mapped data registers dedicated to the analog interface. The 16-bit ADC receive data register is located at address 0x3FED in data memory. The 16-bit DAC transmit data register is located at address 0x3FEC in data memory. These registers must be individually read and written when autobuffering is not in use (autobuffering automatically transfers the data to and from processor data memory).

When autobuffering is disabled, data must be transmitted to the sigma-delta DAC by writing a 16-bit word to the DAC transmit register (0x3FEC) and data must be received from the sigma-delta ADC by reading a 16-bit word from the ADC receive register (0x3FED).

8 Analog Interface

8.4.3 ADC & DAC Interrupts

The analog interface generates two interrupts that signal either: 1) that a 16-bit, 8 kHz analog-to-digital or digital-to-analog conversion has been completed, or 2) that an autobuffer block transfer has been completed (i.e. the entire data buffer contents have been transmitted or received).

When one of the analog interrupts occurs, the processor vectors to the appropriate address:

DAC Transmit interrupt vector address: 0x18
ADC Receive interrupt vector address: 0x1C

These interrupts can be masked out in the processor's IMASK register and can be forced or cleared in the IFC register.

8.4.3.1 Autobuffering Disabled

The ADC receive and DAC transmit interrupts occur at an 8 kHz rate, indicating when the data registers should be accessed, when autobuffering is disabled. On the receive side, the ADC interrupt is generated each time an A/D conversion cycle is completed and the 16-bit data word is available in the ADC receive register. On the transmit side, the DAC interrupt is generated each time a D/A conversion cycle is completed and the DAC transmit register is ready for the next 16-bit data word.

Both interrupts are generated simultaneously at an 8 kHz rate, occurring every 3250 instruction cycles with a 13 MHz internal clock, when autobuffering is disabled. The interrupts are generated continuously, starting when the analog interface is powered up by setting the APWD bits (bits 5, 6) to ones in the analog control register. Because both interrupts occur simultaneously, only one should be enabled (in IMASK) to vector to a single service routine that handles both transmit and receive data. (When autobuffering is enabled, though, both interrupts should be enabled.)

A simple analog loopback program is shown in Listing 8.1.

```
{ ADSP-21msp58/59 Analog Interface Loopback Example   }
{   - configures analog interface                     }
{   - copies ADC receive data to DAC transmit buffer}


.MODULE/ABS=0/BOOT=0  talkthru;

#define  codec_tx_data  0x3FEC
#define  codec_rx_data  0x3FED
#define  codec_ctrl_reg 0x3FEE

resetv:         JUMP setup; NOP; NOP; NOP;
irq2v:          RTI; NOP; NOP; NOP;         {interrupt vectors ...}
hipwv:          RTI; NOP; NOP; NOP;
hiprv:          RTI; NOP; NOP; NOP;
spt0tv:         RTI; NOP; NOP; NOP;
spt0rv:         RTI; NOP; NOP; NOP;
antv:           RTI; NOP; NOP; NOP;
anrv:           SI = DM(codec_rx_data);     {read in data from ADC}
                DM(codec_tx_data) = SI;     {write out data to DAC}
                RTI; NOP;
irq1v:          RTI; NOP; NOP; NOP;
irq0v:          RTI; NOP; NOP; NOP;
timerv:         RTI; NOP; NOP; NOP;
pwrdwnv:        RTI; NOP; NOP; NOP;

setup:          AX1 = 0x0060;
                DM(codec_ctrl_reg) = AX1;  {power up analog interface}
                IMASK = 0x8;               {enable analog receive interrupt}
wait_loop:      IDLE;                      {wait for interrupt}
                JUMP wait_loop;
.ENDMOD;
```

Listing 8.1 ADSP-21msp58/59 Analog Loopback Program

8.4.3.2 Autobuffering Enabled

In some applications it is advantageous to perform block data transfers between the analog converters and processor memory. Analog interface autobuffering allows you to automatically transfer blocks of data from the ADC to on-chip processor data memory or from on-chip processor data memory to the DAC.

An interrupt is generated when an entire block transfer is complete (i.e. when the data buffer is full or empty). Analog interface autobuffering operates in the same way as SPORT autobuffering, described in Chapter 5. Note that data can be autobuffered through the analog converters or through SPORT0 of the ADSP-21msp58/59. Autobuffering is not available on SPORT1 of the ADSP-21msp58/59.

8 Analog Interface

Before autobuffering is enabled, separate circular buffers must be set up in data memory for the ADC receive and DAC transmit data. This is accomplished by selecting I (index) and M (modify) registers in the analog autobuffer/powerdown register; see Figure 8.5.

Transmit data autobuffered to the DAC is addressed with the I register specified in the ATIREG field (bits 9, 10, 11). Receive data autobuffered from the ADC is addressed with the I register specified in the ARIREG field (bits 4, 5, 6). The modify (M) registers are specified in the ARMREG (bits 2, 3) field and ATMREG (bits 7, 8) field. Since the transfer of ADC and DAC data occurs simultaneously, it is possible to use the same I register for transmit and receive autobuffering. In this case, the buffer is shared for both functions and care should be taken when specifiying a value for the M register.

An autobuffering example program is shown in Listing 8.2.

```
{ ADSP-21msp58/59 Analog Interface Autobuffer Example }
{   - configures analog interface                      }
{   - enables analog autobuffer                        }
{   - receive analog data into a 256 word buffer       }
{   - transmit analog data from a 256 word buffer      }

.MODULE/RAM/ABS=0/BOOT=0  auto_example;
.VAR/DM/CIRC  buff1[256];                        {first data buffer}
.VAR/DM/CIRC  buff2[256];                        {second data buffer}
.VAR/DM  flag_bit;                               {tracks buffers}
#define  codec_tx_data  0x3FEC
#define  codec_rx_data  0x3FED
#define  codec_ctrl_reg 0x3FEE
#define codec_auto_ctrl 0x3FEF

resetv:        JUMP setup; NOP; NOP; NOP;
irq2v:         RTI; NOP; NOP; NOP;               {interrupt vectors ...}
hipwv:         RTI; NOP; NOP; NOP;
hiprv:         RTI; NOP; NOP; NOP;
spt0tv:        RTI; NOP; NOP; NOP;
spt0rv:        RTI; NOP; NOP; NOP;
antv:          RTI; NOP; NOP; NOP;
anrv:          JUMP switch; NOP; NOP; NOP;       {call autobuffer switch}
irq1v:         RTI; NOP; NOP; NOP;
irq0v:         RTI; NOP; NOP; NOP;
timerv:        RTI; NOP; NOP; NOP;
pwrdwnv:       RTI; NOP; NOP; NOP;
```

```
setup:          I0 = ^buff1;                {I0 points to first data buffer}
                L0 = %buff1;
                I1 = ^buff2;                {I1 points to second data buffer}
                L1 = %buff2;
                M0 = 0x1;
                SI = 0x0;                   {initialize flag register}
                DM(flag_bit) = SI;
                                            {use I1 and M0 for tranmsit}
                                            {use I0 and M0 for receive}
                AY0 = 0x0203;               {enable rcv and tx autobuffer}
                DM(codec_auto_ctrl) = AY0;
                AX1 = 0x0060;
                DM(codec_ctrl_reg) = AX1;   {power up analog interface}
                IMASK = 0x8;                {enable analog rx interrupt}

wait:           IDLE;                       {wait for autobuffer interrupt}
                JUMP wait;
switch:         AX0 = DM(flag_bit);
                AR = pass AX0;              {check buffer status}
                IF NE JUMP fill_buff2;
fill_buff1:     SI = 0x1;                   {fill buff2 next time}
                AY0 = 0x0013;
                JUMP done;
fill_buff2:     SI = 0x0;                   {fill buff1 next time}
                AY0 = 0x0203;
                JUMP done;
done:           DM(codec_auto_ctrl) = AY0;
                DM(flag_bit) = SI;
                RTI;
.ENDMOD;
```

Listing 8.2 ADSP-21msp58/59 Analog Autobuffer Program

Receive and transmit autobuffering may be independently enabled and the two interrupts can occur (and be serviced) independently. This allows the use of different data buffer lengths when autobuffering both receive and transmit data. It also allows autobuffering to be used on only one side, receive or transmit, while the other is serviced at the 8 kHz interrupt rate.

8 Analog Interface

8.5 CIRCUIT DESIGN CONSIDERATIONS

The following sections discuss interfacing analog signals to the ADSP-21msp58/59.

8.5.1 Analog Signal Input

Figure 8.6 shows the recommended input circuit for the ADSP-21msp58/59's analog input pin (either VIN_{NORM} or VIN_{AUX}). The circuit of Figure 8.6 implements a first-order low pass filter (R_1C_1). The 3 dB point of the filter should be less than 40 kHz. This is the only filter that must be implemented external to the processor to prevent aliasing of the sampled signal. Since the ADSP-21msp58/59's sigma-delta ADC uses a highly oversampled approach that transfers most of the anti-aliasing filtering into the digital domain, the off-chip anti-aliasing filter need only be of low order. Refer to the *ADSP-21msp58/59 Data Sheet* for more detailed information.

The ADSP-21msp58/59's on-chip ADC PGA (programmable gain amplifier) can be used when there is not enough gain in the input circuit. The ADC PGA is configured by bits 9 and 0 (IG1, IG0) of the processor's analog control register. The gain must be selected to ensure that a full-scale input signal (at R_1 in Figure 8.6) produces a signal level at the input to the sigma-delta modulator of the ADC that does not exceed V_{INMAX} (which is specified in the data sheet).

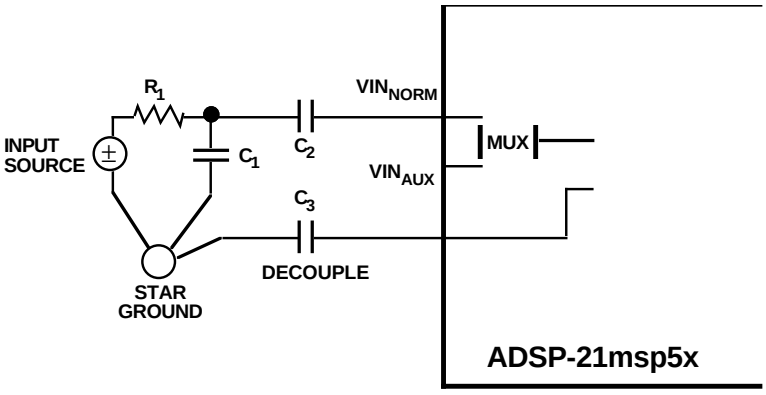


Figure 8.6 Recommended Analog Input Circuit

VIN_{NORM} and VIN_{AUX} are biased at the internal voltage reference (nominally 2.5V) of the ADSP-21msp58/59, which allows the analog interface to operate from a single supply. The input signal should be ac-coupled with an external capacitor (C_2). The value of C_2 is determined by the input resistance of the analog input (VIN_{NORM} , VIN_{AUX}), 200 kΩ, and the desired cutoff frequency. The cutoff frequency should be less than or equal to 30 Hz. The following equations should be used to determine the values for R_1, C_1, and C_2: R_1 should be less than or equal to 2.2 kΩ, C_2 should be greater than or equal to 0.027 µF, C_3 should be equal to C_2.

$$C_2 = \frac{1}{2\pi f_1 R_{IN}}$$

R_{IN} = input resistance of ADSP-21msp58/59 (200 kΩ)

f_1=cutoff frequency ≤ 30 Hz

$$R_1 = \frac{1}{2\pi f_2 C_2}$$

$R_1 \leq$ 2.2 kΩ

20 kHz < f_2 < 40 kHz *

$$C_1 = \frac{1}{2\pi f_2 R_1}$$

$C_3 = C_2$

* If minimum (< 0.1 dB) rolloff at 4 kHz is desired, f_2 should be set to 40 kHz.

8 Analog Interface

8.5.2 Analog Signal Output

The ADSP-21msp58/59's differential analog output ($VOUT_P$ – $VOUT_N$) is produced by an on-chip differential amplifier. The differential amplifier will meet dynamic specifications for loads greater than 2 kΩ ($R_L \geq 2$ kΩ) and has a maximum differential output voltage swing of ±3.156 V peak-to-peak (3.17 dBm0). The DAC will drive loads smaller than 2 kΩ, but with degraded dynamic performance. The differential output can can be ac-coupled directly to a load or dc-coupled to an external amplifier.

Figure 8.7 shows a simple circuit providing a differential output with ac coupling. The capacitor of this circuit (C_{OUT}) is optional; if used, its value can be chosen as follows:

$$C_{OUT} = \frac{1}{60\pi R_L}$$

The $VOUT_P$ – $VOUT_N$ outputs must be used as differential outputs; do not use either as a single-ended output. Figure 8.8 shows an example circuit which can be used to convert the differential output to a single-ended output. The circuit uses a differential-to-single-ended amplifier, the Analog Devices SSM-2141.

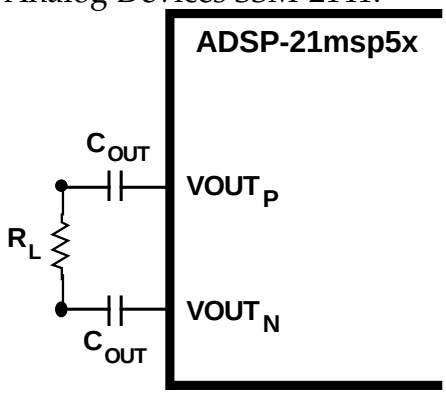


Figure 8.7 Example Circuit For Differential Output With AC Coupling

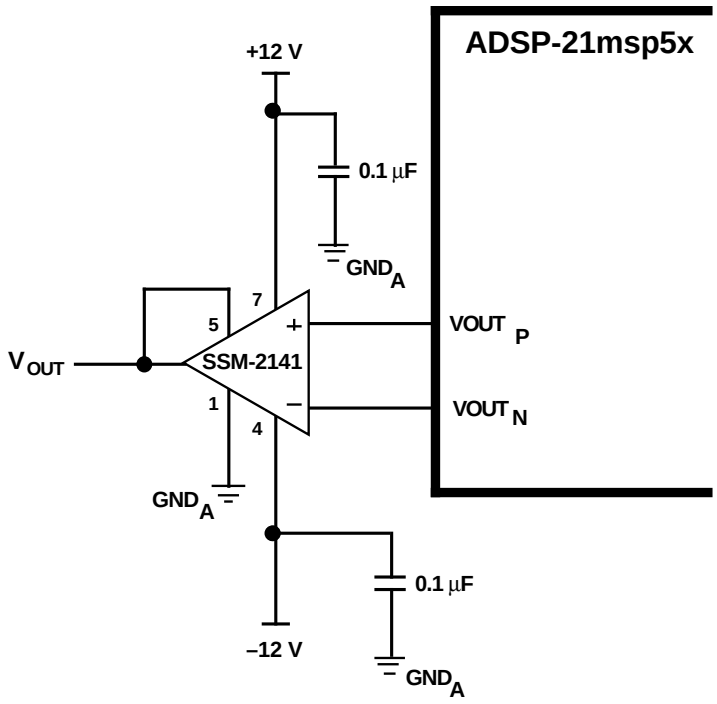


Figure 8.8 Example Circuit For Single-Ended Output

8.5.3 Voltage Reference Filter Capacitance

Figure 8.9 shows the recommended reference filter capacitor connections. The capacitor grounds should be connected to the same star ground point as that of Figure 8.6.

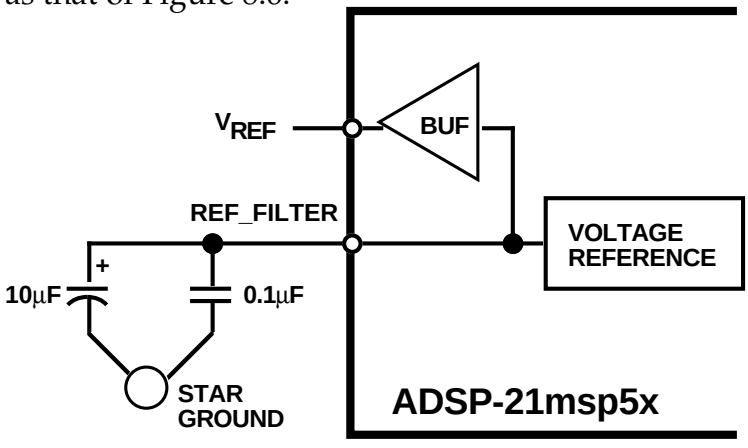


Figure 8.9 Voltage Reference Filter Capacitor

System Interface 9

9.1 OVERVIEW

This chapter describes the basic system interface features of the ADSP-2100 family processors. The system interface includes various hardware and software features used to control the DSP processor.

Processor control pins include a RESET signal, clock signals, flag inputs and outputs, and interrupt requests. This chapter describes only the logical relationships of control signals; consult individual processor data sheets for actual timing specifications.

9.2 CLOCK SIGNALS

The ADSP-2100 family processors may be operated with a TTL-compatible clock signal input to the CLKIN pin or with a crystal connected between the CLKIN and XTAL pins. If an external clock is used, XTAL must be left unconnected. The CLKIN signal may not be halted or changed in frequency during operation.

The ADSP-2101, ADSP-2105, ADSP-2115, and ADSP-2111 processors operate with an input clock frequency equal to the instruction cycle rate. The ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors operate with an input clock frequency equal to half the instruction rate; for example, a 16.67 MHz input clock produces a 33 MHz instruction rate (30 ns cycle time). Device timing is relative to the internal clock rate which is indicated by the CLKOUT signal.

Because these processors include an on-chip oscillator circuit, an external crystal can be used. The crystal should be connected between the CLKIN and XTAL pins, with two capacitors connected as shown in Figure 9.1, which can be found on the following page. A parallel-resonant, fundamental frequency, microprocessor-grade crystal should be used. The frequency value selected for the crystal should be equal to the desired instruction rate for the processor (for the ADSP-2101, ADSP-2105, ADSP-2115, and ADSP-2111) or half the desired instruction rate (for the ADSP-2171, ADSP-2181, and ADSP-21msp58/59).

9 System Interface

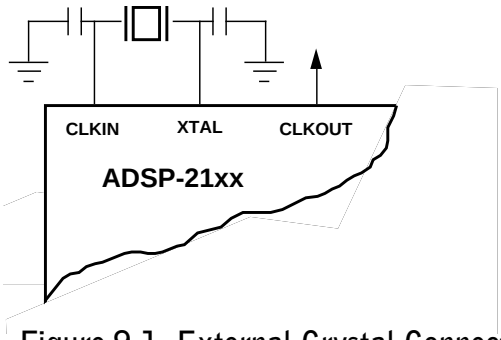


Figure 9.1 External Crystal Connections

The internal phased lock loop of the processors generates an internal clock which is four times the instruction rate.

The processors also generate a CLKOUT signal which is synchronized to the processors' internal cycles and operates at the instruction cycle rate. A phase-locked loop is used to generate CLKOUT and to divide each instruction cycle into a sequence of internal time periods called processor states. The relationship between the phases of CLKIN, CLKOUT, and the processor states is shown in Figure 9.2 for the ADSP-2101, ADSP-2105, ADSP-2115, and ADSP-2111 processors. Figure 9.3 shows the same information for the ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors. The phases of the internal processor clock are dependent upon the period of the external clock.

The CLKOUT output can be disabled on the ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors. This is controlled by the CLKODIS bit in the SPORT0 Autobuffer Control Register.

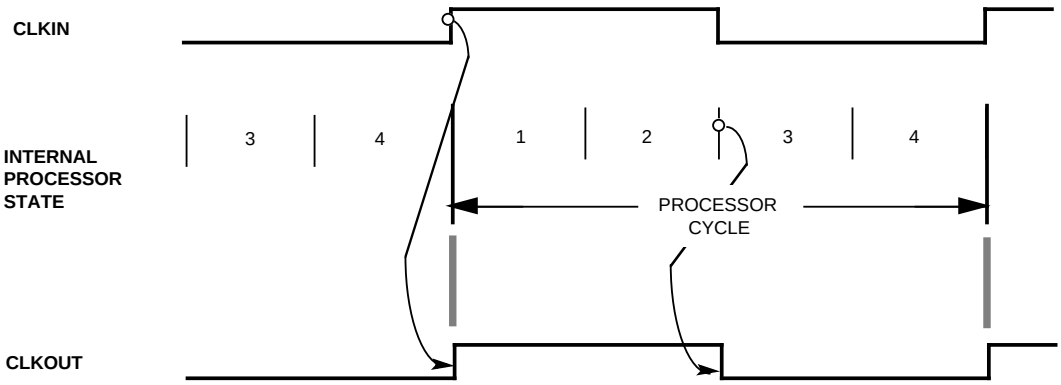


Figure 9.2 Clock Signals & Processor States (ADSP-2101, ADSP-2105, ADSP-2115, ADSP-2111)

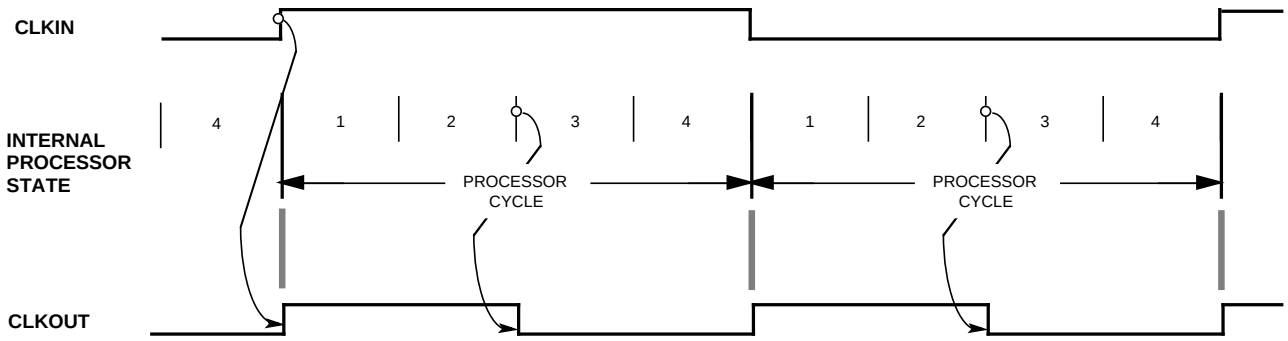


Figure 9.3 Clock Signals & Processor States (ADSP-2171, ADSP-2181, ADSP-21msp58/59)

9.2.1 Synchronization Delay

Each processor has several asynchronous inputs (interrupt requests, for example), which can be asserted in arbitrary phase to the processor clock. The processor synchronizes such signals before recognizing them. The delay associated with signal recognition is called synchronization delay.

Different asynchronous inputs are recognized at different points in the processor cycle. 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 is recognized either in the current cycle or during the next cycle if it remains valid.

Edge-sensitive interrupt requests are latched internally so that the request signal only has to meet the pulse width requirement. To ensure the recognition of any asynchronous input, however, the input must be asserted for at least one full processor cycle plus setup and hold time. Setup and hold times are specified in the data sheet for each individual device.

9.2.2 1x & 1/2x Clock Considerations

Each processor requires only a 1X or 1/2X frequency clock signal. They use what is effectively an on-chip phase-locked loop to generate the higher frequency internal 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 simplified as a result.

9 System Interface

Using a 1X or 1/2X frequency input clock with the phase-locked loop to generate the various internal clocks imposes certain restrictions. The CLKIN signal must be valid long enough to achieve phase lock before RESET can be deasserted. Also, the clock frequency cannot be changed unless the processor is in RESET. Refer to the processor data sheets for details.

9.3 RESET

RESET halts execution and causes a hardware reset of the processor. The RESET signal must be asserted when the processor is powered up to assure proper initialization.

Tables 9.2–9.7 show the RESET state of various registers, including the processors' on-chip memory-mapped status/control registers. The values of any registers not listed are undefined at reset. The contents of on-chip memory are unchanged after RESET, except as shown in Tables 9.2–9.7 for the data-memory-mapped control/status registers. The CLKOUT signal continues to be generated by the processor during RESET, except when disabled on the ADSP-2171, ADSP-2181, or ADSP-21msp58/59.

The contents of the computation unit (ALU, MAC, Shifter) and data address generator (DAG1, DAG2) registers are undefined following RESET.

When RESET is released, the processor's booting operation takes place, depending on the state of the processor's MMAP pin. Program booting is described in Chapter 10, "Memory Interface."

For the ADSP-2111, ADSP-2171, and ADSP-21msp58/59 processors, which include a host interface port, setting the software reset bit in the HSR7 register has the same affect as asserting RESET. This allows either the host processor or the ADSP-21xx to initiate a reset under software control.

In a multiprocessing system with several processors, a synchronous RESET is required.

9.4 SOFTWARE-FORCED REBOOTING

Software-forced reboots can be accomplished in several ways. A software-forced reboot clears the context of the processor and initializes some registers. A *context clear* clears the processor stacks and restart execution at address 0x0000. Table 9.1 shows the different ways each processor can perform a software reboot.

Processor	*Reboot Method*	*Description*
ADSP-2101 ADSP-2105 ADSP-2111 ADSP-2115	Boot Force	Setting the BFORCE bit in the System Control Register causes a reboot
ADSP-2171	Boot Force	Setting the BFORCE bit in the System Control Register causes a reboot
	Powerup Context Reset	Setting the PUCR bit in the SPORT1 Autobuffer & Powerdown Control Register causes a reboot on recovery from powerdown
ADSP-2181	BDMA Context Reset	Setting the BCR bit in the BDMA Control Register *before* writing to the BDMA Word Count Register (BWCOUNT) causes a reboot. Execution starts after the BDMA reboot is completed.
	Powerup Context Reset	Setting the PUCR bit in the SPORT1 Autobuffer & Powerdown Control Register causes a reboot on recovery from powerdown

Table 9.1 Software-Forced Rebooting

Tables 9.2–9.7 show the state of the processor registers after a software-forced reboot. The values of any registers not listed are unchanged by a reboot.

During booting (and rebooting), all interrupts including serial port interrupts are masked and autobuffering is disabled. The serial port(s) 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. If a timer interrupt occurs during the reboot, it is masked. Thus, if more than one timer interrupt occurs during the reboot, the processor latches only the first. A timer overrun can occur.

9 System Interface

Control Field	*Description*	*Reset*	*Reboot*
Bus Exchange Register			
PX	PX register	undefined	undefined
Status Registers			
IMASK	Interrupt service enables	0	0
ASTAT	Arithmetic status	0	0
MSTAT	Mode status	0	unchanged
SSTAT	Stack status	0x55	0x55
ICNTL	Interrupt control	undefined	unchanged
IFC	Interrupt force/clear	0	0
Control Registers (memory-mapped)			
BWAIT	Boot memory wait states	3	unchanged
BPAGE	Boot page	0	unchanged
SPORT1 configure	Configuration	1	unchanged
SPE0	SPORT0 enable	0	unchanged
SPE1	SPORT1 enable	0	unchanged
DWAIT0–4	Data memory wait states	7	unchanged
PWAIT	Program memory wait	7	unchanged
TCOUNT	Timer count register	undefined	operates during reboot
TPERIOD	Timer period register	undefined	unchanged
TSCALE	Timer scale register	undefined	unchanged
Serial Port Control Registers (memory-mapped, one set per SPORT)			
ISCLK	Internal serial clock	0	unchanged
RFSR, TFSR	Frame sync required	0	unchanged
RFSW, TFSW	Frame sync width	0	unchanged
IRFS, ITFS	Internal frame sync	0	unchanged
INVRFS, INVTFS	Invert frame sense	0	unchanged
DTYPE	Companding type, format	0	unchanged
SLEN	Serial word length	0	unchanged
SCLKDIV	Serial clock divide	undefined	unchanged
RFSDIV	RFS divide	undefined	unchanged
Multichannel word enable bits		undefined	unchanged
MCE	Multichannel enable	0	unchanged
MCL	Multichannel length	0	unchanged
MFD	Multichannel frame delay	0	unchanged
INVTDV	Invert transmit data valid	0	unchanged
RBUF, TBUF	Autobuffering enable	0	0
TIREG, RIREG	Autobuffer I index	undefined	unchanged
TMREG, RMREG	Autobuffer M index	undefined	unchanged
FO *(SPORT1 only)*	Flag Out value	undefined	unchanged

Table 9.2 ADSP-2101/ADSP-2115 State After Reset Or Software Reboot

Control Field	*Description*	*Reset*	*Reboot*
Bus Exchange Register			
PX	PX register	undefined	undefined
Status Registers			
IMASK	Interrupt service enables	0	0
ASTAT	Arithmetic status	0	0
MSTAT	Mode status	0	unchanged
SSTAT	Stack status	0x55	0x55
ICNTL	Interrupt control	undefined	unchanged
IFC	Interrupt force/clear	0	0
Control Registers (memory-mapped)			
BWAIT	Boot memory wait states	3	unchanged
BPAGE	Boot page	0	unchanged
SPORT1 configure	Configuration	1	unchanged
SPE1	SPORT1 enable	0	unchanged
DWAIT0–4	Data memory wait states	7	unchanged
PWAIT	Program memory wait	7	unchanged
TCOUNT	Timer count register	undefined	operates during reboot
TPERIOD	Timer period register	undefined	unchanged
TSCALE	Timer scale register	undefined	unchanged
Serial Port 1 Control Registers (memory-mapped)			
ISCLK	Internal serial clock	0	unchanged
RFSR, TFSR	Frame sync required	0	unchanged
RFSW, TFSW	Frame sync width	0	unchanged
IRFS, ITFS	Internal frame sync	0	unchanged
INVRFS, INVTFS	Invert frame sense	0	unchanged
DTYPE	Companding type, format	0	unchanged
SLEN	Serial word length	0	unchanged
SCLKDIV	Serial clock divide	undefined	unchanged
RFSDIV	RFS divide	undefined	unchanged
RBUF, TBUF	Autobuffering enable	0	0
TIREG, RIREG	Autobuffer I index	undefined	unchanged
TMREG, RMREG	Autobuffer M index	undefined	unchanged
FO	Flag Out value	undefined	unchanged

Table 9.3 ADSP-2105 State After Reset Or Software Reboot

Control Field	*Description*	*Reset*	*Reboot*
Bus Exchange Register			
PX	PX register	undefined	undefined
Status Registers			
IMASK	Interrupt service enables	0	0
ASTAT	Arithmetic status	0	0
MSTAT	Mode status	0	unchanged
SSTAT	Stack status	0x55	0x55
ICNTL	Interrupt control	undefined	unchanged
IFC	Interrupt force/clear	0	0
Control Registers (memory-mapped)			
BWAIT	Boot memory wait states	3	unchanged
BPAGE	Boot page	0	unchanged
SPORT1 configure	Configuration	1	unchanged
SPE0	SPORT0 enable	0	unchanged
SPE1	SPORT1 enable	0	unchanged
DWAIT0–4	Data memory wait states	7	unchanged
PWAIT	Program memory wait	7	unchanged
TCOUNT	Timer count register	undefined	operates during reboot
TPERIOD	Timer period register	undefined	unchanged
TSCALE	Timer scale register	undefined	unchanged
Serial Port Control Registers (memory-mapped, one set per SPORT)			
ISCLK	Internal serial clock	0	unchanged
RFSR, TFSR	Frame sync required	0	unchanged
RFSW, TFSW	Frame sync width	0	unchanged
IRFS, ITFS	Internal frame sync	0	unchanged
INVRFS, INVTFS	Invert frame sense	0	unchanged
DTYPE	Companding type, format	0	unchanged
SLEN	Serial word length	0	unchanged
SCLKDIV	Serial clock divide	undefined	unchanged
RFSDIV	RFS divide	undefined	unchanged
Multichannel word enable bits		undefined	unchanged
MCE	Multichannel enable	0	unchanged
MCL	Multichannel length	0	unchanged
MFD	Multichannel frame delay	0	unchanged
INVTDV	Invert transmit data valid	0	unchanged
RBUF, TBUF	Autobuffering enable	0	0
TIREG, RIREG	Autobuffer I index	undefined	unchanged
TMREG, RMREG	Autobuffer M index	undefined	unchanged
FO *(SPORT1 only)*	Flag Out value	undefined	unchanged
Host Interface Port Registers (memory-mapped)			
HDR0-5	HIP data registers	undefined	used during HIP reboot
HSR6	HIP status register	0x0000	used during HIP reboot
HSR7	HIP status register	0x0080	unchanged
HMASK	HIP interrupt enables	0	unchanged

Table 9.4 ADSP-2111 State After Reset Or Software Reboot

Control Field	*Description*	*Reset*	*Reboot*
Bus Exchange Register			
PX	PX register	undefined	undefined
Status Registers			
IMASK	Interrupt service enables	0	0
ASTAT	Arithmetic status	0	0
MSTAT	Mode status	0	unchanged
SSTAT	Stack status	0x55	0x55
ICNTL	Interrupt control	undefined	unchanged
IFC	Interrupt force/clear	0	0
Control Registers (memory-mapped)			
BWAIT	Boot memory wait states	3	unchanged
BPAGE	Boot page	0	unchanged
SPORT1 configure	Configuration	1	unchanged
SPE0	SPORT0 enable	0	unchanged
SPE1	SPORT1 enable	0	unchanged
DWAIT0–4	Data memory wait states	7	unchanged
PWAIT	Program memory wait	7	unchanged
TCOUNT	Timer count register	undefined	operates during reboot
TPERIOD	Timer period register	undefined	unchanged
TSCALE	Timer scale register	undefined	unchanged
ROMENABLE	Program memory ROM enable	0	unchanged
PDFORCE	Powerdown force	0	unchanged
PUCR	Powerup context reset	0	unchanged
XTALDIS	XTAL pindrive disable during powerdown	0	unchanged
XTALDELAY	Delay startup from powerdown (4096 cycles)	0	unchanged
Serial Port Control Registers (memory-mapped, one set per SPORT)			
ISCLK	Internal serial clock	0	unchanged
RFSR, TFSR	Frame sync required	0	unchanged
RFSW, TFSW	Frame sync width	0	unchanged
IRFS, ITFS	Internal frame sync	0	unchanged
INVRFS, INVTFS	Invert frame sense	0	unchanged
DTYPE	Companding type, format	0	unchanged
SLEN	Serial word length	0	unchanged
SCLKDIV	Serial clock divide	undefined	unchanged
RFSDIV	RFS divide	undefined	unchanged
Multichannel word enable bits		undefined	unchanged
MCE	Multichannel enable	0	unchanged
MCL	Multichannel length	0	unchanged
MFD	Multichannel frame delay	0	unchanged
INVTDV	Invert transmit data valid	0	unchanged
RBUF, TBUF	Autobuffering enable	0	0
TIREG, RIREG	Autobuffer I index	undefined	unchanged
TMREG, RMREG	Autobuffer M index	undefined	unchanged

Table 9.5 ADSP-2171 State After Reset Or Software Reboot (cont. on next page)

FO *(SPORT1 only)*	Flag Out value	undefined	unchanged
CLKODIS	CLKOUT disable	0	unchanged
BIASRND	MAC biased rounding	0	unchanged
Host Interface Port Registers (memory-mapped)			
HDR0-5	HIP data registers	undefined	used during HIP reboot
HSR6	HIP status register	0x0000	used during HIP reboot
HSR7	HIP status register	0x0080	unchanged
HMASK	HIP interrupt enables	0	unchanged

Table 9.5 ADSP-2171 State After Reset Or Software Reboot

Control Field	*Description*	*Reset*	*Reboot*
Bus Exchange Register			
PX	PX register	undefined	undefined
Status Registers			
IMASK	Interrupt service enables	0	0
ASTAT	Arithmetic status	0	0
MSTAT	Mode status	0	unchanged
SSTAT	Stack status	0x55	0x55
ICNTL	Interrupt control	undefined	unchanged
IFC	Interrupt force/clear	0	0
Control Registers (memory-mapped)			
BWAIT	Boot memory wait states	3	unchanged
BPAGE	Boot page	0	unchanged
SPORT1 configure	Configuration	1	unchanged
SPE0	SPORT0 enable	0	unchanged
SPE1	SPORT1 enable	0	unchanged
DWAIT0–4	Data memory wait states	7	unchanged
PWAIT	Program memory wait	7	unchanged
TCOUNT	Timer count register	undefined	operates during reboot
TPERIOD	Timer period register	undefined	unchanged
TSCALE	Timer scale register	undefined	unchanged
PDFORCE	Powerdown force	0	unchanged
PUCR	Powerup context reset	0	unchanged
XTALDIS	XTAL pindrive disable during powerdown	0	unchanged
XTALDELAY	Delay startup from powerdown (4096 cycles)	0	unchanged

Table 9.6 ADSP-2181 State After Reset Or Software Reboot (cont. on next page)

Serial Port Control Registers (memory-mapped, one set per SPORT)			
ISCLK	Internal serial clock	0	unchanged
RFSR, TFSR	Frame sync required	0	unchanged
RFSW, TFSW	Frame sync width	0	unchanged
IRFS, ITFS	Internal frame sync	0	unchanged
INVRFS, INVTFS	Invert frame sense	0	unchanged
DTYPE	Companding type, format	0	unchanged
SLEN	Serial word length	0	unchanged
SCLKDIV	Serial clock divide	undefined	unchanged
RFSDIV	RFS divide	undefined	unchanged
Multichannel word enable bits		undefined	unchanged
MCE	Multichannel enable	0	unchanged
MCL	Multichannel length	0	unchanged
MFD	Multichannel frame delay	0	unchanged
INVTDV	Invert transmit data valid	0	unchanged
RBUF, TBUF	Autobuffering enable	0	0
TIREG, RIREG	Autobuffer I index	undefined	unchanged
TMREG, RMREG	Autobuffer M index	undefined	unchanged
FO *(SPORT1 only)*	Flag Out value	undefined	unchanged
CLKODIS	CLKOUT disable	0	unchanged
BIASRND	MAC biased rounding	0	unchanged
External Memory Control Registers (non-memory-mapped)			
DMOVLAY	Data memory overlay select	0	unchanged
PMOVLAY	Program memory overlay select	0	unchanged
(memory-mapped)			
DWAIT	Data memory overlay wait states	0x7	unchanged
PWAIT	Program memory overlay wait states	0x7	unchanged
BMWAIT	Byte memory wait states	0x7	unchanged
IOWAIT0-3	I/O memory wait states	0x7	unchanged
CMSSEL	Composite memory select	0xB	unchanged
Programmable Flag Data & Control Registers (memory-mapped)			
PFDATA	Programmable flag data	undefined	unchanged
PFTYPE	Programmable flag direction	0	unchanged
DMA Control Registers (memory-mapped)			
IDMAA	IDMA Internal Memory Address	0x00	unchanged
IDMAD	IDMA Destination Memory Type	0	unchanged
BIAD	BDMA Internal Memory Address	0	0x20*
BEAD	BDMA External Memory Address	0	0x60*
BTYPE	BDMA Transfer Word Type	0	unchanged
BDIR	BDMA Transfer Direction	0	unchanged
BCR	BDMA Context Reset	1	unchanged
BWCOUNT	BDMA Word Count	0x20	0*
BMPAGE	External Byte Memory Page	0	0*

Table 9.6 ADSP-2181 State After Reset Or Software Reboot

* These values assume that you have just completed an initial BDMA boot load of the ADSP-2181 (MMAP=0 & BMODE=0). For more information on BDMA register contents during the boot loading process see Table 9.8. These values will vary with a processor reboot (other than initial load), since they depend on the previous values.

Control Field	*Description*	*Reset*	*Reboot*
Bus Exchange Register			
PX	PX register	undefined	undefined
Status Registers			
IMASK	Interrupt service enables	0	0
ASTAT	Arithmetic status	0	0
MSTAT	Mode status	0	unchanged
SSTAT	Stack status	0x55	0x55
ICNTL	Interrupt control	undefined	unchanged
IFC	Interrupt force/clear	0	0
Control Registers (memory-mapped)			
BWAIT	Boot memory wait states	3	unchanged
BPAGE	Boot page	0	unchanged
SPORT1 configure	Configuration	1	unchanged
SPE0	SPORT0 enable	0	unchanged
SPE1	SPORT1 enable	0	unchanged
DWAIT0–4	Data memory wait states	7	unchanged
PWAIT	Program memory wait	7	unchanged
TCOUNT	Timer count register	undefined	operates during reboot
TPERIOD	Timer period register	undefined	unchanged
TSCALE	Timer scale register	undefined	unchanged
ROMENABLE	Program memory ROM enable	0	unchanged
PDFORCE	Powerdown force	0	unchanged
PUCR	Powerup context reset	0	unchanged
XTALDIS	XTAL pindrive disable during powerdown	0	unchanged
XTALDELAY	Delay startup from powerdown (4096 cycles)	0	unchanged
Serial Port Control Registers (memory-mapped, one set per SPORT)			
ISCLK	Internal serial clock	0	unchanged
RFSR, TFSR	Frame sync required	0	unchanged
RFSW, TFSW	Frame sync width	0	unchanged
IRFS, ITFS	Internal frame sync	0	unchanged
INVRFS, INVTFS	Invert frame sense	0	unchanged
DTYPE	Companding type, format	0	unchanged
SLEN	Serial word length	0	unchanged
SCLKDIV	Serial clock divide	undefined	unchanged
RFSDIV	RFS divide	undefined	unchanged
Multichannel word enable bits		undefined	unchanged
MCE	Multichannel enable	0	unchanged
MCL	Multichannel length	0	unchanged
MFD	Multichannel frame delay	0	unchanged
INVTDV	Invert transmit data valid	0	unchanged
RBUF, TBUF	Autobuffering enable	0	0
TIREG, RIREG	Autobuffer I index	undefined	unchanged
TMREG, RMREG	Autobuffer M index	undefined	unchanged

Table 9.7 ADSP-21msp58/59 State After Reset Or Software Reboot (cont. on next page)

FO *(SPORT1 only)*	Flag Out value	undefined	unchanged
CLKODIS	CLKOUT disable	0	unchanged
BIASRND	MAC biased rounding	0	unchanged
Host Interface Port Registers (memory-mapped)			
HDR0-5	HIP data registers	undefined	used during HIP reboot
HSR6	HIP status register	0x0000	used during HIP reboot
HSR7	HIP status register	0x0080	unchanged
HMASK	HIP interrupt enables	0	unchanged
Analog Autobuffer/Powerdown Registers			
ARBUF	Receive autobuffer enable	0	0
ATBUF	Transmit autobuffer enable	0	0
control bits	Analog autobuffer control bits	0	unchanged

Table 9.7 ADSP-21msp58/59 State After Reset Or Software Reboot

9.4.1 ADSP-2181 Register Values for BDMA Booting

The state of some ADSP-2181 registers during reset and rebooting is influenced by the MMAP and BMODE pins. If these pins are set for a BDMA boot, the values in the BDMA registers change as shown in Table 9.8.

Register	*Process Description**	*Value Before Boot*	*Value After Boot*
BIAD	BDMA Internal Memory Address. Set for internal address 0.	0	0x20
BEAD	BDMA External Memory Address. Set for external address 0.	0	0x60
BTYPE	BDMA Transfer Word Type. Set for 24-bit program memory words.	0	0
BDIR	BDMA Transfer Direction. Set to transfer data from byte memory.	0	0
BMPAGE	BDMA Page Selection. Set to byte memory page 0.	0	0
BWCOUNT	BDMA Word Count. Set to transfer 32 words.	0x20	0
BMWAIT	BDMA Port Wait States. Set to 7 waits per transfer.	0x7	0x7
BCR	BDMA Context Reset. **	1	1

Table 9.8 BDMA Registers Before And After Initial Boot Loading

* Assuming MMAP=0 and BMODE=0 for a BDMA boot.

** Set to 1 to (a) holdoff instruction execution during BDMA transfer, (b) start execution at address PM(0x0000) after BDMA transfer, and (c) leave a BDMA interrupt pending. This sequence of events occurs if BCR is set before BWCOUNT is written, or after the initial boot.

9 System Interface

9.5 EXTERNAL INTERRUPTS

Each ADSP-2100 family processor has a number of prioritized, individually maskable external interrupts which can be either level- or edge-triggered. These interrupt request pins are named IRQ0, IRQ1, and IRQ2. The IRQ0 and IRQ1 pins are only available as the (optional) alternate configuration of SPORT1. The configuration of SPORT1 as either a serial port or as interrupts (and flags) is determined by bit 10 of the processor's system control register.

The ADSP-2181 processor additionally has two dedicated level-triggered interrupt request pins and one dedicated edge-triggered interrupt request pin; these are IRQL0 , IRQL1, and IRQE.

Internal interrupts, including serial port, timer, host interface port, DMA and analog interface interrupts, are discussed in other chapters. Additional information about interrupt masking, set up, and operation can be found in Chapter 3, "Program Control."

9.5.1 Interrupt Sensitivity

Individual external interrupts can be configured in the ICNTL register as either level-sensitive or edge-sensitive.

Level-sensitive interrupts operate by asserting the interrupt request line (IRQx) until the request is recognized by the processor. Once recognized, the request must be deasserted before unmasking the interrupt so that the DSP does not continually respond to the interrupt.

In contrast, edge-triggered interrupt requests are latched when any high-to-low transition occurs on the interrupt line. The processor 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 than level-sensitive requests since there is never a need to hold or negate the request. With level-sensitive interrupts, however, many interrupting devices can share a single request input; this allows easy system expansion.

An interrupt request will be serviced if it is not masked (in the IMASK register) and a higher priority request is not pending. Valid requests initiate an interrupt servicing sequence that vectors the processor to the appropriate interrupt vector address. The interrupt vector addresses for each family processor are given in Appendix D. There is a synchronization delay

associated with both external interrupt request lines and internal interrupts.

If an interrupt occurs during a waitstated external memory access or 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. Edge-sensitive interrupts are latched, but not serviced, during bus grant (BG) unless the GO mode is enabled.

In order to service an interrupt, the processor must be running and executing instructions. The IDLE instruction can be used to effectively halt processor operations while waiting for an interrupt.

Edge-sensitive and level-sensitive interrupt requests are serviced similarly. Edge-sensitive interrupts may remain active (low) indefinitely, while level-sensitive interrupts must be deasserted before the RTI instruction is executed; otherwise, the same interrupt immediately recurs.

Care must be taken with the serial port (SPORT1) that can be configured for alternate functions (IRQ0 and IRQ1). If the RFS1 or TFS1 input is held low when SPORT1 is configured as the serial port and then is reconfigured as IRQ0 and IRQ1, an interrupt request can be generated. This interrupt request can be cleared with the use of the IFC register.

9.6 FLAG PINS

All ADSP-21xx processors provide flag pins. The alternate configuration of SPORT1 includes a Flag In (FI) pin and a Flag Out (FO) pin. The configuration of SPORT1 as either a serial port or as flags and interrupts is selected by bit 10 of the processor's system control register.

FI can be used to control program branching, using the IF FLAG_IN and IF NOT FLAG_IN conditions of the JUMP and CALL instructions. These conditions are evaluated based on the last state of the FI pin; FLAG_IN is true if FI was last sampled as a 1 and false if last sampled as a 0. FO can be used as a general purpose external signal. The state of FO is also available as a read-only bit of the SPORT1 control register.

The ADSP-2111, ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors have three additional flag output pins: FL0, FL1 and FL2. These flags (and FO) can be controlled in software to signal events or conditions to any external device such as a host processor. The Modify Flag Out instruction, which is conditional, can perform SET, RESET and TOGGLE actions—this instruction allows programs executing on the DSP processor to control the state of its flag output pins. Note that if the condition in the Modify Flag Out instruction is CE

(counter expired), the counter is not decremented as in other IF CE instructions.

Flag outputs FL0, FL1 and FL2 are set to 1 at RESET. The Flag Out (F0) is not affected by RESET.

The ADSP-2181 has eight additional general-purpose flag pins, PF7-0. These flags can be programmed as either inputs or outputs; they default to inputs following reset. The PFx pins are programmed with the use of two memory-mapped registers. The Programmable Flag & Composite Select Control Register determines the flag direction: 1=output and 0=input. The Programmable Flag Data Register is used to read and write the values on the pins. Data being read from a pin configured as an input is synchronized to the processor's clock. Pins configured as outputs drive the appropriate output value. When the PFDATA register is read, any pins configured as outputs will read back the value being driven out.

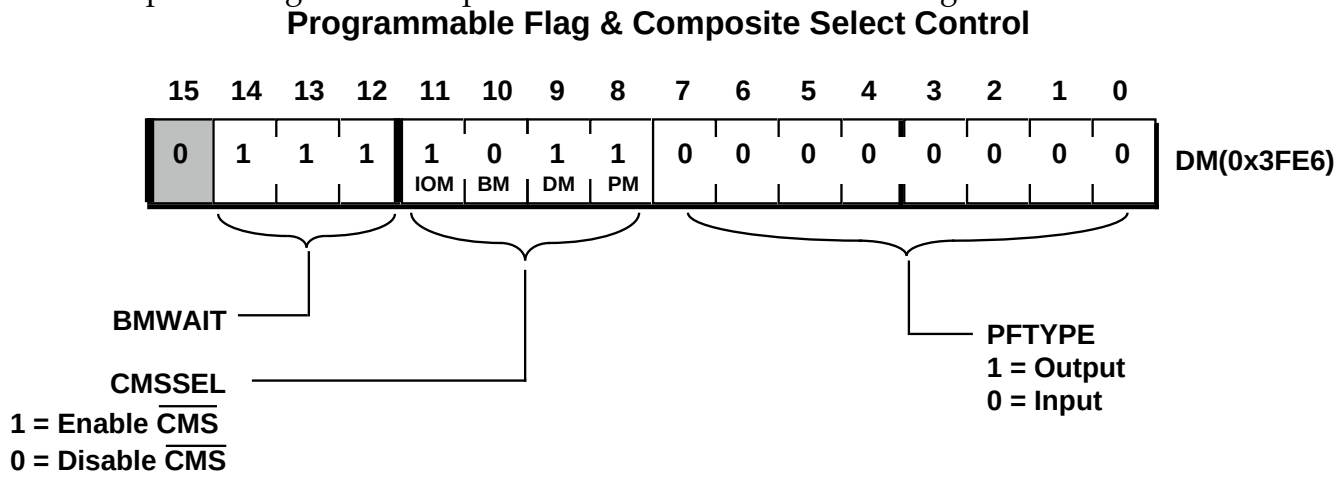


Figure 9.4 Programmable Flag & Composite Select Control Register (ADSP-2181)

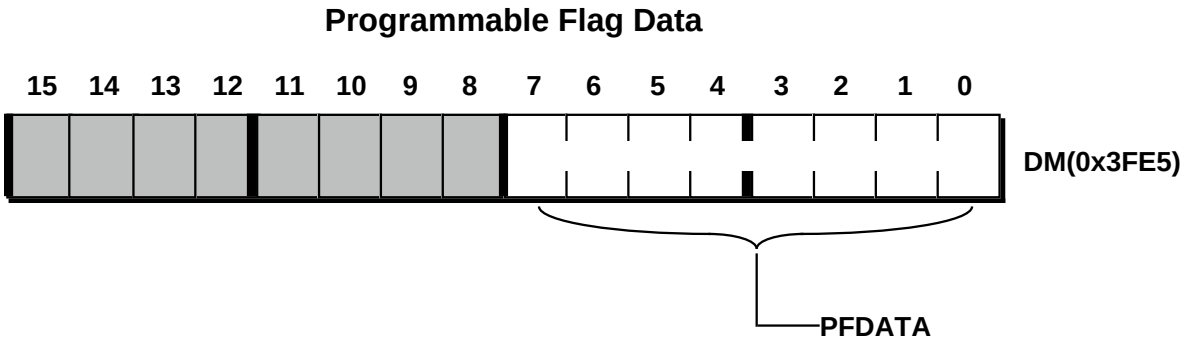


Figure 9.5 Programmable Flag Data Register (ADSP-2181)

9.7 POWERDOWN

The ADSP-2171, ADSP-2181, and ADSP-21msp58/59 provide a powerdown feature that allows the processor to enter a very low power dormant state through hardware or software control. In this CMOS standby state, power consumption is less than 1 mW (approximate). (Refer to the processor data sheet for exact power consumption specifications.)

The powerdown feature is useful for applications where power conservation is necessary, for example in battery-powered operation. Features of powerdown include:

- Internal clocks are disabled
- Processor registers and memory contents are maintained
- Ability to recover from powerdown in less than 100 CLKIN cycles
- Ability to disable internal oscillator when using crystal
- No need to shut down clock for lowest power when using external oscillator
- Interrupt support for executing “housekeeping” code before entering powerdown and after recovering from powerdown
- User selectable powerup context

9 System Interface

Even though the processor is put into the powerdown mode, the lowest level of power consumption still might not be achieved if certain guidelines are not followed. Lowest possible power consumption requires no additional current flow through processor output pins and no switching activity on active input pins. Therefore, a careful analysis of pin loading in your circuit is required. The following sections detail the proper powerdown procedure as well as provide guidelines for clock and output pin connections required for optimum low-power performance.

9.7.1 Powerdown Control

You can control several parameters of powerdown operation through control bits in the SPORT1 Autobuffer/Powerdown Control Register (or Analog Autobuffer/Powerdown Control Register on the ADSP-21msp58/59). This control register is memory-mapped at location 0x3FEF and is shown in Figure 9.6.

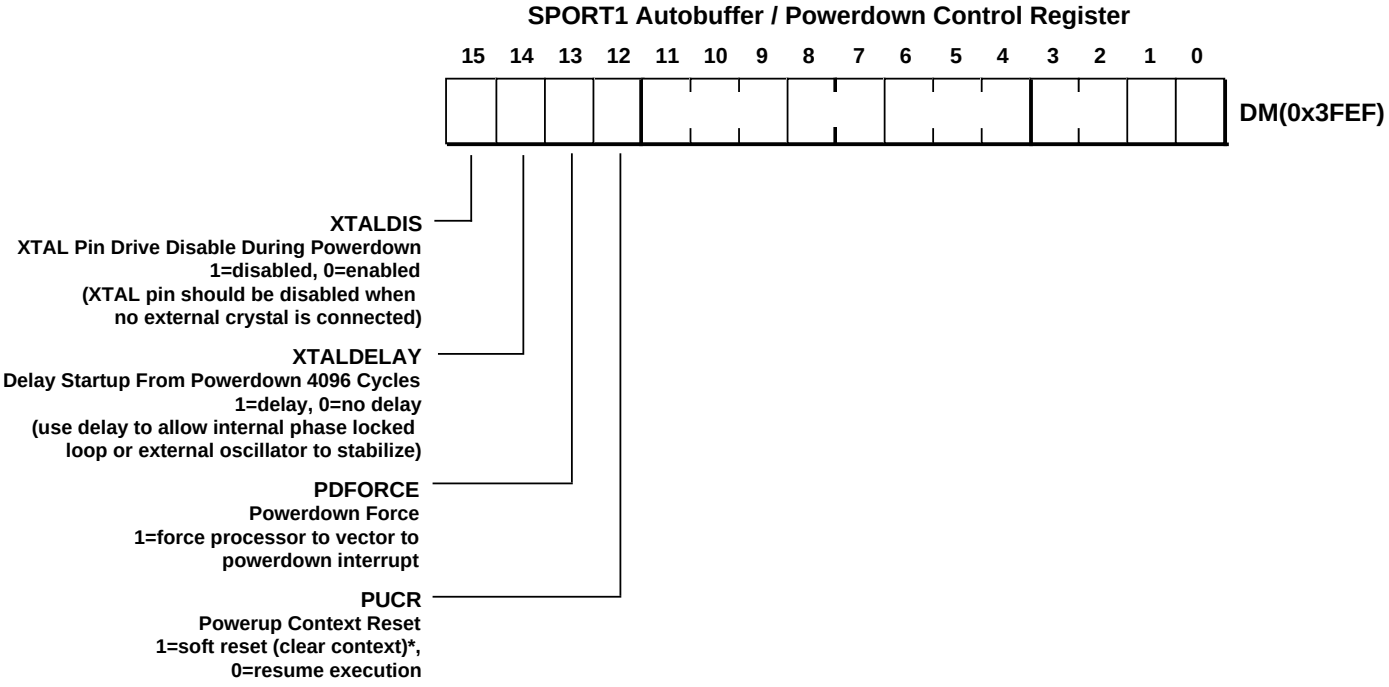


Figure 9.6 SPORT1 Autobuffer / Powerdown Control Register

* PUCR=1: Clears the PC, STATUS, LOOP and CNTR stacks. IMASK and ASTAT registers are cleared to 0 and SSTAT is set to 0x55. The processor will start executing instructions from address 0x0000.

9.7.2 Entering Powerdown

The powerdown sequence is defined as follows.

1.) Initiate the powerdown sequence by applying a high-to-low transition to the PWD pin or by setting the powerdown force control bit (PDFORCE) in the SPORT1 Autobuffer/Powerdown Control Register.

2.) The processor vectors to the non-maskable powerdown interrupt vector at address 0x002C. (Note: The powerdown interrupt is never masked. You must be careful not to cause multiple powerdown interrupts to occur or stack overflow may result. Multiple powerdown interrupts can occur if the PWD input is pulsed while the processor is already servicing the powerdown interrupt.)

3.) Any number of housekeeping instructions, starting at location 0x002C, can be executed prior to the processor entering the powerdown mode. Typically, this section of code is used to configure the powerdown state, disable on-chip peripherals and clear pending interrupts.

4.) The processor now enters powerdown mode when it executes an IDLE instruction (while PWD is asserted). The processor may take either one or two cycles to power down depending upon internal clock states during the execution of the IDLE instruction. All register and memory contents are maintained while in powerdown. Also, all active outputs are held in whatever state they are in before going into powerdown.

If an RTI is executed before the IDLE instruction, then the processor returns from the powerdown interrupt and the powerdown sequence is aborted.

While the processor is in the powerdown mode, the processor is in CMOS standby. This allows the lowest level of power consumption where most input pins are ignored. Active inputs need to be held at CMOS levels to achieve lowest power. More information can be found in the section “Operation During Powerdown” later in this chapter.

9.7.3 Exiting Powerdown

The powerdown mode can be exited with the use of the PWD pin or with RESET. There are also several user-selectable modes for start-up from powerdown which specify a start-up delay as well as specify the program flow after start-up. This allows the program to resume from where it left off before powerdown or for the program context to be cleared.

9.7.3.1 Ending Powerdown With The *PWD* Pin

Applying a low-to-high transition to the PWD pin will take the processor out of powerdown mode. You have the option of selecting the amount of time the processor takes to come out of the powerdown mode with the "delay start-up from powerdown" control bit (XTALDELAY, bit 14 in the Powerdown Control Register.) If this bit is cleared to 0, no additional delay over the quick start-up (100 cycles) is introduced. If this bit is set to 1, a delay of 4096 cycles is introduced. The delay feature is used depending upon the state of an external clock oscillator at the time of powerup or if the internal clock is disabled. This is further discussed in the sections "Systems Using an External TTL/CMOS Clock" and "Systems Using a Crystal and The Internal Oscillator."

You can also program one of two options directing the processor how to resume operation. The context for exiting powerdown is set by bit 12 (PUCR, powerup context reset) of the Powerdown Control Register.

If the PUCR control bit is cleared to 0, the processor will continue to execute instructions following the IDLE instruction. For example, a high-to-low transition is applied to the PWD pin which causes the processor to vector to the powerdown interrupt routine. In this routine, a few housekeeping tasks are performed and the IDLE instruction is executed. The processor powers down. Some time later a low-to-high transition is applied to the PWD pin, causing the processor to exit powerdown mode. Since the PUCR bit is 0, the processor resumes executing instructions in the powerdown interrupt routine, starting at the instruction following the IDLE instruction. When an RTI instruction is encountered, control then passes back to the main routine.

If the PUCR bit is set to 1 for a clear context, the processor resumes operation from powerdown by clearing the PC, STATUS, LOOP and CNTR stacks. The IMASK and ASTAT registers are set to 0 and the SSTAT goes to 0x55. The processor will start executing instructions from address 0x0000.

9.7.3.2 Ending Powerdown With The ***RESET*** Pin

If RESET is asserted while the processor is in the powerdown mode, the processor is reset and instructions are executed from address 0x0000. A boot is performed if the MMAP pin is set to 0. If the RESET pin is used to exit powerdown, then it must be held low for the appropriate number of cycles. If the clock is stopped at powerup or operating at a different frequency at powerup than it was before powerdown, RESET must be held long enough for the oscillator to stabilize plus an additional 1000 CLKIN cycles for the phase locked loop to lock. The time required for the oscillator to stabilize depends upon the type of crystal used and capacitance of the external crystal circuit. Typically 2000 CLKIN cycles is adequate for clock stabilization time.

If the clock was not stopped at powerup and is at a stable frequency at powerup (same as before powerdown), only 5 cycles of RESET are required.

When ending powerdown with RESET, the XTALDELAY (delay start-up from powerdown) control bit is ignored.

9.7.4 Startup Time After Powerdown

The time required to exit the powerdown state depends on whether an internal or external oscillator is used, and the method used to exit powerdown.

9.7.4.1 Systems Using An External TTL/CMOS Clock

When the processor is in powerdown, the external clock signal is ignored if the XTALDIS bit (XTAL pin disable) of the Powerdown Control Register is set to 1. It is therefore not necessary to stop the external clock since no power is wasted while the external clock is running. If the external clock is to be stopped anyway, it must be kept running for (at least) one additional cycle after the IDLE instruction is executed.

The XTALDIS bit should always be set before entering powerdown. This specifies that the XTAL pin is not to be driven by the processor. During powerdown there is no need to drive the XTAL pin when an external oscillator is used. Disabling the XTAL pin drive during powerdown lets the input clock run without wasting power.

After the processor is taken out of the powerdown mode by either the PWD pin or RESET, it will begin executing instructions after a maximum start-up time of 100 CLKIN cycles as long as the clock oscillator is stable and at the same frequency as before powerdown.

If the external clock is unstable when the processor exits powerdown, then the XTALDELAY control bit can be used. This allows time for the external clock to stabilize by inserting an additional 4096-cycle delay before the processor starts to execute instructions. The start-up delay can only be used when the processor is taken out of powerdown mode with the PWD pin.

If the processor is taken out of powerdown by RESET and the clock is stable and at the same frequency as before powerdown, RESET needs to be held for only 5 cycles.

9.7.4.2 Systems Using A Crystal And The Internal Oscillator

A trade-off can be made so that a fast start-up is possible, but power is consumed by leaving the oscillator running during powerdown. If a fast start-up is desired, then you must clear bits 14 (XTALDELAY) and 15 (XTALDIS) of the Powerdown Control Register to 0 before entering powerdown. This selects no additional delay after start-up from powerdown and drives the external crystal during powerdown. In this configuration, the oscillator will continue to operate and the processor will start executing instructions in less than 100 cycles after the low to high signal transition at the PWD pin. The XTAL pin will also be driven and the powerdown power consumption will be higher than the 1 mW specification. The following code example shows the powerdown interrupt routine.

```
{ Sample Powerdown Code                                    }
{ Located at interrupt vector address 0x002C               }
     pwd_int: ax0 = 0x0000; { enable crystal, no delay }
              dm(0x3FEF) = ax0;
              idle;
              rti;
```

If lowest possible power consumption is required, then you must set the XTALDELAY and XTALDIS bits to 1 before entering powerdown. This selects the additional 4096 cycle delay to allow the oscillator to start and the phase locked loop to lock after start-up and disables the drive to the XTAL pin during powerdown. The following code example shows the powerdown interrupt routine.

```
{ Sample Powerdown Code                                     }
{ Located at interrupt vector address 0x002C                }
     pwd_int: ax0 = 0xC000; { disable crystal, delay }
              dm(0x3FEF) = ax0;
              idle;
              rti;
```

Depending on the particular situation and external system conditions, the powerdown modes shown above could be set conditionally. If you want to powerdown for a long time you may want to set the mode for lowest power consumption. If you want to powerdown for a short time, lowest power consumption may not be that important.

If the RESET pin is used to exit powerdown and the clock has been stopped, then RESET must be held low for 1000 CLKIN cycles plus the time required for the phase locked loop to lock and the crystal oscillator to stabilize (typically 2000 CLKIN cycles.) If the clock is running during powerdown, a RESET signal of only 5 cycles is required.

9.7.5 Processor Operation During Powerdown

Some processor circuitry may still be active during powerdown mode. Also, some output pins remain active. A good understanding of these states will allow you to determine the best low-power configuration for your system. By keeping output loading and input switching to a minimum the lowest possible power consumption can be achieved.

9.7.5.1 Interrupts And Flags

Interrupts are latched and can be serviced if the processor exits powerdown without a context reset (PUCR=1). Any activity on the interrupt or flag input pins during powerdown will increase the power consumption. There should also be no resistive load on the flag output pins (as with any active output pin) if lowest power is desired.

9.7.5.2 SPORTS

The circuitry of the serial ports is not directly affected by powerdown. The SPORTs are indirectly affected if an internally generated SCLK or frame sync is required. SPORT circuitry continues to operate during powerdown.

It is possible to clock data into or out of the serial ports during powerdown. You must supply an external serial clock to support operation during powerdown. No interrupts or autobuffer operations will be serviced during powerdown. Instead, the SPORT interrupts are latched and can be serviced if the processor exits powerdown without resetting the processor. Data clocked into the processor will remain in the receive (RX) registers. Autobuffer transfers will occur after the device exits powerdown if the processor is not powered up with RESET. Note that any SPORT activity will increase the power consumption above the 1 mW specification.

If an external serial clock and an external frame sync signal are supplied, data can be clocked into the RX register or out of the TX register during powerdown. Since the TX register can not be updated while the processor is in powerdown, the same value is repeatedly clocked out the serial port. Also, data in the RX register is continually overwritten since the RX register can not be read by the processor during powerdown.

If an external serial clock is used with an internal frame sync, frame sync signals continue to be generated during powerdown since they are derived from the serial clock. Data bits continue to be received with the RX register being overwritten. Since data is only transmitted when the TX register is written, data bits are only transferred out of the processor if the processor is put in powerdown during a serial port transfer. While the processor is being put into powerdown, the serial port transfer in progress is allowed to complete. Since an internally generated transmit frame sync is used, no subsequent frame syncs are generated while in powerdown.

If internal serial clock is used, there is no SPORT activity during powerdown; the serial clock stops.

Lowest power dissipation is achieved when active SPORT pins are not changing during powerdown and are held at CMOS levels.

9.7.5.3 HIP During Powerdown

The circuitry of the Host Interface Port (HIP) is not directly affected by powerdown on the ADSP-2171 and ADSP-21msp58/59. The HIP is indirectly affected since the processor, when in powerdown, is unable to service interrupts or read and write HIP data registers. HIP circuitry continues to operate during powerdown.

The host can write to the HIP register during powerdown but the processor is disabled and cannot service interrupts. Instead, HIP interrupts are latched and can be serviced if the processor exits powerdown without a context reset (PUCR=1).

If the HDR overwrite bit (bit 7 in HSR7) is cleared, a host acknowledge signal will not be asserted until the processor has read data written by the host. During powerdown, the processor is unable to read the data register and the host acknowledge signal will not be asserted. Care must be taken in a system where the host waits for a host acknowledge. In this case, it is possible that the host will “hang” waiting for the acknowledge while the DSP processor is in powerdown.

While in powerdown, the processor can be reset by writing the HSR software reset bit. This will produce the same results as asserting the RESET pin for five cycles (minimum RESET pulse) on the processor. If an external crystal is used and the clock has been stopped, this reset duration is too short; therefore software reset cannot be used in this mode. Note that any HIP activity will increase the power consumption above the 1 mW specification.

Two mode pins, HMD0 and HMD1, are used to put the processor's HIP into one of four possible modes. When HMD0 = 1, the HIP data bus is multiplexed for both address and data. In this case, the HIP data bus inputs are active during powerdown and any bus activity will result in higher power dissipation. Also, inputs must be at CMOS levels. If this host mode is used and there is potential for the bus to be floating, pull-up resistors should be used on the data lines. If you desire the host to communicate with other devices on the bus while the DSP processor is in powerdown, HMD0 should be held low to avoid extra power to be dissipated. When the HIP is put in other modes where data inputs are not active this is not a problem.

Lowest power dissipation is achieved when the HIP pins are not changing during powerdown and are held at CMOS levels.

9.7.5.4 IDMA Port During Powerdown (ADSP-2181)

The IDMA port can receive data during powerdown, but it can not respond with an acknowledge (IACK) signal or increment the IDMA internal address. If you are using a short read or short write and are in the middle of an IDMA transfer, you can complete a single read or write while the processor is in powerdown. If you are using the long read or long write method and are in the middle of an IDMA transfer, your host must be able to handle a "timeout" condition, as the DSP will not return an acknowledge to the transfer in process.

Note that IDMA activity while the DSP is in powerdown uses power and should be avoided to conserve power. For more information on lowest power use, see "Conditions For Lowest Power Consumption."

9.7.5.5 BDMA Port During Powerdown (ADSP-2181)

Do not powerdown the ADSP-2181 during a BDMA transfer. If you do, the DSP will not be able to recover correctly from powerdown and the contents of memory accessed by the ADSP-2181's BDMA port will be corrupted.

If you need to go into powerdown mode, either:

- Verify that the BWCOUNT register contains a zero. If a BDMA transfer is in process, poll the BWCOUNT register to determine when the transfer is done.

  or

- Abort any BDMA transfer in progress by writing 1 to the BWCOUNT register and go into powerdown when the BWCOUNT register contains a zero. (Note that the BDMA transfer is not properly completed in this case.)

9.7.5.6 Analog Interface (ADSP-21msp5x)

You must powerdown the ADSP-21msp58/59's analog interface separately from the processor, as described in the Analog Interface chapter of this manual. The analog interface does not work during powerdown and causes additional power to be dissipated if it is not disabled. The following code example shows a powerdown interrupt routine for the ADSP-21msp58/59:

```
{ Sample Powerdown Code           }
{ located at address 0x002C       }

pwd_int: ax0 = 0x0000;      {powerdown analog interface}
         dm(0x3FEE) = ax0;
         ax0 = 0x0000;      {enable crystal, no delay}
         dm(0x3FEF) = ax0;
         NOP;
         idle;
         rti;
```

It takes three cycles for the analog interface to powerdown. The IDLE instruction should not be executed before these three cycles have elapsed.

9.7.6 Conditions For Lowest Power Consumption

The state of all processor pins during powerdown is shown in Table 9.9.

To assure the lowest power consumption, all active input pins should be held at a CMOS level. All active output pins should be free of resistive load since

load current will increase power dissipation. Some pins will be in one of several states depending upon the connection of mode pins. For example, the ADSP-2171's HIP data bus pins may be either active or inactive depending whether a host write is in progress or how the host mode pins are connected. You must perform a careful analysis of each input and output pin in order to insure lowest power dissipation.

Some inputs are active but ignored. The state of these inputs does not matter as long as they are at a CMOS level.

Pin	*Direction*	*State During Powerdown*
RESET	I	Active
PWD	I	Active
IRQ2	I	Active, latched but not serviced
IRQE	I	(*ADSP-2181*) Active, latched but not serviced
IRQL0	I	(*ADSP-2181*) Active, latched but not serviced
IRQL1	I	(*ADSP-2181*) Active, latched but not serviced
MMAP	I	Active
BR	I	Active, no response until after powerdown
BG	O	Driven HIGH unless bus is granted
CLKIN	I	Input buffer inactive, but XTAL oscillator is active unless XTALDIS bit is set
CLKOUT	O	Driven HIGH
XTAL	O	Driven HIGH if XTALDIS set, inversion of CLKIN otherwise
PWDACK	O	Driven HIGH
PMS	O	Driven HIGH, high impedance if bus granted
DMS	O	Driven HIGH, high impedance if bus granted
BMS	O	Driven HIGH, high impedance if bus granted
IOMS	O	(*ADSP-2181*) Driven HIGH, high impedance if bus granted
CMS	O	(*ADSP-2181*) Driven HIGH, high impedance if bus granted
RD	O	Driven HIGH, high impedance if bus granted
WR	O	Driven HIGH, high impedance if bus granted
ADDR<13:0>	O	High impedance
DATA<23:0>	I	Inactive
DATA<23:0>	O	High impedance
SCLK0	I	Active
SCLK0	O	Driven to static level if internal, high impedance otherwise
TFS0	I	Active if SPORT 0 is enabled
TFS0	O	Driven if configured internal or in multichannel mode and SPORT 0 enabled, high impedance otherwise
RFS0	I	Active if SPORT 0 is enabled
RFS0	O	Driven if configured internal and SPORT 0 enabled, high impedance otherwise
DR0	I	Active if SPORT 0 is enabled
DT0	O	Driven if serial port operating. Output may be static or changing depending upon serial clock, high impedance otherwise

Table 9.9 Pin States During Powerdown (cont. on next page)

9 System Interface

Pin	*Direction*	*State During Powerdown*
SCLK1	I	Active
SCLK1	O	Driven to a static level if internal, high impedance otherwise
TFS1/IRQ1	I	Active if SPORT 1 is enabled or configured alternate (IRQ1)
TFS1	O	Driven if SPORT 1 is enabled and configured for internal transmit framing, high impedance otherwise
RFS1/IRQ0	I	Active if SPORT 1 is enabled or configured alternate (IRQ0)
RFS1	O	Driven if SPORT 1 is enabled and configured for internal receive framing, high impedance otherwise
DR1/FLAGIN	I	Active if SPORT 1 is enabled or configured alternate (FLAGIN)
DT1/FLAGOUT	O	Driven if serial port operating. Output may be static or changing depending upon serial clock. Driven if SPORT 1 is enabled or configured alternate (FLAGOUT)
FL<2:0>	O	Driven to previous value
PF<7:0>	I/O	(*ADSP-2181*) Active
BMODE	I	Active
IRD	I	(*ADSP-2181*) Active, if IS asserted
IWR	I	(*ADSP-2181*) Active, if IS asserted
IS	I	(*ADSP-2181*) Active
IAL	I	(*ADSP-2181*) Active, if IS asserted
IAD	I/O	(*ADSP-2181*) Active, if an operation in progress
IACK	O	(*ADSP-2181*) Active
HSIZE	I	(*ADSP-2171, ADSP-21msp5x*) Active
HMD0	I	(*ADSP-2171, ADSP-21msp5x*) Active
HMD1	I	(*ADSP-2171, ADSP-21msp5x*) Active
HSEL	I	(*ADSP-2171, ADSP-21msp5x*) Active
HRD	I	(*ADSP-2171, ADSP-21msp5x*) Active
HWR	I	(*ADSP-2171, ADSP-21msp5x*) Active
HADR<2:0>	I	(*ADSP-2171, ADSP-21msp5x*) Active
HDATA<15:0>	I	(*ADSP-2171, ADSP-21msp5x*) Active if host writing or HMD1 and HA2/HALE HIGH, inactive otherwise
HDATA<15:0>	O	(*ADSP-2171, ADSP-21msp5x*) Driven if host reading, high impedance otherwise
HACK	O	(*ADSP-2171, ADSP-21msp5x*) Driven
VIN (NORM)	I	(*ADSP-21msp5x*) Inactive, set analog powerdown bit
VIN (AUX)	I	(*ADSP-21msp5x*) Inactive, set analog powerdown bit
VFB (NORM)	O	(*ADSP-21msp5x*) Inactive, set analog powerdown bit
VFB (AUX)	O	(*ADSP-21msp5x*) Inactive, set analog powerdown bit
VOUTP	O	(*ADSP-21msp5x*) Driven low in powerdown
VOUTN	O	(*ADSP-21msp5x*) Driven low in powerdown
VREF	O	(*ADSP-21msp5x*) Reference turned off

Table 9.9 Pin States During Powerdown

9.7.7 PWDACK Pin

The powerdown acknowledge pin (PWDACK) is an output that indicates when the processor is powered down. This pin is driven high by the processor when it has powered down and is driven low when the processor has completed its powerup sequence. A low level on the PWDACK pin also indicates that there is a valid CLKOUT signal and that instruction execution has begun. Figure 9.7 shows an example of timing for the powerdown and restart sequence.

The processor is executing code when the PWD pin is brought low. The processor vectors to the powerdown interrupt vector and an IDLE instruction is executed causing the processor to go into powerdown. The CLKOUT and PWDACK signals are driven high by the processor. At this point, the input clock pin is ignored. If the processor is put into the powerdown mode via the powerdown force bit in the powerdown control register, the result is the same as described above.

The input clock is started and the PWD pin is brought high. After the necessary start-up cycles the processor brings the PWDACK output low, begins driving the CLKOUT pin with a clock signal and begins to fetch the instruction after the IDLE instruction. The processor then resumes normal operation.

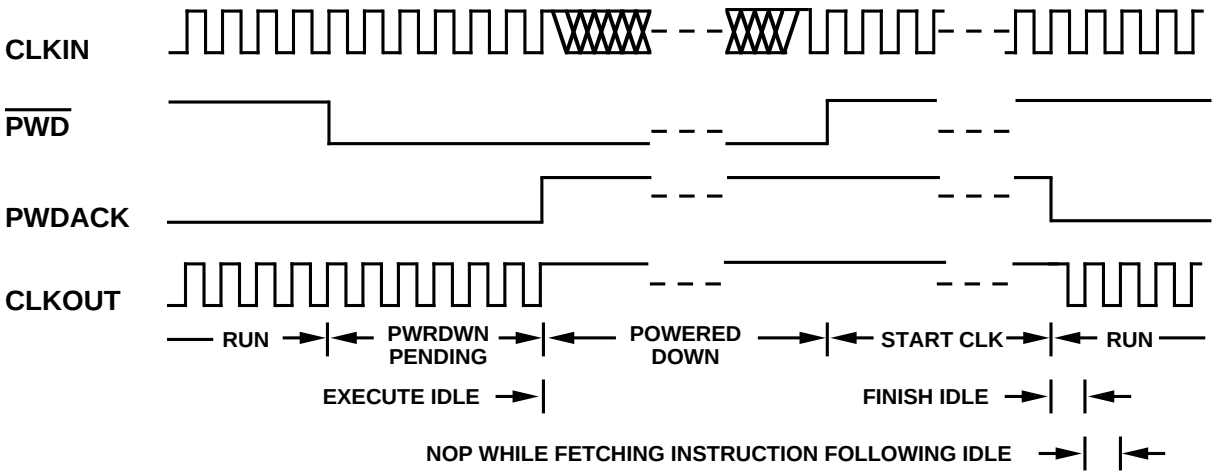


Figure 9.7 Powerdown Timing Example

When powerdown is terminated with the RESET pin or if a start-up delay is selected, a low level on the PWDACK pin only indicates the start of oscillations on the CLKOUT pin. It will not necessarily indicate the start of instruction execution.

The state of PWDACK and also the CLKOUT signal is undefined during the first 100 cycles of initial reset.

9.7.8 Using Powerdown As A Non-Maskable Interrupt

The powerdown interrupt is never masked. It is possible to use this interrupt for other purposes if desired. The processor will not go into powerdown until an IDLE instruction is executed. If an RTI is executed before the IDLE instruction, then the processor returns from the powerdown interrupt and the powerdown sequence is aborted.

It is possible to place a series of instructions at the powerdown interrupt vector location 0x002C. This routine should end with an RTI instruction and not contain an IDLE instruction if the interrupt is to be used for purposes other than powerdown.

Memory Interface 10

10.1 OVERVIEW

The ADSP-2100 family has a modified Harvard architecture in which data memory stores data and program memory stores both instructions and data. Each processor contains on-chip RAM and/or ROM, so that a portion of the program memory space and a portion of the data memory space reside on-chip. Each processor (except the ADSP-2181) also has a boot memory space in addition to the data and program spaces. The ADSP-2181 has a byte memory space instead of the boot memory space. The boot memory space and byte memory space can be used to load on-chip program memory with code from an external EPROM at reset.

In each ADSP-2100 family device, memory is connected with the internal functional units by four on-chip buses: the data memory address bus (DMA), data memory data bus (DMD), program memory address bus (PMA), and program memory data bus (PMD). The internal PMA bus and DMA bus are multiplexed into a single address bus which is extended off-chip. Likewise, the internal PMD bus and DMD bus are multiplexed into a single external data bus. The sixteen MSBs of the external data bus are used as the DMD bus: external bus lines D_{23-8} are used for DMD_{15-0}.

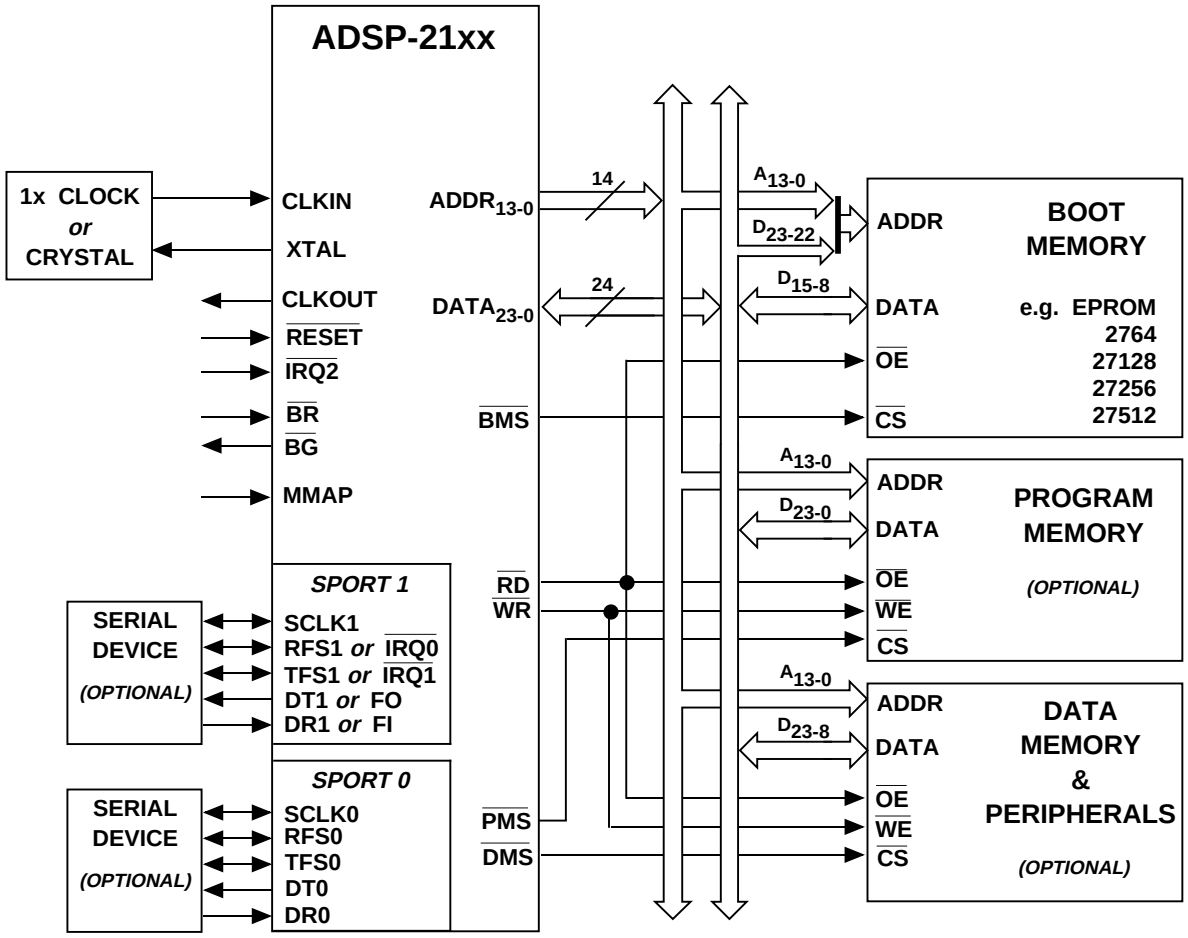
There are three separate memory spaces: data memory, program memory and boot (or byte) memory. The $\overline{PMS}$, $\overline{DMS}$, and $\overline{BMS}$ signals indicate which memory space is being accessed. Because the program memory and data memory buses are multiplexed off-chip, if more than one external transfer must 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. Figure 10.1 shows the external memory buses and control signals (for all ADSP-21xx processors except the ADSP-2181).

All external memories may have automatic wait state generation associated with them. The number of wait states—each equal to one instruction cycle—is programmable.

10 Memory Interface

This chapter includes example timing diagrams for the memory interfaces of the ADSP-21xx processors. For each bus transaction, only the sequence of events is described; you must consult the processor data sheets for actual timing parameters. All timing diagrams use CLKOUT as a reference, which indicates the instruction execution rate.

The memory interfaces of the ADSP-2181 are described separately in the second half this chapter.



NOTES
1. Applies to all ADSP-21xx processors except ADSP-2181.
2. ADSP-2171 and ADSP-21msp58/59 use a 1/2x CLKIN signal.
3. Unused data bus lines may be left floating.
4. The two MSBs of the data bus (D23-22) are used to supply the two MSBs of the boot memory EPROM address. This is only required for the 27256 and 27512.

Figure 10.1 ADSP-21xx System With External Memory

10.2 PROGRAM MEMORY INTERFACE

This section describes the program memory interface of all ADSP-21xx processors except the ADSP-2181.

The processors address 16K of 24-bit wide program memory, up to 2K on-chip and the remainder external, using the control lines shown in Figure 10.1. The processors supply a 14-bit address on the program memory address bus (PMA) which is driven off-chip on the address bus in the case of external program memory accesses. Instructions or data are transferred across the 24-bit program memory data (PMD) bus which is also multiplexed off-chip. For a dual off-chip data fetch, the data from program memory is read first, then the data memory data. 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 $\overline{\text{CE}}$ (Chip Enable), $\overline{\text{RD}}$ to $\overline{\text{OE}}$ (Output Enable) and $\overline{\text{WR}}$ to $\overline{\text{WE}}$ (Write Enable) of your memory.

10.2.1 External Program Memory Read / Write

On-chip memory accesses do not drive any external signals. $\overline{\text{PMS}}$, $\overline{\text{DMS}}$, $\overline{\text{RD}}$, and $\overline{\text{WR}}$ remain high (deasserted); the address and data buses are tristated. Off-chip program memory access happens in this sequence:

1. The processor places the address on the PMA bus, which is multiplexed off-chip, and $\overline{\text{PMS}}$ is asserted.

2. $\overline{\text{RD}}$ or $\overline{\text{WR}}$ is asserted.

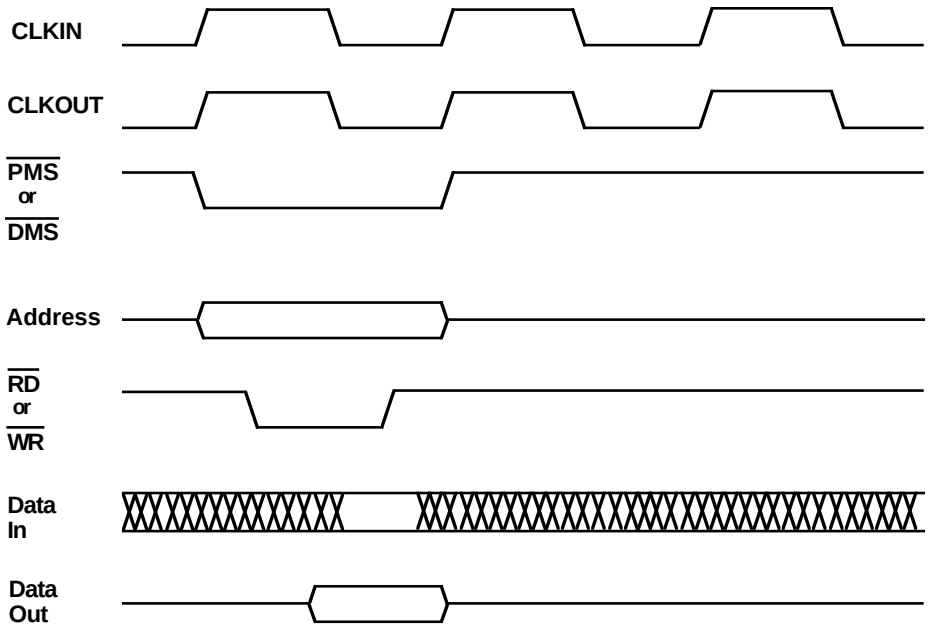
3. Within a specified time, data is placed on the data bus, multiplexed to the internal PMD bus.

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

5. $\overline{\text{PMS}}$ is deasserted.

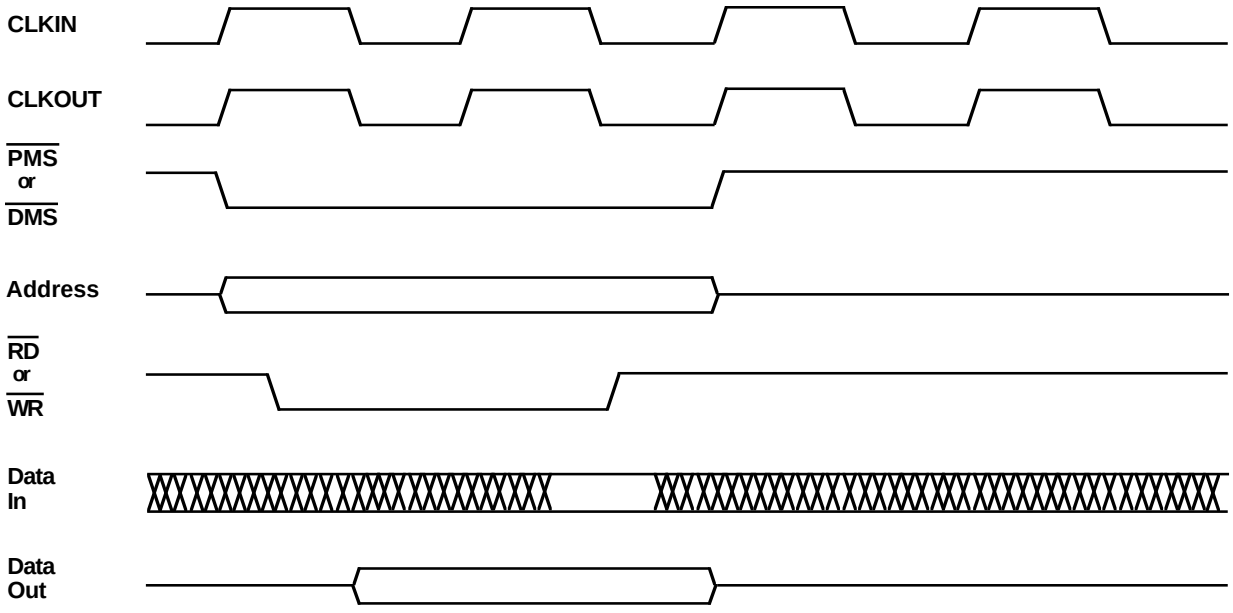
The basic read and write cycles are illustrated in Figure 10.2 on the next page. Figure 10.2A shows zero wait states and 10.2B shows the effect of one wait state.

10 Memory Interface



External Program/Data Memory Read/Write
PWAIT=0, DWAIT=0 (no wait states added)

Figure 10.2A Memory Read And Write, No Wait States



External Program/Data Memory Read/Write
PWAIT=1, DWAIT=1 (one wait state added)

Figure 10.2B Memory Read And Write, One Wait State

The program memory interface can generate 0 to 7 wait states for external memory devices. The program memory wait state field (PWAIT) in the system control register is shown in Figure 10.3. PWAIT defaults (after RESET) to seven wait states for program memory accesses.

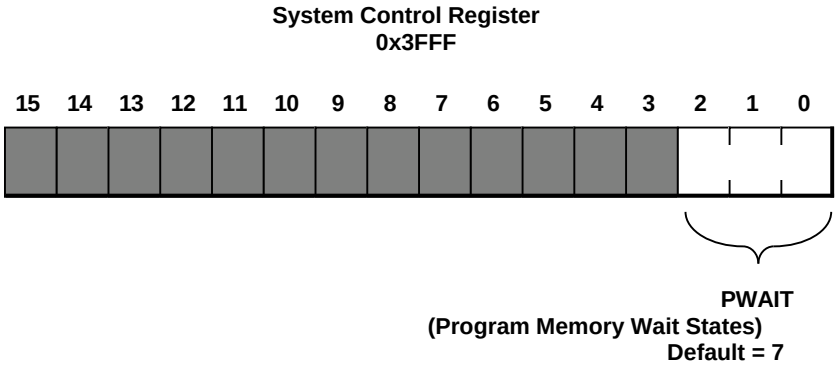


Figure 10.3 Program Memory Wait State Field In System Control Register

10.2.2 Program Memory Maps

For all RAM-based processors except the ADSP-2181, the program memory space is mapped in one of two configurations depending on the state of the MMAP pin. Figure 10.4 shows these configurations for the processors with 2K internal program memory (ADSP-2101, ADSP-2111, ADSP-2171, ADSP-21msp58), and Figure 10.5 shows the same information for the processors with 1K internal program memory (ADSP-2105, ADSP-2115).

When MMAP=0, internal RAM occupies 2K words beginning at address 0x0000. In this configuration, the boot loading sequence is automatically initiated when RESET is released (as described in "Boot Memory Interface").

When MMAP=1, words of external program memory begin at address 0x0000 and internal RAM is located in the upper 2K words, beginning at address 0x3800. In this configuration, program memory is not loaded although it can be written to and read from under program control.

The program memory space can hold instructions and data intermixed in any combination. The ADSP-21xx 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. The restart vector is at program memory address 0x0000. The interrupt vector locations are given in Chapter 3 and in Appendix D.

10 Memory Interface

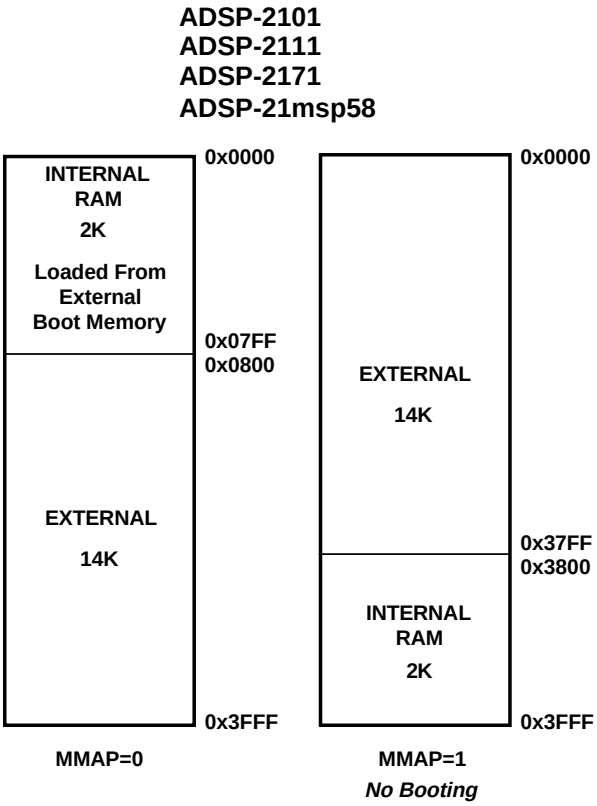


Figure 10.4 Program Memory Maps (2K internal RAM)

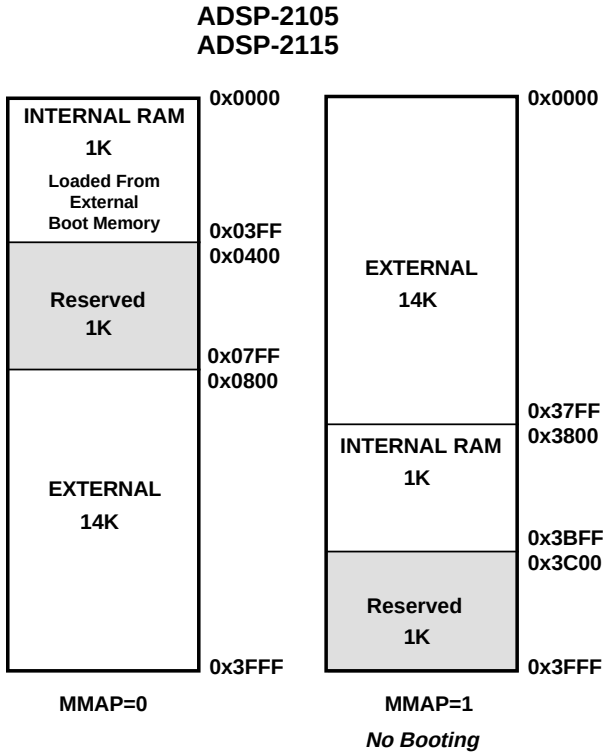


Figure 10.5 Program Memory Maps (1K internal RAM)

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

10.2.3 ROM Program Memory Maps

The ADSP-2172 and ADSP-21msp59 processors contain mask-programmable ROM on-chip. The program memory maps for these processors are shown in Figures 10.6 and 10.7. The ADSP-2172 contains 8K of ROM and the ADSP-21msp59 contains 4K.

On the ADSP-2172 and ADSP-21msp59, the ROM is enabled by setting the ROMENABLE bit in the Data Memory Wait State control register (at address DM[0x3FFE]). When the ROMENABLE bit is set to 1, addressing program memory in the ROM range will access the on-chip ROM. When ROMENABLE is set to 0, addressing program memory in this range will access external program memory. The ROMENABLE bit is initialized to 0 after reset unless MMAP *and* BMODE=1.

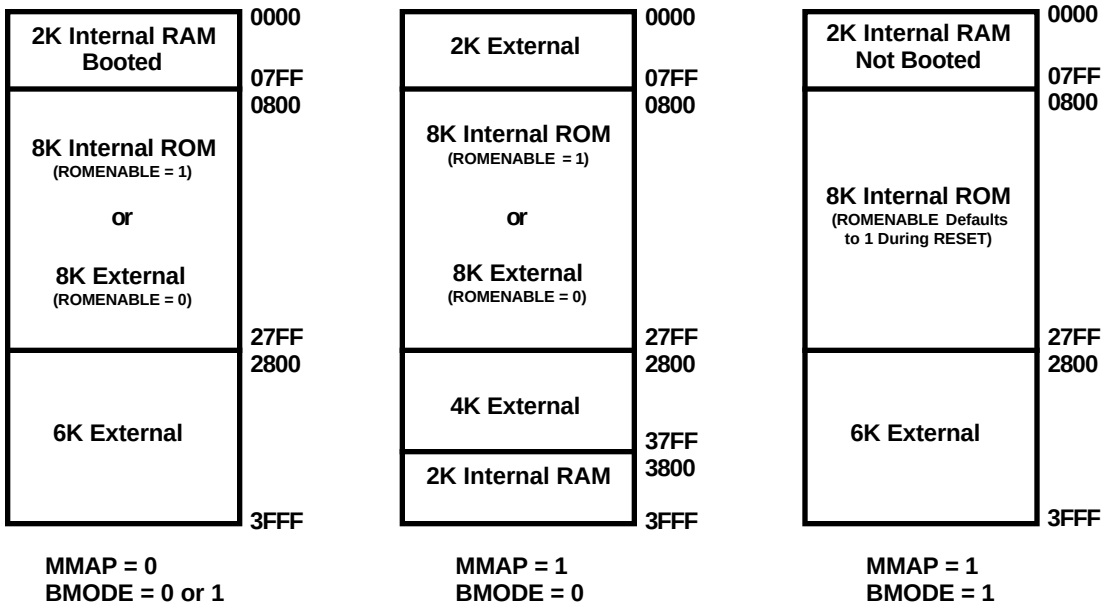


Figure 10.6 ADSP-2172 Program Memory Map

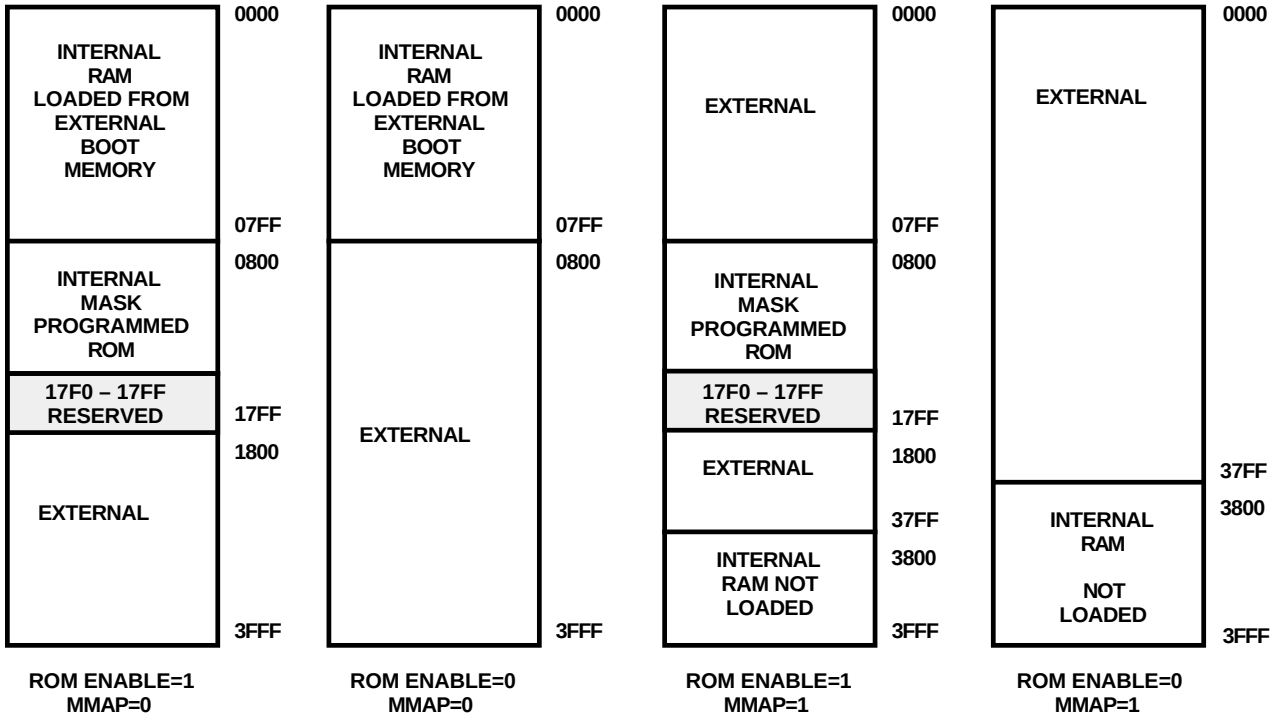


Figure 10.7 ADSP-21msp59 Program Memory Map

10 Memory Interface

When the MMAP and BMODE pins both are set to 1, the ADSP-2172 (or ADSP-21msp59) will operate in *standalone ROM execution* mode. When MMAP=1 and BMODE=1, the ROM is automatically enabled and execution begins from program memory location 0x0800 at the start of ROM. This lets an embedded design operate without external memory components. To operate in this mode, the ROM-coded program must copy an interrupt vector table to the appropriate locations in program memory RAM. In this mode, the ROMENABLE bit defaults to 1 during reset. Table 10.1 summarizes the booting and startup execution modes for the ADSP-2172 and ADSP-21msp59.

	BMODE = 0	*BMODE = 1*
MMAP = 0	Boot from EPROM, then execution starts at internal RAM location 0x0000	Boot from HIP, then execution starts at internal RAM location 0x0000
MMAP = 1	No booting, execution starts at external memory location 0x0000	Standalone mode, execution starts at internal ROM location 0x0800

Table 10.1 Booting Mode for ADSP-2172, ADSP-21msp59

The ADSP-216x processors are memory-variant versions of the ADSP-2101 and ADSP-2103 that contain factory-programmed on-chip ROM program memory. The ADSP-2161, ADSP-2163, and ADSP-2165 are 5.0V supply processors based on the ADSP-2101. The ADSP-2162, ADSP-2164, and ADSP-2166 are 3.3V supply processors based on the ADSP-2103. These devices offer different amounts of on-chip memory for program and data storage, as shown in Table 10.2.

Feature	*2161*	*2162*	*2163*	*2164*	*2165*	*2166*
Data Memory (RAM)	½K	½K	½K	½K	4K	4K
Program Memory (ROM)	8K	8K	4K	4K	12K	12K
Program Memory (RAM)	–	–	–	–	1K	1K

Table 10.2 ADSP-216x ROM-Programmed Processors

Figures 10.8, 10.9, and 10.10 show the program memory maps for the ADSP-2161/62, ADSP-2163/64, and ADSP-2165/66, respectively.

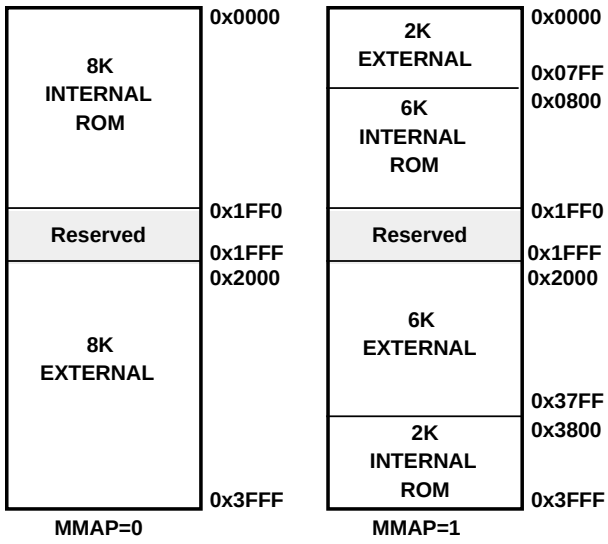


Figure 10.8 ADSP-2161/62 Program Memory Maps

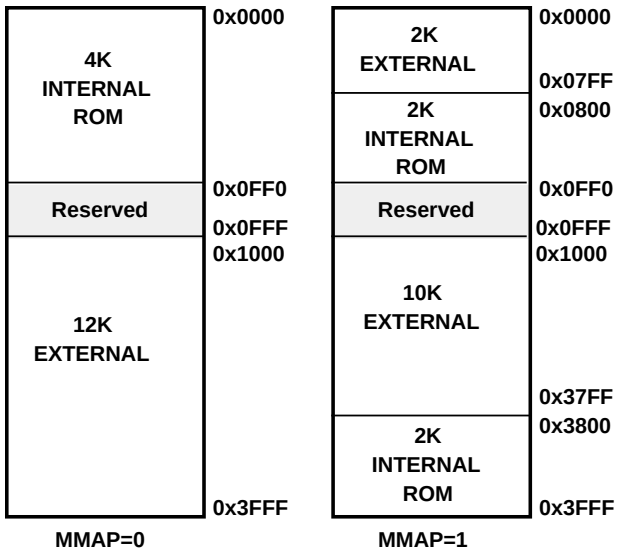


Figure 10.9 ADSP-2163/64 Program Memory Maps

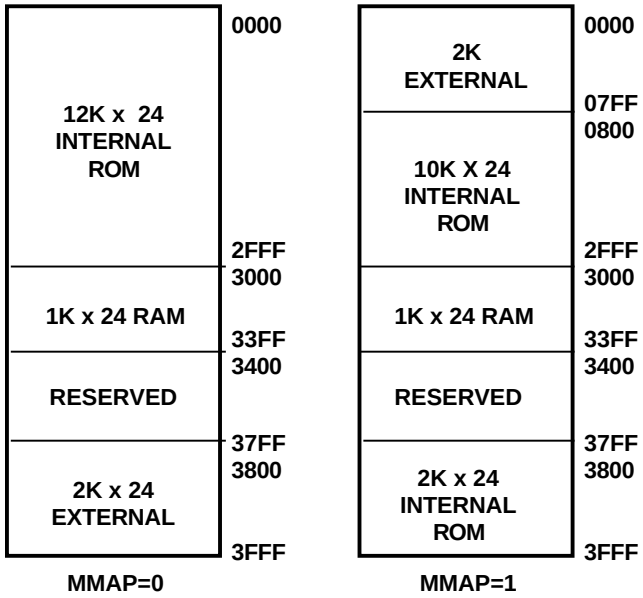


Figure 10.10 ADSP-2165/66 Program Memory Maps

10 Memory Interface

10.3 DATA MEMORY INTERFACE

This section describes the data memory interface of all ADSP-21xx processors except the ADSP-2181.

The processors supply 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 $\overline{\text{CE}}$ (Chip Enable), $\overline{\text{RD}}$ to $\overline{\text{OE}}$ (Output Enable) and $\overline{\text{WR}}$ to $\overline{\text{WE}}$ (Write Enable) of your memory.

10.3.1 External Data Memory Read/Write

Internal data memory accesses are transparent to the external memory interface. Only off-chip accesses drive the memory interface. Off-chip data memory accesses follow the same sequence as off-chip program memory accesses, namely:

1. The processor places the address on the DMA bus, which is multiplexed off-chip, and $\overline{\text{DMS}}$ is asserted.

2. $\overline{\text{RD}}$ or $\overline{\text{WR}}$ is 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 $\overline{\text{RD}}$ (or $\overline{\text{WR}}$) is deasserted.

5. $\overline{\text{DMS}}$ is deasserted.

The basic read and write cycles are illustrated in Figure 10.2.

For a dual off-chip data fetch, the data from program memory is read first, then the data memory data.

10.3.2 Data Memory Maps

The processors can address a total of 16K words of 16-bit data memory. On-chip data memory is 1K in size and starts at address 0x3800 on the ADSP-2101 and ADSP-2111. On-chip data memory is 512 locations in size on the ADSP-2105 and ADSP-2115, again starting at address 0x3800. On-chip data memory is 2K in size on the ADSP-2171 and ADSP-21msp58/59, beginning at address 0x3000.

The processors' control and status registers are mapped into the top 1K of data memory, addresses 0x3C00-0x3FFF. The rest of the top 1K is reserved. External data memory is available for additional data storage. Figures 10.11, 10.12, and 10.13 show the data memory maps for each ADSP-21xx processor.

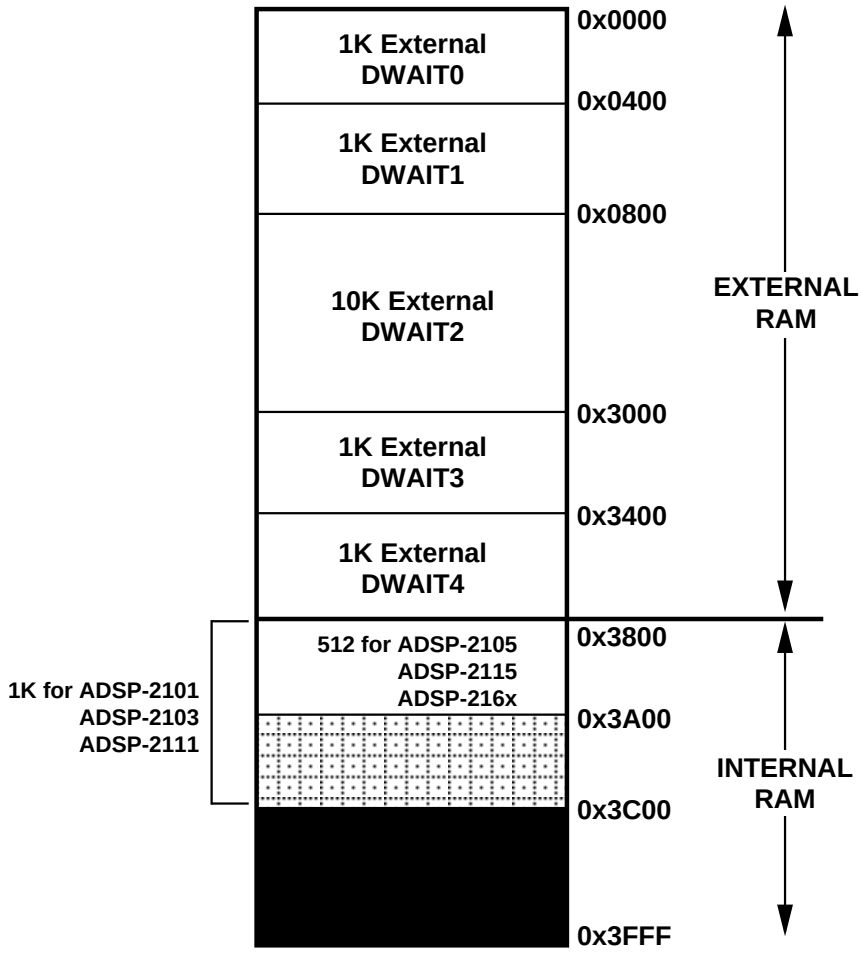


Figure 10.11 Data Memory Map (ADSP-2101, ADSP-2111, ADSP-2105, ADSP-2115, ADSP-2161/62/63/64)

10 Memory Interface

As shown in Figure 10.11, the ADSP-2101, ADSP-2111, ADSP-2105, ADSP-2115, and ADSP-2161/62/63/64 processors have five external wait state zones (DWAIT0–DWAIT4). Each of the five zones of external data memory has its own programmable number of wait states. Wait states are extra cycles that the processor 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.

As shown in Figures 10.12 and 10.13, the ADSP-2171, ADSP-21msp58/59, and ADSP-2165/66 processors each have three wait state zones for external data memory.

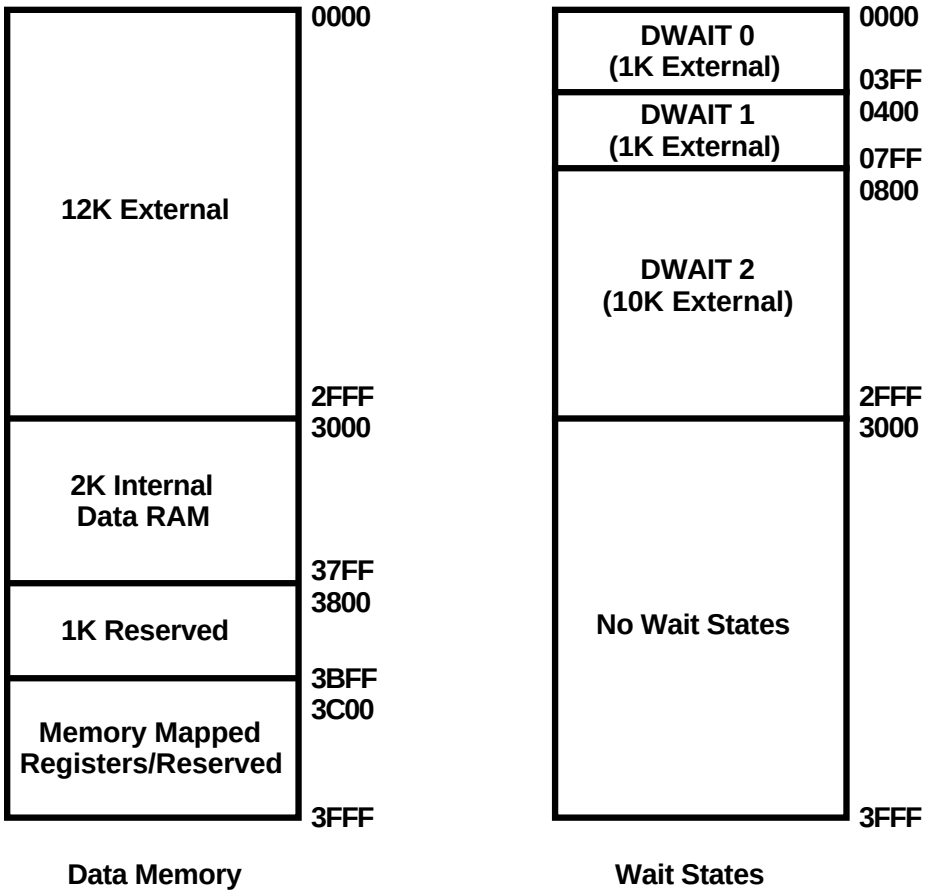


Figure 10.12 Data Memory Map (ADSP-2171, ADSP-21msp58/59)

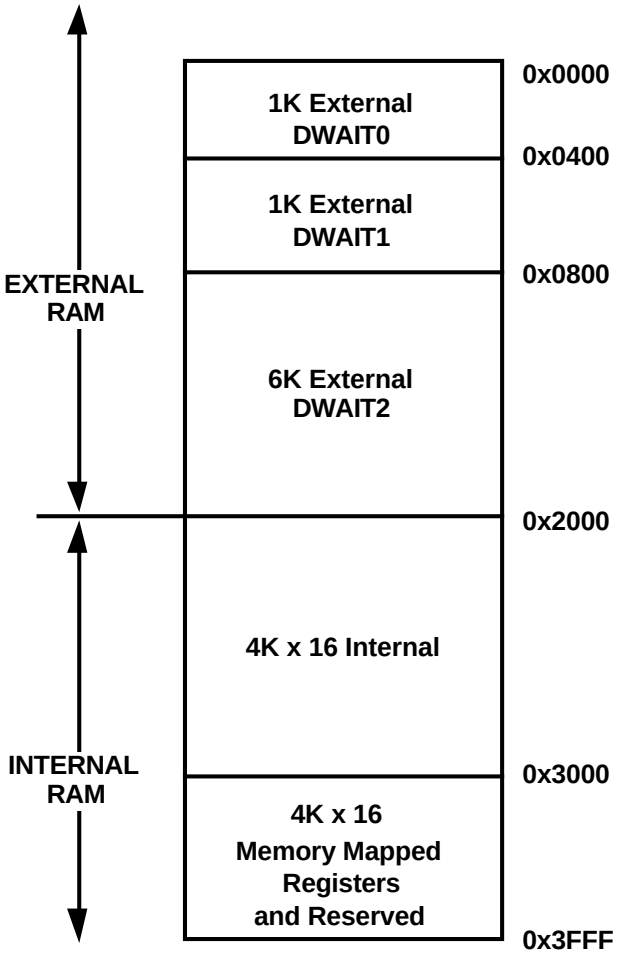


Figure 10.13 Data Memory Map (ADSP-2165/66)

The Data Memory Waitstate control register has a separate field for each zone of external memory. Each 3-bit field specifies the number (0-7) of wait states for the corresponding zone of memory; all zones default to 7 wait states after $\overline{\text{RESET}}$. Figure 10.14 shows this control register for the ADSP-2101, ADSP-2111, ADSP-2105, ADSP-2115, and ADSP-2161/62/63/64 processors. Figure 10.15 shows the register for the ADSP-2171/72 and ADSP-21msp58/59 processors; on the ADSP-2172 and ADSP-21msp59, one bit in this register is used to enable or disable the on-chip ROM.

10 Memory Interface

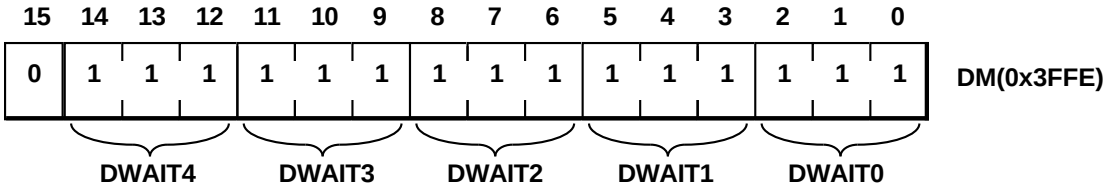


Figure 10.14 Data Memory Waitstate Control Register (ADSP-2101, ADSP-2111, ADSP-2105, ADSP-2115, ADSP-2161/62/63/64)

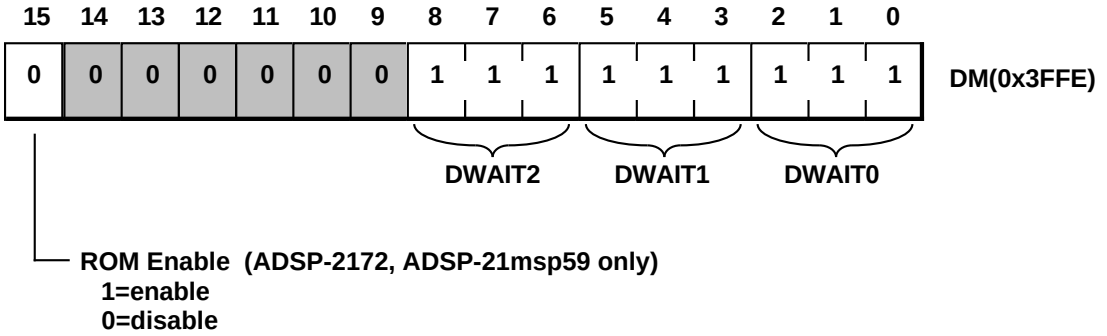


Figure 10.15 Data Memory Waitstate Control Register (ADSP-2171/72, ADSP-21msp58/59)

10.3.3 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 ADSP-2100 Family Development 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.

The 16 MSBs of the external data bus (D_{23-8}) are connected to the 16 LSBs of the internal DMD bus, so D_{23-8} should be used for 16-bit peripherals.

10.4 BOOT MEMORY INTERFACE

This section describes the boot memory interface of all ADSP-21xx processors except the ADSP-2181.

The entire internal program memory, or any portion of it, can be loaded from an external source using a boot sequence. To interface with inexpensive EPROM, the processor loads instructions one byte at a time.

Automatic booting at reset depends on the state of the MMAP pin at the time of processor reset. The boot sequence occurs if the MMAP pin is 0. The boot sequence can also be initiated after reset by software.

The ADSP-2111, ADSP-2171, and ADSP-21msp5x processors, which include a Host Interface Port (HIP), can boot using either the memory interface or the HIP (from a host computer). The state of the BMODE pin determines which method is used: the memory interface if BMODE=0, or the HIP if BMODE=1. Booting through the HIP is described in Chapter 7.

$\overline{BR}$ is recognized during the booting sequence. The bus is granted after completion of loading the current byte.

The ADSP-216x contain on-chip program memory ROM; on these devices, no booting occurs.

10.4.1 Boot Pages

Boot memory is organized into eight pages, each of which can be 8K bytes long. Every fourth byte of a page is an "empty" byte, except the first one, which contains the page length. Each set of three bytes between successive empty bytes contains an instruction. 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:

page length = (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 ROM locations.

The PROM Splitter utility, part of the ADSP-2100 Family Development Software tools, calculates the proper page length for your program and orders the bytes of your program as shown in Figure 10.16 (on the next page).

10 Memory Interface

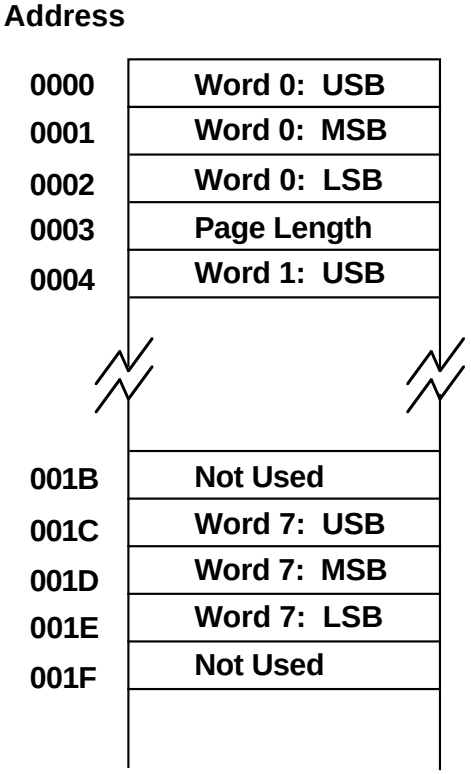


Figure 10.16 EPROM Contents

10.4.2 Powerup Boot & Software Reboot

Upon a hardware or software reset, the boot sequence occurs if the MMAP pin is a logical 0. The boot sequence on reset always loads boot page 0. After reset, boot loading can occur under program control from any one of up to 8 different boot pages. The boot page select field (BPAGE) in the memory-mapped System Control Register (see Figure 10.17) specifies which boot page is to be loaded. To boot from a specific boot page, set BPAGE to the desired page number and, in 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 and (possibly) the number of wait states, there is no difference between a software-forced boot sequence and a reset boot sequence.

Tables 9.2–9.7 in the System Interface chapter show 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 starts at the restart vector, at program memory location 0x0000.

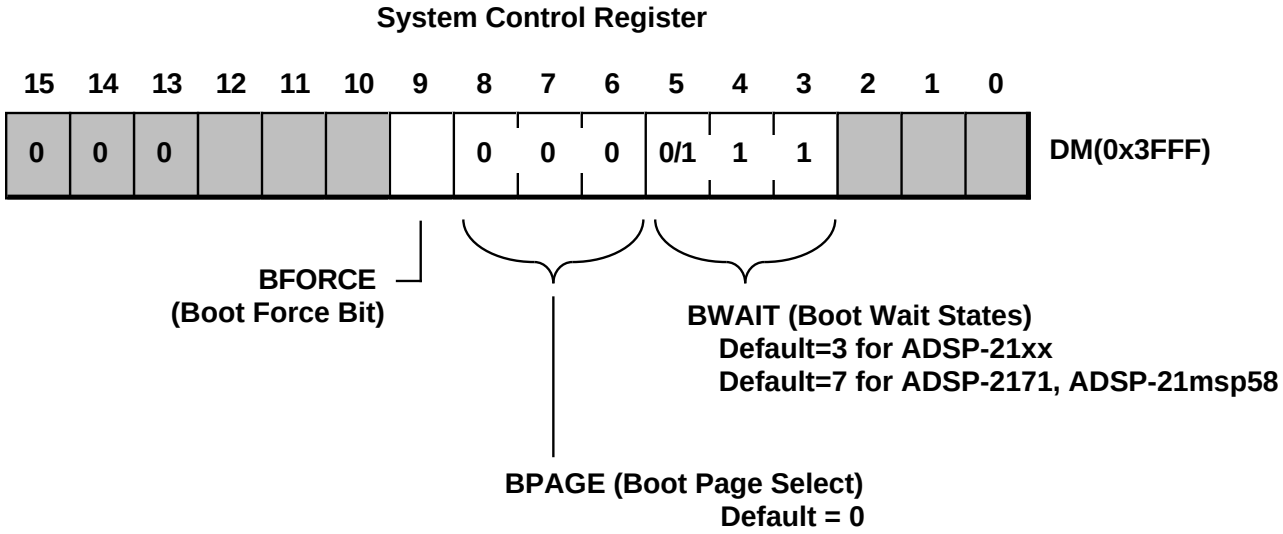


Figure 10.17 Boot Control Fields In System Control Register

10.4.3 Boot Memory Access

The processor can boot its internal memory from a single byte-wide CMOS EPROM, such as the 27C64 and 27C512. A low-cost, commodity-grade EPROM with an industry-standard access time can be used. The number of wait states for the boot memory access is selected in the BWAIT field of the System Control Register (see Figure 10.17). 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 3 wait states on the ADSP-2101, ADSP-2105, ADSP-2111, and ADSP-2115. BWAIT defaults to 7 wait states on the ADSP-2171 and ADSP-21msp58.

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 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 middle eight bits of the data bus.

10.4.4 Boot Loading Sequence

The order in which the processor 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, for instance when a latch is providing data rather than an EPROM.

To execute the boot operation, the boot address generator generates the appropriate byte addresses and loads internal program memory with the contents of the EPROM. The internal program memory is loaded beginning with the high addresses. For example, assume that eight 24-bit words are loaded into the processor during the booting process. The first word written into program memory is written to address 0x0007. The last word loaded is written to internal program memory address 0x0000.

The boot address is made up of several values, as shown in Figures 10.18 and 10.19: 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); a 3-bit word counter value; and a 2-bit code whose value determines which byte of the word is being addressed.

The last 24-bit word (instruction or data value) is loaded into the processor 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 (with eight 24-bit words corresponding to a page length of 0), the following addresses would be generated (see Figure 10.20):

1. The first address generated is 0x0003 which reads the page length.

2. The next address generated in this example is address 0x001C. This is the upper byte of the last word.

3. The byte code is then updated to specify the lower byte (the final two bits are 10) and the address generated is 0x001E.

4. The byte address changes again, this time to address the middle byte (the two bit code is 01) and the address generated is 0x001D.

5. Once all three bytes are loaded, the word counter is decremented. The three succeeding byte addresses generated are 0x0018, 0x001A, and 0x0019.

6. The word counter is decremented again and the next set of byte addresses generated is 0x0014, 0x0016, and 0x0015. This process continues until word 0 is loaded.

The contents of the EPROM, the byte addresses, and the order of addresses generated is shown in Figure 10.20.

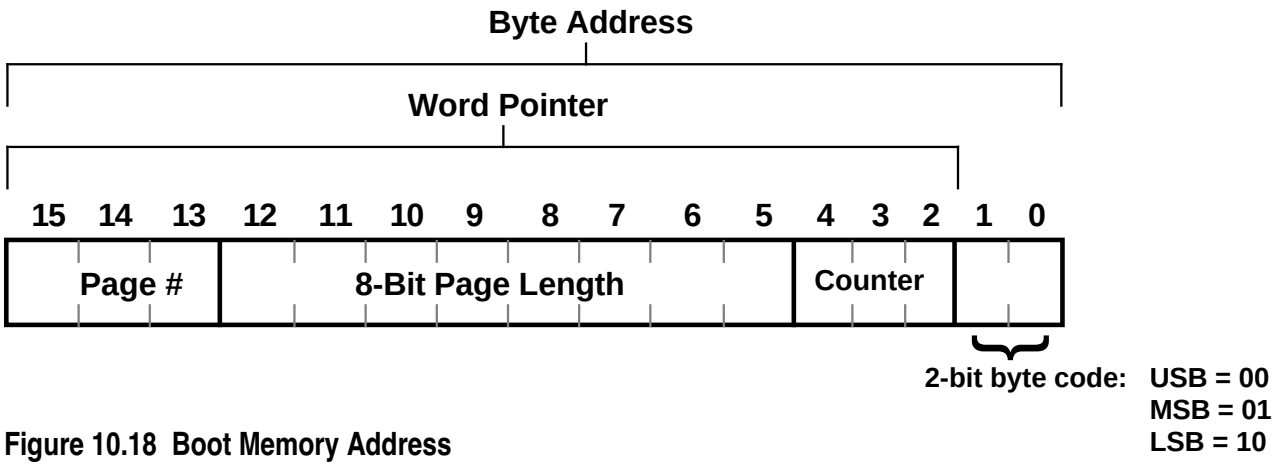


Figure 10.18 Boot Memory Address

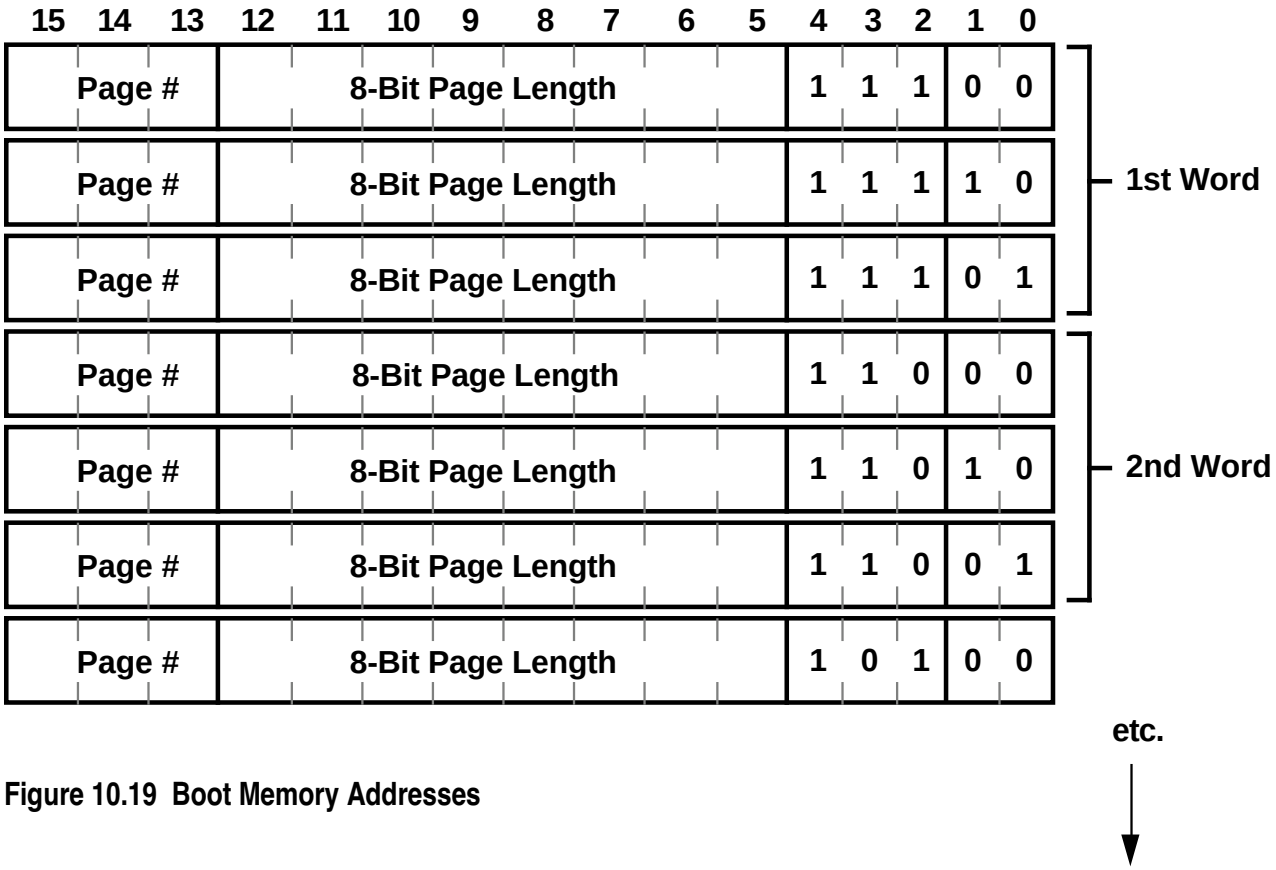


Figure 10.19 Boot Memory Addresses

10 Memory Interface

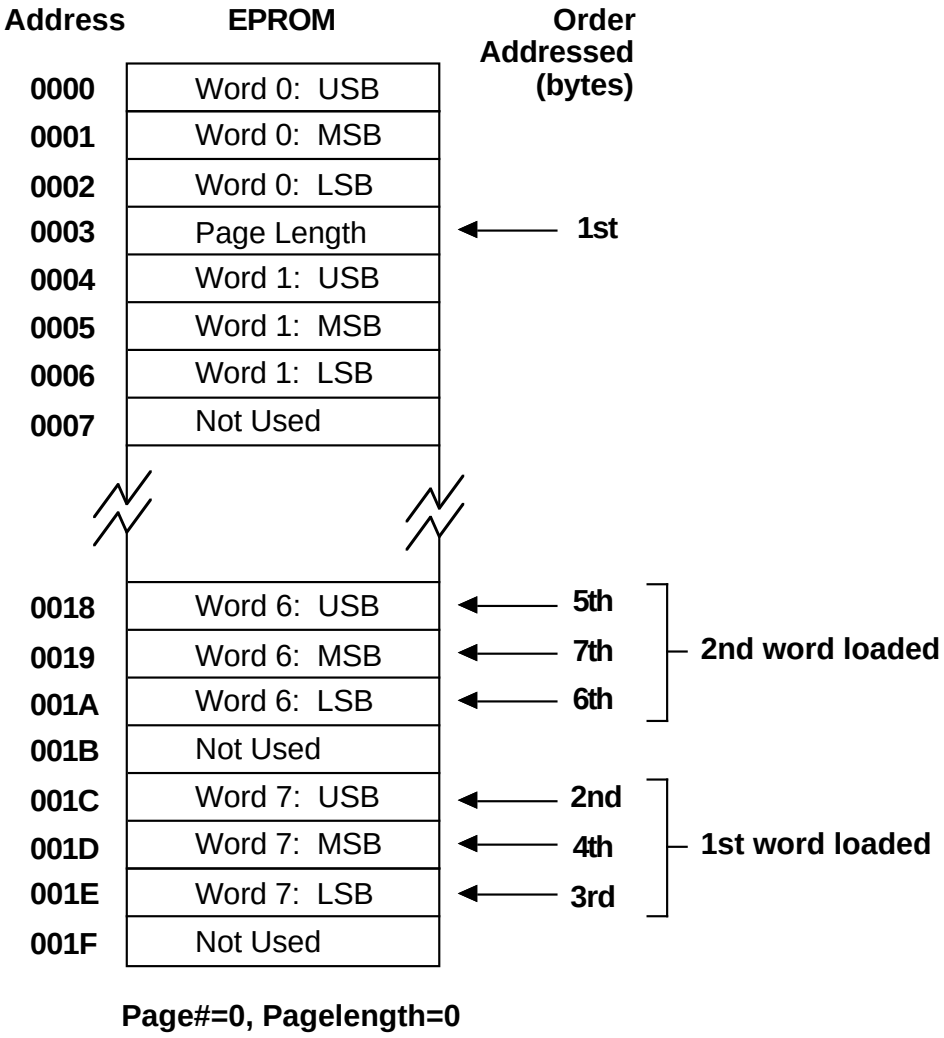


Figure 10.20 Example of Boot Loading Order (with Page#=0, Pagelength=0)

10.5 BUS REQUEST / GRANT

This section describes the bus request and grant feature of all ADSP-21xx processors, including the ADSP-2181.

The ADSP-21xx can relinquish control of its data and address buses to an external device. The external device requests the bus by asserting (low) the bus request signal, $\overline{BR}$. $\overline{BR}$ is an asynchronous input. If the ADSP-21xx is not performing an external access, it responds to the active $\overline{BR}$ input in the following processor cycle by:

- tristating the data and address buses and the $\overline{xMS}$, $\overline{RD}$, $\overline{WR}$ output drivers,
- asserting the bus grant ($\overline{BG}$) signal, and
- halting program execution (unless Go Mode is enabled).

If Go Mode is enabled, the ADSP-21xx continues to execute instructions from its internal memory. It will not halt program execution until it encounters an instruction that requires an external access. (An "external access" may be either a memory device access or, on the ADSP-2181, a memory overlay access, BDMA access, or I/O space access.)

If Go Mode is not enabled, the ADSP-21xx always halts before granting the bus. The processor's internal state is not affected by granting the bus, and the serial ports and host interface port (on the ADSP-2111, ADSP-2171, ADSP-21msp5x) remain active during a bus grant, whether or not the processor core halts.

If the ADSP-21xx is performing an external access when the $\overline{BR}$ signal is asserted, it will not grant the buses until the cycle after the access completes. The sequence of events is illustrated in Figure 10.21. The entire instruction does not need to be completed when the bus is granted. If a single instruction requires two external accesses, the bus will be granted between the two accesses. The second access is performed after $\overline{BR}$ is removed.

When the $\overline{BR}$ input is released, the ADSP-21xx releases the $\overline{BG}$ signal, reenables the output drivers and continues program execution from the point where it stopped. $\overline{BG}$ is always deasserted in the same cycle that the removal of $\overline{BR}$ is recognized. Refer to the data sheet for exact timing relationships.

The bus request feature operates at all times, including when the processor is booting and when $\overline{RESET}$ is active. During $\overline{RESET}$, $\overline{BG}$ is asserted in the same cycle that $\overline{BR}$ is recognized. During booting, the bus is granted after completion of loading of the current byte (including any wait states). Using bus request during booting is one way to bring the booting operation under control of a host computer.

10 Memory Interface

The ADSP-2171 and ADSP-2181 processors have an additional feature, the Bus Grant Hung ($\overline{BGH}$) output, which lets them operate in a multiprocessor system with a minimum number of wasted cycles. The $\overline{BGH}$ pin asserts when the ADSP-21xx is ready to execute an instruction but is stopped because the external bus is granted to another device. The other device can release the bus by deasserting bus request. Once the bus is released, the ADSP-21xx deasserts $\overline{BG}$ and $\overline{BGH}$ and executes the external access. Figure 10.22 shows timing for the $\overline{BGH}$ signal.

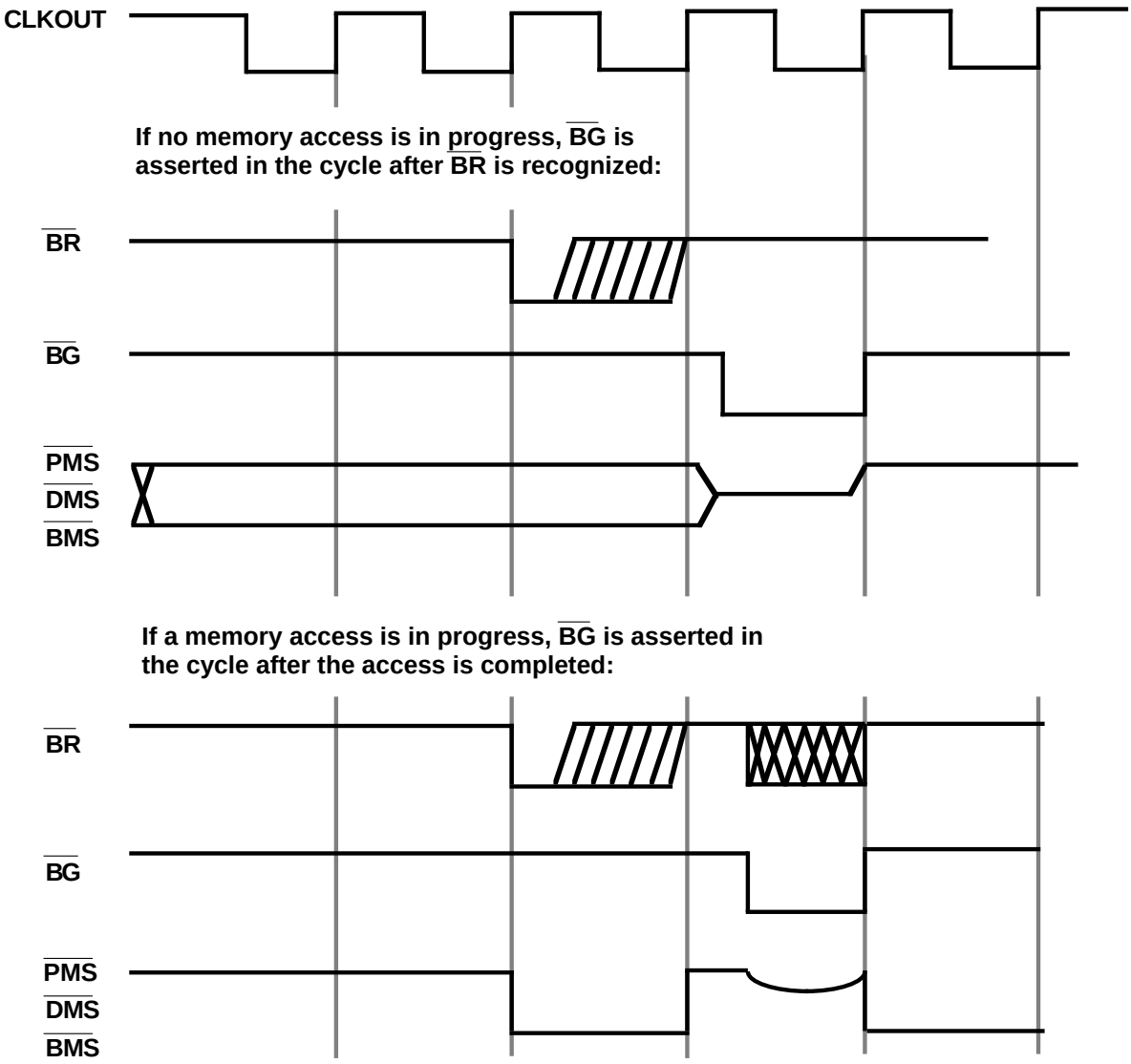


Figure 10.21 Bus Request (with and without external access)

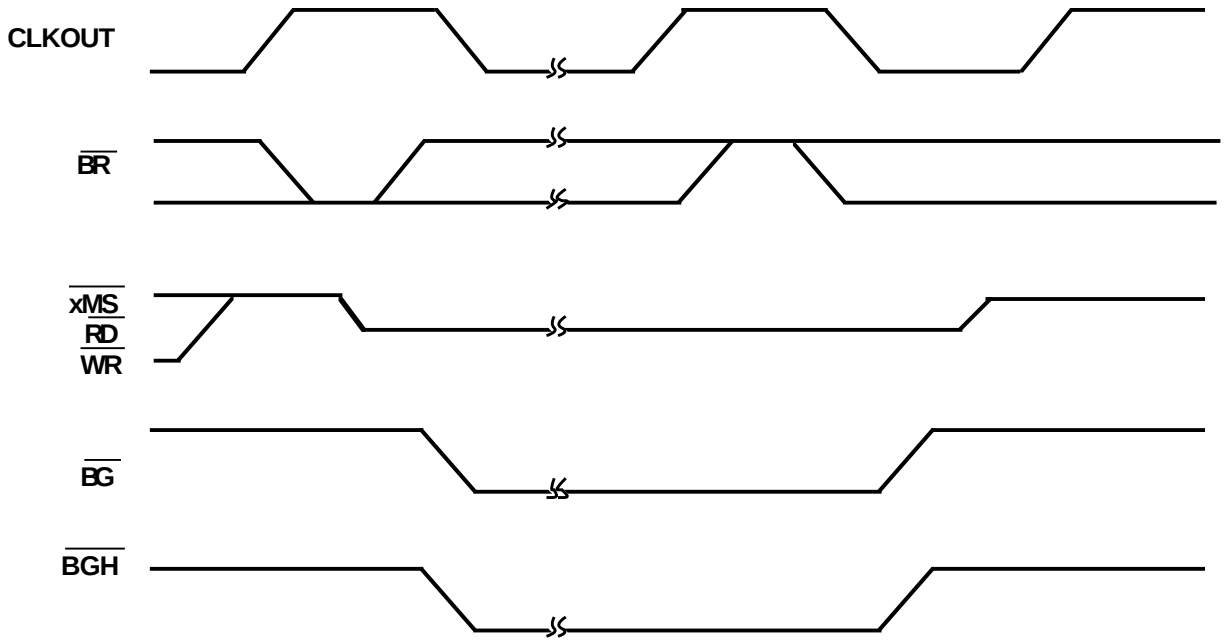


Figure 10.22 Bus Grant Hung ($\overline{BGH}$) Timing (ADSP-2171, ADSP-2181 only)

10.6 ADSP-2181 MEMORY INTERFACES

The ADSP-2181 has the same modified Harvard architecture for internal memory as the other processors of the ADSP-2100 family. In this architecture, Data Memory stores data values and Program Memory stores both instructions and data. The ADSP-2181 has as its full base memory on-chip: 16K x 24-bit words of internal program memory RAM and 16K x 16-bit words of internal data memory RAM.

There are four separate memory spaces: data memory, program memory, byte memory, and I/O memory. To provide external access to these memory spaces, the ADSP-2181 extends the internal address and data buses off-chip and provides the $\overline{PMS}$, $\overline{DMS}$, $\overline{BMS}$, and $\overline{IOMS}$ select lines. The $\overline{PMS}$, $\overline{DMS}$, $\overline{BMS}$, and $\overline{IOMS}$ signals indicate which memory space is being accessed.

The composite memory space (and its $\overline{CMS}$ select line) lets a single off-chip memory be accessed as multiple memory spaces. The Composite Memory Select register lets you define which memory spaces are selected by the $\overline{CMS}$ signal.

10 Memory Interface

Figure 10.23 shows the external memory buses and control signals in an ADSP-2181 system. Two control lines determine the direction of external memory transfers: $\overline{RD}$ is active low signaling a read and $\overline{WR}$ is active low for a write operation. Typically, you would connect $\overline{RD}$ to $\overline{OE}$ (Output Enable) and $\overline{WR}$ to $\overline{WE}$ (Write Enable) of your memory.

Internal memory accesses do not drive any external signals: $\overline{PMS}$, $\overline{DMS}$, $\overline{BMS}$, $\overline{IOMS}$, $\overline{RD}$, and $\overline{WR}$ remain high (deasserted), and the address and data buses are tristated.

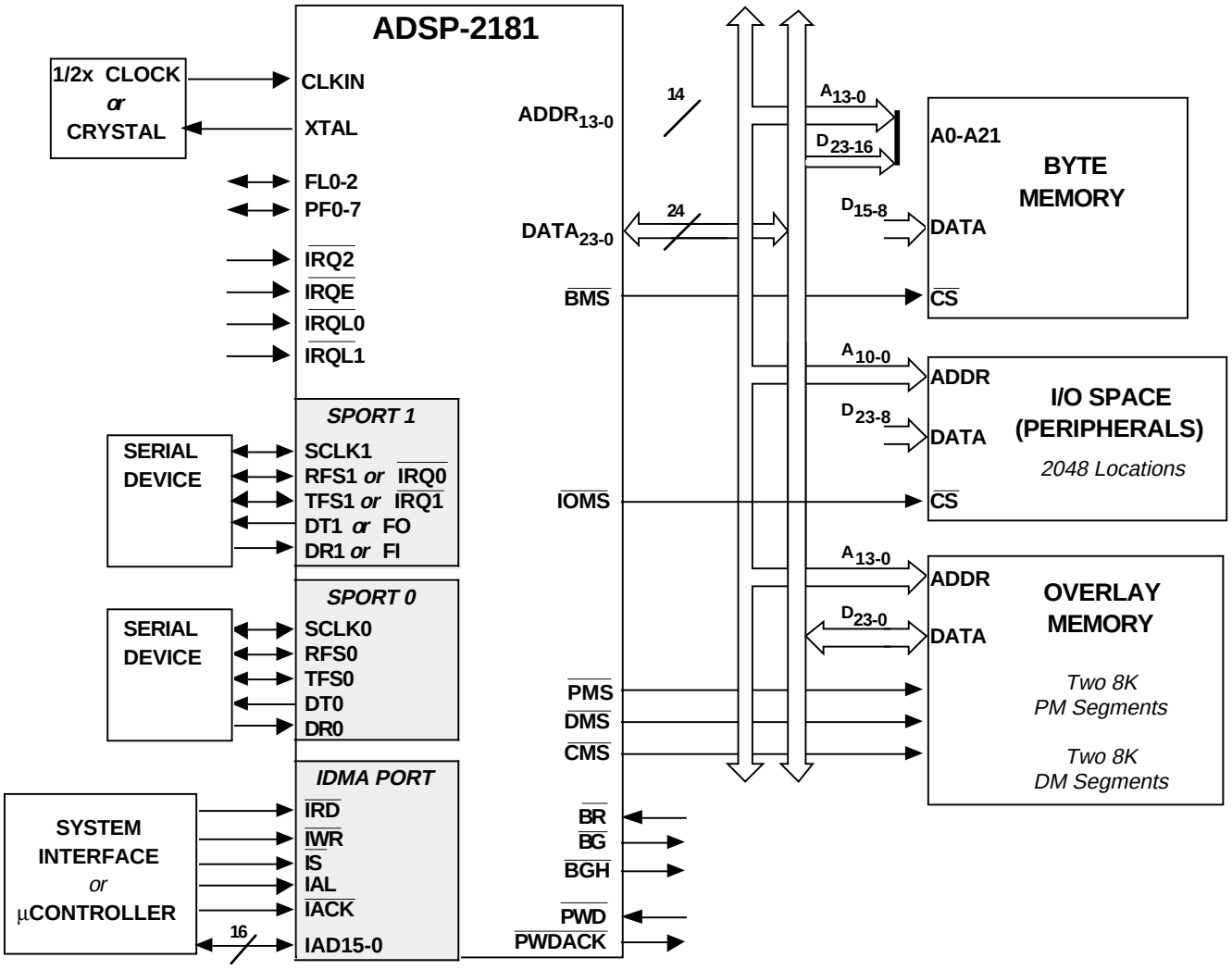


Figure 10.23 ADSP-2181 System Diagram

Unlike other processors of the ADSP-2100 family, the ADSP-2181 supports several additional memory interfacing features. These features include:

- **External Overlay Memory** in 8K segments: these segments can be swapped for the upper 8K of internal program memory or lower 8K of data memory.
- **I/O Memory** space: this memory space is for peripheral I/O, has 2K (16-bit wide) locations, and has four user-assignable waitstate ranges.
- **Byte Memory & Byte Memory DMA (BDMA)**: this memory space can address up to 4M bytes. The byte memory interface supports booting from and runtime access to inexpensive 8-bit memories. The DMA feature lets you define the number of memory locations the DSP will transfer to/from internal memory in the background while continuing foreground processing.
- **Internal Direct Memory Access (IDMA) Port**: this port supports booting from and runtime access to host systems (for example, PC Bus Interface ASICs). The DMA feature of this port lets you define the number of memory locations the DSP will transfer to/from internal memory in the background while continuing foreground processing.

For complete information on the BDMA port, including booting, and IDMA port, refer to the *DMA Ports* chapter of this manual.

The ADSP-2181 uses a half-instruction-rate clock input from which it generates a full-instruction-rate internal clock. For example, from a 16.67 MHz clock input (CLKIN) the ADSP-2181 generates a 33.33 MHz instruction rate clock. All timing diagrams for the processor use the full-instruction-rate output clock (CLKOUT) as a reference.

All external memories may have automatic wait state generation associated with them. The number of wait states—each equal to one instruction cycle—is programmable.

10.6.1 ADSP-2181 Program Memory Interface

The ADSP-2181 processor addresses its 16K of internal program memory as well as two 8K external program memory overlays. All program memory is 24 bits wide. Up to two accesses to internal program memory can be completed per instruction cycle; this lets the DSP complete all operations in a single cycle. The PWAIT field of the System Control Register (shown in Figure 10.24) sets the number of waitstates for each access to program memory overlays. PWAIT defaults (after reset) to seven.

10 Memory Interface

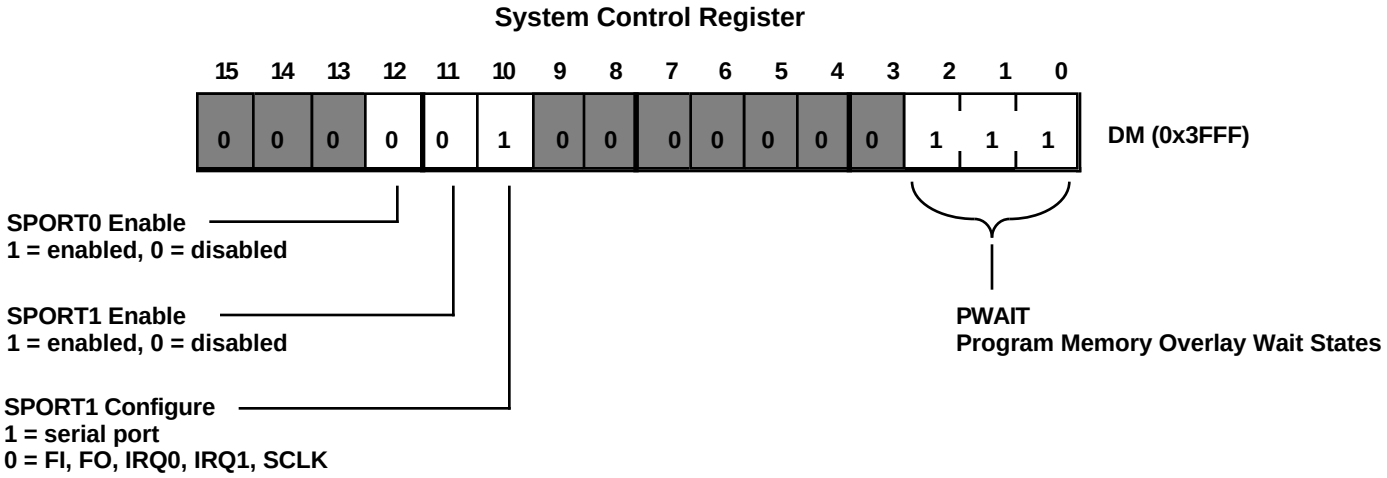


Figure 10.24 PWAIT Field in System Control Register

The on-chip program memory and overlays can hold instructions and data intermixed in any combination. The ADSP-21xx 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. The restart vector is at program memory address 0x0000.

The ADSP-2181's MMAP pin lets you select from two program memory configurations. The MMAP pin also controls whether the ADSP-2181 boots after $\overline{\text{RESET}}$ is released. Figure 10.25 shows the MMAP options and the resulting memory maps for program memory.

The program memory overlay select register (PMOVLAY) lets you choose a memory overlay to map from address PM(0x2000) to address PM(0x3FFF). The memory mapped to this space and corresponding PMOVLAY register values are shown in Figure 10.25. Table 10.3 shows how PMOVLAY relates to the addressing of memory locations (with address line A13).

PMOVLAY	*Memory*	*A13*	*A12:0*
0	Internal	—	—
1	External overlay 1	0	13 LSBs of address between 0x2000 and 0x3FFF
2	External overlay 2	1	13 LSBs of address between 0x2000 and 0x3FFF

Table 10.3 PMOVLAY and Program Memory Overlay Addressing

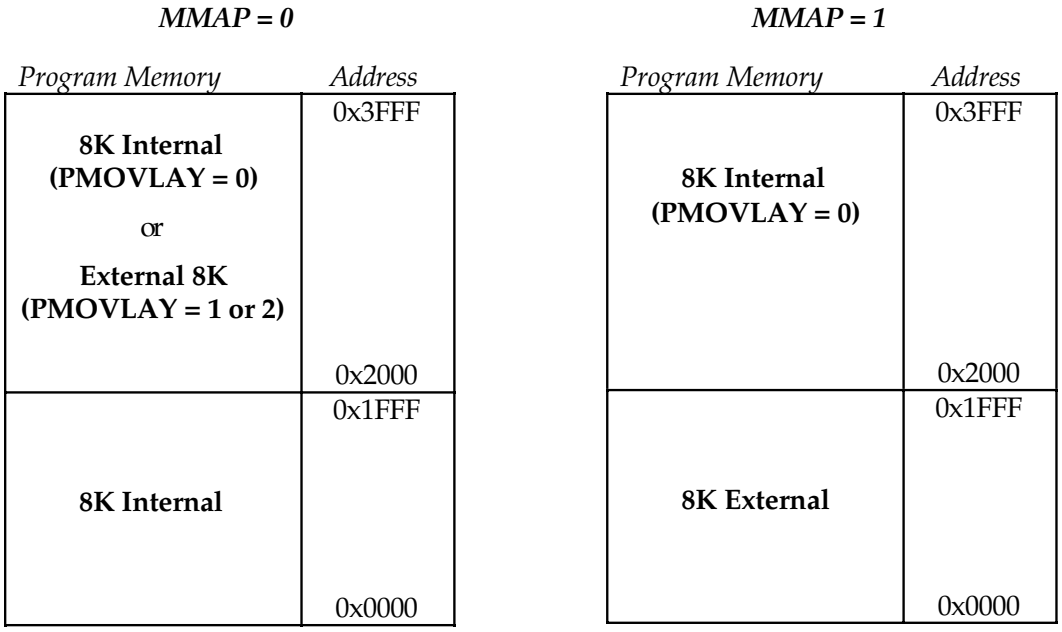


Figure 10.25 ADSP-2181 Program Memory Map

The following example instructions demonstrate how to use the PMOVLAY register.

```
PMOVLAY=DM(0x1234);  {type 3 instruction, PMOVLAY is loaded
}
                     { with the contents of address
DM(0x1234)}

PMOVLAY=2;           {type 7 instruction, PMOVLAY is loaded }
                     { with the value 2. }

PMOVLAY=AX0;         {PMOVLAY is loaded from AX0 register.}

AX0=PMOVLAY;         {AX0 is loaded from PMOVLAY register.}
```

If you are using a system design that sets MMAP=1, note that the first 8K is used to support a single segment of external memory. This allows an external ROM-based system to operate properly. In this mode, the external program memory address always has A13 set to 0 and 8K of internal PM is available. Set PMOVLAY=0 and MMAP=1. This mode is available on other

ADSP-2100 family processors.

Figure 10.26 shows a memory design that provides full external program and data memory overlays for an ADSP-2181 processor, assuming that MMAP=0. The important points to note about this design are:

- Three 32K x 8-bit SRAMs are required for full external program and data memory overlays; glue logic is *not* required.
- Four control lines are required for read ($\overline{RD}$), write ($\overline{WR}$), chip select ($\overline{CMS}$), and data/program memory select ($\overline{PMS}$ or $\overline{DMS}$).
- Composite Memory Select (CMSSEL) is configured to assert the $\overline{CMS}$ control line when Program Memory Select ($\overline{PMS}$) or Data Memory Select ($\overline{DMS}$) are asserted.
- The order of overlays stored in this design (from lowest address to highest) is PM Overlay 1, PM Overlay 2, DM Overlay 1, and DM Overlay 2. Address line 13 (A13) of the ADSP-2181 selects between overlay 1 or 2. Figure 10.27 shows a memory map of this design.

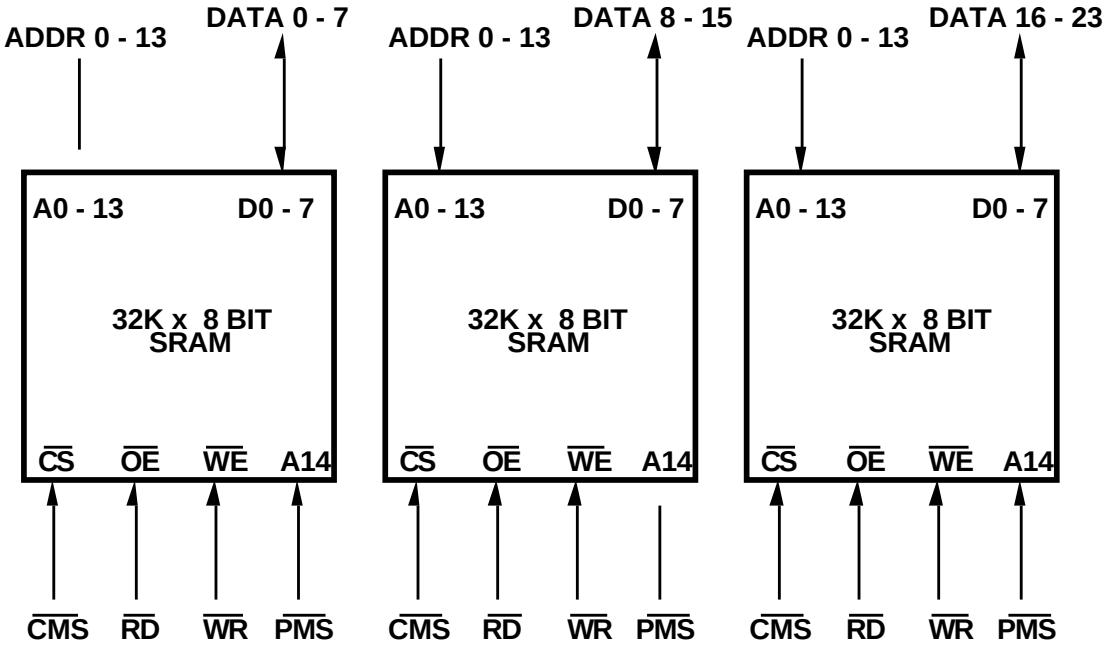


Figure 10.26 Example Program and Data Memory Overlay Design

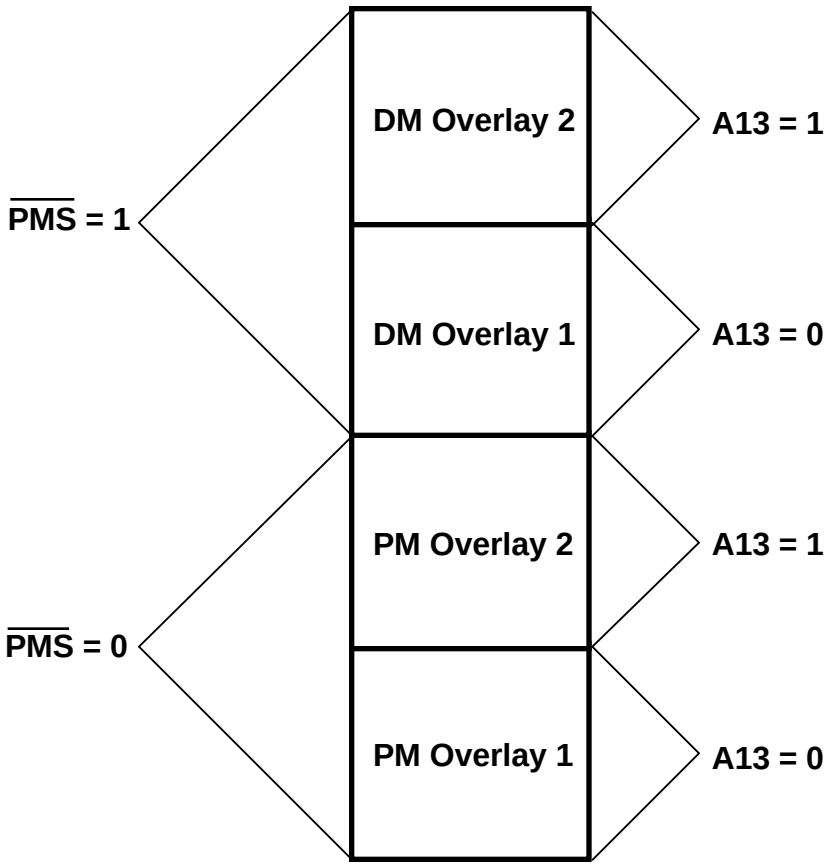


Figure 10.27 Memory Overlay Addressing For Example Design

There are some restrictions on using program memory overlays:

- The ADSP-2181's program sequencer does *not* consider the value in the PMOVLAY register. Switching pages during operations that are sensitive to the current PMOVLAY register value can result in program execution errors. For example, if your program is performing a loop operation on one of the external overlays and the program changes to another external or internal overlay, an incorrect loop operation could occur.

- The contents of the PMOVLAY register are *not* automatically saved and restored on the processors status stack when the processor responds to an interrupt. If your program uses overlays, you must save and restore the

contents of PMOVLAY as part of your interrupt service routine.

10.6.2 ADSP-2181 Data Memory Interface

The ADSP-2181 addresses 16K x 16-bit wide internal data memory and two 8K x 16-bit wide external data memory overlays. All accesses to internal data memory are completed in a single processor instruction cycle. The DWAIT field of the Waitstate Control Register (shown in Figure 10.28) sets the number of waitstates for each access to data memory overlays. Figure 10.29 shows the data memory map of the ADSP-2181.

The processor's memory-mapped control/status registers are mapped into the top locations of internal data memory, addresses 0x3FE0-0x3FFF. Most of the ADSP-2181's control registers correspond to those found on other ADSP-21xx processors. Note that the ADSP-2181's System Control Register does not have the boot memory control fields found on other ADSP-21xx processors. Also note that the Waitstate Control Register

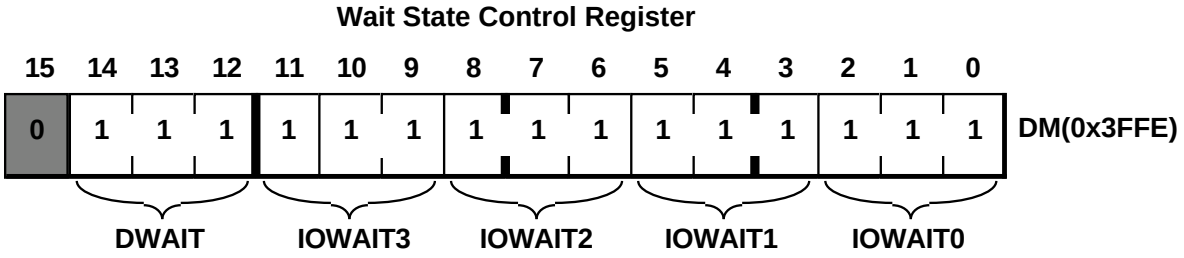


Figure 10.28 ADSP-2181 Wait State Control Register

Data Memory	*Address*
32 Memory-Mapped Control Registers	0x3FFF 0x3FE0
Internal 8160 words	0x3FDF 0x2000
8K Internal (DMOVLAY=0) or External 8K (DMOVLAY=1,2)	0x1FFF 0x0000

Figure 10.29 ADSP-2181 Data Memory Map

includes four fields for the ADSP-2181's I/O memory space.

The Data Memory overlay select (DMOVLAY) register lets you choose a memory overlay to map from address DM(0x0000) to address DM(0x1FFF). The DMOVLAY register is unique to the ADSP-2181. The memory mapped to this space and corresponding DMOVLAY contents are shown in Figure 10.29. Table 10.4 shows how DMOVLAY relates to memory addressing (address line A13).

DMOVLAY	*Memory*	*A13*	*A12:0*
0	Internal	—	—
1	External overlay 1	0	13 LSBs of address between 0x0000 and 0x1FFF
2	External overlay 2	1	13 LSBs of address between 0x0000 and 0x1FFF

Table 10.4 DMOVLAY and Data Memory Overlay Addressing

The following example instructions demonstrate how to use the DMOVLAY register:

```
DMOVLAY=DM(0x1234);  {type 3 instruction, DMOVLAY is loaded
}
                     { with the contents of address
DM(0x1234)}

DMOVLAY=2;           {type 7 instruction, DMOVLAY is loaded }
                     { with the value 2. }

DMOVLAY=AX0;         {DMOVLAY is loaded from AX0 register.}

AX0=DMOVLAY;         {AX0 is loaded from DMOVLAY register.}
```

For an example memory design that provides full external program and data memory overlays for an ADSP-2181 processor, see the previous section "Program Memory Interface."

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

$\overline{RD}$ to $\overline{OE}$ (Output Enable) and $\overline{WR}$ to $\overline{WE}$ (Write Enable) of your memory.

10.6.3 ADSP-2181 Byte Memory Interface

The ADSP-2181's byte memory space is 8 bits wide and can address up to 4M bytes of program code or data. This memory space takes the place of the boot memory space found on other ADSP-2100 family processors. Unlike boot memory space, byte memory has read/write access through the ADSP-2181's BDMA port.

Byte memory space consists of 256 pages, each containing 16K x 8-bit wide locations. This memory can be written and read in four different formats: 24-bit, 16-bit, 8-bit MSB alignment, and 8-bit LSB alignment.

Each read/write to byte memory consists of data (on data bus lines 15:8) and address (on address bus lines 13:0 plus data lines 23:16). The 22-bit byte memory address lets you access up to 4M bytes of ROM or RAM.

For complete information on the ADSP-2181's byte memory and BDMA port, refer to the *DMA Ports* chapter of this manual.

10.6.4 ADSP-2181 I/O Memory Space

The ADSP-2181 has a dedicated I/O Memory Space instead of the memory-mapped I/O used on other ADSP-21xx processors. The I/O memory space consists of 2048 locations with four associated programmable waitstate regions. Figure 10.30 shows the Wait State

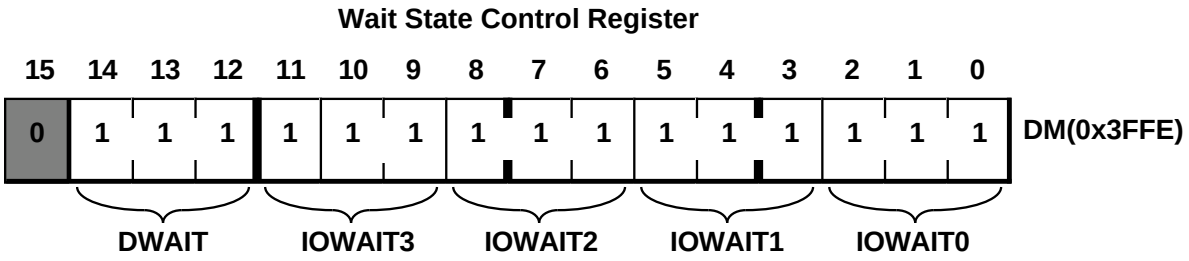


Figure 10.30 ADSP-2181 Waitstate Control Register

Control Register and the IOWAIT0-3 bit fields that control I/O memory waitstate regions.

The Wait State Control Register is divided into the following fields:

- **IOWAIT0**. This 3-bit field sets the number of waitstates (0-7) for accesses to I/O memory addresses 0x000–0x1FF.
- **IOWAIT1**. This 3-bit field sets the number of waitstates (0-7) for accesses to I/O memory addresses 0x200–0x3FF.
- **IOWAIT2**. This 3-bit field sets the number of waitstates (0-7) for accesses to I/O memory addresses 0x400–0x5FF.
- **IOWAIT3**. This 3-bit field sets the number of waitstates (0-7) for accesses to I/O memory addresses 0x600–0x7FF.
- **DWAIT**. This 3-bit field sets the number of waitstates (0-7) for accesses to external program and data memory overlays.

Note: The PWAIT field of the System Control Register sets the number of waitstates for access to external program memory overlays.

When you connect a parallel I/O device to the ADSP-2181 as shown in

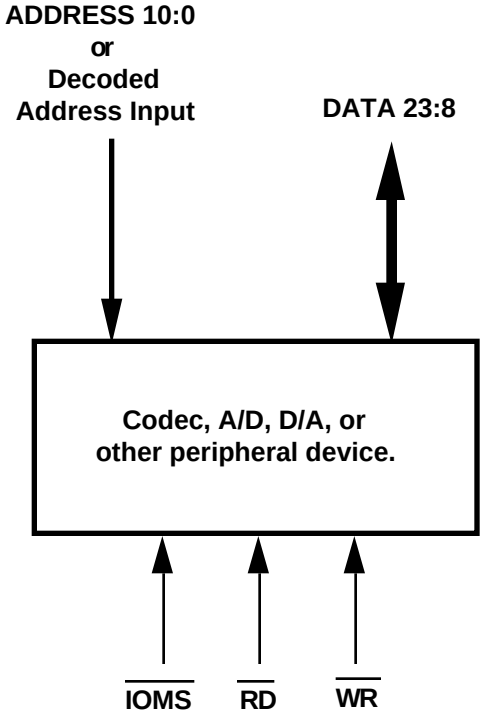


Figure 10.31 I/O Memory Space Peripheral Connection Example

10 Memory Interface

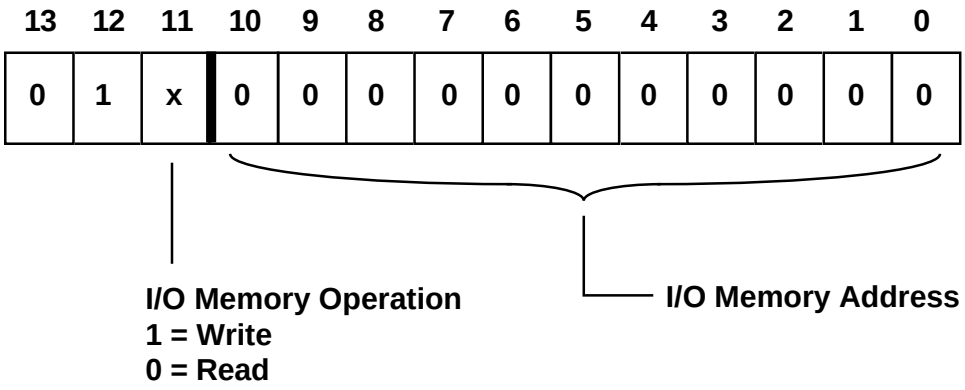


Figure 10.32 I/O Memory Address Word

Figure 10.31, the address sent to the device appears on the external address bus as shown in Figure 10.32.

Host interfaces can use the additional communications channel provided by the ADSP-2181's I/O memory space. If your system bus interface ASIC uses a set of data registers for passing control information from the system bus and must also pass large amounts of sample data, map the control registers as I/O memory peripherals and transfer the sample data using IDMA. This combination of the I/O memory and IDMA channels reduces system bus transfer rate limitations.

Note: As with other ADSP-2100 Family processors, on the ADSP-2181 you can define memory-mapped I/O ports with the assembler's .PORT directive. On the ADSP-2181, this directive defines memory-mapped I/O ports in external program memory overlays or data memory overlays. If you want to use this feature, you must make sure at runtime that you are on the correct program memory overlay or data memory overlay when accessing the port; the assembler and linker will not flag errors in .PORT accesses related to overlays because the issue is resolved at runtime. The "IO" keyword does not work with the .PORT directive; to assign symbolic

labels to I/O memory addresses, use a `#define` macro. The best use of the .PORT directive is in porting non-ADSP-2181 applications to the ADSP-2181; otherwise, use I/O memory space for memory-mapped I/O.

10.6.5 ADSP-2181 Composite Memory Select

The ADSP-2181 has a programmable memory select signal, Composite Memory Select ($\overline{CMS}$). This signal lets you generate a memory select for devices mapped to more than one memory space, with the same timing as other individual memory select signals ($\overline{PMS}$, $\overline{DMS}$, $\overline{BMS}$, and $\overline{IOMS}$).

Based on the value of CMSSEL in the Programmable Flag & Composite Select Control register (see Figure 10.33), the ADSP-2181 asserts $\overline{CMS}$

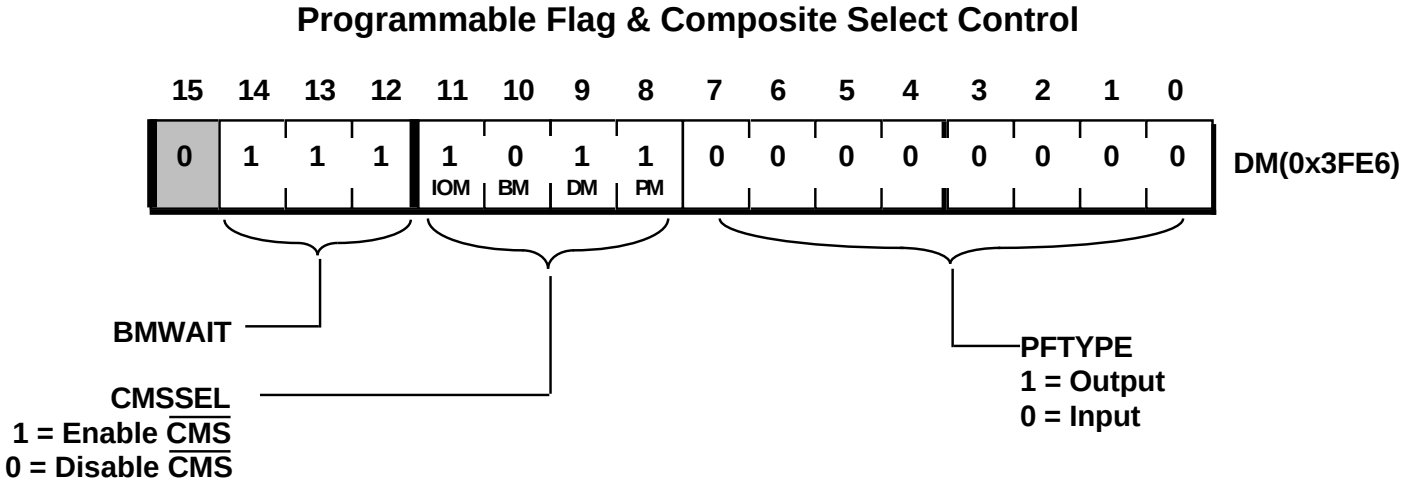


Figure 10.33 CMSSEL Selection for $\overline{CMS}$ Signal

when the corresponding memory select signal (or signals) are asserted. Each $\overline{xMS}$ signal can be individually enabled. After reset, CMSSEL is initialized to enable $\overline{PMS}$, $\overline{DMS}$, and $\overline{IOMS}$ (with $\overline{BMS}$ disabled).

Figure 10.26 (earlier in this chapter) shows an example of how to use the

$\overline{CMS}$ signal. In this system the $\overline{CMS}$ line drives the chip select for all three SRAMs. This lets you use three 32K x 8-bit SRAMs, with no glue logic, for complete program and data memory overlays.

10.6.6 External Memory Read – Overlays & I/O Memory

External memory reads may access either PM overlays, DM overlays, or I/O memory space. These read operations occur in the following sequence (see Figure 10.34):

1) The ADSP-2181 executes a read from an external memory address; the address is driven on the address bus and $\overline{PMS}$, $\overline{DMS}$, $\overline{BMS}$, or $\overline{IOMS}$, and $\overline{RD}$ is asserted. ($\overline{CMS}$ may also be asserted, depending how it is configured.)

2) The external peripheral drives the data onto the data bus.

3) The ADSP-2181 reads the data and deasserts $\overline{RD}$.

$\overline{WR}$ remains high (deasserted) throughout the external memory read operation.

Note that ADSP-2181 internal memory accesses do not drive any

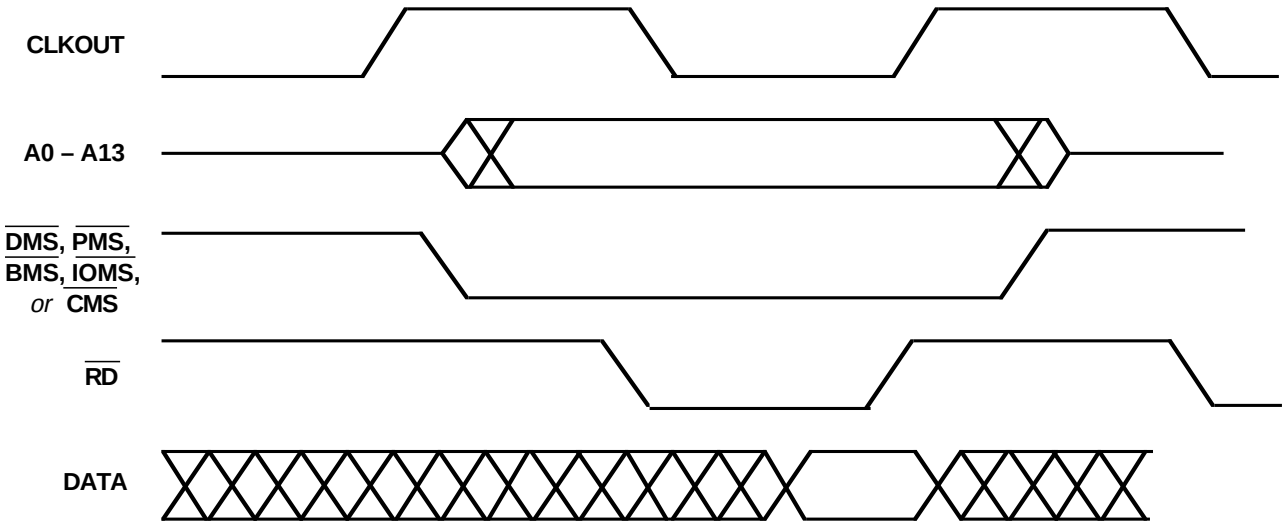


Figure 10.34 External Memory Read Timing

external signals: $\overline{PMS}$, $\overline{DMS}$, $\overline{IOMS}$, $\overline{BMS}$, $\overline{CMS}$, $\overline{RD}$, and $\overline{WR}$ remain high (deasserted), and the address and data buses are tristated.

10.6.7 External Memory Write – Overlays & I/O Memory

External memory writes may access either PM overlays, DM overlays, or I/O memory space. These read operations occur in the following sequence (see Figure 10.35):

1) The ADSP-2181 executes a write to an external memory address; the address is driven on the address bus, data is driven on the data bus, and $\overline{PMS}$, $\overline{DMS}$, $\overline{BMS}$, or $\overline{IOMS}$, and $\overline{WR}$ is asserted. ($\overline{CMS}$ may also be asserted, depending how it is configured.)

2) The external peripheral stores the data.

3) The ADSP-2181 stops driving the address and data buses and deasserts $\overline{WR}$.

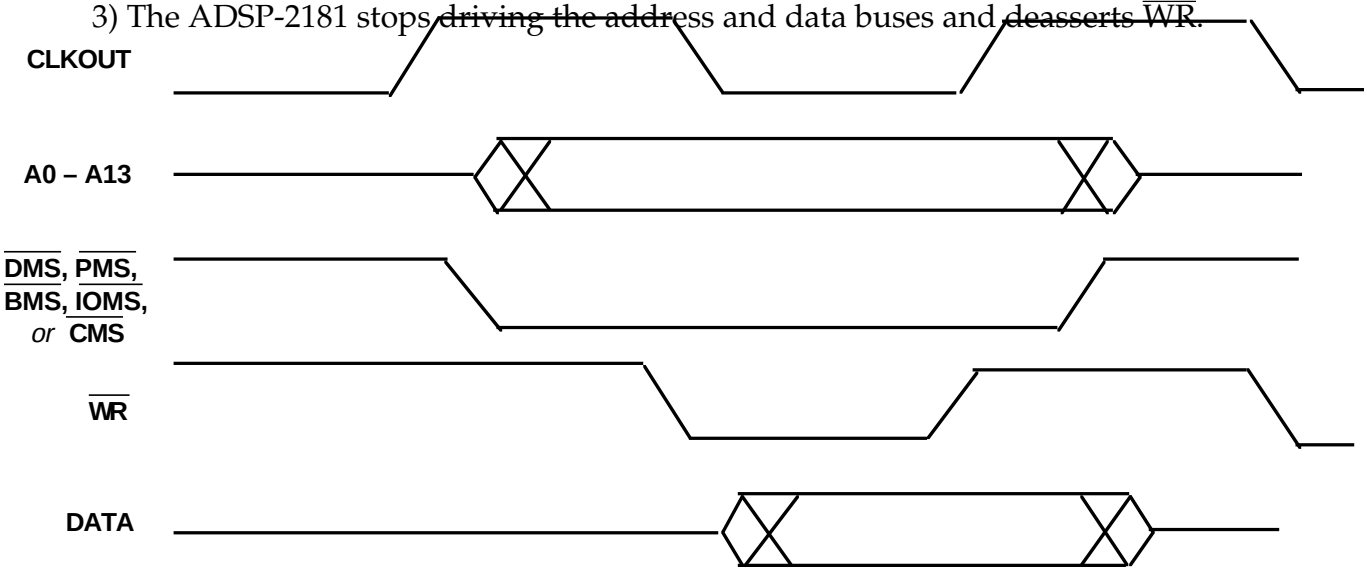


Figure 10.35 External Memory Write Timing

$\overline{RD}$ remains high (deasserted) throughout the external memory write operation.

10.7 MEMORY INTERFACE SUMMARY (ALL PROCESSORS)

Table 10.5 summarizes the states of the memory interface pins for various combinations of program memory and data memory accesses. Table 10.6 summarizes the states of the memory interface and control pins during reset, booting (ADSP-21xx boot memory booting, not ADSP-2181 byte memory booting), and bus grant.

10 Memory Interface

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
External boot memory	high	high	low	low (for read)	high	Boot address	Boot data, Boot page address

Table 10.5 Pin States During Memory Accesses

Operation	*Address*	*Data*	$\overline{PMS}$ $\overline{DMS}$ $\overline{BMS}$	$\overline{RD}$ $\overline{WR}$	*CLKOUT*	*SPORTs*	$\overline{BG}$
Reset	tristated	tristated	high	high	active	tristated	high
Booting* after Reset	active	active	$\overline{BMS}$ active $\overline{PMS}$, $\overline{DMS}$ high	$\overline{RD}$ active $\overline{WR}$ high	active	tristated	high
Reboot*	active	active	$\overline{BMS}$ active $\overline{PMS}$, $\overline{DMS}$ high	$\overline{RD}$ active $\overline{WR}$ high	active	active	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 10.6 Pin States During Reset, Booting*, and Bus Grant

* ADSP-21xx boot memory booting, not ADSP-2181 byte memory booting.

DMA Ports 11

11.1 OVERVIEW

The ADSP-2181 supports several DMA interfacing features:

- **Byte Memory & Byte Memory DMA (BDMA):** this memory space can address up to 4M bytes. The byte memory interface supports booting from and runtime access to inexpensive 8-bit memories. The BDMA feature lets you define the number of memory locations the ADSP-2181 will transfer to/from internal memory in the background while continuing foreground processing.

- **Internal Direct Memory Access (IDMA) Port:** this parallel port supports booting from and runtime access to host systems (for example, PC Bus Interface ASICs). The DMA feature of this port lets you transfer data to/from internal memory in the background while continuing foreground processing.

These DMA transfers are accomplished internally by "cycle stealing," in the same way as serial port autobuffering. This means that the ADSP-2181 uses internal bus cycles to transfer the data to and from memory. The stolen cycles will only occur at instruction cycle boundaries, i.e. not between cycles of a multiple-cycle instruction. See "$\overline{\text{IACK}}$ Acknowledge & DMA Cycle Stealing" at the end of this chapter for additional details.

The ADSP-2181 uses a half-instruction-rate clock input from which it generates a full-instruction-rate internal clock. For example, from a 16.67 MHz clock input (CLKIN) the ADSP-2181 generates a 33.33 MHz instruction rate clock. All timing diagrams for the processor use the full-instruction-rate output clock (CLKOUT) as a reference.

Figure 11.1 shows an ADSP-2181 system and the interfaces to byte memory space and the IDMA port.

11 DMA Ports

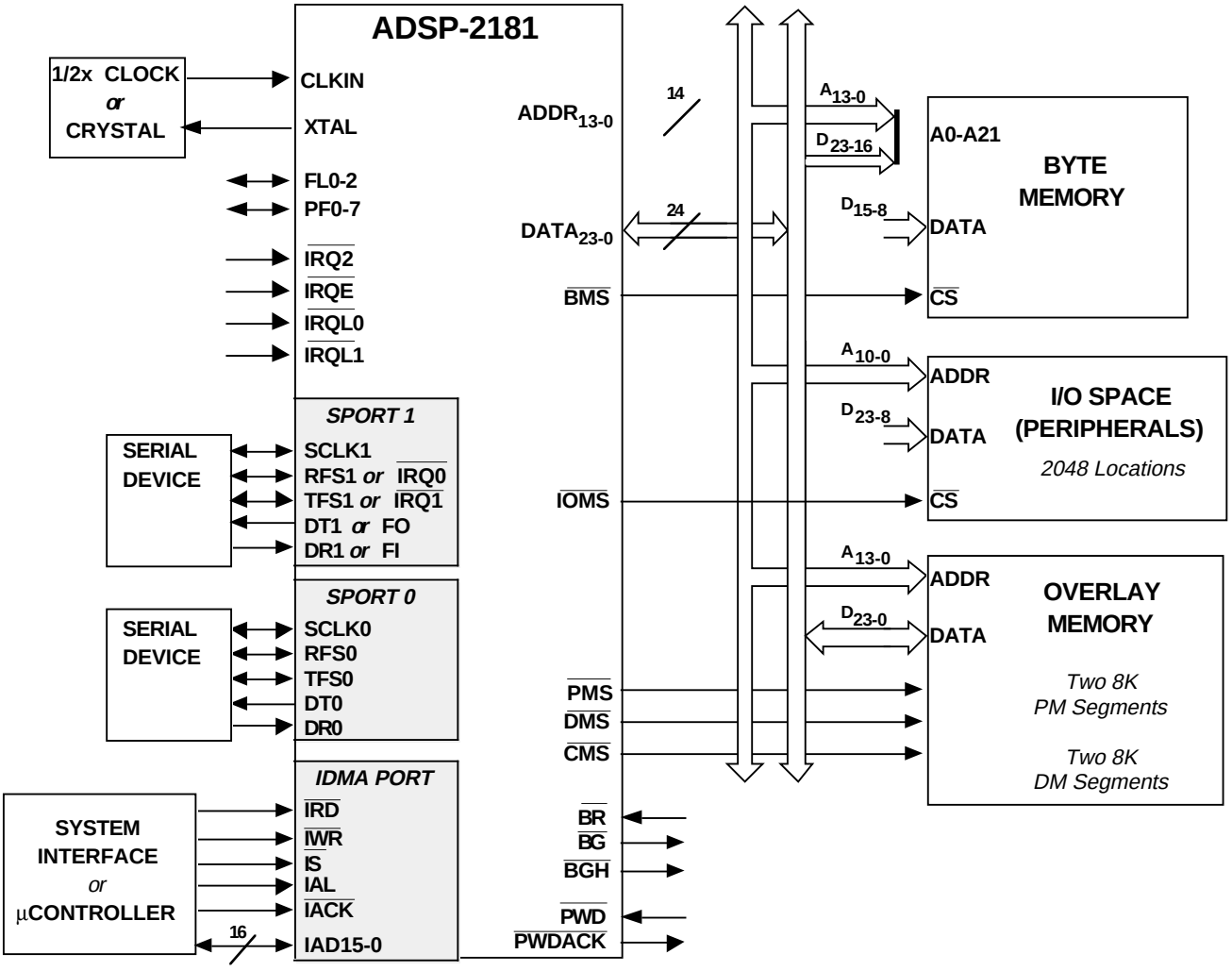


Figure 11.1 ADSP-2181 System

11.2 BDMA PORT

The ADSP-2181's byte memory space is 8 bits wide and can address up to 4M bytes of program code or data. This memory space takes the place of the boot memory space found on other ADSP-2100 family processors. Unlike boot memory space, byte memory has read/write access through the ADSP-2181's BDMA port.

Each read/write to byte memory consists of data (on data bus lines 15:8) and address (on address bus lines 13:0 plus data lines 23:16). The 22-bit byte memory address lets you access up to 4M bytes of ROM or RAM.

Byte memory space consists of 256 pages, each containing 16K x 8-bit wide locations. This memory can be written and read in four different formats: 24-bit, 16-bit, 8-bit MSB alignment, and 8-bit LSB alignment.

To use byte memory for purposes other that boot loading, for example runtime access to bulk data storage, you must know the page (BMPAGE) that the code/data is stored on, the number of words (BWCOUNT) to read from that page, and the word format (BTYPE) of the data. Use the following procedure to prepare a runtime-accessible byte memory EPROM:

- Develop the data/code to be accessed at runtime
- Use the ADSP-2100 Family PROM Splitter utility to split the file into single page (or smaller) 16K x 8-bit-wide segments
- Program these pages into your EPROM, noting the offset (page number) of each
- Use these page numbers when doing BDMA accesses

Note: For more information on the ADSP-2100 Family Development Software Tools, see the *ADSP-2100 Family Assembler Tools & Simulator Manual* and current software release note.

When using BDMA for non-boot-loading transfers, a BDMA transfer begins when data is written to the BWCOUNT register and a BDMA interrupt is issued when the transfer is complete.

The following restrictions apply to BDMA transfers:

- The source or target of BDMA transfer is always internal program or data memory. The contents of the PMOVLAY and DMOVLAY registers do not influence BDMA source (or target selection).
- Do not access the BEAD or BIAD registers during BDMA transfers.
- Other external memory accesses (PM overlay, DM overlay, or I/O space) take precedence over BDMA port accesses. These accesses cannot occur at the same time because they also use the processor's external bus.
- Do not enter powerdown mode with the BDMA port active. For information on powerdown restrictions on BDMA port access, see the System Interface chapter of this manual.

11 DMA Ports

11.2.1 BDMA Port Functional Description

The BDMA Port lets you load (and store) program instructions and data from (and to) byte memory with very low processor overhead. While the ADSP-2181 is executing program instructions, the BDMA port reads (or writes) code or data from (or to) byte memory—stealing one ADSP-2181 cycle per word when it needs to write to (or read from) internal memory. You can calculate BDMA transfer time from the formula:

$$\begin{pmatrix}\text{Number}\\ \text{of PM}\\ \text{or DM}\\ \text{Words}\end{pmatrix}\left[\begin{pmatrix}\text{Number}\\ \text{of Bytes}\\ \text{per Word}\end{pmatrix}\begin{pmatrix}\text{Number}\\ \text{of Added}\\ \text{Waitstates}\\ \text{per Byte}\end{pmatrix}+\begin{pmatrix}1\\ \text{Cycle}\\ \text{for}\\ \text{Transfer}\end{pmatrix}+\begin{pmatrix}1\\ \text{Cycle for}\\ \text{Internal}\\ \overline{\text{RD}}/\overline{\text{WR}}\end{pmatrix}\right]+\begin{pmatrix}\text{Hold}\\ \text{Offs}\end{pmatrix}$$

If, for example, you wanted to transfer 100 24-bit program memory words through the BDMA port, assuming five waitstates and no hold offs, the operation would take 1900 cycles. This is shown in the following equation:

$$\begin{pmatrix}100\\ \text{PM}\\ \text{Words}\end{pmatrix}\left[\begin{pmatrix}3\\ \text{Bytes}\\ \text{per}\\ \text{Word}\end{pmatrix}\begin{pmatrix}5\\ \text{Added}\\ \text{Waitstates}\\ \text{per Byte}\end{pmatrix}+\begin{pmatrix}1\\ \text{Cycle}\\ \text{for}\\ \text{Transfer}\end{pmatrix}+\begin{pmatrix}1\\ \text{Cycle for}\\ \text{Internal}\\ \overline{\text{RD}}/\overline{\text{WR}}\end{pmatrix}\right]+\begin{pmatrix}0\\ \text{Hold}\\ \text{Offs}\end{pmatrix}$$

Hold offs for DMA transfers are defined in the section “DMA Cycle Stealing, DMA Hold Offs, and $\overline{\text{IACK}}$ Acknowledge” at the end of this chapter.

11.2.2 BDMA Control Registers

A set of memory-mapped registers are used to setup and control transfers through the BDMA port. Figures 11.2 through 11.6 show these registers.

The BDMA Internal Address Register (BIAD) lets you set the 14-bit internal memory starting address for a BDMA transfer. The BDMA External Address Register (BEAD) lets you set the 14-bit external memory starting address for a BDMA transfer.

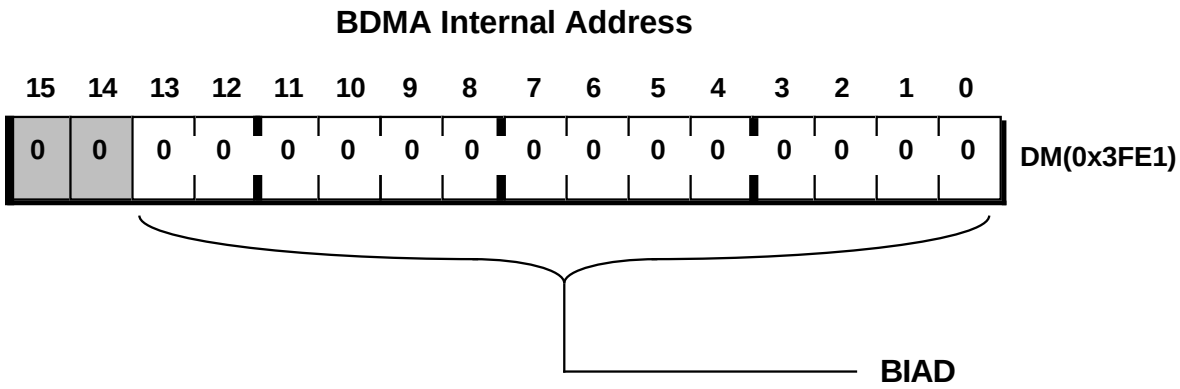


Figure 11.2 BDMA Internal Address Register

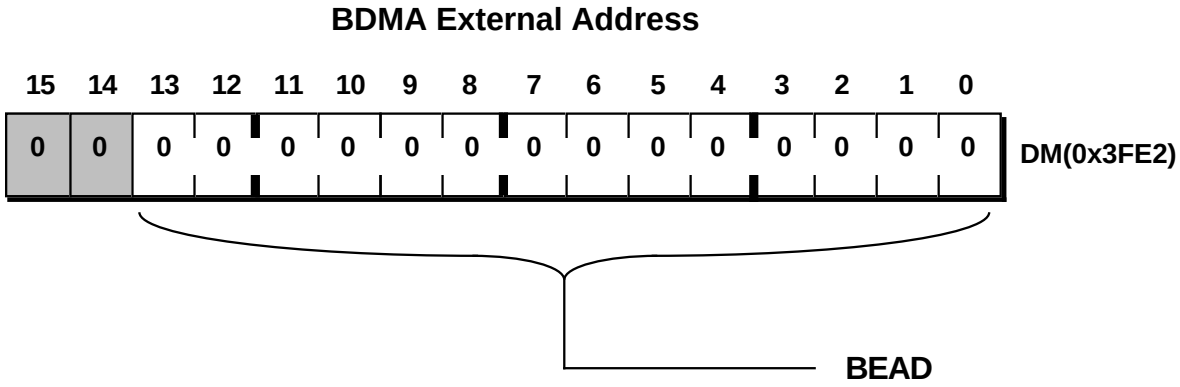
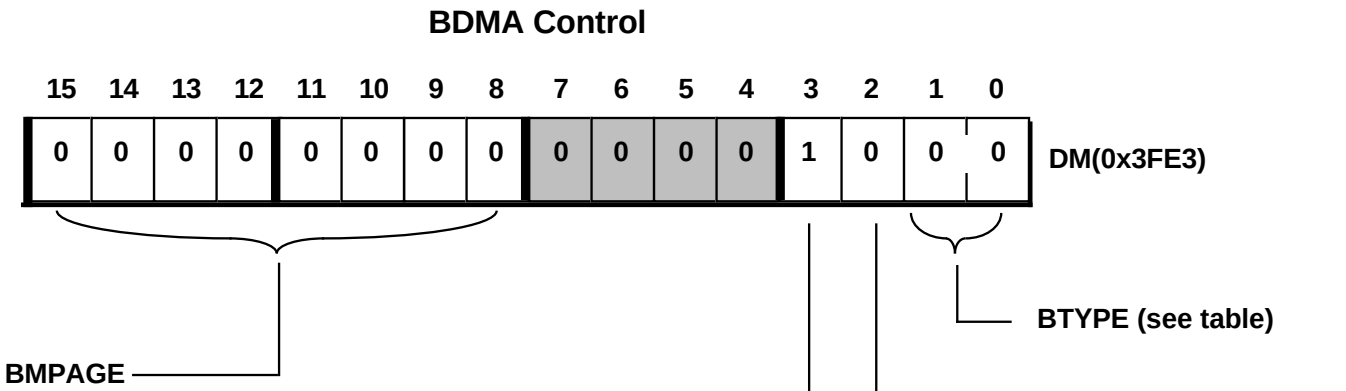


Figure 11.3 BDMA External Address Register

11 DMA Ports



BTYPE	00	01	10	11
Internal Memory Space	PM	DM	DM	DM
Word Size	24	16	8	8
Alignment	full word	full word	MSB	LSB

Figure 11.4 BDMA Control Register

The BDMA Control Register lets you set:

- The BDMA Transfer Type (BTYPE)
- The BDMA Direction (BDIR)
- The BDMA Context Reset (BCR)
- The BDMA Page (BMPAGE)

BTYPE can be:

00 24-bit Program Memory Words
01 16-bit Data Memory
10 8-bit bytes for Data Memory, MSB alignment
10 8-bit bytes for Data Memory, LSB alignment

BDIR can be:

0 from Byte Memory
1 to Byte Memory

BCR can be set to:

0 Allow program execution during BDMA
1 Inhibit program execution during BDMA transfers and cause a context reset after transfer is complete

BMPAGE lets you select the starting page for BDMA transfer.

Note: Rebooting with BDMA Context Reset (BCR=1) is similar to a Powerup Context Reset. For more details on processor states during reset and reboot, see the System Interface chapter of this manual.

The BWCOUNT register lets you start a BDMA transfer by writing the number of words for the transfer to this register. The count automatically decrements as the transfer proceeds. When the count is zero (i.e. transfer complete), the processor issues a BDMA interrupt. When MMAP and BMODE are set to zero on boot, a value of 32 (decimal) is written to this register directing the ADSP-2181 to load the first 32 locations of its internal program memory.

Two useful control techniques using this register are:

- Poll the BWCOUNT register to determine when the DMA transfer is complete (BWCOUNT=0), instead of waiting for the BDMA interrupt.

- Abort the DMA operation by writing a 1 to the BWCOUNT register and poll to determine when the transfer is complete (BWCOUNT=0), instead of waiting for the BDMA interrupt. (Note that the DMA transfer is aborted, and cannot be resumed later.)

BMWAIT consists of bits 12, 13, and 14 of the Programmable Flag & Composite Select Control Register. BMWAIT lets you select 0-7 waitstates (each equal to a single instruction cycle) to apply to each byte memory access. BMWAIT is set to 7 after a reboot.

11 DMA Ports

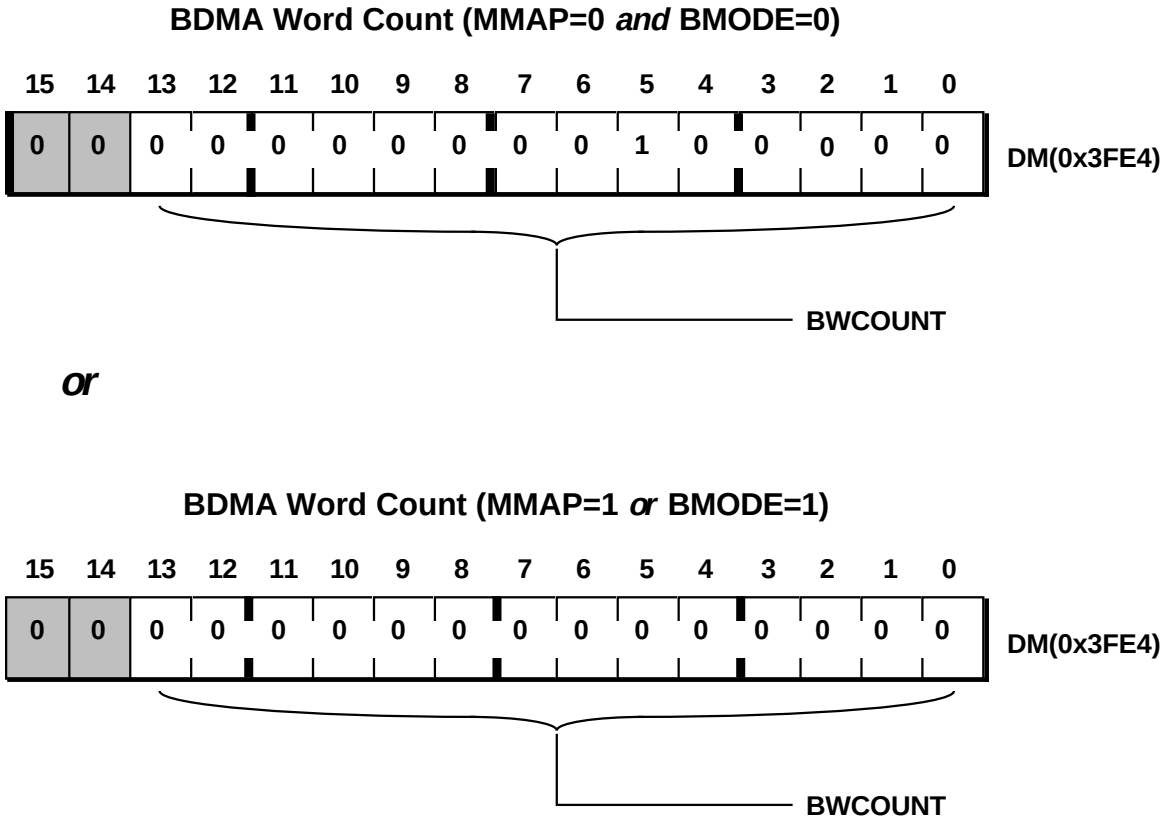


Figure 11.5 BDMA Word Count Register

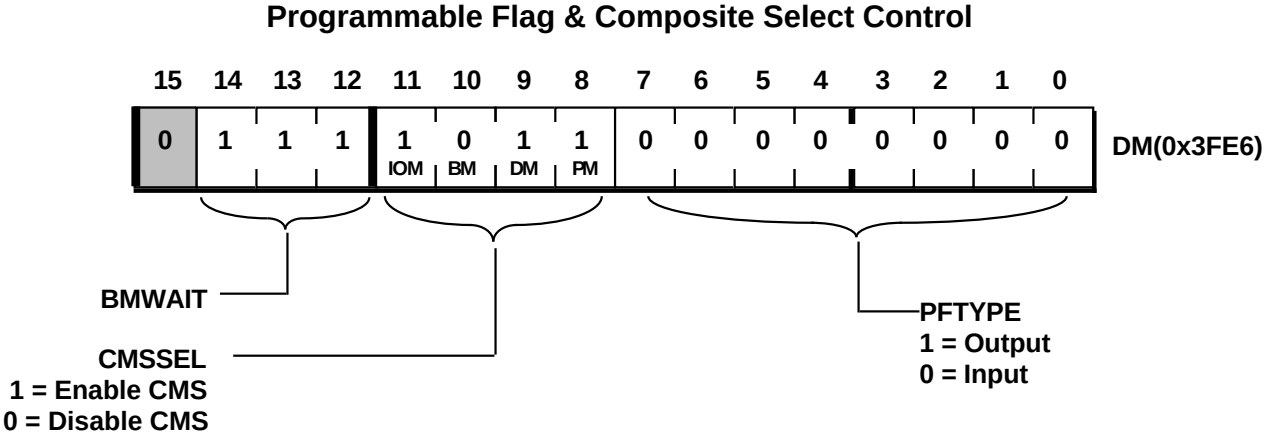


Figure 11.6 BMWAIT Field (in Programmable Flag & Composite Select Control Register)

11.2.3 Byte Memory Word Formats

In your byte memory ROM or RAM, data is stored by the ADSP-21xx PROM Splitter according to the data format you select: 24-bit program memory words, 16-bit data memory words, 8-bit data memory bytes with MSB-alignment, or 8-bit data memory bytes with LSB-alignment. The byte order for 24-bit program memory words and 16-bit data memory words stored in byte memory is most-significant-byte in the lower address. Table 11.1 shows an example of byte memory storage of all four code/data formats.

Note: When transferring either of the data memory byte formats, the unused byte of data memory is zero-filled.

BTYPE	*Internal Memory Address*	*Internal Memory Contents*	*Byte Memory Address (page 0x00)*	*Byte Memory Contents*
00	PM(0x0000)	0xABCDEF	BM(0x0000)	0xAB
			BM(0x0001)	0xCD
			BM(0x0002)	0xEF
00	PM(0x0001)	0x123456	BM(0x0003)	0x12
			BM(0x0004)	0x34
			BM(0x0005)	0x56
01	DM(0x0000)	0x9876	BM(0x0006)	0x98
			BM(0x0007)	0x76
01	DM(0x0001)	0x3456	BM(0x0008)	0x34
			BM(0x0009)	0x56
10	DM(0x0002)	0x9800	BM(0x000A)	0x98
10	DM(0x0003)	0x7600	BM(0x000B)	0x76
11	DM(0x0004)	0x0034	BM(0x000C)	0x34
11	DM(0x0005)	0x0056	BM(0x000D)	0x56

Table 11.1 Byte Memory Storage Formats

11.2.4 BDMA Booting

The entire on-chip program memory of the ADSP-2181, or any portion of it, can be loaded from an external source using a byte memory booting sequence. Booting from byte memory is one of two methods available for automatic booting after a reset.

Table 11.2 shows how to select the post-reset booting method using the ADSP-2181's MMAP and BMODE pins.

11 DMA Ports

MMAP Pin	*BMODE Pin*	*Booting Method*
0	0	**Boot through BDMA Port.** Boot sequence loads the first 32 program memory words from the byte memory space. After all 32 words are loaded, program execution begins at internal address PM(0x0000) with a BDMA interrupt pending.
0	1	**Boot through IDMA Port.** Boot sequence holds off execution while the host processor loads Program Memory using writes through the IDMA Port. Program execution begins when internal address PM(0x0000) is loaded.
1	–	**No Booting.** Boot sequence does *not* load memory *or* hold off execution. Program execution starts at external address PM(0x0000). The PMOVLAY register must be cleared (to zero).

Table 11.2 Selecting The ADSP-2181 Boot Method

The ADSP-2181 uses a BDMA boot sequence after reset when the BMODE and MMAP pins are held low. The BDMA port is initialized for booting as follows:

- BWCOUNT is set to 32
- BDIR, BMPAGE, BEAD, BIAD, and BTYPE are set to zero
- BCR is set to 1
- BMWAIT is set to 7

These initializations set the BDMA port to load 32 *words* (BWCOUNT)—*from* (BDIR)—byte memory *page* zero (BMPAGE)—byte memory *address* zero (BEAD) —to internal Program Memory *address* zero (BIAD)—using 24-bit program memory *word* format (BTYPE). The BDMA context reset bit (BCR) set to 1 inhibits program execution during BDMA transfer and causes execution to begin at address PM(0x0000) after the transfer. The number of waitstates (BMWAIT) for BDMA access is set to the maximum of 7. After the boot sequence is complete (32 words transferred), program execution begins at internal PM address 0x0000.

The ADSP-2100 Family PROM Splitter utility provides a boot loader option for ADSP-2181 based designs; see “Development Software Features for BDMA Booting” below.

If you are developing your own boot-loading software for the ADSP-2181, however, you should note that the BDMA Context Reset bit (BCR) is set to 1 (inhibiting program execution during BDMA transfer) and a BDMA interrupt is pending (signalling the first 32 word were sent) after the boot sequence is complete. Your program will have to process the interrupt (if you unmask the BDMA interrupt with the IMASK register) *or* clear the interrupt (with the IFC register).

In an alternate method, using the BDMA interrupt without context clear, a loader program could suspend program execution with the IDLE instruction while BDMA boot loading. If the loader sets the PM boot-load parameters, enables only the BDMA interrupt in the IMASK register, and then executes an IDLE instruction—the IDLE instruction suspends program execution until the BDMA interrupt occurs. At that point all of program memory is loaded.

11.2.4.1 Development Software Features for BDMA Booting

The ADSP-21xx PROM Splitter utility lets you create BDMA boot-loader programs for ADSP-2181-based designs. This provides a low overhead method for BDMA boot-loading your program. The boot loader program adds memory loader code to your executable program. The PROM Splitter generates loader code that initializes up to 6 pages of program memory and 4 pages of data memory, where each page is 16k bytes in size. Typically, the code generated by the PROM Splitter is burned into an EPROM and used as the ADSP-2181's Byte Memory space.

When the MMAP and BMODE pins equal 0, the ADSP-2181 will load the first 32 program memory words from the Byte memory space and then begin execution. The loader routine is in those first 32 words; it continues to load from the Byte Port until your whole program is loaded.

Refer to the *ADSP-2100 Family Assembler Tools & Simulator Manual* as well as the software release note for complete information on the PROM Splitter features.

11 DMA Ports

11.3 IDMA PORT

The IDMA Port of the ADSP-2181 is a parallel I/O port that lets the processor's internal memory be read or written by a host system. The IDMA Port architecture eases host bus interface design.

Think of the IDMA port as a gateway to all internal memory locations on the DSP (except for the processor's memory-mapped control registers). The IDMA Port has a 16-bit multiplexed address and data bus that supports access to both 16-bit Data Memory and 24-bit Program Memory. IDMA Port read/write access is completely asynchronous and a host can access the DSP's internal memory while the ADSP-2181 is operating at full speed.

Unlike the Host Interface Port (HIP) of the ADSP-2171 and ADSP-2111, the IDMA port does not require any ADSP-2181 processor intervention to maintain data flow. The host system can access ADSP-2181 internal memory directly, without going through a set of mailbox registers. Direct access to DSP memory increases throughput for block data transfers. Through the IDMA port, internal memory accesses can be performed with an overhead of one DSP processor cycle per word.

The ADSP-2181 supports boot loading through the IDMA port, through the BDMA port, or from an external Program Memory Overlay. The BMODE and MMAP pins select the DSP's boot mode and memory map. Setting BMODE=1 and MMAP=0 directs the ADSP-2181 to boot through the IDMA Port. For information on IDMA booting, see "Boot Loading Through The IDMA Port" at the end of this chapter.

Note: The IDMA port cannot be used to read or write the ADSP-2181's memory-mapped control registers. See "Modifying Control Registers for IDMA."

11.3.1 IDMA Port Pin Summary

The IDMA Port pins are shown below in Table 11.3.

Pin Name(s)	*Input/ Output*	*Function*
$\overline{\text{IRD}}$	I	IDMA Port Read Strobe
$\overline{\text{IWR}}$	I	IDMA Port Write Strobe
$\overline{\text{IS}}$	I	IDMA Port Select
IAL	I	IDMA Port Address Latch Enable
IAD0-15	I/O	IDMA Port Address/Data Bus
$\overline{\text{IACK}}$	O	IDMA Port Access Ready Acknowledge*

Table 11.3 IDMA Port Pins

* After reset, $\overline{\text{IACK}}$ is asserted (low). It stays low until an IDMA transfer is initiated. After each IDMA operation is completed, $\overline{\text{IACK}}$ will again be low.

Four IDMA port inputs control when the port is selected ($\overline{IS}$) for read ($\overline{IRD}$), write ($\overline{IWR}$), or address latch (IAL) operations on its address/data bus (IAD0-15). The IDMA Port Select ($\overline{IS}$) line acts as a chip select for all IDMA operations.

Asserting the IDMA Port Select ($\overline{IS}$) and address latch enable (IAL) directs the ADSP-2181 to write the address on the IAD0-15 bus into the IDMA Control Register. This register, shown in Figure 11.7, is memory-mapped at address DM(0x3FE0). Note that the latched address (IDMAA) cannot be read back by the host.

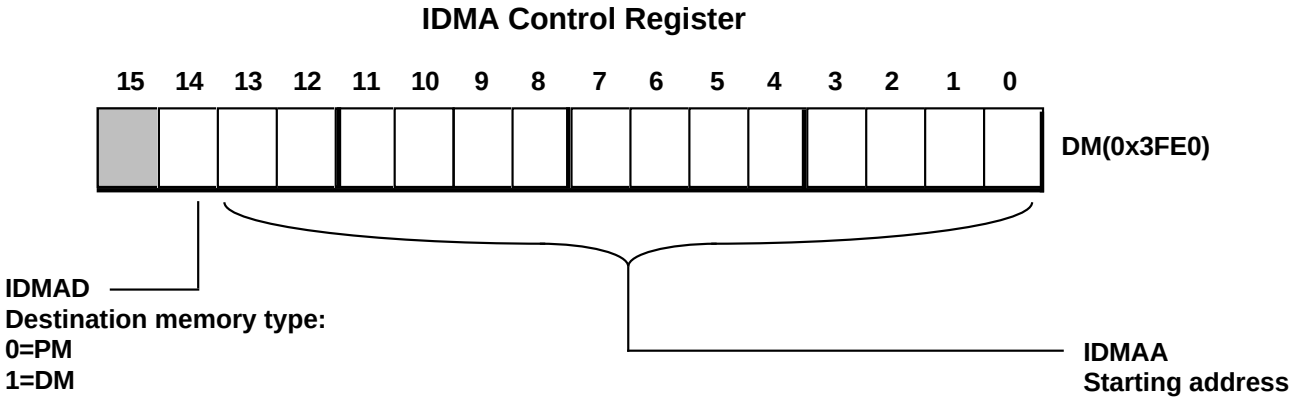


Figure 11.7 IDMA Control Register

Asserting the IDMA Port Select ($\overline{IS}$) and Read strobe ($\overline{IRD}$) inputs directs the ADSP-2181 to output the contents of the memory location pointed to by the IDMA Control register onto the IDMA data bus.

Asserting the IDMA Port Select ($\overline{IS}$) and Write strobe ($\overline{IWR}$) inputs directs the ADSP-2181 to write the input from the IDMA data bus to the address pointed to by the IDMA register.

When reading/writing to Data Memory, the IDMA data bus pins make up a 16-bit Data Memory word. When reading/writing to Program Memory, the upper 16 bits of the 24-bit Program Memory word are sent first on the IDMA data bus pins. On the next IDMA Port read/write, the lowest 8 bits of the Program Memory word are sent on bits 0-7 of the IDMA data bus. For reads, the ADSP-2181 sets data bus lines 8-15 to 0; for writes, the ADSP-2181 ignores bits 8-15 from the host.

The IDMA Port Access Acknowledge ($\overline{IACK}$) line identifies completion of data reads/write operations. It also acts as a busy signal for the IDMA Port. External devices must wait for this signal to go low *before* modifying IDMA Control register or starting the next read/write operation.

11 DMA Ports

11.3.2 IDMA Port Functional Description

The IDMA Port lets a host system directly access internal ADSP-2181 memory locations (but *not* the memory-mapped control registers). Figure 11.8 shows a flow chart of the most general case for IDMA transfers.

In the case shown in Figure 11.8, the host system starts an IDMA transfer by checking the state of the $\overline{\text{IACK}}$ line to determine port status (ready/busy). When the IDMA port is ready, the host directs the ADSP-2181 (with the $\overline{\text{IS}}$ and IAL lines) to latch the IDMA internal memory address from the IDMA address/data bus to the IDMA Control Register. (Note that the latched address cannot be read back by the host.)

Next, the host (using the $\overline{\text{IS}}$ and $\overline{\text{IRD}}$ or $\overline{\text{IS}}$ and $\overline{\text{IWR}}$ lines) begins reading (or writing) the DSP's internal memory until done. With each IDMA read or write operation, the the ADSP-2181 automatically increments the IDMA internal memory address. Note that the ADSP-2181 continues program execution throughout the IDMA transfer operation, *except* during the "stolen" cycle used to do the memory access.

The case shown in Figure 11.8 is not the only way to use the IDMA port. Some variations on this scheme include:

- After completing an IDMA port read/write operation, the host could change the IDMA internal memory address and start a new operation from a different starting address.
- After latching an IDMA internal memory address, the host could stop the operation and come back at a later time to proceed with the read/write operation. The IDMA starting memory address remains in the IDMA Control Register until the host or DSP changes it.
- The ADSP-2181 can also read and write the IDMA Control Register as part of your program. This means that the host could just control read/write operations and let the ADSP-2181 control the IDMA starting memory address.
- Using the *IDMA short read cycle* (which does not wait for the data-ready assertion of the $\overline{\text{IACK}}$ signal), you could set up a single-location data buffer for IDMA read transfers. For information on how this data buffer would work, see "IDMA Port Short Read Cycle" below.
- For ADSP-2181 applications with a host processor or host ASIC that does *not* use a data-ready or write-complete acknowledge, use the *IDMA short read/write cycles*.

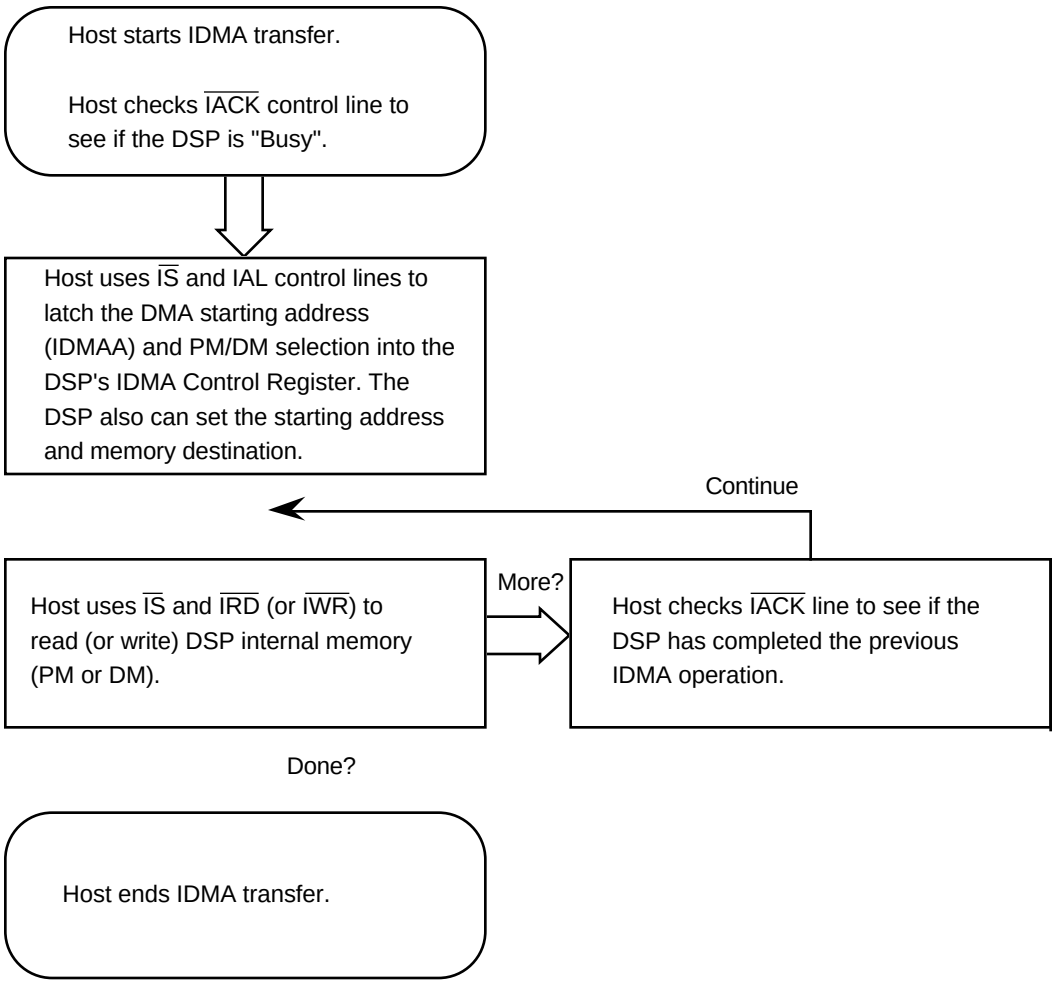


Figure 11.8 General IDMA Transfer Flow Chart

There are some restrictions on IDMA operations. These hardware/ software design restrictions include:

- If your design has both the host and ADSP-2181 writing to the IDMA Control Register, do *not* let both write to this register at the same time; the results of this are indeterminate.

- Host reads of internal Program Memory take two IDMA reads (for a 24-bit word through a 16-bit port). If an IDMA address latch cycle or a ADSP-2181 write to the IDMA Control Register occurs after the first Program Memory read cycle, the IDMA port "loses" the second half of the 24-bit Program Memory word. The next IDMA read or write uses the address selected by the new contents of the IDMA Control Register. Note that writing to the IDMA Control Register after the first half of a Program Memory IDMA read lets you read just 16-bit data from Program Memory.

- Host writes to internal Program Memory take two IDMA writes (for a 24-bit word through a 16-bit port). If an IDMA address latch cycle or a ADSP-2181 write to the IDMA Control Register occurs after a first Program Memory write cycle, the IDMA port "loses" the Program Memory word without changing the contents of memory. The next IDMA read or write accesses the address selected by the new contents of the IDMA Control Register.

- Host memory accesses through the IDMA port that occur while the ADSP-2181 is in powerdown have some restrictions. For information on powerdown restrictions on IDMA port transfers, see the *System Interface* chapter of this manual.

11.3.3 Modifying Control Registers for IDMA

The ADSP-2181's memory-mapped control registers are protected from DMA transfers to prevent accidental corruption. You may want the host processor to read and write these registers, however, in order to determine the ADSP-2181's configuration and then change it.

To read the memory-mapped control registers, you must first transfer the contents of these locations to another area of internal RAM. The following code segment shows a loop that performs this task:

```
.const NUM_REG=32;
.var/dm/ram  temp_array[NUM_REG];

            i0=^temp_array;
            l0=0;
            i1=0x3fe0;
            l1=0;
            m1=1;
            cntr=NUM_REG;
            do transfer until ce;
            ax0=dm(i1,m1);
transfer:   dm(i0,m1)=ax0;
```

To have the host write to the memory-mapped control registers, you must first load the values to a temporary buffer (through the IDMA port) and then signal the ADSP-2181 to transfer the contents of the temporary buffer to the memory-mapped control registers. This transfer is performed in a similar manner as the code shown above. You should set up some form of signalling between the host and the ADSP-2181, either interrupts, flag I/O, or a mailbox register. This will provide a mechanism for the host to tell the DSP when to perform an operation and vice versa.

11.3.4 IDMA Timing

From the host system interface point of view, there are three IDMA port operations with critical timing parameters. These operations are:

- latching the IDMA internal memory address,
- reading from the IDMA port, and
- writing to the IDMA port.

The following sections cover the timing details of each of these operations.

11.3.4.1 Address Latch Cycle

The host writes the DMA starting address and destination memory type (DM or PM) using the IDMA address latch cycle. The address latch cycle, shown in Figure 11.9, consists of the following steps:

1. Host ensures that $\overline{\text{IACK}}$ line is low.
2. Host asserts IAL and $\overline{\text{IS}}$, directing the ADSP-2181 to latch the IDMA starting address from the IAD15-0 address/data bus into the IDMA Control Register.
3. Host drives the starting address (bits 0-13) and destination memory type (bit 14) onto the IAD15-0 bus. (Bit 15 must be a 0.)

Note that $\overline{\text{IRD}}$ and $\overline{\text{IWR}}$ remain high (inactive) throughout the latch operation.

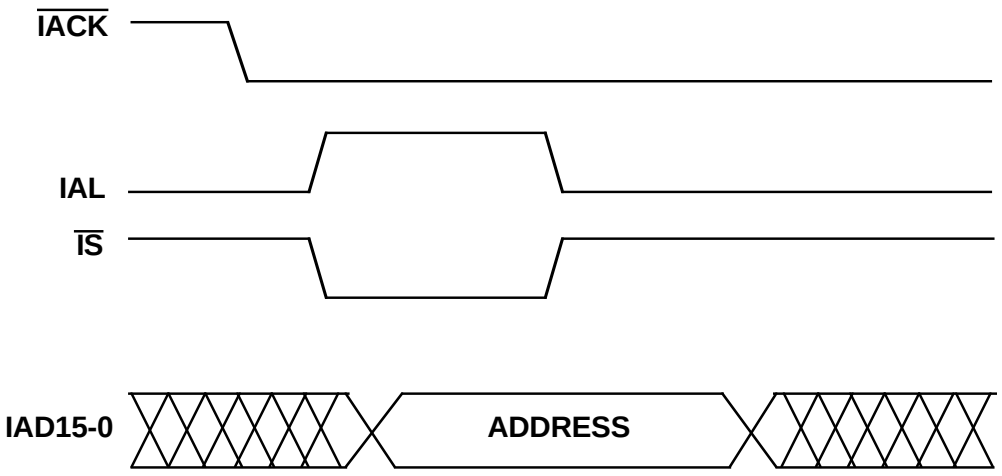


Figure 11.9 IDMA Address Latch Cycle Timing

Note: The IDMA starting address and destination memory type is available to the host and to the ADSP-2181 in the IDMA Control Register. For Data Memory accesses, the ADSP-2181 increments the address automatically after each IDMA read or write transfer (16-bit word). For Program Memory accesses, the ADSP-2181 increments the address automatically after each *pair* of IDMA read or write transfers (24-bit word).

Warning: Both the ADSP-2181 and the host can specify the starting address by writing to the IDMA Control Register. Do not let the ADSP-2181 access the IDMA Control Register while it is being written by the host; this operation will have an indeterminate result.

11.3.4.2 Long Read Cycle

The host reads the contents of an ADSP-2181 internal memory location using the IDMA port long read cycle. The read cycle, shown in Figure 11.10, consists of the following steps:

1. Host ensures that $\overline{\text{IACK}}$ line is low.
2. Host asserts $\overline{\text{IRD}}$ and $\overline{\text{IS}}$ (low), causing the ADSP-2181 to put the contents of the location pointed to by the IDMA address on the IAD15-0 address/data bus.
3. ADSP-2181 deasserts $\overline{\text{IACK}}$ line, indicating the requested data is being fetched. When the ADSP-2181 asserts the $\overline{\text{IACK}}$ line, the requested data is driven on the IAD address/data bus.
4. Host detects the $\overline{\text{IACK}}$ line is now low and reads the data (READ DATA) from the IAD15-0 address/data bus. After reading the data, the host deasserts $\overline{\text{IRD}}$ and $\overline{\text{IS}}$.

Note that IAL is low (inactive) and $\overline{\text{IWR}}$ is high (inactive) throughout the read operation.

IDMA memory accesses “steal” one processor cycle, but may only occur on instruction cycle boundaries. The best-case response for a 16-bit Data Memory read or the first 16 bits of a Program Memory read is 2.5 processor cycles; worst case is 3.5 cycles. One cycle is for synchronization, one is for reading the memory internally, and one-half cycle is for $\overline{\text{IACK}}$ setup time. A second cycle of synchronization may be required. Thus the best-case and worst-case response times are determined as follows:

Best Case: 1 cycle (sync) + 1 cycle (internal memory read) + 0.5 cycle ($\overline{\text{IACK}}$ setup) = **2.5 cycles**

Worst Case: 1 cycle (sync) + 1 cycle (sync) + 1 cycle (internal memory read) + 0.5 cycle ($\overline{\text{IACK}}$ setup) = **3.5 cycles**

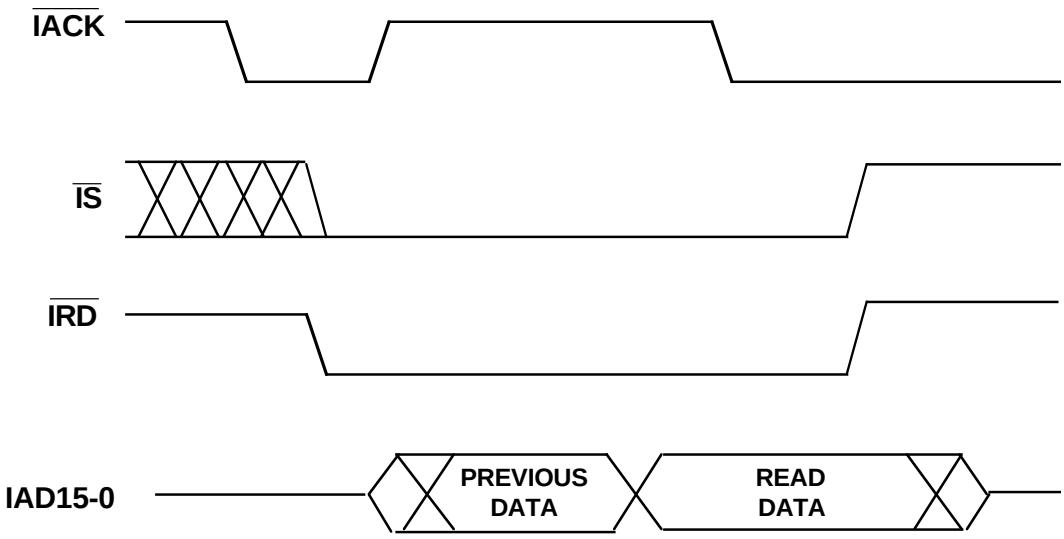


Figure 11.10 IDMA Long Read Cycle Timing

In the case of a Program Memory operation, the second IDMA port read cycle for a given internal 24-bit word does not require an internal memory access, does not wait for an instruction cycle boundary, and takes 1.5 or 2.5 cycles.

The best- and worst-case response times given above assume no system hold offs. Hold offs for DMA transfers are defined in the section "DMA Cycle Stealing, DMA Hold Offs, and $\overline{\text{IACK}}$ Acknowledge" at the end of this chapter.

Warning: If an IDMA address latch cycle or an ADSP-2181 write to the IDMA Control Register occurs after a first Program Memory read cycle (16 bits), the IDMA port will lose the second half of the Program Memory word. The ADSP-2181 treats the next IDMA access as the first operation for the new IDMA address and destination.

11.3.4.3 Short Read Cycle

The host reads the contents of a ADSP-2181 internal memory location using the IDMA short read cycle. The read cycle, shown in Figure 11.11, consists of the following steps:

1. Host ensures that $\overline{\text{IACK}}$ line is low.
2. Host asserts $\overline{\text{IRD}}$ and $\overline{\text{IS}}$ (low), directing the ADSP-2181 to put the contents of the location pointed to by the target IDMA address on the IAD15-0 address/data bus.
3. ADSP-2181 deasserts $\overline{\text{IACK}}$ line, indicating the requested data is being fetched.
4. Host detects the $\overline{\text{IACK}}$ line is now high and reads the data (PREVIOUS DATA) from the IAD15-0 address/data bus, *before* the requested data (READ DATA) is driven on the IAD address/data bus—*not* waiting for the ADSP-2181 to assert the $\overline{\text{IACK}}$ line. After reading the data, the host deasserts $\overline{\text{IRD}}$ and $\overline{\text{IS}}$.

The host must do an initial "dummy" read, to make the ADSP-2181 put the first data word (PREVIOUS DATA) on the IAD15-0 bus.

Note that IAL is low (inactive) and $\overline{\text{IWR}}$ is high (inactive) throughout the read operation.

The IDMA Short Read and Long Read cycles provide different alternatives for implementing your DMA transfers. Short reads are useful for hosts that can handle the faster timing of these accesses, while long reads allow slower hosts more time.

The IDMA short read cycle also serves as a single-location data buffer. If you are using the ADSP-2181 in a multiprocessing environment, using this buffer is one way to avoid tying up the IAD bus (waiting for $\overline{\text{IACK}}$ signal).

Warning: If an IDMA address latch cycle or a ADSP-2181 write to the IDMA Control register occurs after a first Program Memory read cycle, the IDMA port will lose the second half of the Program Memory word. The ADSP-2181 treats the next host data on the IAD address/data bus as the new contents of the IDMA Control Register.

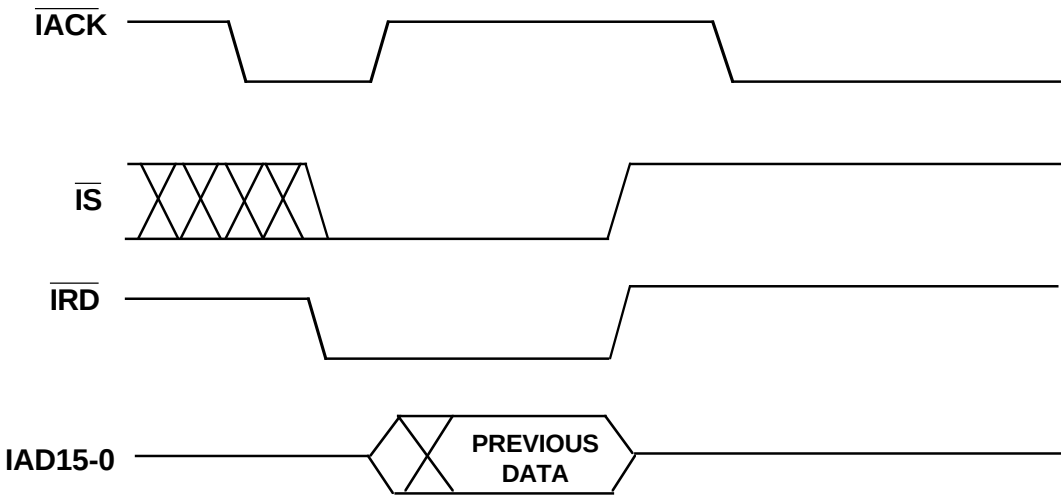


Figure 11.11 IDMA Short Read Cycle Timing

11.3.4.4 Long Write Cycle

The host writes the contents of an internal memory location using the IDMA long write cycle. The write cycle, shown in Figure 11.12, consists of the following steps:

1. Host ensures that $\overline{\text{IACK}}$ line is low.
2. Host asserts $\overline{\text{IWR}}$ and $\overline{\text{IS}}$ (low), directing the ADSP-2181 to write the data on the IAD15-0 address/data bus to the location pointed to by the target IDMA address .
3. ADSP-2181 deasserts the $\overline{\text{IACK}}$ line, indicating it recognizes the IDMA write operation.
4. Host drives the data on the IAD address/data bus.
5. ADSP-2181 asserts $\overline{\text{IACK}}$ line, indicating it latched the data on the IAD15-0 address/data bus.
6. Host recognizes the $\overline{\text{IACK}}$ line is now low, stops driving the data on the IDMA address/data bus and deasserts $\overline{\text{IWR}}$ and $\overline{\text{IS}}$ (ending the IDMA Long Write Cycle).

Note that IAL is low (inactive) and $\overline{\text{IRD}}$ is high (inactive) throughout the write operation.

11 DMA Ports

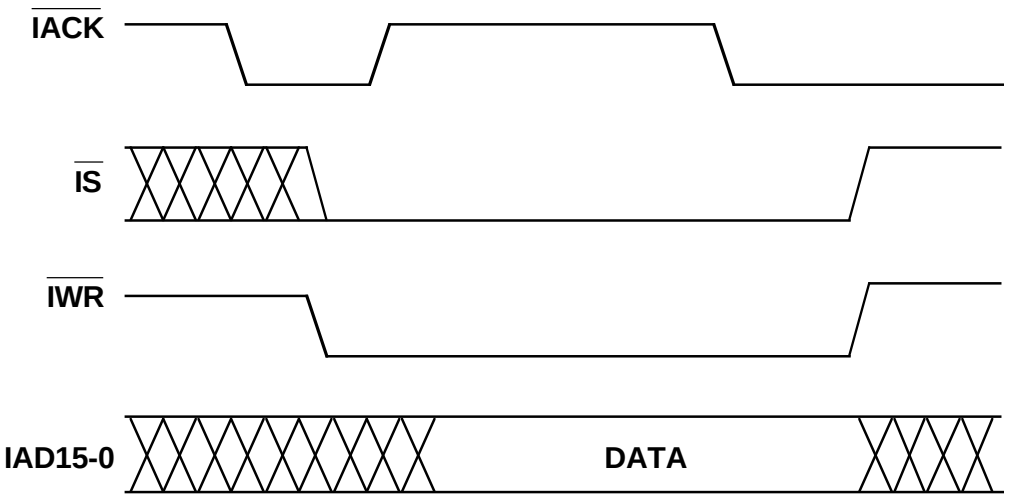


Figure 11.12 IDMA Long Write Cycle Timing

Note: IDMA port writes to Program Memory require two IDMA port write cycles to write a word to ADSP-2181 internal Program Memory. The ADSP-2181 acknowledges the IDMA port write of the first 16 bits (MSBs of PM word) as they are written to a temporary holding latch, *not* waiting for an instruction cycle boundary. The ADSP-2181 does not assert the IACK line after the second Program Memory write (or all Data Memory writes) *until* the internal memory write is complete and the IDMA port is ready for another transaction.

Warning: Host IDMA write accesses to internal Program Memory take two IDMA port writes (24-bit word through a 16-bit port). If an IDMA address latch cycle or a ADSP-2181 write to the IDMA Control register occurs after a first program memory write cycle, the IDMA port "loses" the Program Memory word without changing the contents of ADSP-2181 internal memory. The next IDMA read or write uses the address selected by the new contents of the IDMA Control register.

11.3.4.5 Short Write Cycle

The host writes the contents of a ADSP-2181 internal memory location using the IDMA short write cycle. The write cycle, shown in Figure 11.13, consists of the following steps:

1. Host ensures that $\overline{\text{IACK}}$ line is low.
2. Host asserts $\overline{\text{IWR}}$ and $\overline{\text{IS}}$ (low), directing the ADSP-2181 to write the data on the IAD15-0 address/data bus to the location pointed to by the target IDMA address .
3. ADSP-2181 deasserts $\overline{\text{IACK}}$ line (high), indicating it recognizes the IDMA write operation.
4. Host drives the data on the IAD address/data bus.
5. Host deasserts $\overline{\text{IWR}}$ and $\overline{\text{IS}}$ *after* meeting the short write timing requirements (ending the short write cycle).
6. ADSP-2181 detects $\overline{\text{IWR}}$ and $\overline{\text{IS}}$ have gone high, then latches the data on the IAD address/data bus.
7. Host stops driving the data on the IAD15-0 address/data bus *after* meeting the short write timing requirements.

Note that IAL is low (inactive) and IRD is high (inactive) throughout the write operation.

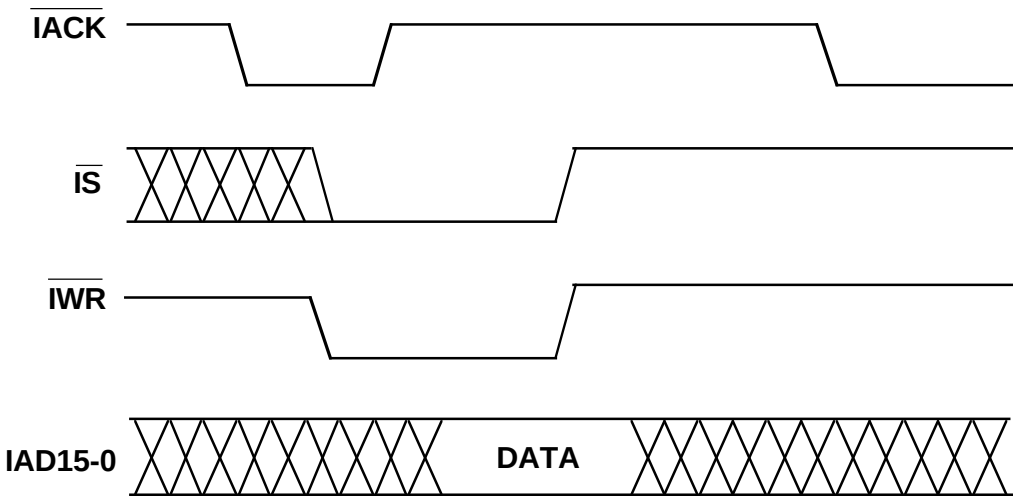


Figure 11.13 IDMA Short Write Cycle Timing

Note: IDMA port writes to Program Memory require two IDMA port write cycles to write a word to ADSP-2181 internal Program Memory. The ADSP-2181 acknowledges the IDMA port write of the first 16 bits (MSBs of PM word) as they are written to a temporary holding latch, *not* waiting for an instruction cycle boundary. The ADSP-2181 does not assert the $\overline{\text{IACK}}$ line after the second Program Memory write (or all Data Memory writes) *until* the internal memory write is complete and the IDMA port is ready for another transaction.

Warning: If an IDMA address latch cycle or a ADSP-2181 write to the IDMA Control register occur after a first Program Memory write cycle, the IDMA port will lose the first half of the Program Memory word. The next Program Memory write will be considered the first half of a Program Memory write pair.

There are two features that differentiate between the IDMA Port long write and short write. The long write supports hosts (processors or ASICs) that allow a data-written acknowledge. If your host needs the ADSP-2181 to signal that it has written the data, use the IDMA long read cycle.

The short write lets your host hold data on the bus just until it is latched and then release the bus. If you are using the ADSP-2181 in a multiprocessing environment, using the short write is one way to avoid tying up the IAD15-0 data bus (waiting for $\overline{\text{IACK}}$ signal). Short writes are also useful for hosts that can handle the short write timing, but can't extend the accesses with $\overline{\text{IACK}}$ (when holdoffs occur).

11.3.5 Boot Loading Through The IDMA Port

The ADSP-2181 supports boot loading through the IDMA port. To boot through the IDMA Port, use the following steps:

- Reset the processor (assert $\overline{\text{RESET}}$).
- Set MMAP=0 and BMODE=1. These pin settings select IDMA booting.
- Deassert $\overline{\text{RESET}}$.
- Load ADSP-2181 internal memory through the IDMA port. Program execution is held off until you write to Program Memory address zero, PM(0x0000). The ADSP-2181 responds to IDMA control signals (IAL, $\overline{\text{IS}}$, $\overline{\text{IWR}}$, and $\overline{\text{IRD}}$) and provides acknowledge ($\overline{\text{IACK}}$) in the same manner as during non-booting IDMA transfers.
- Write to PM(0x0000) to begin program execution.

Warning: Make certain to load all of the necessary memory locations with the proper data *before* writing to PM(0x0000).

11.3.6 DMA Cycle Stealing, DMA Hold Offs, and $\overline{\text{IACK}}$ Acknowledge

The $\overline{\text{IACK}}$ signal is generated by the ADSP-2181 to signal that it is safe to read or write through the IDMA port. After reset, $\overline{\text{IACK}}$ is asserted (low). It stays low until an IDMA transfer is initiated. After each IDMA operation is completed, $\overline{\text{IACK}}$ will again be low.

In order for $\overline{\text{IACK}}$ to be asserted (low) during the IDMA operation, the IDMA port must have completed the internal memory access by either writing data to memory or reading data from memory. The IDMA port must "steal" a processor cycle to do this. In order to steal a processor cycle, the IDMA port must wait for an instruction completion boundary. *Thus if $\overline{\text{IACK}}$ is not asserted, it is not safe for the host to access the IDMA port.*

In most cases, there is an instruction boundary on every clock cycle (CLKOUT period) and the IDMA port can complete its transfer in a given period of time. There are, however, some instances where either the ADSP-2181 does not complete an instruction in one clock cycle or the IDMA port cannot access memory. These are *DMA hold offs*:

- **Bus Request** – If the ADSP-2181 is being held in Bus Request when it attempts an external access (DM overlay, PM overlay, or I/O memory space), or if it is not in GO mode, processor execution stops in the middle of the cycle and no instruction boundary is encountered. Therefore, the IDMA port cannot complete its internal memory access and $\overline{\text{IACK}}$ will be held off.

- **External Access with Wait State(s)** – If the ADSP-2181 is performing a wait-stated external access (DM overlay, PM overlay, or I/O memory space), then the instruction cycle will not complete until the access has completed; the IDMA port cannot steal a cycle, and $\overline{\text{IACK}}$ will be held off.

- **Multiple External Accesses** – If the ADSP-2181 is executing a multifunction instruction where more than one of the required elements (PM instruction fetch, PM data access, or DM data access) resides externally, it will require more than one cycle to complete the instruction and $\overline{\text{IACK}}$ will be held off. Likewise, if the ADSP-2181 is executing an instruction from external PM that initiates an I/O memory space access, $\overline{\text{IACK}}$ will be held off until the cycle completes.

- **IDLE n (clock-reducing IDLE instruction)** – Because this instruction slows down the effective cycle time of the ADSP-2181, $\overline{\text{IACK}}$ may be delayed.

- **SPORT Autobuffering to External Memory with Waitstated Access** – When one of the processor's serial ports needs to access external memory for autobuffering and the external access takes more than one cycle, the IDMA transfer will be held off.

- **EZ-ICE Emulation** – When the EZ-ICE emulator is controlling your ADSP-2181 target system, IDMA transfers may be held off for periods of time.

Using the $\overline{\text{IACK}}$ signal simplifies your system design by allowing you to ignore hold-off conditions. If you always wait for $\overline{\text{IACK}}$ to assert before accessing the IDMA port, the DMA transfers will always operate properly.

You can ignore $\overline{\text{IACK}}$, however, *if you are sure that no hold-offs occur in your system or if your IDMA accesses are longer than any hold-offs*. To be sure of this, you must carefully analyze all possible hold-off conditions of your system.

Programming Model 12

12.1 OVERVIEW

From a programming standpoint, the ADSP-21xx processors consist of three computational units, two data address generators, and a program sequencer, plus on-chip peripherals and memory that vary with each processor. Almost all operations using these architectural components involve one or more registers—to store data, to keep track of values such as pointers, or to specify operating modes, for example.

Internal registers hold data, addresses, control information or status information. For example, AX0 stores an ALU operand (data); I4 stores a DAG2 pointer (address); ASTAT contains status flags from arithmetic operations; and fields in the Wait State register control the number of wait states for different zones of external memory.

There are two types of accesses for registers. Dedicated registers such as MX0 and IMASK can be read and written explicitly in assembly language. For example:

```
MX0=1234;
IMASK=0xF;
```

Memory-mapped registers—the System Control Register, Wait State Control Register, timer registers, SPORT registers, etc.—are accessed by reading and writing the corresponding data memory locations. For example, this code clears the Wait State Control Register, which is mapped to data memory location 0x3FFE:

```
AX0=0;
DM(0x3FFE)=AX0;
```

(AX0 is used to hold the constant 0 because there is no instruction to write an immediate data value to memory using an immediate address.)

12 Programming Model

The ADSP-21xx registers are shown in Figure 12.1. Not all of these registers are available on every processor. The registers are grouped by function: data address generators (DAGs), program sequencer, computational units (ALU, MAC and shifter), bus exchange (PX), memory interface, timer, SPORTs, host interface and DMA interfaces.

12.1.1 Data Address Generators

DAG1 and DAG2 each have twelve 14-bit registers: four index (I) registers for storing pointers, four modify (M) registers for updating pointers and four length (L) registers for implementing circular buffers. DAG1 addresses data memory only and has the capability of bit-reversing its outputs. DAG2 addresses both program and data memory and can provide addresses for indirect branching (jumps and calls) as well as for accessing data.

For example:

```
AX0=DM(I0,M0);
```

is an indirect data memory read from the location pointed to by I0. Once the read is complete, I0 is updated by M0.

```
PM(I4,M5)=MR1;
```

is an indirect program memory data write to the address pointed to by I4 with a post modify by M5. The instruction

```
JUMP (I4);
```

is an example of an indirect jump.

12.1.1.1 Always Initialize L Registers

The ADSP-21xx processors allow two addressing modes for data memory accesses: direct and register indirect. Indirect addressing is accomplished by loading an address into an I (index) register and specifying one of the available M (modify) registers.

The L registers are provided to facilitate wraparound addressing of circular data buffers. A circular buffer is only implemented when an L register is set to a non-zero value. *For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.* Do not assume that the L registers are automatically initialized or may be ignored; the I, M, and L registers contain random values following processor reset. Your program must initialize the L registers corresponding to any I registers it uses.

Programming Model 12

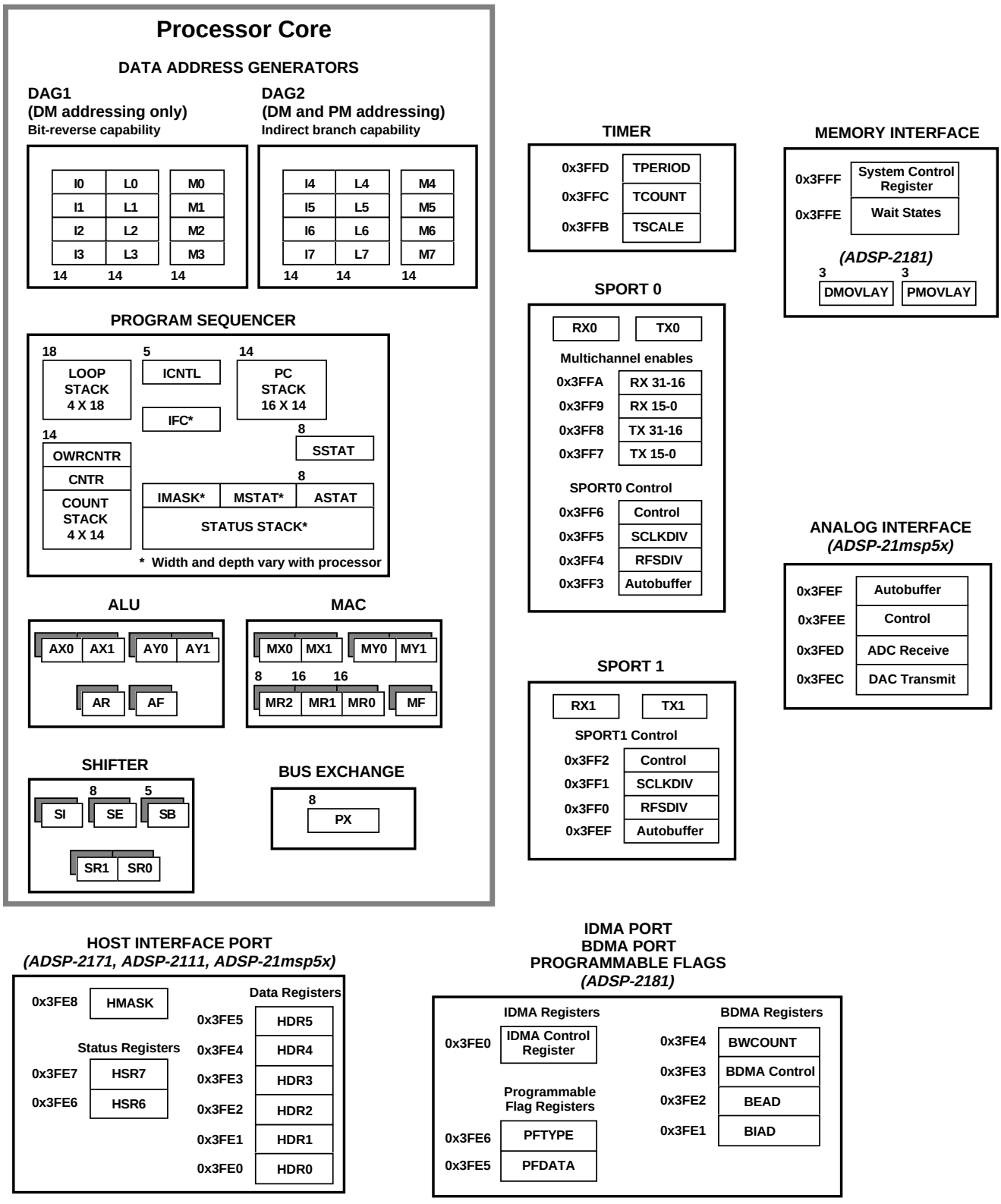


Figure 12.1 ADSP-21xx Registers

Shading denotes secondary (alternate) registers.
Registers are 16 bits wide (unless otherwise marked).

12.1.2 Program Sequencer

Registers associated with the program sequencer control subroutines, loops, and interrupts. They also indicate status and select modes of operation.

12.1.2.1 Interrupts

The ICNTL register controls interrupt nesting and external interrupt sensitivity; the IFC register lets you force and clear interrupts in software; the IMASK register masks (disables) individual interrupts. The widths of the IFC and IMASK registers depend on the processor, since different ADSP-21xx processors support different numbers of interrupts.

The ADSP-2171, ADSP-2181, and ADSP-21msp58/59 support a global interrupt enable instruction (`ENA INTS`) and interrupt disable instruction (`DIS INTS`).

Interrupts are enabled by default at reset. Executing the disable interrupt instruction causes all interrupts to be masked without changing the contents of the IMASK register. Disabling interrupts does not affect serial port autobuffering, which will operate normally whether or not interrupts are enabled. The disable interrupt instruction masks all user interrupts including the powerdown interrupt.

The interrupt enable instruction allows all unmasked interrupts to be serviced again.

12.1.2.2 Loop Counts

The CNTR register stores the count value for the currently executing loop. The count stack allows the nesting of count-based loops to four levels. A write to CNTR pushes the current value onto the count stack before writing the new value. For example:

```
CNTR=10;
```

pushes the current value of CNTR on the count stack and then loads CNTR with 10.

OWRCNTR is a special syntax with which you can overwrite the count value for the current loop without pushing CNTR on the count stack. OWRCNTR cannot be read (i.e. used as a source register), and must not be written in the last instruction of a DO UNTIL loop.

12.1.2.3 Status And Mode Bits

The stack status (SSTAT) register contains full and empty flags for stacks. The arithmetic status (ASTAT) register contains status flags for the computational units. The mode status (MSTAT) register contains control bits for various options. MSTAT contains 4 bits that control alternate register selection for the computational units, bit-reverse mode for DAG1, and overflow latch and saturation modes for the ALU. MSTAT also has 3 bits to control the MAC result placement, timer enable, and Go mode enable.

Use the Mode Control instruction (ENA, DIS) to conveniently enable or disable processor modes.

12.1.2.4 Stacks

The program sequencer contains four stacks that allow loop, subroutine and interrupt nesting.

The PC stack is 14 bits wide and 16 locations deep. It stores return addresses for subroutines and interrupt service routines, and top-of-loop addresses for loops. PC stack handling is automatic for subroutine calls and interrupt handling. In addition, the PC stack can be manually pushed or popped using the PC Stack Control instructions `TOPPCSTACK=`*`reg`* and *`reg`*`=TOPPCSTACK`.

The loop stack is 18 bits wide, 14 bits for the end-of-loop address and 4 bits for the termination condition code. The loop stack is four locations deep. It is automatically pushed during the execution of a DO UNTIL instruction. It is popped automatically during a loop exit if the loop was nested. The loop stack may be manually popped with the `POP LOOP` instruction.

The status stack, which is automatically pushed when the processor services an interrupt, accommodates the interrupt mask (IMASK), mode status (MSTAT) and arithmetic status (ASTAT) registers. The depth and width of the status stack varies with each processor, since different processors have different numbers of interrupts. The status stack is automatically popped when the return from interrupt (RTI instruction) is executed. The status stack can be pushed and popped manually with the `PUSH STS` and `POP STS` instructions.

The count stack is 14 bits wide and holds counter (CNTR) values for nested counter-based loops. This stack is pushed automatically with the current CNTR value when there is a write to CNTR. The counter stack may be manually popped with the `POP CNTR` instruction.

12.1.3 Computational Units

The registers in the computational units store data.

The ALU and MAC require two inputs for most operations. The AX0, AX1, MX0 and MX1 registers store X inputs, and the AY0, AY1, MY0 and MY1 registers store Y inputs.

The AR and AF registers store ALU results; AF can be fed back to the ALU Y input, whereas AR can provide the X input of any computational unit. Likewise, the MR0, MR1, MR2 and MF register store MAC results and can be fed back for other computations. The 16-bit MR0 and MR1 registers together with the 8-bit MR2 register can store a 40-bit multipy/accumulate result.

The shifter can receive input from the ALU or MAC, from its own result registers, or from a dedicated shifter input (SI) register. It can store a 32-bit result in the SR0 and SR1 registers. The SB register stores the block exponent for block floating-point operations. The SE register holds the shift value for normalize and denormalize operations.

Registers in the computational units have secondary registers, shown in Figure 12.1 as second set of registers behind the first set. Secondary registers are useful for single-cycle context switches. The selection of these secondary registers is controlled by a bit in the MSTAT (mode status) register; the bit is set and cleared by these instructions:

```
ENA SEC_REG;        {select secondary registers}
DIS SEC_REG;        {select primary registers}
```

12.1.4 Bus Exchange

The PX register is an 8-bit register that allows data transfers between the 16-bit DMD bus and the 24-bit PMD bus. In a transfer between program memory and a 16-bit register, PX provides or receives the lower eight bits.

12.1.5 Timer

The TPERIOD, TCOUNT and TSCALE hold the timer period, count and scale factor values, respectively. These registers are memory-mapped at locations 0x3FFD, 0x3FFC, and 0x3FFB respectively.

12.1.6 Serial Ports

SPORT0 and SPORT1 each have receive (RX), transmit (TX) and control registers. The control registers are memory-mapped registers at locations 0x3FEF–0x3FFA in data memory. SPORT0 also has registers for controlling its multichannel functions. Each SPORT control register contains bits that control frame synchronization, companding, word length and, in SPORT0, multichannel options. The SCLKDIV register for each SPORT determines the frequency of the internally generated serial clock, and the RFSDIV register determines the frequency of the internally generated receive frame sync signal for each SPORT. The autobuffer registers control autobuffering in each SPORT.

Programming a SPORT consists of writing its control register and, depending on the modes selected, its SCLKDIV and/or RFSDIV registers as well. The following example code programs SPORT0 for 8-bit μ-law companding, normal framing, and an internally generated serial clock. RFSDIV is set to 255, for 256 SCLK cycles between RFS assertions. SCLKDIV is set to 2, resulting in an SCLK frequency that is 1/6 of the CLKOUT frequency.

```
SI=0xB27;
DM(0X3FF6)=SI;    {SPORT0 control register}

SI=2;
DM(0x3FF5)=SI;    {SCLKDIV = 2}

SI=255;
DM(0x3FF4)=SI;    {RFSDIV = 255}
```

12.1.7 Memory Interface & SPORT Enables

The System Control Register, memory-mapped at DM(0x3FFF), contains SPORT enables as well as the SPORT1 configuration selection. On all ADSP-21xx processors except the ADSP-2181, it also contains fields for controlling the booting operation: selecting the page, specifying the number of wait states and forcing the boot in software. The System Control Register also contains the PWAIT field which specifies the number of wait states for external program memory accesses.

The Wait State Control Register, memory-mapped at data memory location 0x3FFE, contains fields that specify the number of wait states for each bank of data memory. On the ADSP-2181, it also specifies the number of wait states for I/O memory space. In processors with optional on-chip ROM, it also contains a bit for enabling the ROM.

On the ADSP-2181, wait states are applied to external memory overlay accesses. Other memory-mapped registers control the IDMA port and byte memory DMA port for booting operations—selecting the byte memory page, specifying the number of wait states, and forcing the boot from software—and runtime access of byte memory.

12.1.8 Host Interface

The ADSP-2171, ADSP-2111, ADSP-21msp58/59 processors contain a host interface port (HIP). The host interface has six data registers, two status registers and an interrupt mask register. These registers are memory-mapped at data memory locations 0x3FE7 – 0x3FE0. The status registers contains status flags for each of the data registers. The HMASK register lets you enable or disable the generation of HIP read or HIP write interrupts independently for each HIP data register. HMASK is memory-mapped at data memory location 0x3FE8.

12.1.9 Analog Interface

The analog interface of the ADSP-21msp58/59 has four memory-mapped registers. These registers are memory-mapped in data memory locations 0x3FEC – 0x3FEF. The transmit register sends data to the DAC for transmitting. The receive register receives data from the ADC. The analog control register contains bits that select amplifier, gain, analog input and filter options.

12.2 PROGRAM EXAMPLE

Listing 12.1 presents an example of an FIR filter program written for the ADSP-2111 with discussion of each part of the program. The program can also be executed on any other ADSP-21xx processor, with minor modifications. This FIR filter program demonstrates much of the conceptual power of the ADSP-2100 family architecture and instruction set.

```
{ADSP-2111 FIR Filter Routine

   -serial port 0 used for I/O
   -internally generated serial clock
   -12.288 MHz processor clock rate is divided to 1.536 MHz serial clock
   -serial clock divided to 8 kHz frame sampling rate}

   .MODULE/RAM/ABS=0            main_routine;    {program loaded from }
                                                  {EPROM, with MMAP=0  }
A  .INCLUDE                     <const.h>;
B  .VAR/DM/RAM/ABS=0x3800/CIRC  data_buffer[taps]; {on-chip data buffer}
   .VAR/PM/RAM/CIRC             coefficient[taps];
   .GLOBAL                      data_buffer, coefficient;
   .EXTERNAL                    fir_start;
   .INIT                        coefficient:<coeff.dat>;
```

```
{code starts here}
{load interrupt vector addresses}

C               JUMP restarter; NOP; NOP; NOP;     {restart interrupt}
                RTI; NOP; NOP; NOP;                {IRQ2 interrupt}
                RTI; NOP; NOP; NOP;                {HIP write interrupt}
                RTI; NOP; NOP; NOP;                {HIP read interrupt}
                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 through buffers}
                M4=1;

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

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

{set up memory-mapped control registers}
E               AX0=191;
                DM(0x3FF4)=AX0;          {set up divide value for 8KHz RFS}
                AX0=3;
                DM(0x3FF5)=AX0;          {1.536MHz internal serial clock}
                AX0=0x69B7;
                DM(0x3FF6)=AX0;          {multichannel disabled}
                                         {internally generated 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}
                AX0=0x7000;
                DM(0x3FFE)=AX0;          {DM wait states: }
                                         {  0x3400-0x37FF 7 waits}
                                         {  all else 0 waits}
                AX0=0x1000;
                DM(0x3FFF)=AX0;          {SPORT0 enabled}
                                         {boot from boot page 0}
                                         {0 PM waits}
                                         {0 boot memory waits}
                ICNTL = 0x00;
                IMASK = 0x0018;          {enable SPORT0 interrupt only}
mainloop:       IDLE;                    {wait for interrupt}
                JUMP mainloop;
.ENDMOD;
```

Listing 12.1 Program Example Listing (Setup & Main Loop Routine)

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

Listing 12.1 (cont.) Include File, Constants Initialization

12.2.1 Example Program: Setup Routine Discussion

The setup and main loop routine performs initialization and then loops on the IDLE instruction to wait until the receive interrupt from SPORT0 occurs. The filter is interrupt-driven. When the interrupt occurs control shifts to the interrupt service routine (shown in Listing 12.2).

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

Section B of the program includes the assembler directives defining 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 is required.

Section C shows the setup of interrupts. Since this code module is located at absolute address zero (as indicated by the ABS qualifier in the .MODULE directive), the first instruction is placed at the restart vector: address 0x0000. The first location is the restart vector instruction, which jumps to the routine *restarter.* Interrupt vectors 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.

Section D, *restarter,* sets up the index (I), length (L), and modify (M) registers used to address the two circular buffers. A non-zero value for length activates the processor's modulus logic. Each time the interrupt occurs, the I register pointers advance one position through the buffers. The *clear* loop zeroes all values in the data memory buffer.

Section E, after *clear*, sets up the processor's memory-mapped control registers used in this system. See Appendix E for control register initialization information.

SPORT0 is set up to generate the serial clock internally at 1.536 MHz, based on a processor clock rate of 12.288 MHz. The RFS and TFS signals are both required and the RFS signal is generated internally at 8 kHz, while the TFS signal comes from the external device communicating with the processor.

Finally, SPORT0 is enabled and the interrupts are enabled. Now the IDLE instruction causes the processor to 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, shown in Listing 12.2.

```
.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}

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}
             MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0);      {set up multiplier for loop}
             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;
```

Listing 12.2 Interrupt Routine

12.2.2 Example Program: Interrupt Routine Discussion

This subroutine transfers the received data to the next location in the circular buffer (overwriting the oldest sample). All samples and coefficients are then 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.

The first four lines of the listing declare the code module (which is relocatable rather than placed at an absolute address), include the same file of constants, and make the entry point visible to the main routine with the .ENTRY directive. Likewise, the .EXTERNAL directive 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.

The fourth instruction of the routine, `MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0)`, zeroes the multiplier result register (MR) and fetches the first two operands. This instruction accesses both program and data memory but still executes in a single cycle because of the processor's architecture.

The *convolution* label identifies the loop itself, consisting of only two instructions, one setting up the loop (DO UNTIL) and one "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.

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

Hardware Examples 13

13.1 OVERVIEW

This chapter describes some hardware examples of circuits that can be interfaced to the ADSP-21xx serial ports, host interface port (HIP), or the memory port. As with any hardware design, it is important that timing information be carefully analyzed. Therefore, the data sheet for the particular ADSP-2100 family processor used should be used in addition to the information presented in this chapter.

13 Hardware Examples

13.2 BOOT LOADING FROM HOST USING BUS REQUEST & GRANT

All ADSP-2100 family processors that have internal program memory RAM support boot loading. With boot loading, the processor reads instructions from a byte-wide external memory device (usually an EPROM) over the memory interface and stores the instructions in the 24-bit wide internal program memory. Once the external memory device is set up to provide bytes in the proper order, the boot operation can run automatically and transparently at reset or when forced in software. See Chapter 10, "Memory Interface."

In some systems where the ADSP-21xx is controlled by a host processor, it is necessary to boot the DSP directly from the host. In this case the host, rather than an EPROM, is the source of bytes to be loaded into on-chip memory. If the ADSP-21xx has a host interface port (such as the ADSP-2111), it can perform automatic boot loading through this port. If the processor does not have a host interface port, however, it can still boot through the memory interface using the bus request signal, as described below.

This example shows a simple way to download programs from a host processor to the internal program memory of an ADSP-21xx. There are several techniques for connecting a DSP processor to a host. The choice of which technique to use depends upon the I/O structure of the host, availability of I/O port lines, and the amount of address decoding logic already available in the system.

Figure 13.1 illustrates a minimal system implementation to allow a microcontroller to boot an ADSP-21xx. The only hardware required is a D-type flip-flop and a 5 kΩ resistor. The resistor is used to pull the ADSP-21xx's $\overline{\text{BMS}}$ pin (Boot Memory Select) high.

The ADSP-21xx automatically enters its booting sequence after the processor is reset (when the MMAP pin is tied low) or when software initiates a reboot operation. When the ADSP-21xx begins to fetch a byte from external boot memory (in this case, the host processor), it asserts $\overline{\text{BMS}}$. When $\overline{\text{BMS}}$ goes low, the flip-flop is preset and the $\overline{\text{Q}}$ output brought low. This low signal asserts $\overline{\text{BR}}$ (bus request) on the ADSP-21xx. When bus request is recognized by the ADSP-21xx, the current execution cycle is allowed to finish and then processor operation is suspended. The ADSP-21xx then asserts $\overline{\text{BG}}$ (bus grant) in the next cycle (after $\overline{\text{BR}}$ is recognized).

Hardware Examples 13

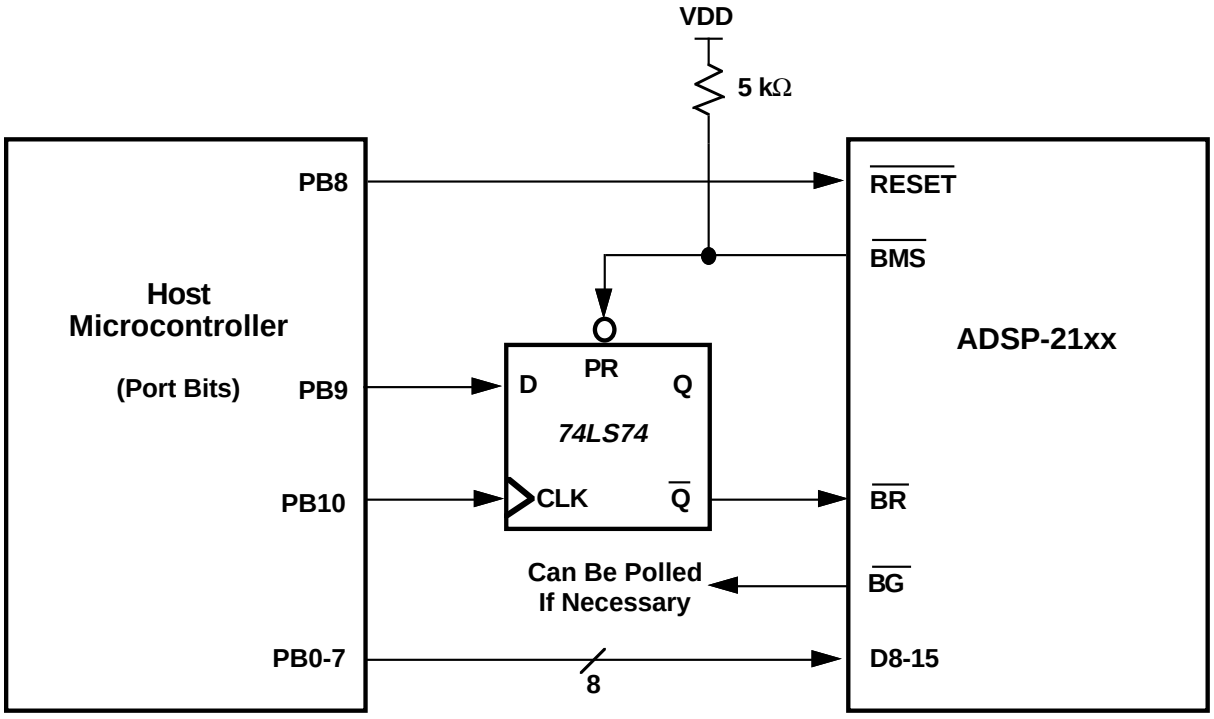


Figure 13.1 ADSP-21xx Booting From Host

13 Hardware Examples

When a low-level signal at the D input is clocked into the flip-flop, the $\overline{Q}$ output is brought high, deasserting $\overline{BR}$.

The bus request pin ($\overline{BR}$) of the ADSP-21xx is used to stop and synchronize the booting process. The host releases bus request, causing the ADSP-21xx to read one byte of boot data. During the read operation the $\overline{BMS}$ pin is asserted, which in turn causes the $\overline{BR}$ pin to be asserted and the ADSP-21xx to be put back into a bus request state. The ADSP-21xx remains suspended, waiting for the next byte of boot data.

Three programmable port bits of the microcontroller (PB 8-10) are used to provide the handshake mechanism for the transfer of each byte of boot data. Alternately, PB9 and PB10 could be implemented as a memory-mapped port location. PB8 is used to bring the ADSP-21xx out of reset, starting the boot process. Note that if PB8 is not low at power-up, the ADSP-21xx will start executing undefined instructions until PB8 is brought low.

The boot data is presented by the microcontroller either through 8 port bits (PB0-7) or through a memory-mapped port. The PB0-7 bits should be put into a high-impedance state after the boot is complete, to prevent bus contention if the ADSP-21xx tries to write to external memories or peripherals.

A typical boot sequence for this system is as follows:

1.) Bring PB8 low to reset the ADSP-21xx.

2.) Clock a high state into the flip-flop with PB9 and PB10 to bring $\overline{BR}$ low.

3.) Bring PB8 high to bring the ADSP-21xx out of reset.

4.) Place a byte of boot data on the data bus (PB0-7.).

5.) Clock a low state into the flip-flop with PB9 and PB10 to bring $\overline{BR}$ high.

6.) Wait a minimum of six processor cycles while the ADSP-21xx fetches the data byte and the flip-flop asserts $\overline{BR}$.

7.) Repeat steps 4, 5, and 6 for each byte of boot data. After the last iteration, the ADSP-21xx will automatically start execution.

Note: The proper loading sequence for boot data must be followed (i.e. the order in which the host passes bytes to the ADSP-21xx). This sequence is described in the Chapter 10, "Memory Interface." To create a file for booting, use the PROM Splitter utility of the ADSP-2100 Family Development Software. The PROM Splitter automatically organizes the bytes in the proper order for booting.

13.3 SERIAL PORT TO CODEC INTERFACE

A codec (COder/DECoder) incorporates analog-to-digital conversion, digital-to-analog conversion, and filtering in one device. The codec shown in this example also performs pulse-code modulation (PCM) encoding and decoding according to the CCITT μ-law standard. PCM compresses digital data so that fewer bits are needed to store the same information. The ADSP-21xx serial ports have both μ-law and A-law companding (compressing/expanding) capability.

In the example described here, a codec converts its analog input to digital data, compresses it and sends it serially to the SPORT on an ADSP-21xx processor. At the same time, the processor sends compressed serial data via the SPORT to the codec, which expands the data and converts the result to an analog signal.

13 Hardware Examples

Figure 13.2 shows an industry standard μ-law companding codec connected to a serial port (in this case, SPORT0) on an ADSP-21xx processor. The codec's analog input at VFXI+ is internally amplified by a gain which is controlled by the resistor combination at GSX and VFXI–. The gain is

$$20 \times \log (R1 + R2)/R2$$

in this case, 20 log 2.

The ADSP-21xx controls codec operation by supplying master and bit clock signals. In the configuration shown, the codec transmit and receive sections operate synchronously. MCLKR and MCLKX are the master clocks for the receive and transmit sections of the codec, respectively. BCLKX is the bit clock and in this configuration is used for clocking both received and transmitted serial data. MCLKR, MCLKX and BCLKX must be synchronous and in this case they are the same signal, namely the SCLK0 output generated by the ADSP-21xx processor. The BCLKR/CLKSEL input, tied low, selects the frequency of MCLKX to be 2.048 MHz. The ADSP-21xx must be programmed for internal SCLK0 generation at 2.048 MHz.

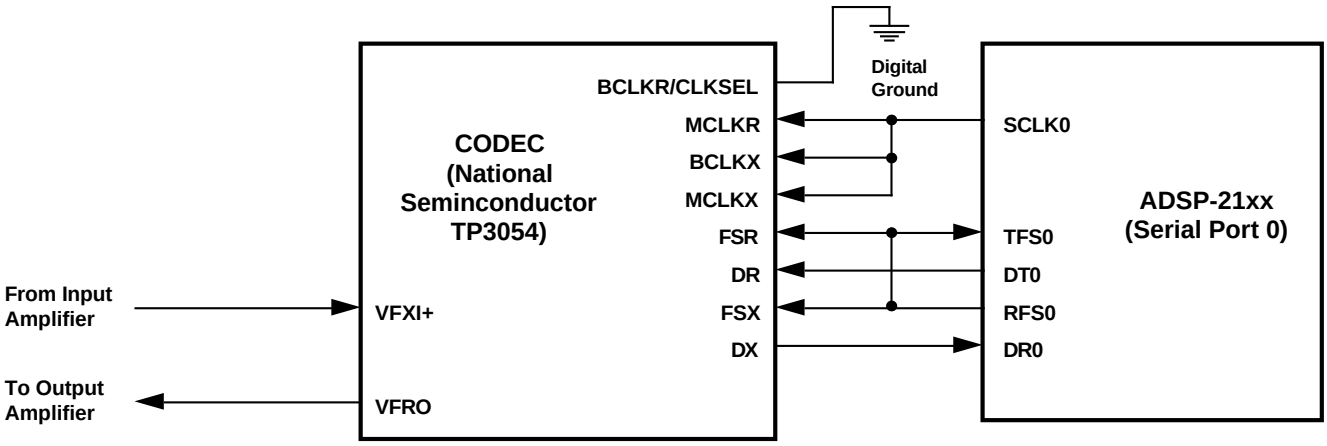


Figure 13.2 ADSP-21xx Serial Port (SPORT0) To CODEC

The processor uses frame synchronization signals to tell the codec to send and receive data. To transmit data to the codec, it sends a TFS0 pulse to the FSR input of the codec and then outputs the eight bits on DT0 on the next eight serial clock periods. The codec receives the data on its DR input. Likewise, the processor initiates a data receive operation by sending an RFS0 pulse to the codec's FSX input, which causes the codec to output eight bits on its DX output on the next eight serial clock periods. The processor receives the data on its DR0 input. The ADSP-21xx must be programmed to use normal framing, 8-bit data words, and internal, active-high frame sync generation.

The ADSP-21xx code shown in Listing 13.1 configures SPORT0 for operation as required in this example:

- Internally generated serial clock
- 2.048 MHz serial clock frequency
- Both transmit and receive frame syncs required
- Use normal framing for both transmit and receive
- Internally generated transmit and receive frame syncs
- Both frame syncs active high
- Word length of eight bits
- μ-law companding

This code assumes the processor operating at 12.288 MHz. The code also sets up the processor to request data from the codec at an 8 kHz rate (this register is not initialized at reset and should always be written before the SPORT is enabled if RFS is generated internally). The processor transmits data as needed by the program it is executing.

```
AX0=0x6927;       {Int SCLK, RFS/TFS req, norm framing,}
DM(0x3FF6)=AX0;   {generate RFS, active HI, Mu-law, word length 8}

AX0=2;            {value of SCLKDIV for 2.048 MHz}
DM(0x3FF5)=AX0;   {with a 12.888 MHz CLKOUT}

AX0=255;          {RFSDIV=256, 256 SCLKs between}
DM(0x3FF4)=AX0;   {frame syncs, 8 kHz framing}

AX0=0x1038;       {enable SPORT0 only, leave defaults}
DM(0x3FFF)=AX0;
```

Listing 13.1 Serial Port Initialization Example

13 Hardware Examples

13.4 SERIAL PORT TO DAC INTERFACE

Any DSP process must ultimately output analog information. The serial port of the ADSP-21xx processors can send data directly to a DAC (digital-to-analog converter) for conversion to an analog signal.

Analog Devices' AD766 is a DAC that requires no extra logic to interface to the SPORT. The AD766 receives 16-bit data words serially, MSB first, which it then converts to an analog signal. Its digital interface consists of three inputs: DATA, the serial data input; $\overline{\text{CLK}}$, for clocking data into the DAC (active low because data is clocked on the falling edge) and LE (latch enable), which latches each 16-bit word into the conversion section of the DAC.

The serial port connection to the AD766 is shown in Figure 13.3. In this configuration, the processor generates SCLK internally and provides it to the DAC. Serial data is output from the DT pin to the DATA input of the DAC. The TFS signal provides the DAC's LE input.

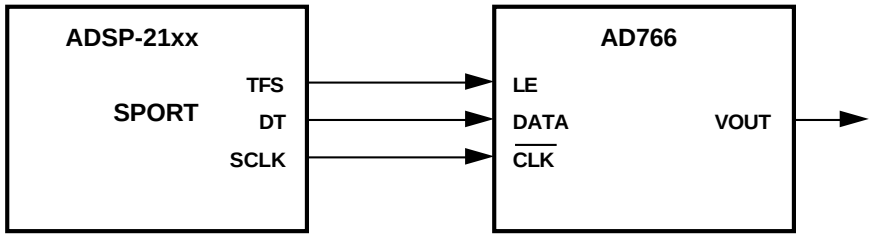


Figure 13.3 Serial Port Interface To AD766 DAC

LE should go low on the clock cycle after the LSB (sixteenth bit) of a word is transmitted, to latch the 16-bit word into the DAC. To provide this timing, TFS is configured for the alternate framing mode, non-inverted; it goes high when the first bit is transmitted and low after the last bit is transmitted. This low-going edge latches the word into the AD766. The only restriction is that the SPORT cannot transmit continuously; there must be a break in between the last bit of one word and the first bit of the next so that TFS can go low. Figure 13.4 shows the timing.

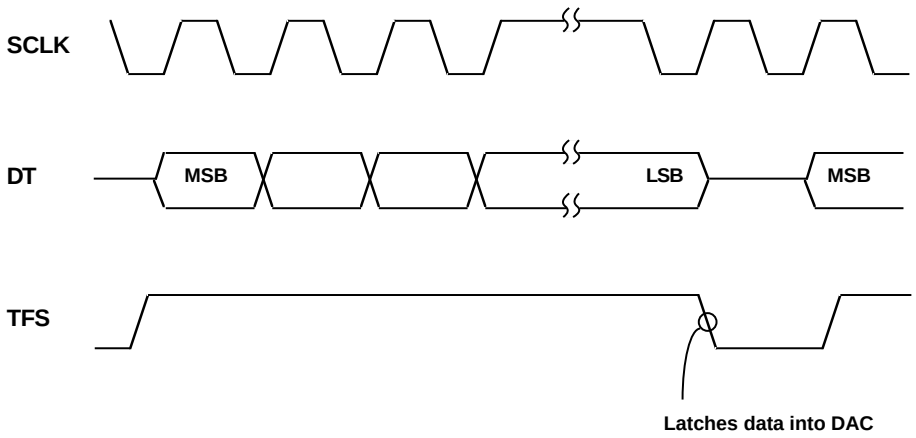


Figure 13.4 SPORT To AD766 DAC Timing

The configuration of the SPORT control registers for this application is shown in Figure 13.5.

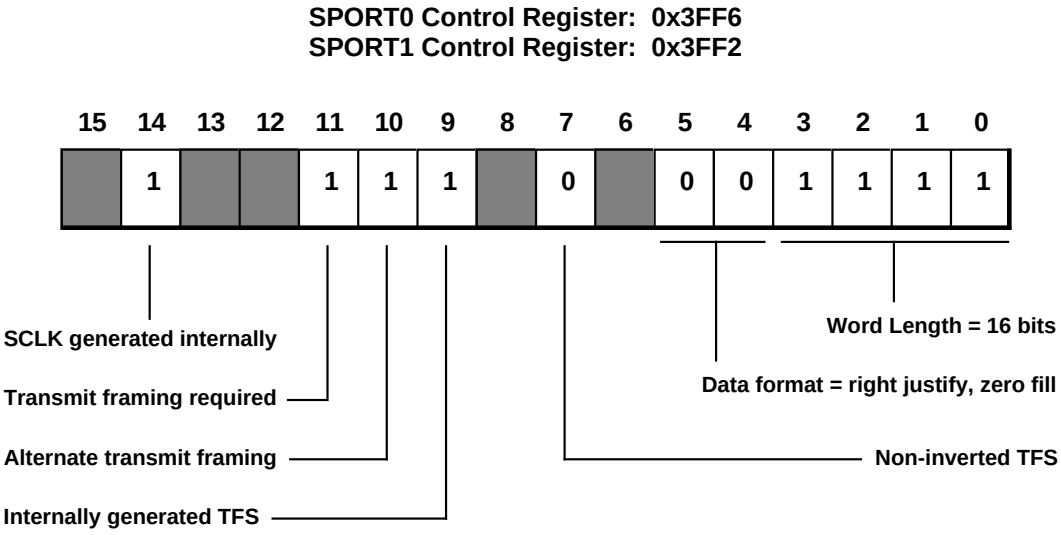


Figure 13.5 SPORT To AD766 DAC Control Register Settings

13 Hardware Examples

13.5 SERIAL PORT TO ADC INTERFACE

An ADC (analog-to-digital converter) converts an analog signal to digital samples that a DSP processor can operate on. The ADSP-21xx processors can receive data from an ADC directly through a serial port.

Analog Devices' AD7872 is an ADC that requires no extra logic to interface to the SPORT. The AD7872 converts an analog signal to 14-bit samples. Each sample is padded with two zero MSBs to yield 16-bit samples. The AD7872 outputs each sample serially, MSB first. Its digital interface consists of three pins: SDATA, the serial data output; SCLK, for clocking data out; and $\overline{\text{SSTRB}}$, (serial strobe), which frames each serial word.

The serial port connection to the AD7872 is shown in Figure 13.6. The timer regulates sampling via the $\overline{\text{CONVST}}$ input at a constant frequency. Instead of the timer, an unused serial clock or flag output from the ADSP-21xx processor can be programmed to generate the $\overline{\text{CONVST}}$ signal. The AD7872 generates SCLK internally and provides it to the processor. With the CONTROL input held at –5 V, the SCLK signal is continuous, running even when no data is being output.

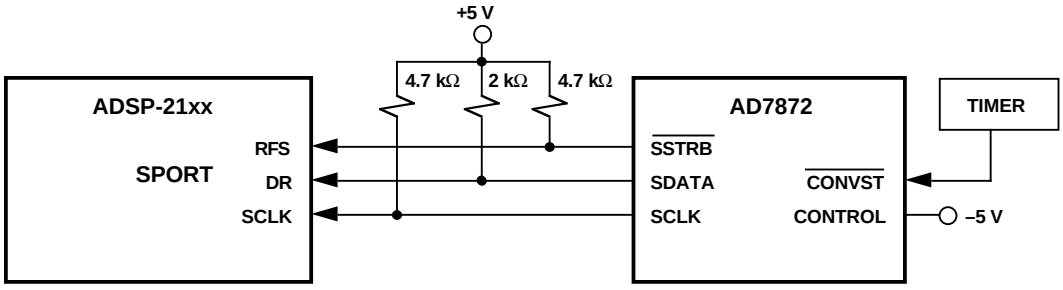


Figure 13.6 Serial Port Interface To AD7872 ADC

Serial data is output from the SDATA output of the ADC to the processor's DR pin. The $\overline{\text{SSTRB}}$ signal provides the RFS input to the processor. $\overline{\text{SSTRB}}$ goes low when the first bit is transmitted to the processor. Figure 13.7 shows the timing of the serial data transfer.

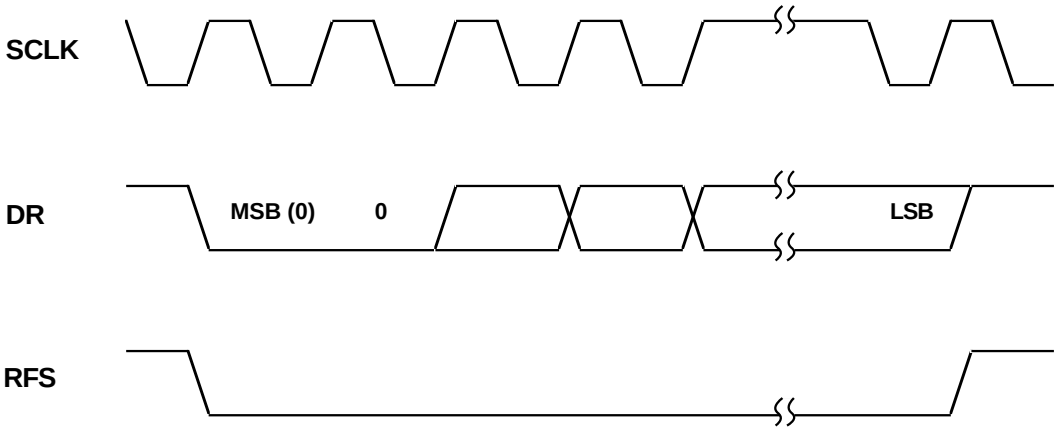


Figure 13.7 SPORT To AD7872 ADC Timing

RFS is configured for the alternate framing mode, externally generated, with inverted (active low) logic. The SPORT must also be programmed for external serial clock and a serial word length of 16 bits. The configuration of the SPORT control register for this application is shown in Figure 13.8.

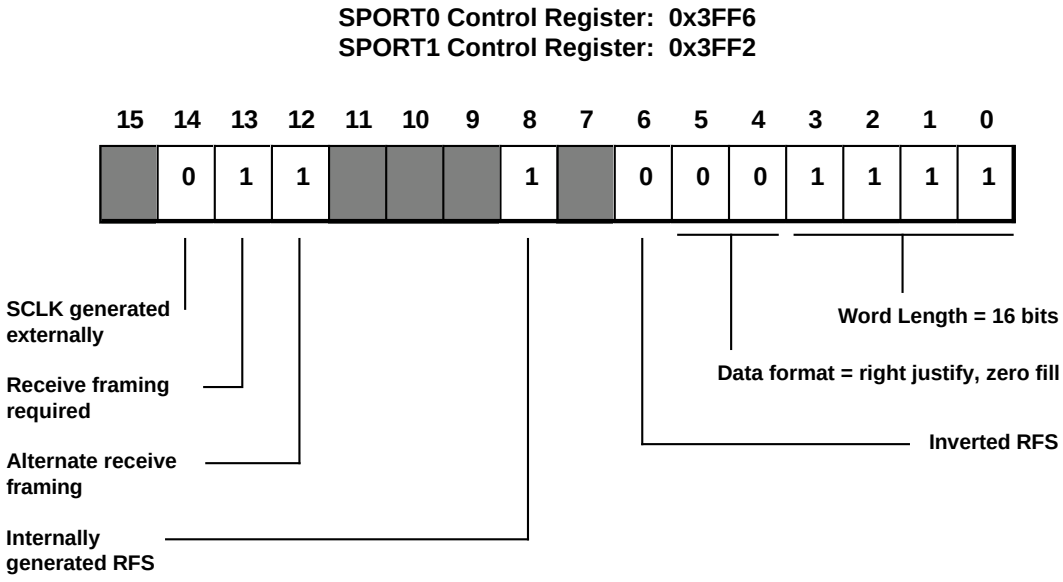


Figure 13.8 SPORT To AD7872 ADC Control Register Settings

13 Hardware Examples

13.6 SERIAL PORT TO SERIAL PORT INTERFACE

The serial ports provide a convenient way to transfer data between ADSP-21xx processors without using external memory or the memory bus and without halting either processor. The serial ports are connected as shown in Figure 13.9—in this example, SPORT1 of processor #1 is connected to SPORT0 of processor #2.

The serial clock used by both processors is generated internally by processor #1. Processor #2 is configured to receive its serial clock externally. The serial port control registers should be set up with the following parameters.

Processor 1, SPORT1

SCLKDIV = system-dependent
SLEN = system-dependent
ISCLK = 1
TFSR = 1
RFSR = 1
IRFS = 0
ITFS = 1
RFSDIV = don't care

Processor 2, SPORT0

SCLKDIV = system-dependent
SLEN = system-dependent
ISCLK = 0
TFSR = 1
RFSR = 1
IRFS = 0
ITFS = 1
RFSDIV = don't care

TFSW1 = RFSW1 = TFSW2 = RFSW2 = system-dependent
INVRFS1 = INVTFS1 = INVRFS2 = INVTFS2 = system-dependent

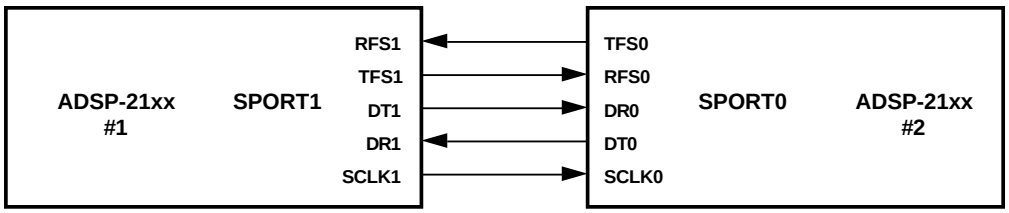


Figure 13.9 Serial Port Interface Between Two ADSP-21xx Processors

Frame synchronization is used to coordinate the transfer of serial data. Each processor generates a transmit frame sync (TFS) signal internally and expects to receive its receive frame sync (RFS) signal externally, from the other processor. The framing mode can be normal or alternate, but must be the same for both SPORTs. Likewise, the SPORTs must be configured for the same serial word length and companding type, if companding is used, or data format if companding is not used.

The autobuffering capability of the serial ports can be used in this configuration to transfer an entire buffer of data from the data memory space of one processor to the other's, *without interrupt overhead*. The serial ports handshake automatically—when one processor writes its' TX0 register, the data is automatically transmitted to the other processor's RX0 register and an autobuffer cycle is generated.

In fact, autobuffer transfers can occur in both directions at the same time, in the background, while each processor is executing some other primary function. Each SPORT will generate an interrupt when the autobuffer transfer is complete. The description of autobuffering in the Serial Port chapter shows an example of the code for setting up autobuffering.

13.7 80C51 INTERFACE TO HOST INTERFACE PORT

The host interface port (HIP) on the ADSP-2111, ADSP-2171, and ADSP-21msp5x processors facilitates communication with a host microcomputer such as the Intel 80C51. An example connection is shown in Figure 13.10. In this example, the HIP data registers (HDRs) and HIP status registers (HSRs) of the ADSP-2111 occupy eight contiguous locations in the memory space of the 80C51.

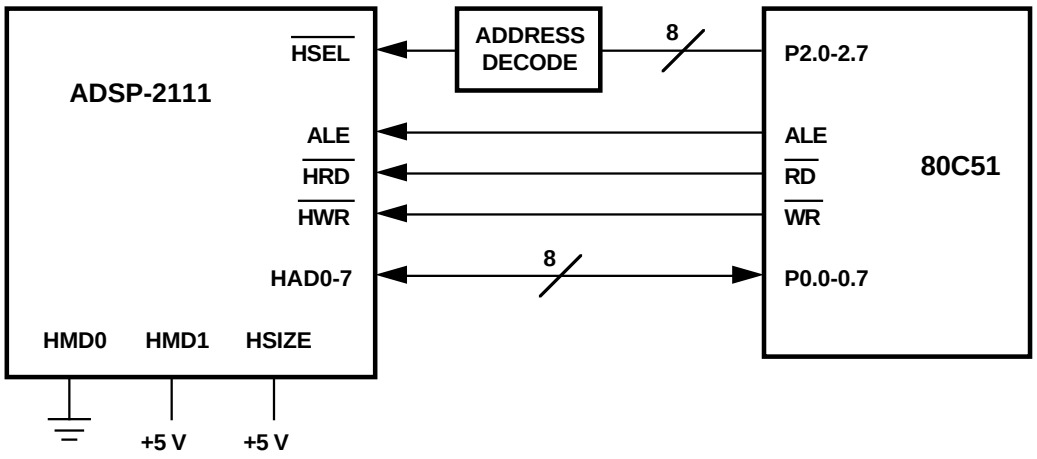


Figure 13.10 Host Port Interface to 80C51 Microcomputer

13 Hardware Examples

To access one of the HIP registers, the 80C51 asserts ALE and outputs a 16-bit address, with the upper half on P2.0-2.7 and the lower half on P0.0-0.7. The upper half is decoded to select the HIP via $\overline{\text{HSEL}}$, and the lower half selects the HIP register via HAD0-7. The ALE assertion causes the HIP to latch the address so that the 8-bit data can then be transferred on the HAD0-7 lines. The 80C51 asserts $\overline{\text{WR}}$ for a write or $\overline{\text{RD}}$ for a read.

In this example, the 80C51 reads and writes 8-bit data, so the ADSP-2111's HSIZE input is tied high. Only the lower eight bits of each HIP register are used. HMD0 is tied low because the 80C51 uses separate read and write strobes rather than a single Read/$\overline{\text{Write}}$ line. HMD1 is tied high because the address and data use the same bus (time-multiplexed using ALE) rather than separate buses.

Software Examples 14

14.1 OVERVIEW

This chapter provides a brief summary of the development process that you use to create executable programs for the ADSP-2100 family processors. The summary is followed by a number of software examples that can give you an idea of how to write your own applications.

The software examples presented in this chapter are used a variety of DSP operations. The FIR filter and cascaded biquad IIR filter are general filter algorithms that can be tailored to many applications. Matrix multiplication is used in image processing and other areas requiring vector operations. The sine function is required for many scientific calculations. The FFT (fast Fourier transform) has wide application in signal analysis. Each of these examples is described in greater detail in *Digital Signal Processing Applications Using The ADSP-2100 Family, Volume 1*, available from Prentice Hall. They are presented here to show some aspects of typical programs.

The FFT example is a complete program, showing a subroutine that performs the FFT and a main calling program that initializes registers and calls the FFT subroutine as well as an auxiliary routine.

Each of the other examples is shown as a subroutine in its own module. The module starts with a .MODULE directive that names the module and ends with the .ENDMOD directive. The subroutine can be called from a program in another module that declares the starting label of the subroutine as an external symbol. This is the same label that is declared with the .ENTRY directive in the subroutine module. The last instruction in each subroutine is the RTS instruction, which returns control to the calling program.

14 Software Examples

Each module is prefaced by a comment block that provides the following information:

Calling Parameters	Register values that the calling program must set before calling the subroutine
Return Values	Registers that hold the results of the subroutine
Altered Registers	Registers used by the subroutine. The calling program must save them before calling the subroutine and restore them afterward if it needs to preserve their values.
Computation Time	The number of instruction cycles needed to perform the subroutine

14.2 SYSTEM DEVELOPMENT PROCESS

The ADSP-2100 family of processors is supported by a complete set of development tools. Programming aids and processor simulators facilitate software design and debug. In-circuit emulators and demonstration boards help in hardware prototyping.

The software development system includes several programs: System Builder, Assembler, Linker, PROM Splitter, Simulators and C Compiler with Runtime Library. These programs are described in detail in the *ADSP-2100 Family Assembler Tools & Simulator Manual*, *ADSP-2100 Family C Tools Manual*, and *ADSP-2100 Family C Runtime Library Manual*.

Figure 14.1 shows a flow chart of the system development process.

The development process begins with the task of describing the hardware environment for the development software. You create a system specification file using a text editor. This file contains simple directives that describe the locations of memory and I/O ports, the type of processor, and the state of the MMAP pin in the target hardware configuration. The system builder reads this file and generates an architecture description file which passes information to the linker, simulator and emulator.

You begin code generation by creating source code files in C language or

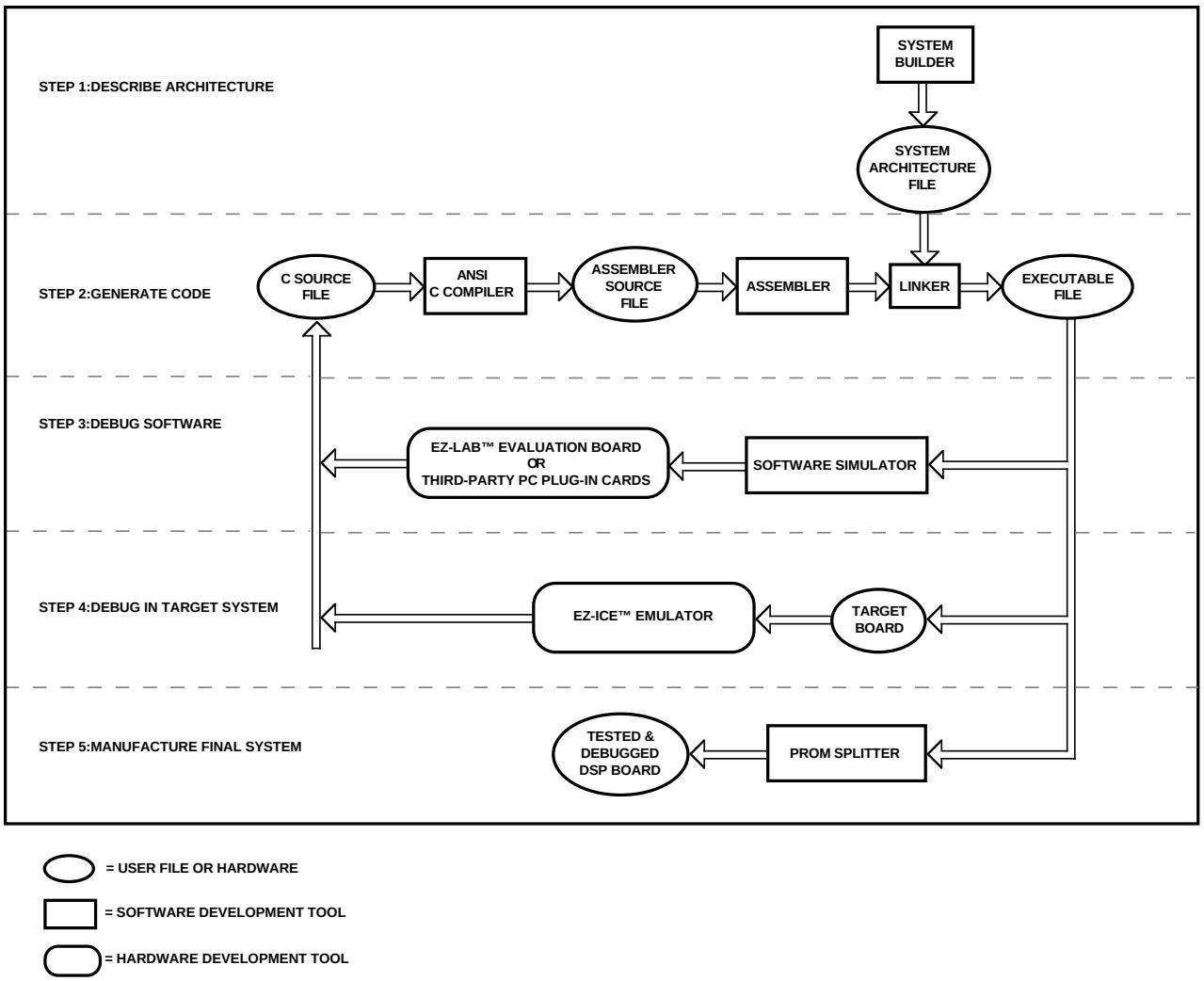


Figure 14.1 ADSP-2100 Family System Development Process

assembly language. A module is a unit of assembly language comprising a main program, subroutine, or data variable declarations. C programmers write C language files and use the C compiler to create assembly code modules from them. Assembly language programmers write assembly code modules directly. Each code module is assembled separately by the assembler.

The linker links several modules together to form an executable program (memory image file). The linker reads the target hardware information from the architecture description file to determine appropriate addresses for code and data. In the assembly modules you may specify each code/data fragment as completely relocatable, relocatable within a defined memory segment, or non-

relocatable (placed at an absolute address).

The linker places non-relocatable code or data modules at the specified memory addresses, provided the memory area has the correct attributes. Relocatable objects are placed at addresses selected by the linker. The linker generates a memory image file containing a single executable program which may be loaded into a simulator or emulator for testing.

The simulator provides windows that display different portions of the hardware environment. To replicate the target hardware, the simulator configures its memory according to the architecture description file generated by the system builder, and simulates memory-mapped I/O ports. This simulation allows you to debug the system and analyze performance before committing to a hardware prototype.

After fully simulating your system and software, you can use an EZ-ICE in-circuit emulator in the prototype hardware to test circuitry, timing, and real-time software execution.

The PROM splitter software tool translates the linker-output program (memory image file) into an industry-standard file format for a PROM programmer. Once you program the code in PROM devices and install an ADSP-21xx processor into your prototype, it is ready to run.

14.3 SINGLE-PRECISION FIR TRANSVERSAL FILTER

An FIR transversal filter structure can be obtained directly from the equation for discrete-time convolution.

$$y(n) = \sum_{k=0}^{N-1} h_k(n)\, x(n-k)$$

In this equation, x(n) and y(n) represent the input to and output from the filter at time *n*. The output y(n) is formed as a weighted linear combination of the current and past input values of x, x(n–k). The weights, $h_k(n)$, are the transversal filter coefficients at time *n*. In the equation, x(n–k) represents the past value of the input signal "contained" in the (k+1)th tap of the transversal filter. For example, x(n), the present value of the input signal, would correspond to the first tap, while x(n–42) would

correspond to the forty-third filter tap.

The subroutine that realizes the sum-of-products operation used in computing the transversal filter is shown in Listing 14.1.

```
.MODULE fir_sub;

{
   FIR Transversal Filter Subroutine

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

   Return Values
        MR1 = Sum of products (rounded and saturated)
        I0 –> Oldest input data value in delay line
        I4 –> Beginning of filter coefficient table

   Altered Registers
        MX0,MY0,MR

   Computation Time
        N - 1 + 5 + 2 cycles

   All coefficients and data values are assumed to be
   in 1.15 format.
}

.ENTRY  fir;

fir:    MR=0, MX0=DM(I0,M1), MY0=PM(I4,M5);
        DO sop UNTIL CE;
sop:       MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M5);
        MR=MR+MX0*MY0(RND);
        IF MV SAT MR;
        RTS;
.ENDMOD;
```

Listing 14.1 Single-Precision FIR Transversal Filter

14 Software Examples

14.4 CASCADED BIQUAD IIR FILTER

A second-order biquad IIR filter section is represented by the transfer function (in the z-domain):

$$H(z) = Y(z)/X(z) = (B_0 + B_1z^{-1} + B_2z^{-2})/(1 + A_1z^{-1} + A_2z^{-2})$$

where A_1, A_2, B_0, B_1 and B_2 are coefficients that determine the desired impulse response of the system H(z). The corresponding difference equation for a biquad section is:

$$Y(n) = B_0X(n) + B_1X(n–1) + B_2X(n–2) – A_1Y(n–1) – A_2Y(n–2)$$

Higher-order filters can be obtained by cascading several biquad sections with appropriate coefficients. The biquad sections can be scaled separately and then cascaded in order to minimize the coefficient quantization and the recursive accumulation errors.

A subroutine that implements a high-order filter is shown in Listing 14.2. A circular buffer in program memory contains the scaled biquad coefficients. These coefficients are stored in the order: B_2, B_1, B_0, A_2 and A_1 for each biquad. The individual biquad coefficient groups must be stored in the order that the biquads are cascaded.

```
.MODULE     biquad_sub;

{     Nth order cascaded biquad filter subroutine

      Calling Parameters:

         SR1=input X(n)
         I0 –> delay line buffer for X(n-2), X(n-1),
            Y(n-2), Y(n-1)
         L0 = 0
         I1 –> scaling factors for each biquad section
         L1 = 0  (in the case of a single biquad)
         L1 = number of biquad sections
            (for multiple biquads)
         I4 –> scaled biquad coefficients
         L4 = 5 x [number of biquads]
         M0, M4 = 1
         M1 = -3
         M2 = 1 (in the case of multiple biquads)
         M2 = 0 (in the case of a single biquad)
```

```
         M3 = (1 - length of delay line buffer)

      Return Value:
         SR1 = output sample Y(n)

      Altered Registers:
         SE, MX0, MX1, MY0, MR, SR

      Computation Time (with N even):
         ADSP-2101/2102: (8 x N/2) + 5 cycles
         ADSP-2100/2100A: (8 x N/2) + 5 + 5 cycles

      All coefficients and data values are assumed to
      be in 1.15 format
}

.ENTRY      biquad;

biquad:     CNTR = number_of_biquads
            DO sections UNTIL CE;   {Loop once for each biquad}
               SE=DM(I1,M2);        {Scale factor for biquad}
               MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MX0*MY0(SS), MX1=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX1*MY0(SS), MY0=PM(I4,M4);
               MR=MR+SR1*MY0(SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
               MR=MR+MX0*MY0(SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
               DM(I0,M0)=MX1, MR=MR+MX0*MY0(RND);
sections:      DM(I0,M0)=SR1, SR=ASHIFT MR1 (HI);
            DM(I0,M0)=MX0;
            DM(I0,M3)=SR1;
            RTS;
.ENDMOD;
```

Listing 14.2 Cascaded Biquad IIR Filter

14.5 SINE APPROXIMATION

The following formula approximates the sine of the input variable x:

$$\sin(x) = 3.140625x + 0.02026367x^2 - 5.325196x^3 + 0.5446778x^4 + 1.800293x^5$$

The approximation is accurate for any value of x from 0° to 90° (the first quadrant). However, because sin(–x) = –sin(x) and sin(x) = sin(180° – x), you can infer the sine of any angle from the sine of an angle in the first

quadrant.

The routine that implements this sine approximation, accurate to within two LSBs, is shown in Listing 14.3. This routine accepts input values in 1.15 format. The coefficients, which are initialized in data memory in 4.12 format, have been adjusted to reflect an input value scaled to the maximum range allowed by this format. On this scale, 180˚ equals the maximum positive value, 0x7FFF, and –180˚ equals the maximum negative value, 0x8000.

The routine shown in Listing 14.3 first adjusts the input angle to its equivalent in the first quadrant. The sine of the modified angle is calculated by multiplying increasing powers of the angle by the appropriate coefficients. The result is adjusted if necessary to compensate for the modifications made to the original input value.

```
.MODULE  Sin_Approximation;

{
   Sine Approximation
         Y = Sin(x)

   Calling Parameters
         AX0 = x in scaled 1.15 format
         M3 = 1
         L3 = 0

   Return Values
         AR = y in 1.15 format

   Altered Registers
         AY0,AF,AR,MY1,MX1,MF,MR,SR,I3

   Computation Time
         25 cycles
```

```
}

.VAR/DM  sin_coeff[5];

.INIT    sin_coeff : 0x3240, 0x0053, 0xAACC, 0x08B7, 0x1CCE;

.ENTRY   sin;

sin:     I3=^sin_coeff;                {Pointer to coeff. buffer}
         AY0=0x4000;
         AR=AX0, AF=AX0 AND AY0;       {Check 2nd or 4th quad.}
         IF NE AR=-AX0;                {If yes, negate input}
         AY0=0x7FFF;
         AR=AR AND AY0;                {Remove sign bit}
         MY1=AR;
         MF=AR*MY1 (RND), MX1=DM(I3,M3);  {MF = x^2}
         MR=MX1*MY1 (SS), MX1=DM(I3,M3);  {MR = C1x}
         CNTR=3;
         DO approx UNTIL CE;
            MR=MR+MX1*MF (SS);
approx:     MF=AR*MF (RND), MX1=DM(I3,M3);
         MR=MR+MX1*MF (SS);
         SR=ASHIFT MR1 BY 3 (HI);
         SR=SR OR LSHIFT MR0 BY 3 (LO);   {Convert to 1.15 format}
         AR=PASS SR1;
         IF LT AR=PASS AY0;            {Saturate if needed}
         AF=PASS AX0;
         IF LT AR=-AR;                 {Negate output if needed}
         RTS;
.ENDMOD;
```

Listing 14.3 Sine Approximation

14.6 SINGLE-PRECISION MATRIX MULTIPLY

The routine presented in this section multiplies two input matrices: X, an RxS (R rows, S columns) matrix stored in data memory and Y, an SxT (S rows, T columns) matrix stored in program memory. The output Z, an RxT (R rows, T columns) matrix, is written to data memory.

The routine is shown in Listing 14.4. It requires a number of registers to be initialized, as listed in the "Calling Parameters" section of the initial comment. SE must contain the value necessary to shift the result of each multiplication into the desired format. For example, SE would be set to zero to obtain a matrix of 1.31 values from the multiplication of two matrices of 1.15 values.

14 Software Examples

```
.MODULE     matmul;

{
            Single-Precision Matrix Multiplication

                        S
               Z(i,j) = Σ [X(i,k) × Y(k,j)]   i=0 to R; j=0 to T
                        k=0

               X is an RxS matrix
               Y is an SxT matrix
               Z is an RxT matrix

            Calling Parameters
               I1 –> Z buffer in data memory                L1 = 0
               I2 –> X, stored by rows in data memory       L2 = 0
               I6 –> Y, stored by rows in program memory    L6 = 0
               M0 = 1         M1 = S
               M4 = 1         M5 = T
               L0,L4,L5 = 0
               SE = Appropriate scale value
               CNTR = R

            Return Values
               Z Buffer filled by rows

            Altered Registers
               I0,I1,I2,I4,I5,MR,MX0,MY0,SR

            Computation Time
               ((S + 8) × T + 4) × R + 2 + 2 cycles
```

```
}

.ENTRY      spmm;

spmm:    DO row_loop UNTIL CE;
            I5=I6;                        {I5 = start of Y}
            CNTR=M5;
            DO column_loop UNTIL CE;
               I0=I2;                     {Set I0 to current X row}
               I4=I5;                     {Set I4 to current Y col}
               CNTR=M1;
               MR=0, MX0=DM(I0,M0), MY0=PM(I4,M5); {Get 1st data}
               DO element_loop UNTIL CE;
element_loop:     MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0),
MY0=PM(I4,M5);
               SR=ASHIFT MR1 (HI), MY0=DM(I5,M4);  {Update I5}
               SR=SR OR LSHIFT MR0 (LO);           {Finish shift}
column_loop:   DM(I1,M0)=SR1;                      {Save output}
row_loop:   MODIFY(I2,M1);              {Update I2 to next X row}
         RTS;
.ENDMOD;
```

Listing 14.4 Single-Precision Matrix Multiply

14.7 RADIX-2 DECIMATION-IN-TIME FFT

The FFT program includes three subroutines. The first subroutine scrambles the input data (places the data in bit-reversed address order), so that the FFT output will be in the normal, sequential order. The next subroutine computes the FFT and the third scales the output data to maintain the block floating-point data format.

The program is contained in four modules. The main module declares and initializes data buffers and calls subroutines. The other three modules contain the FFT, bit reversal, and block floating-point scaling subroutines. The main module calls the FFT and bit reversal subroutines. The FFT module calls the data scaling subroutine.

The FFT is performed in place; that is, the outputs are written to the same buffer that the inputs are read from.

14.7.1 Main Module

The dit_fft_main module is shown in Listing 14.5. N is the number of points in the FFT (in this example, N=1024) and N_div_2 is used for specifying the lengths of buffers. To change the number of points in the

14 Software Examples

FFT, you change the value of these constants and the twiddle factors.

The data buffers twid_real and twid_imag in program memory hold the twiddle factor cosine and sine values. The inplacereal, inplaceimag, inputreal and inputimag buffers in data memory store real and imaginary data values. Sequentially ordered input data is stored in inputreal and inputimag. This data is scrambled and written to inplacereal and inplaceimag. A four-location buffer called padding is placed at the end of inplaceimag to allow data accesses to exceed the buffer length. This buffer assists in debugging but is not necessary in a real system. Variables (one-location buffers) named groups, bflys_per_group, node_space and blk_exponent are declared last.

The real parts (cosine values) of the twiddle factors are stored in the buffer twid_real. This buffer is initialized from the file twid_real.dat. Likewise, twid_imag.dat values initialize the twid_imag buffer that stores the sine values of the twiddle factors. In an actual system, the hardware would be set up to initialize these memory locations.

The variable called groups is initialized to N_div_2, and bflys_per_group and node_space are each initialized to 2 because there are two butterflies per group in the second stage of the FFT. The blk_exponent variable is initialized to zero. This exponent value is updated when the output data is scaled.

After the initializations are complete, two subroutines are called. The first subroutine places the input sequence in bit-reversed order. The second performs the FFT and calls the block floating-point scaling routine.

```
.MODULE/ABS=4           dit_fft_main;
.CONST                  N=1024, N_div_2=512; {For 1024 points}
.VAR/PM/RAM/CIRC        twid_real [N_div_2];
.VAR/PM/RAM/CIRC        twid_imag [N_div_2];
.VAR/DM/RAM/ABS=0       inplacereal [N], inplaceimag [N], padding
[4];
.VAR/DM/RAM/ABS=H#1000  inputreal [N], inputimag [N];
.VAR/DM/RAM             groups, bflys_per_group, node_space,
                        blk_exponent;

.INIT       twid_real: <twid_real.dat>;
.INIT       twid_imag: <twid_imag.dat>;
.INIT       inputreal: <inputreal.dat>;
.INIT       inputimag: <inputimag.dat>;
```

```
.INIT       inplaceimag: <inputimag.dat>;
.INIT       groups: N_div_2;
.INIT       bflys_per_group: 2;
.INIT       node_space: 2;
.INIT       blk_exponent: 0;
.INIT       padding: 0,0,0,0;          {Zeros after inplaceimag}

.GLOBAL     twid_real, twid_imag;
.GLOBAL     inplacereal, inplaceimag;
.GLOBAL     inputreal, inputimag;
.GLOBAL     groups, bflys_per_group, node_space, blk_exponent;


.EXTERNAL   scramble, fft_strt;

            CALL scramble;             {subroutine calls}
            CALL fft_strt;
            TRAP;                      {halt program}
.ENDMOD;
```

Listing 14.5 Main Module, Radix-2 DIT FFT

14.7.2 DIT FFT Subroutine

The radix-2 DIT FFT routine is shown in Listing 14.6. The constants N and log2N are the number of points and the number of stages in the FFT, respectively. To change the number of points in the FFT, you modify these constants.

The first and last stages of the FFT are performed outside of the loop that executes all the other stages. Treating the first and last stages individually allows them to be executed faster. In the first stage, there is only one butterfly per group, so the butterfly loop is unnecessary, and the twiddle factors are all either 1 or 0, so no multiplications are necessary. In the last stage, there is only one group, so the group loop is unnecessary, as are the

setup operations for the next stage.

```
{1024 point DIT radix 2 FFT}
{Block Floating Point Scaling}

.MODULE     fft;

{     Calling Parameters
         inplacereal=real input data in scrambled order
         inplaceimag=all zeroes (real input assumed)
         twid_real=twiddle factor cosine values
         twid_imag=twiddle factor sine values
         groups=N/2
         bflys_per_group=1
         node_space=1

      Return Values
         inplacereal=real FFT results, sequential order
         inplaceimag=imag. FFT results, sequential order

      Altered Registers
         I0,I1,I2,I3,I4,I5,L0,L1,L2,L3,L4,L5
         M0,M1,M2,M3,M4,M5
         AX0,AX1,AY0,AY1,AR,AF
         MX0,MX1,MY0,MY1,MR,SB,SE,SR,SI

      Altered Memory
         inplacereal, inplaceimag, groups, node_space,
         bflys_per_group, blk_exponent
}

.CONST      log2N=10, N=1024, nover2=512, nover4=256;

.EXTERNAL   twid_real, twid_imag;
.EXTERNAL   inplacereal, inplaceimag;
.EXTERNAL   groups, bflys_per_group, node_space;
.EXTERNAL   bfp_adj;
.ENTRY      fft_strt;

fft_strt:   CNTR=log2N - 2;   {Initialize stage counter}
            M0=0;
            M1=1;
            L1=0;
            L2=0;
            L3=0;
            L4=%twid_real;
            L5=%twid_imag;
            L6=0;
```

```
            SB=-2;

{─────── STAGE 1 ──────}

            I0=^inplacereal;
            I1=^inplacereal + 1;
            I2=^inplaceimag;
            I3=^inplaceimag + 1;
            M2=2;

            CNTR=nover2;
            AX0=DM(I0,M0);
            AY0=DM(I1,M0);
            AY1=DM(I3,M0);

            DO group_lp UNTIL CE;
               AR=AX0+AY0, AX1=DM(I2,M0);
               SB=EXPADJ AR, DM(I0,M2)=AR;
               AR=AX0-AY0;
               SB=EXPADJ AR;
               DM(I1,M2)=AR, AR=AX1+AY1;
               SB=EXPADJ AR, DM(I2,M2)=AR;
               AR=AX1-AY1, AX0=DM(I0,M0);
               SB=EXPADJ AR, DM(I3,M2)=AR;
               AY0=DM(I1,M0);
group_lp:      AY1=DM(I3,M0);
            CALL bfp_adj;

{──────────STAGES 2 TO N-1──────────────────────}

         DO stage_loop UNTIL CE;    {Compute all stages in FFT}
            I0=^inplacereal;        {I0 ->x0 in 1st grp of stage}
            I2=^inplaceimag;        {I2 ->y0 in 1st grp of stage}
            SI=DM(groups);
            SR=ASHIFT SI BY -1(LO); {groups / 2}
            DM(groups)=SR0;         {groups=groups / 2}
            CNTR=SR0;               {CNTR=group counter}
            M4=SR0;                 {M4=twiddle factor modifier}
            M2=DM(node_space);      {M2=node space modifier}
            I1=I0;
            MODIFY(I1,M2);          {I1 ->y0 of 1st grp in stage}
            I3=I2;
```

14 Software Examples

```
              MODIFY(I3,M2);          {I3 ->y1 of 1st grp in stage}

              DO group_loop UNTIL CE;
                 I4=^twid_real;                   {I4 -> C of W0}
                 I5=^twid_imag;                   {I5 -> (-S) of W0}
                 CNTR=DM(bflys_per_group);        {CNTR=bfly count}
                 MY0=PM(I4,M4),MX0=DM(I1,M0);     {MY0=C,MX0=x1 }
                 MY1=PM(I5,M4),MX1=DM(I3,M0);     {MY1=-S,MX1=y1}
                 DO bfly_loop UNTIL CE;
                    MR=MX0*MY1(SS),AX0=DM(I0,M0);
                                                  {MR=x1(-S),AX0=x0}
                    MR=MR+MX1*MY0(RND),AX1=DM(I2,M0);
                                      {MR=(y1(C)+x1(-S)),AX1=y0}
                    AY1=MR1,MR=MX0*MY0(SS);
                                     {AY1=y1(C)+x1(-S),MR=x1(C)}
                    MR=MR-MX1*MY1(RND);            {MR=x1(C)-y1(-S)}
                    AY0=MR1,AR=AX1-AY1;
                             {AY0=x1(C)-y1(-S),AR=y0-[y1(C)+x1(-S)]}
                    SB=EXPADJ AR,DM(I3,M1)=AR;
                        {Check for bit growth, y1=y0-[y1(C)+x1(-S)]}
                    AR=AX0-AY0,MX1=DM(I3,M0),MY1=PM(I5,M4);
                    {AR=x0-[x1(C)-y1(-S)], MX1=next y1,MY1=next (-S)}
                    SB=EXPADJ AR,DM(I1,M1)=AR;
                        {Check for bit growth, x1=x0-[x1(C)-y1(-S)]}
                    AR=AX0+AY0,MX0=DM(I1,M0),MY0=PM(I4,M4);
                      {AR=x0+[x1(C)-y1(-S)], MX0=next x1,MY0=next C}
                    SB=EXPADJ AR,DM(I0,M1)=AR;
                        {Check for bit growth, x0=x0+[x1(C)-y1(-S)]}
                    AR=AX1+AY1;                {AR=y0+[y1(C)+x1(-S)]}
bfly_loop:          SB=EXPADJ AR,DM(I2,M1)=AR;
                        {Check for bit growth, y0=y0+[y1(C)+x1(-S)]}
                 MODIFY(I0,M2);           {I0 ->1st x0 in next group}
                 MODIFY(I1,M2);           {I1 ->1st x1 in next group}
                 MODIFY(I2,M2);           {I2 ->1st y0 in next group}
group_loop:      MODIFY(I3,M2);           {I3 ->1st y1 in next group}

              CALL bfp_adj;               {Compensate for bit growth}
              SI=DM(bflys_per_group);
              SR=ASHIFT SI BY 1(LO);
              DM(node_space)=SR0;         {node_space=node_space / 2}
stage_loop:   DM(bflys_per_group)=SR0;
```

```
                              {bflys_per_group=bflys_per_group / 2}

{——— LAST STAGE ———}

         I0=^inplacereal;
         I1=^inplacereal+nover2;
         I2=^inplaceimag;
         I3=^inplaceimag+nover2;

         CNTR=nover2;
         M2=DM(node_space);
         M4=1;
         I4=^twid_real;
         I5=^twid_imag;

         MY0=PM(I4,M4),MX0=DM(I1,M0);        {MY0=C,MX0=x1}
         MY1=PM(I5,M4),MX1=DM(I3,M0);        {MY1=-S,MX1=y1}
         DO bfly_lp UNTIL CE;
            MR=MX0*MY1(SS),AX0=DM(I0,M0);       {MR=x1(-S),AX0=x0}
            MR=MR+MX1*MY0(RND),AX1=DM(I2,M0);
                                          {MR=(y1(C)+x1(-S)),AX1=y0}
            AY1=MR1,MR=MX0*MY0(SS);    {AY1=y1(C)+x1(-S),MR=x1(C)}
            MR=MR-MX1*MY1(RND);                 {MR=x1(C)-y1(-S)}
            AY0=MR1,AR=AX1-AY1;
                        {AY0=x1(C)-y1(-S), AR=y0-[y1(C)+x1(-S)]}
            SB=EXPADJ AR,DM(I3,M1)=AR;
                      {Check for bit growth, y1=y0-[y1(C)+x1(-S)]}
            AR=AX0-AY0,MX1=DM(I3,M0),MY1=PM(I5,M4);
                 {AR=x0-[x1(C)-y1(-S)], MX1=next y1,MY1=next (-S)}
            SB=EXPADJ AR,DM(I1,M1)=AR;
                      {Check for bit growth, x1=x0-[x1(C)-y1(-S)]}
            AR=AX0+AY0,MX0=DM(I1,M0),MY0=PM(I4,M4);
                    {AR=x0+[x1(C)-y1(-S)], MX0=next x1,MY0=next C}
            SB=EXPADJ AR,DM(I0,M1)=AR;
                      {Check for bit growth, x0=x0+[x1(C)-y1(-S)]}
            AR=AX1+AY1;                        {AR=y0+[y1(C)+x1(-S)]}
bfly_lp:    SB=EXPADJ AR,DM(I2,M1)=AR;      {Check for bit growth}

         CALL bfp_adj;

         RTS;
.ENDMOD;
```

Listing 14.6 Radix-2 DIT FFT Routine, Conditional Block Floating-Point

14 Software Examples

14.7.3 Bit-Reverse Subroutine

The bit-reversal routine, called scramble, puts the input data in bit-reversed order so that the results will be in sequential order. This routine uses the bit-reverse capability of the ADSP-2100 family processors.

```
.MODULE  dit_scramble;

{  Calling Parameters
         Sequentially ordered input data in inputreal

   Return Values
         Scrambled input data in inplacereal

   Altered Registers
         I0,I4,M0,M4,AY1

   Altered Memory
         inplacereal
}

.CONST      N=1024,mod_value=H#0010; {Initialize constants}

.EXTERNAL   inputreal, inplacereal;

.ENTRY      scramble;


scramble:   I4=^inputreal;   {I4—>sequentially ordered data}
            I0=^inplacereal;     {I0—>scrambled data}
            M4=1;
            M0=mod_value;    {M0=modifier for reversing N bits}
            L4=0;
            L0=0;
            CNTR = N;
            ENA BIT_REV;     {Enable bit-reversed outputs on DAG1}
            DO brev UNTIL CE;
               AY1=DM(I4,M4);    {Read sequentially ordered data}
brev:          DM(I0,M0)=AY1;
                              {Write data in bit-reversed location}
            DIS BIT_REV;     {Disable bit-reverse}
            RTS;             {Return to calling program}
.ENDMOD;
```

Listing 14.7 Bit-Reverse Routine (Scramble)

14.7.4 Block Floating-Point Scaling Subroutine

The bfp_adj routine checks the FFT output data for bit growth and scales the entire set of data if necessary. This check prevents data overflow for each stage in the FFT. The routine, shown in Listing 14.8, uses the exponent detection capability of the shifter.

```
.MODULE  dit_radix_2_bfp_adjust;

{  Calling Parameters
      Radix-2 DIT FFT stage results in inplacereal and inplaceimag

   Return Parameters
      inplacereal and inplaceimag adjusted for bit growth

   Altered Registers
      I0,I1,AX0,AY0,AR,MX0,MY0,MR,CNTR

   Altered Memory
      inplacereal, inplaceimag, blk_exponent
}

.CONST      Ntimes2 = 2048;
.EXTERNAL   inplacereal, blk_exponent; {Begin declaration section}

.ENTRY      bfp_adj;

bfp_adj:    AY0=CNTR;            {Check for last stage}
            AR=AY0-1
            IF EQ RTS;           {If last stage, return}
            AY0=-2;
            AX0=SB;
            AR=AX0-AY0;          {Check for SB=-2}
            IF EQ RTS;           {IF SB=-2, no bit growth, return}
            I0=^inplacereal;     {I0=read pointer}
            I1=^inplacereal;     {I1=write pointer}
            AY0=-1;
            MY0=H#4000;          {Set MY0 to shift 1 bit right}
```

14 Software Examples

```
            AR=AX0-AY0,MX0=DM(I0,M1);
                                {Check if SB=-1; Get 1st sample}
            IF EQ JUMP strt_shift;
                               {If SB=-1, shift block data 1 bit}
            AX0=-2;            {Set AX0 for block exponent update}
            MY0=H#2000;           {Set MY0 to shift 2 bits right}
strt_shift: CNTR=Ntimes2 - 1;           {initialize loop counter}
            DO shift_loop UNTIL CE;        {Shift block of data}
               MR=MX0*MY0(RND),MX0=DM(I0,M1);
                                {MR=shifted data,MX0=next value}
shift_loop:    DM(I1,M1)=MR1;    {Unshifted data=shifted data}
            MR=MX0*MY0(RND);            {Shift last data word}
            AY0=DM(blk_exponent);       {Update block exponent and}
            DM(I1,M1)=MR1,AR=AY0-AX0;  {store last shifted sample}
            DM(blk_exponent)=AR;
            RTS;
.ENDMOD;
```

Listing 14.8 Radix-2 Block Floating-Point Scaling Routine

Instruction Set Reference 15

15.1 QUICK LIST OF INSTRUCTIONS

This chapter is a complete reference for the instruction set of the ADSP-2100 family. The instruction set is organized by instruction group and, within each group, by individual instruction. The list below shows all of the instructions and the reference page for each.

ALU
- Add / Add with Carry (p. 15-21)
- Subtract X-Y / Subtract X-Y with Borrow (p. 15-23)
- Subtract Y-X / Subtract Y-X with Borrow (p. 15-25)
- AND, OR, XOR (p. 15-27)
- Test Bit, Set Bit, Clear Bit, Toggle Bit (p. 15-29)
- Pass / Clear (p. 15-31)
- Negate (p. 15-33)
- NOT (p. 15-34)
- Absolute Value (p. 15-35)
- Increment (p. 15-36)
- Decrement (p. 15-37)
- Divide (p. 15-38)
- Generate ALU Status (p. 15-40)

MAC
- Multiply (p. 15-41)
- Multiply / Accumulate (p. 15-43)
- Multiply / Subtract (p. 15-45)
- Clear (p. 15-47)
- Transfer MR (p. 15-48)
- Conditional MR Saturation (p. 15-49)

SHIFTER
- Arithmetic Shift (p. 15-50)
- Logical Shift (p. 15-52)
- Normalize (p. 15-54)
- Derive Exponent (p. 15-56)
- Block Exponent Adjust (p. 15-58)
- Arithmetic Shift Immediate (p. 15-60)
- Logical Shift Immediate (p. 15-62)

MOVE
- Register Move (p. 15-63)
- Load Register Immediate (p. 15-65)
- Data Memory Read (Direct Address) (p. 15-67)
- Data Memory Read (Indirect Address) (p. 15-68)
- Program Memory Read (Indirect Address) (p. 15-69)
- Data Memory Write (Direct Address) (p. 15-70)
- Data Memory Write (Indirect Address) (p. 15-71)
- Program Memory Write (Indirect Address) (p. 15-73)
- I/O Space Read/Write (p. 15-74)

PROGRAM FLOW
- JUMP (p. 15-75)
- CALL (p. 15-76)
- JUMP or CALL on Flag In Pin (p. 15-77)
- Modify Flag Out Pin (p. 15-78)
- Return from Subroutine (p. 15-79)
- Return from Interrupt (p. 15-80)
- Do Until (p. 15-81)
- IDLE (p. 15-83)

MISC
- Stack Control (p. 15-84)
- Mode Control (p. 15-87)
- Modify Address Register (p. 15-89)
- NOP (p. 15-90)
- Interrupt Enable & Disable (p. 15-91)

MULTIFUNCTION
- ALU/MAC/SHIFT with Memory Read (p. 15-92)
- ALU/MAC/SHIFT with Data Register Move (p. 15-96)
- ALU/MAC/SHIFT with Memory Write (p. 15-99)
- Data & Program Memory Read (p. 15-103)
- ALU/MAC with Data & Program Memory Read (p. 15-104)

15 Instruction Set Reference

15.2 OVERVIEW

This chapter provides an overview and detailed reference for the instruction set of the ADSP-2100 family of DSP microprocessors.

For information regarding the ADSP-2100 Family Development Software, refer to the *ADSP-2100 Family Assembler Tools & Simulator Manual*, *ADSP-2100 Family C Tools Manual*, and *ADSP-2100 Family C Runtime Library Manual*. These manuals provide a complete guide to the development software. The handbooks *Digital Signal Processing Applications Using The ADSP-2100 Family, Volume 1* and *Volume 2* present DSP applications programs with source code and discussion.

The 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 processors' three computational units: the ALU, MAC and Shifter. Arithmetic instructions can process single-precision 16-bit operands directly; provisions for multiprecision operations are available.

The high-level syntax of ADSP-2100 family source code is both readable and efficient. Unlike many assembly languages, the ADSP-2100 family 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 instruction set. (If memory access times require, 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 instruction set's 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 IDLE instruction is provided 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 address values; indirect addressing uses the I registers of the two data address generators (DAGs).

Instruction Set Reference 15

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

- any ALU, MAC or Shifter operation (conditional or non-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 instruction set allows maximum flexibility. It provides moves from any register to any other register, and from most registers to/from memory. In addition, 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.

15.3 INSTRUCTION TYPES & NOTATION CONVENTIONS

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

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

Because the multifunction instructions best illustrate the power of the processors' architecture, in the next section we begin with a discussion of this group of instructions.

Throughout this chapter you will find tables summarizing the syntax of the instruction groups. The following notation conventions are used in these tables and in the reference page for each instruction.

Square Brackets [ ]	Anything within square brackets is an optional part of the instruction statement.
Parallel Lines | |	Lists of operands are enclosed by vertical parallel bars. One of the operands listed must be chosen. If the parallel bars are within square brackets, then the operand is optional for that instruction.

15 Instruction Set Reference

CAPITAL LETTERS	Capital letters denote a literal in the instruction. Literals are the instruction name (e.g. ADD), register names, or operand selections. Literals must be typed exactly as shown.
operands	Some instruction operands are shown in lowercase letters. These operands may take different values in assembly code. For example, the operand *yop* may be one of several registers: AY0, AY1, or AF.
<exp>	Denotes exponent (shift value) in Shift Immediate instructions; must be an 8-bit signed integer constant.
<data>	Denotes an immediate data value. Can also be a symbol (address label or variable/buffer name) dereferenced by the '%' or '^' operators.
<addr>	Denotes an immediate address value to be encoded in the instruction. The <addr> may be either an immediate value (a constant) or a program label.
<reg>	Refers to any accessible register; see Table 15.7.
<dreg>	Refers to any data register; see Table 15.7.

Immediate values, <exp>, <data>, or <addr>, may be a constant in decimal, hexadecimal, octal or binary format. Default is to decimal.

15.4 MULTIFUNCTION INSTRUCTIONS

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

15.4.1 ALU/MAC With Data & Program Memory Read

Perhaps the single most common operation in DSP algorithms is the sum of products, performed as follows:

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

Instruction Set Reference 15

The ADSP-2100 family processors can execute both data fetches and the multiplication/accumulation in a single-cycle. Typically, a loop of multiply/accumulates can be expressed in ADSP-21xx source code in just two program lines. Since the on-chip program memory of the ADSP-21xx processors 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).

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).

15.4.2 Data & Program Memory Read

This variation of a multifunction instruction is a special case of the multifunction instruction described above in which the computation is omitted. It executes only the dual operand 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, for the processors with on-chip memory).

15.4.3 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 kind of multifunction 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.

15.4.4 Computation With Memory Write

The computation with memory write instruction is similar in structure to the computation with memory read: 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 this 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 causes the assembler to generate a 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 given above. All ALU operations except division, all MAC operations, and all Shifter operations except SHIFT IMMEDIATE are legal. Computations must be unconditional.

15.4.5 Computation With Data Register Move

This final type of 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 also 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 is given in Table 15.7. A complete list of the permissible *xops* and *yops* for computational operations is given in the reference page for each instruction. Table 15.1 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 15.1 Summary Of Valid Combinations For Multifunction Instructions

15 Instruction Set Reference

Multifunction Instructions

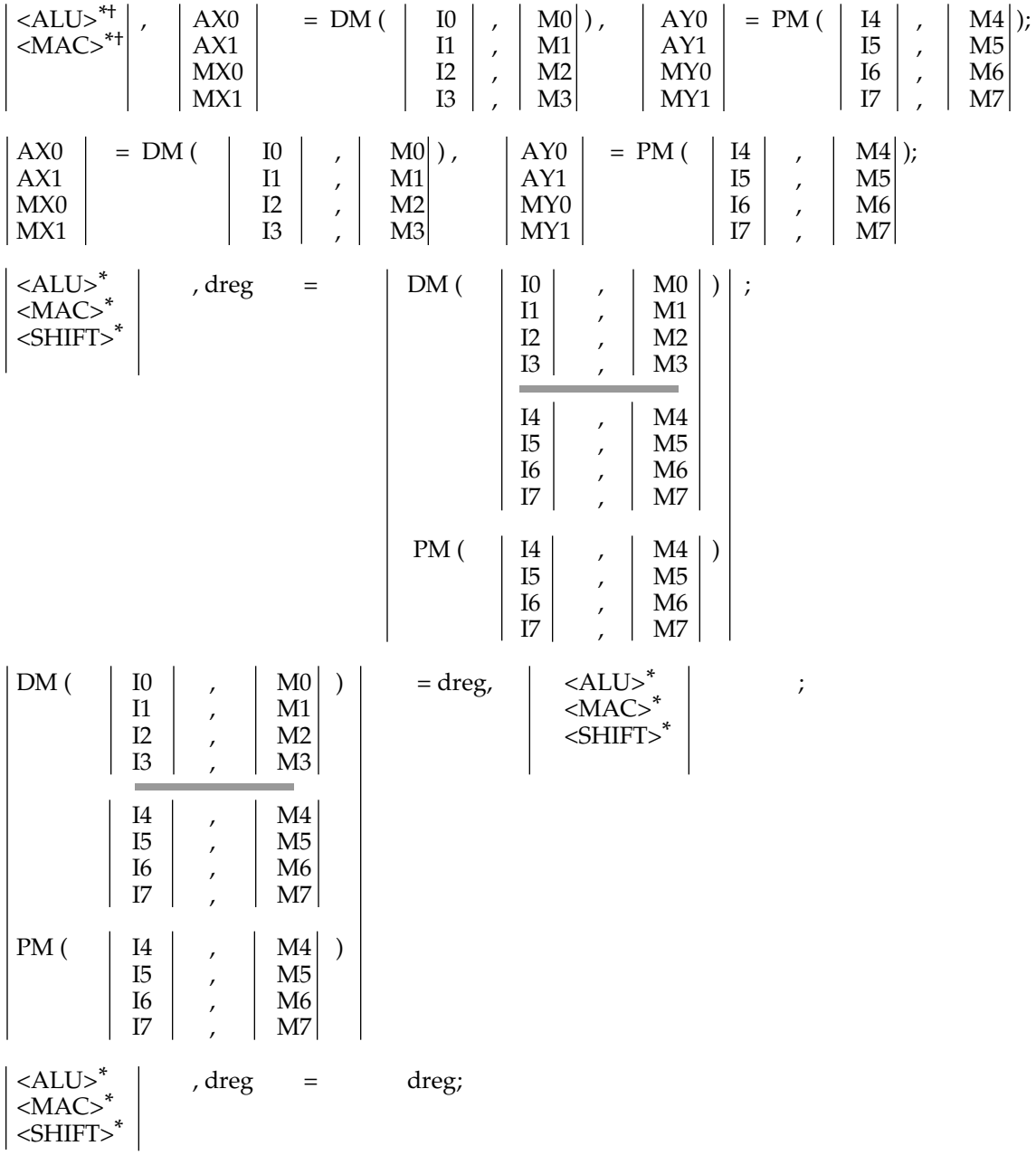


Table 15.2 Multifunction Instructions

<ALU>	Any ALU instruction *(except DIVS, DIVQ)*
<MAC>	Any multiply/accumulate instruction
<SHIFT>	Any shifter instruction *(except Shift Immediate)*

* May not be conditional instruction

† AR, MR result registers must be used—not AF, MF feedback registers. (See Section 15.4.1, "ALU/MAC with Data & Program Memory Read.")

15.5 ALU, MAC & SHIFTER INSTRUCTIONS

This group of instructions performs computations. All of these instructions can be executed conditionally except the ALU division instructions and the Shifter SHIFT IMMEDIATE instructions.

15.5.1 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.

Table 15.3 gives 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 reference page for each instruction. A complete list of conditions is given in Table 15.9.

ALU Instructions

[IF condition]	AR AF	=	xop	+ yop + C + yop + C + constant + constant + C	;
[IF condition]	AR AF	=	xop	– yop – yop + C – 1 + C – 1 – constant – constant + C – 1	;
[IF condition]	AR AF	=	yop	– xop – xop + C – 1 – xop + C – 1 – xop + constant – xop + constant + C – 1	;

15 Instruction Set Reference

[IF condition]	AR AF	=	xop	AND OR XOR	yop constant	;
[IF condition]	AR AF	=	TSTBIT n OF xop SETBIT n OF xop CLRBIT n OF xop TGLBIT n OF xop			;
[IF condition]	AR AF	=	PASS	xop yop constant		;
[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 ;

NONE = <ALU> ;

Table 15.3 ALU Instructions

15.5.2 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.

Table 15.4 gives 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 reference page for each instruction.
MAC Instructions

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

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

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

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

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

IF MV SAT MR ;

Table 15.4 MAC Instructions

15.5.3 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.

15 Instruction Set Reference

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.

Table 15.5 gives 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 <exp>	(	HI / LO	);
SR	=		[SR OR] LSHIFT xop BY <exp>	(	HI / LO	);

Table 15.5 Shifter Instructions

15.6 MOVE: READ & WRITE

MOVE instructions, shown in Table 15.6, move data to and from data registers and external memory. 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 15.7 shows which registers belong to these groups. Many of the system control registers are memory-mapped (for the processors with on-chip memory); these registers are read and written as memory locations instead of with register names.

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 15.6 MOVE Instructions

Registers: ***reg***	*Data Registers:* ***dreg***
SB	
PX	
I0 – I7, M0 – M7, L0 – L7	AX0, AX1, AY0, AY1, AR
CNTR	MX0, MX1, MY0, MY1, MR0, MR1, MR2
ASTAT, MSTAT, SSTAT	SI, SE, SR0, SR1
IMASK, ICNTL, IFC	
TX0, TX1, RX0, RX1	

Table 15.7 Processor Registers: reg & dreg

15.7 PROGRAM FLOW CONTROL

Program flow control on the ADSP-2100 family processors is simple but powerful. Here is an example of one instruction:

```
IF EQ JUMP my_label;
```

JUMP, of course, is a familiar construct from many other languages. *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 source code 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 IDLE instruction 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.

Table 15.8 gives a summary of all program flow control instructions. The *condition* codes are described in Table 15.9.

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 |

[IF condition]      RTS ;

[IF condition]      RTI ;

DO <address>  [UNTIL termination] ;

IDLE [(n)];
```

Table 15.8 Program Flow Control Instructions

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	
FLAG_IN*	FI pin	Last sample of FI pin = 1
NOT FLAG_IN*	Not FI pin	Last sample of FI pin = 0

Table 15.9 IF Condition Codes

* Only available on JUMP and CALL instructions

15 Instruction Set Reference

15.8 MISCELLANEOUS INSTRUCTIONS

There are several miscellaneous instructions. NOP is a no operation instruction. The PUSH/POP instructions 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 instruction enables and disables processor modes of operation: bit-reversal on DAG1, latching ALU overflow, saturating the ALU result register, choosing the primary or secondary register set, 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 4, “Data Transfer,” for more information).

The FO (Flag Out), FL0, FL1 and FL2 pins can each be set, cleared, or toggled. This instruction provides a control structure for multiprocessor communication.

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 |

[IF condition]      | SET    |    | FLAG_OUT |   [ , ]   ;
                    | RESET  |    |    FL0   |
                    | TOGGLE |    |    FL1   |
                                  |    FL2   |

| ENA | INTS ;
| DIS |
```

Table 15.10 Miscellaneous Instructions

15 Instruction Set Reference

15.9 EXTRA CYCLE CONDITIONS

All instructions execute in a single cycle except under certain conditions, as explained below.

15.9.1 Multiple Off-Chip Memory Accesses

The data and address busses of the ADSP-21xx processors are multiplexed off-chip. Because of this, the processors can perform only one off-chip access per instruction in a single cycle. If two off-chip accesses are required—the instruction fetch and one data fetch, for example, or data fetches from both program and data memory—then one overhead cycle occurs. In this case the program memory access occurs first, then the data memory access. If three off-chip accesses are required—the instruction fetch as well as data fetches from both program and data memory—then two overhead cycles occur.

A multifunction instruction requires three items to be fetched from memory: the instruction itself and two data words. No extra cycle is needed to execute the instruction as long as only one of the fetches is from external memory. (Two fetches must be from on-chip memory, either PM or DM.)

15.9.2 Wait States

All family processors allow the programming of wait states for external memory chips. Up to seven extra wait state cycles may be added to the processor's access time for external memory. Extra cycles inserted due to wait states are in addition to any caused by multiple off-chip accesses (as described above). Wait state programming is described in the "Memory Interface" chapter.

Wait states and multiple off-chip memory accesses are the two cases when an extra cycle is generated during instruction execution. The following case, SPORT autobuffering and DMA, causes the insertion of extra cycles *between* instructions.

15.9.3 SPORT Autobuffering & DMA

If serial port autobuffering or ADSP-2181 DMA is being used to transfer data words to or from internal memory, then one memory access is "stolen" for each transfer. The stolen memory access occurs only between complete instructions. If extra cycles are required to execute any instruction (for one of the two reasons above), the processor waits until it is completed before "stealing" the access cycle.

15.10 INSTRUCTION SET SYNTAX

The following sections describe instruction set syntax and other notation conventions used in the reference page of each instruction.

15.10.1 Punctuation & Multifunction Instructions

All instructions terminate with a semicolon. A comma separates the clauses of a multifunction instruction but does not terminate it. For example, the statements below in Example A comprise one multifunction instruction (which can execute in a single cycle). Example B shows two separate instructions, requiring two instruction cycles.

Example A: One multifunction instruction

```
AX0 = DM(I0, M0), a comma is used in multifunction instructions
AY0 = PM(I4, M4);
```

Example B: Two separate instructions

```
AX0 = DM(I0, M0); a semicolon terminates an instruction
AY0 = PM(I4, M4);
```

15.10.2 Syntax Notation Example

Here is an example of one instruction, the ALU Add/Add with Carry instruction:

[IF cond] | AR / AF | = xop + | yop / C / yop + C | ;

The permissible *conds*, *xops* and *yops* are given in a list. The conditional IF clause is enclosed in square brackets, indicating that it is optional.

The destination register for the add operation must be either AR or AF. These are listed within parallel bars, indicating that one of the two must be chosen.

Similarly, the *yop* term may consist of a Y operand, the carry bit, or the sum of both. One of these three terms must be used.

15.10.3 Status Register Notation

The following notation is used in the discussion of the effect each instruction has on the processors' status registers:

*	An asterisk indicates a bit in the status word that is changed by the execution of the instruction.
–	A dash indicates that a bit is not affected by the instruction.
0 or 1	Indicates that a bit is unconditionally cleared or set.

For example, the status word ASTAT is shown below:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	0	–	–

Here the MV bit is updated and the AV bit is cleared.

Syntax: [IF cond] | AR / AF | = xop | + yop / + C / + yop + C / + constant / + constant + C | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example:
```
IF EQ AR = AX0 + AY0 + C;
AR = AR + 512;
```

Description: Test the optional condition and, if true, perform the specified addition. If false then perform a no-operation. Omitting the condition performs the addition unconditionally. The addition operation adds the first source operand to the second source operand along with the ALU carry bit, AC, (if designated by the "+C" notation), using binary addition. The result is stored in the destination register. The operands are contained in the data registers or constant specified in the instruction.

The *xop + constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

(instruction continues on next page)

ADD / ADD with CARRY

Instruction Format:

Conditional ALU/MAC operation, Instruction Type 9:

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

AMF specifies the ALU or MAC operation, in this case:

AMF = 10010 for xop + yop + C
AMF = 10011 for xop + yop

(Note that xop + C is a special case of xop + yop + C with yop=0.)

Z: Destination register
Yop: Y operand
Xop: X operand
COND: condition

(xop + constant) Conditional ALU/MAC operation, Instruction Type 9: *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

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	YY	Xop	CC	BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 10010 for xop + constant + C
AMF = 10011 for xop + constant

Z: Destination register
COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

SUBTRACT X-Y / SUBTRACT X-Y with BORROW

Syntax: [IF cond] | AR / AF | = xop | – yop / – yop + C–1 / + C–1 / – constant / – constant + C–1 | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example: IF GE AR = AX0 – AY0;

Description: Test the optional condition and, if true, then perform the specified subtraction. If the condition is not true then perform a no-operation. Omitting the condition performs the subtraction unconditionally. The subtraction operation subtracts the second source operand from the first source operand, and optionally adds the ALU Carry bit (AC) minus 1 (H#0001), and stores the result in the destination register. The (C–1) quantity effectively implements a borrow capability for multiprecision subtractions. The operands are contained in the data registers or constant specified in the instruction.

The *xop – constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	-	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.

(instruction continues on next page)

AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC operation, Instruction type 9:

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

AMF specifies the ALU or MAC operation. In this case,

AMF = 10110 for xop – yop + C – 1 operation.
AMF = 10111 for xop – yop operation.

Note that xop + C – 1 is a special case of xop – yop + C – 1 with yop=0.

Z: Destination register Yop: Y operand
Xop: X operand COND: condition

(xop – constant) Conditional ALU/MAC operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

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	YY	Xop	CC	BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 10110 for xop – constant + C – 1
AMF = 10111 for xop – constant

Z: Destination register COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

SUBTRACT Y-X / SUBTRACT Y-X with BORROW

Syntax: [IF cond] | AR / AF | = | yop – | xop / xop + C – 1 | | ;
| –xop + C – 1 |
| –xop + constant |
| –xop + constant + C – 1 |

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example: IF GT AR = AY0 – AX0 + C – 1;

Description: Test the optional condition and, if true, then perform the specified subtraction. If the condition is not true then perform a no-operation. Omitting the condition performs the subtraction unconditionally. The subtraction operation subtracts the second source operand from the first source operand, optionally adds the ALU Carry bit (AC) minus 1 (H#0001), and stores the result in the destination register. The (C–1) quantity effectively implements a borrow capability for multiprecision subtractions. The operands are contained in the data registers or constant specified in the instruction.

The *–xop + constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.

(instruction continues on next page)

SUBTRACT Y-X / SUBTRACT Y-X with BORROW

AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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

AMF specifies the ALU or MAC operation. In this case,

AMF = 11010 for yop – xop + C – 1
AMF = 11001 for yop – xop

(Note that –xop + C – 1 is a special case of yop – xop + C – 1 with yop=0.)

Z: Destination register Yop: Y operand
Xop: X operand COND: condition

(–xop + constant) Conditional ALU/MAC operation, Instruction Type 9: *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

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	YY	Xop	CC BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 11010 for constant – xop + C – 1
AMF = 11001 for constant – xop

Z: Destination register COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

Syntax: [IF cond] | AR / AF | = xop | AND / OR / XOR | | yop / constant | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32767
–2, –3, –5, –9, –17, –33, –65, –129, –257, –513, –1025, –2049, –4097, –8193, –16385, –32768

Example:
```
AR = AX0  XOR  AY0;
IF FLAG_IN  AR = MR0 AND 8192;
```

Description: Test the optional condition and if true, then perform the specified bitwise logical operation (logical AND, inclusive OR, or exclusive OR). If the condition is not true then perform a no-operation. Omitting the condition performs the logical operation unconditionally. The operands are contained in the data registers or constant specified in the instruction.

The *xop AND/OR/XOR constant* operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Always cleared.
AC Always cleared.

(instruction continues on next page)

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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			

AMF specifies the ALU or MAC operation. In this case,

AMF = 11100 for AND operation.
AMF = 11101 for OR operation.
AMF = 11110 for XOR operation.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

(xop AND/OR/XOR constant)
Conditional ALU/MAC operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

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					YY		Xop			CC		BO	COND				

AMF specifies the ALU or MAC operation, in this case:

AMF = 11100 for AND operation.
AMF = 11101 for OR operation.
AMF = 11110 for XOR operation.

Z: Destination register
Xop: X operand
COND: condition

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

TEST BIT, SET BIT, CLEAR BIT, TOGGLE BIT
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Syntax: [IF cond] | AR / AF | = | TSTBIT n OF xop / SETBIT n OF xop / CLRBIT n OF xop / TGLBIT n OF xop | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
AX0	MR2	EQ	LE	AC
AX1	MR1	NE	NEG	NOT AC
AR	MR0	GT	POS	MV
	SR1	GE	AV	NOT MV
	SR0	LT	NOT AV	NOT CE

Permissible n values (0=LSB)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

Examples:
```
AF=TSTBIT 5 OF AR;
AR=TGLBIT 13 OF AX0;
```

Description: Test the optional condition and if true, then perform the specified bit operation. If the condition is not true then perform a no-operation. Omitting the condition performs the operation unconditionally. These operations cannot be used in multifunction instructions.

These operations are defined as follows:

TSTBIT is an AND operation with a 1 in the selected bit
SETBIT is an OR operation with a 1 in the selected bit
CLRBIT is an AND operation with a 0 in the selected bit
TGLBIT is an XOR operation with a 1 in the selected bit

The ASTAT status bits are affected by these instructions. The following instructions could be used, for example, to test a bit and branch accordingly:

```
AF=TSTBIT 5 OF AR;
IF NE JUMP set;     /*Jump to "set" if bit 5 of AR is
set*/
```

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

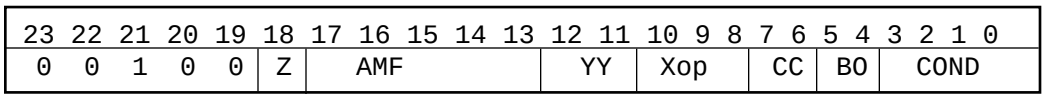
(instruction continues on next page)

TEST BIT, SET BIT, CLEAR BIT, TOGGLE BIT
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Always cleared.
AC Always cleared.

Instruction Format:

(xop AND/OR/XOR constant)
Conditional ALU/MAC operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)



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	YY	Xop	CC	BO	COND

AMF specifies the ALU or MAC operation, in this case:

AMF = 11100 for AND operation.
AMF = 11101 for OR operation.
AMF = 11110 for XOR operation.

Z: Destination register
Xop: X operand
COND: condition

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

Syntax: [IF cond] | AR / AF | = PASS | xop / yop / constant | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Permissible constants (all ADSP-21xx processors)
–1, 0, 1

Permissible constants (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
2, 3, 4, 5, 7, 8, 9, 15, 16, 17, 31, 32, 33, 63, 64, 65, 127, 128, 129, 255, 256, 257, 511, 512, 513, 1023, 1024, 1025, 2047, 2048, 2049, 4095, 4096, 4097, 8191, 8192, 8193, 16383, 16384, 16385, 32766, 32767

–2, –3, –4, –5, –6, –8, –9, –10, –16, –17, –18, –32, –33, –34, –64, –65, –66, –128, –129, –130, –256, –257, –258, –512, –513, –514, –1024, –1025, –1026, –2048, –2049, –2050, –4096, –4097, –4098, –8192, –8193, –8194, –16384, –16385, –16386, –32767, –32768

Examples:
```
IF GE AR = PASS AY0;
AR = PASS 0;
AR = PASS 8191;      (ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)
```

Description: Test the optional condition and if true, pass the source operand unmodified through the ALU block and store in the destination register. If the condition is not true perform a no-operation. Omitting the condition performs the PASS unconditionally. The source operand is contained in the data register or constant specified in the instruction.

PASS 0 is one method of clearing AR. PASS 0 can also be combined with memory reads and writes in a multifunction instruction to clear AR.

The PASS instruction performs the transfer to the AR or AF register and affects the ASTAT status flags (for xop, yop, –1, 0, 1 only). This instruction is different from a register move operation which does not affect any status flags. The *PASS constant*

(instruction continues on next page)

operation (using any constant other than –1, 0, or 1) causes the ASTAT status flags to be undefined.

The *PASS constant* operation (using any constant other than –1, 0, or 1) is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors and may not be used in multifunction instructions.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV, AC Always cleared.

Note: The *PASS constant* operation (using any constant other than –1, 0, or 1) causes the ASTAT status flags to be undefined.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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			

AMF specifies the ALU or MAC operation. In this case,

AMF = 10000 for PASS yop
AMF = 10011 for PASS xop
AMF = 10001 for PASS 1
AMF = 11000 for PASS –1

Note that PASS xop is a special case of xop + yop, with yop=0.
Note that PASS 1 is a special case of yop + 1, with yop=0.
Note that PASS –1 is a special case of yop – 1, with yop=0.

Z: Destination register
Xop: X operand
Yop: Y operand
COND: condition

Conditional ALU/MAC operation, Instruction Type 9:
(PASS constant; constant ≠ 0,1,–1)
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

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					YY		Xop			CC		BO		COND			

AMF specifies the ALU or MAC operation. In this case,

AMF = 10000 for PASS yop (special case of yop, with yop=constant)
AMF = 10001 for PASS yop + 1 (special case of yop + 1, with yop=constant)
AMF = 11000 for PASS yop – 1 (special case of yop – 1, with yop=constant)

Z: Destination register COND: condition
Xop: X operand

BO, CC, and YY specify the constant (see Appendix A, *Instruction Coding*).

Syntax: [IF cond] | AR / AF | = – | xop / yop | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1		GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF LT AR = – AY0;

Description: Test the optional condition and if true, then NEGATE the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the NEGATE operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if operand = H#8000. Cleared otherwise.
AC Set if operand equals zero. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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			

AMF specifies the ALU or MAC operation. In this case,
AMF = 10101 for –yop operation.
AMF = 11001 for –xop operation

Note that –xop is a special case of yop – xop, with yop specified to be 0.

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | AR / AF | = NOT | xop / yop | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
AX0	MR2	AY0	EQ	LE	AC
AX1	MR1	AY1	NE	NEG	NOT AC
AR	MR0	AF	GT	POS	MV
	SR1	0	GE	AV	NOT MV
	SR0		LT	NOT AV	NOT CE

Example: IF NE AF = NOT AX0;

Description: Test the optional condition and if true, then perform the logical complement (ones complement) of the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the complement operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	0	0	*	*

AZ	Set if the result equals zero. Cleared otherwise.
AN	Set if the result is negative. Cleared otherwise.
AV	Always cleared.
AC	Always cleared.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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

AMF specifies the ALU or MAC operation. In this case,
AMF = 10100 for NOT yop operation.

AMF = 11011 for NOT xop operation.

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | AR / AF | = ABS xop ;

Permissible xops		*Permissible conds (see Table 15.9)*		
AX0	MR2	EQ	LE	AC
AX1	MR1	NE	NEG	NOT AC
AR	MR0	GT	POS	MV
	SR1	GE	AV	NOT MV
	SR0	LT	NOT AV	NOT CE

Example: IF NEG AF = ABS AX0 ;

Description: Test the optional condition and, if true, then take the absolute value of the source operand and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the absolute value operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	0	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if *xop* is H#8000. Cleared otherwise.
AV Set if *xop* is H#8000. Cleared otherwise.
AC Always cleared.
AS Set if the source operand is negative. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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	0 0	Xop	0 0 0 0	COND

AMF specifies the ALU or MAC operation. In this case,
AMF = 11111 for ABS xop operation.

15 ALU INCREMENT

Z: Destination register
Xop: X operand COND: condition

Syntax: [IF cond] | AR / AF | = yop + 1 ;

Permissible yops	*Permissible conds (see Table 15.9)*		
AY0	EQ	LE	AC
AY1	NE	NEG	NOT AC
AF	GT	POS	MV
	GE	AV	NOT MV
	LT	NOT AV	NOT CE

Example: IF GT AF = AF + 1;

Description: Test the optional condition and if true, then increment the source operand by H#0001 and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the increment operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an overflow is generated. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:
Conditional ALU/MAC Operation, Instruction Type 9:

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

AMF specifies the ALU or MAC operation. In this case,

AMF = 10001 for yop + 1 operation.

Note that the xop field is ignored for the increment operation.

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax: [IF cond] | AR / AF | = yop – 1 ;

Permissible yops	*Permissible conds (see Table 15.9)*		
AY0	EQ	LE	AC
AY1	NE	NEG	NOT AC
AF	GT	POS	MV
	GE	AV	NOT MV
	LT	NOT AV	NOT CE

Example: IF EQ AR = AY1 – 1 ;

Description: Test the optional condition and if true, then decrement the source operand by H#0001 and store in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the decrement operation unconditionally. The source operand is contained in the data register specified in the instruction.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ	Set if the result equals zero. Cleared otherwise.
AN	Set if the result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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

AMF specifies the ALU or MAC operation. In this case,

AMF = 11000 for yop – 1 operation.

Note that the xop field is ignored for the decrement operation.

DIVIDE

Z:	Destination register	Yop:	Y operand
Xop:	X operand	COND:	condition

Syntax:
```
DIVS  yop ,  xop  ;
DIVQ  xop  ;
```

Permissible xops		*Permissible yops*
AX0	MR2	AY1
AX1	MR1	AF
AR	MR0	
	SR1	
	SR0	

Description: These instructions implement **yop ÷ xop**. There are two divide primitives, DIVS and DIVQ. A single precision divide, with a 32-bit numerator and a 16-bit denominator, yielding a 16-bit quotient, executes in 16 cycles. Higher precision divides are also possible.

The division can be either signed or unsigned, but both the numerator and denominator must be the same; both signed or unsigned. The programmer sets up the divide by sorting the upper half of the numerator in any permissible *yop* (AY1 or AF), the lower half of the numerator in AY0, and the denominator in any permissible *xop*. The divide operation is then executed with the divide primitives, DIVS and DIVQ. Repeated execution of DIVQ implements a non-restoring conditional add-subtract division algorithm. At the conclusion of the divide operation the quotient will be in AY0.

To implement a signed divide, first execute the DIVS instruction once, which computes the sign of the quotient. Then execute the DIVQ instruction for as many times as there are bits remaining in the quotient (e.g., for a signed, single-precision divide, execute DIVS once and DIVQ 15 times).

To implement an unsigned divide, first place the upper half of the numerator in AF, then set the AQ bit to zero by manually clearing it in the Arithmetic Status Register, ASTAT. This indicates that the sign of the quotient is positive. Then execute the DIVQ instruction for as many times as there are bits in the quotient (e.g., for an unsigned single-precision divide, execute DIVQ 16 times).

The quotient bit generated on each execution of DIVS and DIVQ is the AQ bit which is written to the ASTAT register at the end of each cycle. The final remainder produced by this algorithm (and left over in the AF register) is not valid and must be corrected if it is needed. For more information, consult the *Division Exceptions* appendix of this manual.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	*	–	–	–	–	–

AQ Loaded with the bit value equal to the AQ bit computed on each cycle from execution of the DIVS or DIVQ instruction.

Instruction Format:

DIVQ, Instruction Type 23:

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

DIVS, Instruction Type 24:

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

GENERATE ALU STATUS
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Xop: X operand Yop: Y operand

Syntax: NONE = <ALU> ;

*<ALU> may be any unconditional ALU operation except DIVS or DIVQ.**

Examples:
```
NONE = AX0 – AY0;
NONE = PASS SR0;
```

Description: Perform the designated ALU operation, generate the ASTAT status flags, then discard the result value. This instruction allows the testing of register values without disturbing the contents of the AR or AF registers.

* Note that the additional-constant ALU operations of the ADSP-217x, ADSP-218x, ADSP-21msp58/59 processors are also not allowed:

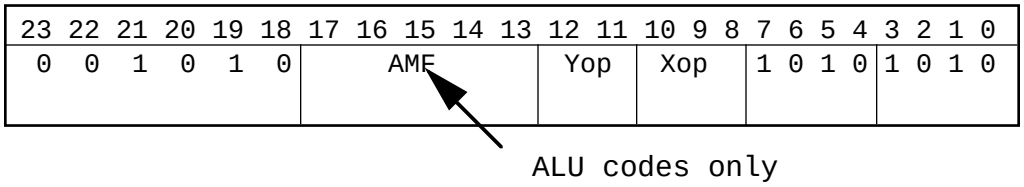
ADD (*xop + constant*)
SUBTRACT X–Y (*xop – constant*)
SUBTRACT Y–X (*–xop + constant*)
AND, OR, XOR (*xop ● constant*)
PASS (*PASS constant, using any constant other than –1, 0, or 1*)
TSTBIT, SETBIT, CLRBIT, TGLBIT.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	–	–	–	*	*	*	*

AZ Set if the result equals zero. Cleared otherwise.
AN Set if the result is negative. Cleared otherwise.
AV Set if an arithmetic overflow occurs. Cleared otherwise.
AC Set if a carry is generated. Cleared otherwise.

Instruction Format:
ALU/MAC operation with Data Register Move, Instruction Type 8:



AMF specifies the ALU or MAC operation (only ALU operations are allowed).

Xop: X operand Yop: Y operand

Syntax: [IF cond] | MR / MF | = xop * | yop / xop | | (SS) / (SU) / (US) / (UU) / (RND) | ;

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Examples:
IF EQ MR = MX0 * MF (UU); *xop * yop*
MF = SR0 * SR0 (SS); *xop * xop*

Description: Test the optional condition and, if true, then multiply the two source operands and store in the destination location. If the condition is not true perform a no-operation. Omitting the condition performs the multiplication unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 31-16 of the product are stored in MF.

The *xop * xop* squaring operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors. Both *xops* must be the same register. This option allows single-cycle X^2 and $\sum X^2$ instructions.

The data format selection field following the two operands specifies whether each respective operand is in Signed (S) or Unsigned (U) format. The *xop* is specified first and *yop* is second. If the *xop * xop* operation is used, the data format selection field must be (UU), (SS), or (RND) only. There is no default; one of the data formats must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, rounds the result to the most significant 24 bits (or rounds bits 31-16 to 16 bits if there is no overflow from the multiply), and stores the result in the destination register. The two multiplication operands *xop* and *yop* (or *xop* and *xop*) are considered to be in twos complement format. All rounding is unbiased, except on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors, which offer a biased rounding mode. For a discussion of

(instruction continues on next page)

biased vs. unbiased rounding, see "Rounding Mode" in the "Multiplier/ Accumulator" section of Chapter 2, *Computation Units*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

*(xop * yop)* Conditional ALU/MAC Operation, Instruction Type 9:

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

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 0 1 0 0	xop * yop	(SS)	Signed	Signed
0 0 1 0 1	xop * yop	(SU)	Signed	Unsigned
0 0 1 1 0	xop * yop	(US)	Unsigned	Signed
0 0 1 1 1	xop * yop	(UU)	Unsigned	Unsigned
0 0 0 0 1	xop * yop	(RND)	Signed	Signed

Z: Destination register — Yop: Y operand register
Xop: X operand register — COND: condition

*(xop * xop)* Conditional ALU/MAC Operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

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	0 0	Xop	0 0 0 1	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*
0 0 1 0 0	xop * xop	(SS)	Signed
0 0 1 1 1	xop * xop	(UU)	Unsigned
0 0 0 0 1	xop * xop	(RND)	Signed

Z: Destination register
Xop: X operand register

COND: condition

Syntax:

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

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Examples:

```
IF GE MR = MR + MX0 * MY1 (SS) ;        xop * yop
MR = MR + MX0 * MX0  (SS);              xop * xop
```

Description: Test the optional condition and, if true, then multiply the two source operands, add the product to the present contents of the MR register, and store the result in the destination location. If the condition is not true then perform a no-operation. Omitting the condition performs the multiply/accumulate unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 31-16 of the 40-bit result are stored in MF.

The *xop * xop* squaring operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors. Both *xops* must be the same register. This option allows single-cycle X^2 and $\sum X^2$ instructions.

The data format selection field to the right of the two operands specifies whether each respective operand is in signed (S) or unsigned (U) format. The *xop* is specified first and *yop* is second. If the *xop * xop* operation is used, the data format selection field must be (UU), (SS), or (RND) only. There is no default; one of the data formats must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, adds the product to the current contents of the MR register, rounds the result to the most significant 24 bits (or rounds bits 31-16 to the nearest 16 bits if there is no overflow from the multiply/accumulate), and stores the result in the destination register. The two multiplication operands *xop* and *yop* (or *xop* and *xop*) are considered to be in twos complement format. All

(instruction continues on next page)

rounding is unbiased, except on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors, which offer a biased rounding mode. For a discussion of biased vs. unbiased rounding, see "Rounding Mode" in the "Multiplier/Accumulator" section of Chapter 2, *Computation Units*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

*(xop * yop)* Conditional ALU/MAC Operation, Instruction Type 9:

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

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 1 0 0 0	MR+xop * yop	(SS)	Signed	Signed
0 1 0 0 1	MR+xop * yop	(SU)	Signed	Unsigned
0 1 0 1 0	MR+xop * yop	(US)	Unsigned	Signed
0 1 0 1 1	MR+xop * yop	(UU)	Unsigned	Unsigned
0 0 0 1 0	MR+xop * yop	(RND)	Signed	Signed

Z: Destination register — Yop: Y operand register
Xop: X operand register — COND: condition

(*xop * xop*) Conditional ALU/MAC Operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

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

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*
0 1 0 0 0	MR+xop * xop	(SS)	Signed
0 1 0 1 1	MR+xop * xop	(UU)	Unsigned
0 0 0 1 0	MR+xop * xop	(RND)	Signed

Z: Destination register
Xop: X operand register
COND: condition

Syntax:

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

Permissible xops		*Permissible yops*	*Permissible conds (see Table 15.9)*		
MX0	AR	MY0	EQ	LE	AC
MX1	SR1	MY1	NE	NEG	NOT AC
MR2	SR0	MF	GT	POS	MV
MR1			GE	AV	NOT MV
MR0			LT	NOT AV	NOT CE

Examples:

```
IF LT MR = MR – MX1 * MY0 (SU) ;        xop * yop
MR = MR – MX0 * MX0  (SS);              xop * xop
```

Description: Test the optional condition and, if true, then multiply the two source operands, subtract the product from the present contents of the MR register, and store the result in the destination location. If the condition is not true perform a no-operation. Omitting the condition performs the multiply/subtract unconditionally. The operands are contained in the data registers specified in the instruction. When MF is the destination operand, only bits 16-31 of the 40-bit result are stored in MF.

The *xop * xop* squaring operation is only available on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors. Both *xops* must be the same register.

The data format selection field to the right of the two operands specifies whether each respective operand is in signed (S) or unsigned (U) format. The *xop* is specified first and *yop* is second. If the *xop * xop* operation is used, the data format selection field must be (UU), (SS), or (RND) only. There is no default; one of the data formats must be specified.

If RND (Round) is specified, the MAC multiplies the two source operands, subtracts the product from the current contents of the MR register, rounds the result to the most significant 24 bits (or rounds bits 31-16 to 16 bits if there is no overflow from the multiply/accumulate), and stores the result in the destination register. The two multiplication operands *xop* and *yop* (or *xop* and *xop*) are considered to be in twos complement format. All

(instruction continues on next page)

rounding is unbiased, except on the ADSP-217x, ADSP-218x, and ADSP-21msp58/59 processors, which offer a biased rounding mode. For a discussion of biased vs. unbiased rounding, see "Rounding Mode" in the "Multiplier/Accumulator" section of Chapter 2, *Computation Units*.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of the upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

*(xop * yop)* Conditional ALU/MAC Operation, Instruction Type 9:

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

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*	*Y-Operand*
0 1 1 0 0	MR–xop * yop	(SS)	Signed	Signed
0 1 1 0 1	MR–xop * yop	(SU)	Signed	Unsigned
0 1 1 1 0	MR–xop * yop	(US)	Unsigned	Signed
0 1 1 1 1	MR–xop * yop	(UU)	Unsigned	Unsigned
0 0 0 1 1	MR–xop * yop	(RND)	Signed	Signed

Z: Destination register — Yop: Y operand register
Xop: X operand register — COND: condition

*(xop * xop)* Conditional ALU/MAC Operation, Instruction Type 9:
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

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	0 0	Xop	0 0 0 1	COND

AMF: Specifies the ALU or MAC Operation. In this case,

AMF	*FUNCTION*	*Data Format*	*X-Operand*
0 1 1 0 0	MR–xop * xop	(SS)	Signed
0 1 1 1 1	MR–xop * xop	(UU)	Unsigned
0 0 0 1 1	MR–xop * xop	(RND)	Signed

Z: Destination register COND: condition
Xop: X operand register

Syntax: [IF cond] | MR / MF | = 0 ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF GT MR = 0;

Description: Test the optional condition and, if true, then set the specified register to zero. If the condition is not true perform a no-operation. Omitting the condition performs the clear unconditionally. The entire 40-bit MR or 16-bit MF register is cleared to zero.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	0	–	–	–	–	–	–

MV Always cleared.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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					1	1	0	0	0	0	0	0	0	COND			

AMF: Specifies the ALU or MAC Operation. In this case,
AMF = 00100 for clear operation.

TRANSFER MR

Note that this instruction is a special case of xop * yop, with yop set to zero.

Z: Destination register COND: condition

Syntax: [IF cond] | MR / MF | = MR [(RND)] ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF EQ MF = MR (RND);

Description: Test the optional condition and, if true, then perform the MR transfer according to the description below. If the condition is not true then perform a no-operation. Omitting the condition performs the transfer unconditionally.

This instruction actually performs a multiply/accumulate, specifying *yop = 0* as a multiplicand and adding the zero product to the contents of MR. The MR register may be optionally rounded at the boundary between bits 15 and 16 of the result by specifying the RND option. If MF is specified as the destination, bits 31-16 of the result are stored in MF. If MR is the destination, the entire 40-bit result is stored in MR.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	–	*	–	–	–	–	–	–

MV — Set on MAC overflow (if any of upper 9 bits of MR are not all one or zero). Cleared otherwise.

Instruction Format:

Conditional ALU/MAC Operation, Instruction Type 9:

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	1	1	0	0	0	0	0	0	0	COND			

AMF: Specifies the ALU or MAC Operation. In this case,

AMF = 01000 for Transfer MR operation

Note that this instruction is a special case of MR + xop * yop, with yop set to zero.

Z: Destination register COND: condition
Syntax: IF MV SAT MR ;

Description: Test the MV (MAC Overflow) bit in the Arithmetic Status Register (ASTAT), and if set, then saturate the lower-order 32 bits of the 40-bit MR register; if the MV is not set then perform a no-operation.

Saturation of MR is executed with this instruction for one cycle only; MAC saturation is not a continuous mode that is enabled or disabled. The saturation instruction is intended to be used at the completion of a series of multiply/accumulate operations so that temporary overflows do not cause the accumulator to saturate.

The saturation result depends on the state of MV and on the sign of MR (the MSB of MR2). The possible results after execution of the saturation instruction are shown in the table below.

MV	*MSB of MR2*	*MR contents after saturation*		
0	0	No change		
0	1	No change		
1	0	00000000	0111111111111111	1111111111111111
1	1	11111111	1000000000000000	0000000000000000

Status Generated: No status bits affected.

Instruction Format:

Saturate MR operation, Instruction Type 25:

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

ARITHMETIC SHIFT

Syntax: [IF cond] SR = [SR OR] ASHIFT xop | (HI) / (LO) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF LT SR = SR OR ASHIFT SI (LO);

Description: Test the optional condition and, if true, then perform the designated arithmetic shift. If the condition is not true then perform a no-operation. Omitting the condition performs the shift unconditionally. The operation arithmetically shifts the bits of the operand by the amount and direction specified in the Shift Code from the SE register. Positive Shift Codes cause a left shift (upshift) and negative codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For ASHIFT with a positive Shift Code (i.e. positive value in SE), the operand is shifted left; with a negative Shift Code (i.e. negative value in SE), the operand is shifted right. The number of positions shifted is the count in the Shift Code. The 32-bit output field is sign-extended to the left (the MSB of the input is replicated to the left), and the output is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using an ASHIFT with the HI option; on the following cycle, the lower half of the number is shifted using an LSHIFT with the LO and OR options. This prevents sign bit extension of the lower word's MSB.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

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			

SF *Shifter Function*
0 1 0 0 ASHIFT (HI)
0 1 0 1 ASHIFT (HI, OR)
0 1 1 0 ASHIFT (LO)
0 1 1 1 ASHIFT (LO, OR)

Xop: shifter operand COND: condition

LOGICAL SHIFT

Syntax: [IF cond] SR = [SR OR] LSHIFT xop | (HI) / (LO) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GE SR = LSHIFT SI (HI) ;

Description: Test the optional condition and, if true, then perform the designated logical shift. If the condition is not true then perform a no-operation. Omitting the condition performs the shift unconditionally. The operation logically shifts the bits of the operand by the amount and direction specified in the Shift Code from the SE register. Positive Shift Codes cause a left shift (upshift) and negative Codes cause a right shift (downshift).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For LSHIFT with a positive Shift Code, the operand is shifted left; the numbers of positions shifted is the count in the Shift Code. The 32-bit output field is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped.

For LSHIFT with a negative Shift Code, the operand is shifted right; the number of positions shifted is the count in the Shift Code. The 32-bit output field is zero-filled from the left. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using the HI option; on the following cycle, the lower half of the number is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

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			

SF	*Shifter Function*
0 0 0 0	LSHIFT (HI)
0 0 0 1	LSHIFT (HI, OR)
0 0 1 0	LSHIFT (LO)
0 0 1 1	LSHIFT (LO, OR)

Xop: shifter operand COND: condition

NORMALIZE

Syntax: [IF cond] SR = [SR OR] NORM xop | (HI) | (LO) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: SR = NORM SI (HI) ;

Description: Test the optional condition and, if true, then perform the designated normalization. If the condition is not true then perform a no-operation. Omitting the condition performs the normalize unconditionally. The operation arithmetically shifts the input operand to eliminate all but one of the sign bits. The amount of the shift comes from the SE register. The SE register may be loaded with the proper Shift Code to eliminate the redundant sign bits by using the Derive Exponent instruction; the Shift Code loaded will be the negative of the quantity: (the number of sign bits minus one).

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option. When the LO reference is selected, the 32-bit output field is zero-filled to the left. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped.

The 32-bit output field is zero-filled from the right. If the exponent of an overflowed ALU result was derived with the HIX modifier, the 32-bit output field is filled from left with the ALU Carry (AC) bit in the Arithmetic Status Register (ASTAT) during a NORM (HI) operation. In this case (SE=1 from the exponent detection on the overflowed ALU value) a downshift occurs.

To normalize a double precision number, the same Shift Code is used for both halves of the number. On the first cycle, the upper half of the number is shifted using the HI option; on the following cycle, the lower half of the number is shifted using the LO and OR options.

Status Generated: None affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

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			

SF *Shifter Function*
1 0 0 0 NORM (HI)
1 0 0 1 NORM (HI, OR)
1 0 1 0 NORM (LO)
1 0 1 1 NORM (LO, OR)

Xop: shifter operand COND: condition

DERIVE EXPONENT

Syntax: [IF cond] SE = EXP xop | (HI) (LO) (HIX) | ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GT SE = EXP MR1 (HI) ;

Description: Test the optional condition and, if true, perform the designated exponent operation. If the condition is not true then perform a no-operation. Omitting the condition performs the exponent operation unconditionally.

The EXP operation derives the effective exponent of the input operand to prepare for the normalization operation (NORM). EXP supplies the source operand to the exponent detector, which generates a Shift Code from the number of leading sign bits in the input operand. The Shift Code, stored in SE at the completion of the EXP instruction, is the effective exponent of the input value. The Shift Code depends on which exponent detector mode is used (HI, HIX, LO).

In the HI mode, the input is interpreted as a single precision signed number, or as the upper half of a double precision signed number. The exponent detector counts the number of leading sign bits in the source operand and stores the resulting Shift Code in SE. The Shift Code will equal the negative of the number of redundant sign bits in the input.

In the HIX mode, the input is interpreted as the result of an add or subtract which may have overflowed. HIX is intended to handle shifting and normalization of results from ALU operations. The HIX mode examines the ALU Overflow bit (AV) in the Arithmetic Status Register: if AV is set, then the effective exponent of the input is +1 (indicating that an ALU overflow occurred before the EXP operation), and +1 is stored in SE. If AV is not set, then HIX performs exactly the same operations as the HI mode.

In the LO mode, the input is interpreted as the lower half of a double precision number. In performing the EXP operation on a double precision number, the higher half of the number must first be processed with EXP in the HI or HIX mode, and then the lower half can be processed with EXP in the LO mode. If the upper half contained a non-sign bit, then the correct Shift Code was generated in the HI or HIX operation and that is the code that is stored in SE. If, however, the upper half was all sign bits, then EXP in the LO mode totals the number of leading sign bits in the double precision word and stores the resulting Shift Code in SE.

Status Generated:

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	–	–	–	–	–	–	–

SS — Set by the MSB of the input for an EXP operation in the HI or HIX mode with AV = 0. Set by the MSB inverted in the HIX mode with AV = 1. Not affected by operations in the LO mode.

Instruction Format:

Conditional Shift Operation, Instruction Type 16:

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			

SF	*Shifter Function*
1 1 0 0	EXP (HI)
1 1 0 1	EXP (HIX)
1 1 1 0	EXP (LO)

Xop: shifter operand COND: condition

BLOCK EXPONENT ADJUST

Syntax: [IF cond] SB = EXPADJ xop ;

Permissible xops		*Permissible conds (see Table 15.9)*		
SI	AR	EQ	LE	AC
SR1	MR2	NE	NEG	NOT AC
SR0	MR1	GT	POS	MV
	MR0	GE	AV	NOT MV
		LT	NOT AV	NOT CE

Example: IF GT SB = EXPADJ SI ;

Description: Test the optional condition and, if true, perform the designated exponent operation. If the condition is not true then perform a no-operation. Omitting the condition performs the exponent operation unconditionally. The Block Exponent Adjust operation, when performed on a series of numbers, derives the effective exponent of the number largest in magnitude. This exponent can then be associated with all of the numbers in a block floating point representation.

The Block Exponent Adjust circuitry applies the input operand to the exponent detector to derive its effective exponent. The input must be a signed twos complement number. The exponent detector operates in HI mode (see the EXP instruction, above).

At the start of a block, the SB register should be initialized to –16 to set SB to its minimum value. On each execution of the EXPADJ instruction, the effective exponent of each operand is compared to the current contents of the SB register. If the new exponent is greater than the current SB value, it is written to the SB register, updating it. Therefore, at the end of the block, the SB register will contain the largest exponent found. EXPADJ is only an inspection operation; no actual shifting takes place since the true exponent is not known until all the numbers in the block have been checked. However, the numbers can be shifted at a later time after the true exponent has been derived.

Extended (overflowed) numbers and the lower halves of double precision numbers can not be processed with the Block Exponent Adjust instruction.

Status Generated: Not affected.

Instruction Format:
Conditional Shift Operation, Instruction Type 16:

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			

SF = 1111.

Xop: shifter operand COND: condition

15 SHIFTER
ARITHMETIC SHIFT IMMEDIATE

Syntax: SR = [SR OR] ASHIFT xop BY <exp> | (HI) / (LO) | ;

Permissible xops		*<exp>*
SI	MR0	Any constant between –128 and 127*
SR1	MR1	
SR0	MR2	
AR		

Example: SR = SR OR ASHIFT SR0 BY 3 (LO); {do not use "+3"}

Description: Arithmetically shift the bits of the operand by the amount and direction specified by the constant in the exponent field. Positive constants cause a left shift (upshift) and negative constants cause a right shift (downshift). A positive constant must be entered **without** a "+" sign.

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the present contents of the SR register by selecting the SR OR option.

For ASHIFT with a positive shift constant the operand is shifted left; with a negative shift constant the operand is shifted right. The 32-bit output field is sign-extended to the left (the MSB of the input is replicated to the left), and the output is zero-filled from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped. Bits shifted out of the low order bit in the destination field (SR_0) are dropped.

To shift a double precision number, the same shift constant is used for both halves of the number. On the first cycle, the upper half of the number is shifted using an ASHIFT with the HI option; on the following cycle, the lower half is shifted using an LSHIFT with the LO and OR options. This prevents sign bit extension of the lower word's MSB.

* See Table 2.4 in Chapter 2.

Status Generated: None affected.

Instruction Format:
Shift Immediate Operation, Instruction Type 15:

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			<exp>							

SF	*Shifter Function*
0 1 0 0	ASHIFT (HI)
0 1 0 1	ASHIFT (HI, OR)
0 1 1 0	ASHIFT (LO)
0 1 1 1	ASHIFT (LO, OR)

Xop: Shifter Operand <exp>: 8-bit signed shift value

15 SHIFTER
LOGICAL SHIFT IMMEDIATE

Syntax: SR = [SR OR] LSHIFT xop BY <exp> | (HI) / (LO) | ;

Permissible xops		*<exp>*
SI	MR0	Any constant between –128 and 127*
SR1	MR1	
SR0	MR2	
AR		

Example: SR = LSHIFT SR1 BY –6 (HI) ;

Description: Logically shifts the bits of the operand by the amount and direction specified by the constant in the exponent field. Positive constants cause a left shift (upshift); negative constants cause a right shift (downshift). A positive constant must be entered **without** a "+" sign.

The shift may be referenced to the upper half of the output field (HI option) or to the lower half (LO option). The shift output may be logically ORed with the contents of the SR register by selecting the SR OR option.

For LSHIFT with a positive shift constant, the operand is shifted left. The 32-bit output field is zero-filled to the left and from the right. Bits shifted out of the high order bit in the 32-bit destination field (SR_{31}) are dropped. For LSHIFT with a negative shift constant, the operand is shifted right. The 32-bit output field is zero-filled from the left and to the right. Bits shifted out of the low order bit are dropped.

To shift a double precision number, the same shift constant is used for both parts of the number. On the first cycle, the upper half of the number is shifted using the HI option; on the following cycle, the lower half is shifted using the LO and OR options.

* See Table 2.4 in Chapter 2.

Status Generated: None affected.

Instruction Format:
Shift Immediate Operation, Instruction Type 15:

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			<exp>							

SF	*Shifter Function*
0 0 0 0	LSHIFT (HI)
0 0 0 1	LSHIFT (HI, OR)
0 0 1 0	LSHIFT (LO)
0 0 1 1	LSHIFT (LO, OR)

Xop: Shifter Operand

<exp>: 8-bit signed shift value

Syntax: reg = reg ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	OWRCNTR(*write only*)
AY0	MY0	SR1	ASTAT	RX0
AY1	MY1	SR0	MSTAT	RX1
AR	MR2	I0-I7	SSTAT(*read only*)	TX0
	MR1	M0-M7	IMASK	TX1
	MR0	L0-L7	ICNTL	IFC(*write only*)

Example: I7 = AR;

Description: Move the contents of the source to the destination location. The contents of the source are always right-justified in the destination location after the move.

When transferring a smaller register to a larger register (e.g., an 8-bit register to a 16-bit register), the value stored in the destination is either sign-extended to the left if the source is a signed value, or zero-filled to the left if the source is an unsigned value. The unsigned registers which (when used as the source) cause the value stored in the destination to be zero-filled to the left are: I0 through I7, L0 through L7, CNTR, PX, ASTAT, MSTAT, SSTAT, IMASK, and ICNTL. All other registers cause sign-extension to the left.

When transferring a larger register to a smaller register (e.g., a 16-bit register to a 14-bit register), the value stored in the destination is right-justified (bit 0 maps to bit 0) and the higher-order bits are dropped.

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

Status Generated: None affected.

(instruction continues on next page)

REGISTER MOVE

Instruction Format:

Internal Data Move, Instruction Type 17:

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			

SRC RGP (Source Register Group) and SOURCE REG (Source Register) select the source register according to the Register Selection Table (see Appendix A).

DST RGP (Destination Register Group) and DEST REG (Destination Register) select the destination register according to the Register Selection Table (see Appendix A).

Syntax: reg = <data> ;
dreg = <data> ;

data: <constant>
'%' <symbol>
'^' <symbol>

Permissible registers

dregs (*Instruction Type 6*)
(*16-bit load*)

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

regs (*Instruction Type 7*)
(*maximum 14-bit load*)

SB	CNTR
PX	OWRCNTR (*write only*)
ASTAT	RX0
MSTAT	RX1
IMASK	TX0
ICNTL	TX1
I0-I7	IFC(*write only*)
M0-M7	
L0-L7	

Example: I0 = ^*data_buffer*;
L0=%*data_buffer*;

Description: Move the data value specified to the destination location. The data may be a constant, or any symbol referenced with the "length of" (%) or "pointer to" (^) operators. The data value is contained in the instruction word, with 16 bits for data register loads and up to 14 bits for other register loads. The value is always right-justified in the destination location after the load (bit 0 maps to bit 0). When a value of length less than the length of the destination is moved, it is sign-extended to the left to fill the destination width.

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

For this instruction only, the RX and TX registers may be loaded with a maximum of 14 bits of data (although the registers themselves are 16 bits wide). To load these registers with 16-bit data, use the register-to-register move instruction or the data memory-to-register move instruction with direct addressing.

Status Generated: None affected.

(instruction continues on next page)

Instruction Format :

Load Data Register Immediate, Instruction Type 6:

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			

DATA contains the immediate value to be loaded into the Data Register destination location. The data is right-justified in the field, so the value loaded into an N-bit destination register is contained in the lower-order N bits of the DATA field.

DREG selects the destination Data Register for the immediate data value. One of the 16 Data Registers is selected according to the DREG Selection Table (see Appendix A).

Load Non-Data Register Immediate Instruction Type 7:

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			

DATA contains the immediate value to be loaded into the Non-Data Register destination location. The data is right-justified in the field, so the value loaded into an N-bit destination register is contained in the lower-order N bits of the DATA field.

RGP (Register Group) and REG (Register) select the destination register according to the Register Selection Table (see Appendix A).

DATA MEMORY READ (Direct Address)

Syntax: reg = DM (<addr>) ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	OWRCNTR (*write only*)
AY0	MY0	SR1	ASTAT	RX0
AY1	MY1	SR0	MSTAT	RX1
AR	MR2	I0-I7		TX0
	MR1	M0-M7	IMASK	TX1
	MR0	L0-L7	ICNTL	IFC(*write only*)

Example: SI = DM(*ad_port0*);

Description: The Read instruction moves the contents of the data memory location to the destination register. The addressing mode is direct addressing (designated by an immediate address value or by a label). The data memory address is stored directly in the instruction word as a full 14-bit field. The contents of the source are always right-justified in the destination register after the read (bit 0 maps to bit 0).

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

Status Generated: None affected.

Instruction Format:
Data Memory Read (Direct Address), Instruction Type 3:

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	0	RGP																			

DATA MEMORY READ (Indirect Address)

Syntax: dreg = DM (I0 | I1 | I2 | I3 , M0 | M1 | M2 | M3) ;
I4 | I5 | I6 | I7 , M4 | M5 | M6 | M7

Permissible dregs

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

Example: AY0 = DM (I3, M1);

Description: The Data Memory Read Indirect instruction moves the contents of the data memory location to the destination register. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register after the read (bit 0 maps to bit 0).

Status Generated: None affected.

Instruction Format:
ALU / MAC Operation with Data Memory Read, Instruction Type 4:

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	0	0	AMF					0	0	0	0	0	DREG				I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Read. In this case, AMF = 00000, indicating a no-operation for the ALU/MAC function.

DREG selects the destination Data Register . One of the 16 Data Registers is selected according to the DREG Selection Table (see Appendix A).

G specifies which Data Address Generator the I and M registers are selected from. These registers must be from the same DAG as separated by the gray bar above. I specifies the indirect address pointer (I register). M specifies the modify register (M register).

PROGRAM MEMORY READ (Indirect Address)

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

Permissible dregs

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

Example: MX1 = PM (I6, M5);

Description: The Program Memory Read Indirect instruction moves the contents of the program memory location to the destination register. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The 16 most significant bits of the Program Memory Data bus (PMD_{23-8}) are loaded into the destination register, with bit PMD_8 lining up with bit 0 of the destination register (right-justification). If the destination register is less than 16 bits wide, the most significant bits are dropped. Bits PMD_{7-0} are always loaded into the PX register. You may ignore these bits or read them out on a subsequent cycle.

Status Generated: None affected

Instruction Format:
ALU / MAC Operation with Program Memory Read, Instruction Type 5:

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	0	0	AMF					0	0	0	0	0	DREG					I		M

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Read. In this case, AMF = 00000, indicating a no-operation for the ALU/MAC function.

DREG selects the destination Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

I specifies the indirect address pointer (I register). M specifies the modify register (M register).

15 MOVE
DATA MEMORY WRITE (Direct Address)

Syntax: DM (<addr>) = reg ;

Permissible registers

AX0	MX0	SI	SB	CNTR
AX1	MX1	SE	PX	RX0
AY0	MY0	SR1	ASTAT	RX1
AY1	MY1	SR0	MSTAT	TX0
AR	MR2	I0-I7	SSTAT(*read only*)	TX1
	MR1	M0-M7	IMASK	
	MR0	L0-L7	ICNTL	

Example: DM (*cntl_port0*) = AR;

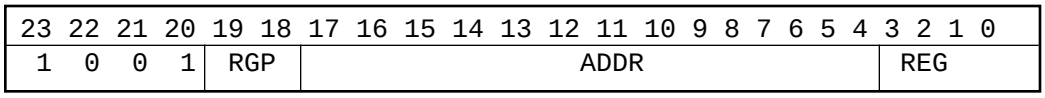
Description: Moves the contents of the source register to the data memory location specified in the instruction word. The addressing mode is direct addressing (designated by an immediate address value or by a label). The data memory address is stored directly in the instruction word as a full 14-bit field. Whenever a register less than 16 bits in length is written to memory, the value written is either sign-extended to the left if the source is a signed value, or zero-filled to the left if the source is an unsigned value. The unsigned registers which are zero-filled to the left are: I0 through I7, L0 through L7, CNTR, PX, ASTAT, MSTAT, SSTAT, IMASK, and ICNTL. All other registers are sign-extended to the left.

The contents of the source are always right-justified in the destination location after the write (bit 0 maps to bit 0).

Note that whenever MR1 is loaded with data, it is sign-extended into MR2.

Status Generated: None affected.

Instruction Format:
Data Memory Read (Direct Address), Instruction Type 3:



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	1	RGP				ADDR												REG			

ADDR contains the direct address of the destination location in Data Memory.

RGP (Register Group) and REG (Register) select the source register according to the Register Selection Table (see Appendix A).

DATA MEMORY WRITE (Indirect Address)

Syntax:

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

data: <constant>
'%' <symbol>
'^' <symbol>

Permissible dregs

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

Example: DM (I2, M0) = MR1;

Description: The Data Memory Write Indirect instruction moves the contents of the source to the data memory location specified in the instruction word. The immediate data may be a constant or any symbol referenced with the "length of" (%) or "pointer to" (^) operators.

The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** When a register of less than 16 bits is written to memory, the value written is sign-extended to form a 16-bit value. The contents of the source are always right-justified in the destination location after the write (bit 0 maps to bit 0).

Status Generated: None affected.

(instruction continues on next page)

DATA MEMORY WRITE (Indirect Address)

Instruction Format:
ALU / MAC Operation with Data Memory Write, Instruction Type 4:

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	1	0	AMF					0	0	0	0	0	DREG				I		M	

Data Memory Write, Immediate Data, Instruction Type 2:

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	

AMF specifies the ALU or MAC operation to be performed in parallel with the Data Memory Write. In this case, AMF = 00000, indicating a no-operation for the ALU / MAC function.

Data represents the actual 16-bit value.

DREG selects the source Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

G specifies which Data Address Generator the I and M registers are selected from. These registers must be from the same DAG as separated by the gray bar in the Syntax description above. I specifies the indirect address pointer (I register). M specifies the modify register (M register).

PROGRAM MEMORY WRITE (Indirect Address)

Syntax: PM (I4 / I5 / I6 / I7 , M4 / M5 / M6 / M7) = dreg ;

Permissible dregs

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

Example: PM (I6, M5) = AR;

Description: The Program Memory Write Indirect instruction moves the contents of the source to the program memory location specified in the instruction word. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The 16 most significant bits of the Program Memory Data bus (PMD_{23-8}) are loaded from the source register, with bit PMD_8 aligned with bit 0 of the source register (right justification). The 8 least significant bits of the Program Memory Data bus (PMD_{7-0}) are loaded from the PX register. Whenever a source register of length less than 16 bits is written to memory, the value written is sign-extended to form a 16-bit value.

Status Generated: None affected.

Instruction Format:
ALU / MAC Operation with Program Memory Write, Instruction Type 5 (see Appendix A), as shown below:

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	1	0	AMF					0	0	0	0	0	DREG				I		M	

AMF specifies the ALU or MAC operation to be performed in parallel with the Program Memory Write. In this case, AMF = 00000, indicating a no-operation for the ALU / MAC function.

DREG selects the source Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

I specifies the indirect address pointer (I register). M specifies the modify register (M register).

15 MOVE
I/O SPACE READ/WRITE
(ADSP-218x only)

Syntax:
IO (<addr>) = dreg ; *I/O write*
dreg = IO (<addr>) ; *I/O read*

<addr> is an 11-bit direct address value between 0 and 2047

Permissible dregs

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

Examples:
IO(23) = AX0;
MY1 = IO(2047);

Description: The I/O space read and write instructions are used to access the ADSP-218x's I/O memory space. These instructions move data between the processor data registers and the I/O memory space.

Status Generated: None affected.

Instruction Format:
I/O Memory Space Read/Write, Instruction Type 29:

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	D	ADDR											DREG			

ADDR contains the 11-bit direct address of the source or destination location in I/O Memory Space.

DREG selects the Data Register. One of the 16 Data Registers is selected according to the Register Selection Table (see Appendix A).

D specifies the direction of the transfer (0=read, 1=write).

Syntax: [IF cond] JUMP | (I4) / (I5) / (I6) / (I7) / <addr> | ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF NOT CE JUMP *top_loop*; {CNTR is decremented}

Description: Test the optional condition and, if true, perform the specified jump. If the condition is not true then perform a no-operation. Omitting the condition performs the jump unconditionally. The JUMP instruction causes program execution to continue at the effective address specified by the instruction. The addressing mode may be direct or register indirect.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field. For register indirect jumps, the selected I register provides the address; it is not post-modified in this case.

If JUMP is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled. If NOT CE is used as the condition, execution of the JUMP instruction decrements the processor's counter (CNTR register).

Status Generated: None affected.

Instruction Field:
Conditional JUMP Direct Instruction Type 10:

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	0	ADDR														COND			

Conditional JUMP Indirect Instruction Type 19:

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	0	COND			

I specifies the I register (Indirect Address Pointer).

ADDR: immediate jump address COND: condition

CALL

Syntax: [IF cond] CALL | (I4) (I5) (I6) (I7) <addr> | ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF AV CALL *scale_down;*

Description: Test the optional condition and, if true, then perform the specified call. If the condition is not true then perform a no-operation. Omitting the condition performs the call unconditionally. The CALL instruction is intended for calling subroutines. CALL pushes the PC stack with the return address and causes program execution to continue at the effective address specified by the instruction. The addressing modes available for the CALL instruction are direct or register indirect.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field. For register indirect jumps, the selected I register provides the address; it is not post-modified in this case.

If CALL is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

Status Generated: None affected.

Instruction Field:
Conditional JUMP Direct Instruction Type 10:

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	1	ADDR														COND			

Conditional JUMP Indirect Instruction Type 19:

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	1	COND			

I specifies the I register (Indirect Address Pointer).

ADDR: immediate jump address COND: condition

Syntax: IF | FLAG_IN / NOT FLAG_IN | | JUMP / CALL | | <addr> | ;

Example: IF FLAG_IN JUMP *service_proc_three*;

Description: Test the condition of the FI pin of the processor and, if set to one, perform the specified jump or call. If FI is zero then perform a no-operation. Omitting the flag in condition reduces the instruction to a standard JUMP or CALL.

The JUMP instruction causes program execution to continue at the address specified by the instruction. The addressing mode for the JUMP on FI must be direct.

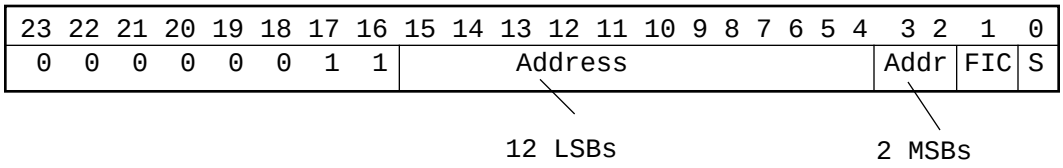
The CALL instruction is intended for calling subroutines. CALL pushes the PC stack with the return address and causes program execution to continue at the address specified by the instruction. The addressing mode for the CALL on FI must be direct.

If JUMP or CALL is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

For direct addressing (using an immediate address value or a label), the program address is stored directly in the instruction word as a full 14-bit field.

Status Generated: None affected.

Instruction Field:
Conditional JUMP or CALL on Flag In Direct Instruction Type 27:



S: specifies JUMP (0) or CALL (1)

FIC: latched state of FI pin

MODIFY FLAG OUT PIN

Syntax: [IF cond] | SET / RESET / TOGGLE | | FLAG_OUT / FL0 / FL1 / FL2 | [, …] ;

Example: IF MV SET FLAG_OUT, RESET FL1;

Description: Evaluate the optional condition and if true, set to one, reset to zero, or toggle the state of the specified flag output pin(s). Otherwise perform a no-operation and continue with the next instruction. Omitting the condition performs the operation unconditionally. Multiple flags may be modified by including multiple clauses, separated by commas, in a single instruction. This instruction does not directly alter the flow of your program—it is provided to signal external devices.

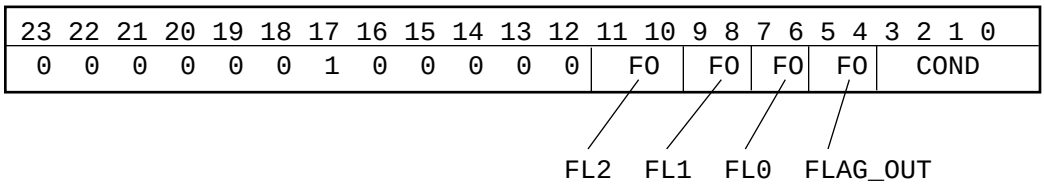
(Note that the FO pin is specified by "FLAG_OUT" in the instruction syntax.)

The following table shows which flag outputs are present on each ADSP-21xx processor:

processor	*flag pin(s)*
ADSP-2101	FO
ADSP-2105	FO
ADSP-2115	FO
ADSP-2111	FO, FL0, FL1, FL2
ADSP-217x	FO, FL0, FL1, FL2
ADSP-218x	FO, FL0, FL1, FL2
ADSP-21msp5x	FO, FL0, FL1, FL2

Status Generated: None affected.

Instruction Field:
Flag Out Mode Control Instruction Type 28:



FO: Operation to perform on flag output pin

COND: Condition code

Syntax: [IF cond] RTS ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF LE RTS ;

Description: Test the optional condition and, if true, then perform the specified return. If the condition is not true then perform a no-operation. Omitting the condition performs the return unconditionally. RTS executes a program return from a subroutine. The address on top of the PC stack is popped and is used as the return address. The PC stack is the only stack popped.

If RTS is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

Status Generated: None affected.

Instruction Field:
Conditional Return, Instruction Type 20:

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	0	COND			

RTI

COND: condition

Syntax: [IF cond] RTI ;

Permissible conds (see Table 15.9)

EQ	NE	GT	GE	LT
LE	NEG	POS	AV	NOT AV
AC	NOT AC	MV	NOT MV	NOT CE

Example: IF MV RTI ;

Description: Test the optional condition and, if true, then perform the specified return. If the condition is not true then perform a no-operation. Omitting the condition performs the return unconditionally. RTI executes a program return from an interrupt service routine. The address on top of the PC stack is popped and is used as the return address. The value on top of the status stack is also popped, and is loaded into the arithmetic status (ASTAT), mode status (MSTAT) and the interrupt mask (IMASK) registers.

If RTI is the last instruction inside a DO UNTIL loop, you must ensure that the loop stacks are properly handled.

Status Generated: None affected.

Instruction Field:
Conditional Return, Instruction Type 20:

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	1	COND			

COND: condition

Syntax: DO <addr> [UNTIL term] ;

Permissible terms

EQ	NE	GT	GE	LT	FOREVER
LE	NEG	POS	AV	NOT AV	
AC	NOT AC	MV	NOT MV	CE	

Example: DO *loop_label* UNTIL CE ; {CNTR is decremented each pass through loop}

Description: DO UNTIL sets up looping circuitry for zero-overhead looping. The program loop begins at the program instruction immediately following the DO instruction, ends at the address designated in the instruction and repeats execution until the specified termination condition is met (if one is specified) or repeats in an infinite loop (if none is specified). The termination condition is tested during execution of the last instruction in the loop, the status having been generated upon completion of the previous instruction. The address (<addr>) of the last instruction in the loop is stored directly in the instruction word.

If CE is used for the termination condition, the processor's counter (CNTR register) is decremented once for each pass through the loop.

When the DO instruction is executed, the address of the last instruction is pushed onto the loop stack along with the termination condition and the current program counter value plus 1 is pushed onto the PC stack.

Any nesting of DO loops continues the process of pushing the loop and PC stacks, up to the limit of the loop stack size (4 levels of loop nesting) or of the PC stack size (16 levels for subroutines plus interrupts plus loops). With either or both the loop or PC stacks full, a further attempt to perform the DO instruction will set the appropriate stack overflow bit and will perform a no-operation.

Status Generated:

ASTAT: Not affected.

SSTAT:	7	6	5	4	3	2	1	0
	LSO	LSE	SSO	SSE	CSO	CSE	PSO	PSE
	*	0	–	–	–	–	*	0

LSO Loop Stack Overflow: set if the loop stack overflows; otherwise not affected.
LSE Loop Stack Empty: always cleared (indicating loop stack not empty)
PSO PC Stack Overflow: set if the PC stack overflows; otherwise not affected.
PSE PC Stack Empty: always cleared (indicating PC stack not empty)

(instruction continues on next page)

DO UNTIL

Instruction Format:

Do Until, Instruction Type 11:

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			

ADDR specifies the address of the last instruction in the loop. In the Instruction Syntax, this field may be a program label or an immediate address value.

TERM specifies the termination condition, as shown below:

TERM	*Syntax*	*Condition Tested*
0 0 0 0	NE	Not Equal to Zero
0 0 0 1	EQ	Equal Zero
0 0 1 0	LE	Less Than or Equal to Zero
0 0 1 1	GT	Greater Than Zero
0 1 0 0	GE	Greater Than or Equal to Zero
0 1 0 1	LT	Less Than Zero
0 1 1 0	NOT AV	Not ALU Overflow
0 1 1 1	AV	ALU Overflow
1 0 0 0	NOT AC	Not ALU Carry
1 0 0 1	AC	ALU Carry
1 0 1 0	POS	X Input Sign Positive
1 0 1 1	NEG	X Input Sign Negative
1 1 0 0	NOT MV	Not MAC Overflow
1 1 0 1	MV	MAC Overflow
1 1 1 0	CE	Counter Expired
1 1 1 1	FOREVER	Always

Syntax: IDLE ;
IDLE (n); *Slow Idle*

Description: IDLE causes the processor to wait indefinitely in a low-power state, waiting for interrupts. When an interrupt occurs it is serviced and execution continues with the instruction following IDLE. Typically this next instruction will be a JUMP back to IDLE, implementing a low-power standby loop. (Note the restrictions on JUMP or IDLE as the last instruction in a DO UNTIL loop, detailed in Chapter 3.)

IDLE (n) is a special version of IDLE that slows the processor's internal clock signal to further reduce power consumption. The reduced clock frequency, a programmable fraction of the normal clock rate, is specified by a selectable divisor *n* given in the instruction: *n* = 16, 32, 64, or 128. The instruction leaves the processor fully functional, but operating at the slower rate during execution of the IDLE (n) instruction. While it is in this state, the processor's other internal clock signals (such as SCLK, CLKOUT, and the timer clock) are reduced by the same ratio.

When the IDLE (n) instruction is used, it slows the processor's internal clock and thus its response time to incoming interrupts—the 1-cycle response time of the standard IDLE state is increased by *n*, the clock divisor. When an enabled interrupt is received, the ADSP-21xx will remain in the IDLE state for up to a maximum of *n* CLKIN cycles (where *n* =16, 32, 64, or 128) before resuming normal operation.

When the IDLE (n) instruction is used in systems that have an externally generated serial clock, the serial clock rate may be faster than the processor's reduced internal clock rate. Under these conditions, interrupts must not be generated at a faster rate than can be serviced, due to the additional time the processor takes to come out of the IDLE state (a maximum of *n* CLKIN cycles).

Serial port autobuffering continues during IDLE without affecting the idle state.

Status Generated: None affected.

Instruction Field:
Idle, Instruction Type 31:

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

Slow Idle, Instruction Type 31:

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	DV			

15 MISC
STACK CONTROL

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

Example: POP CNTR, POP PC, POP LOOP;

Description: Stack Control pushes or pops the designated stack(s). The entire instruction executes in one cycle regardless of how many stacks are specified.

The PUSH STS (Push Status Stack) instruction increments the status stack pointer by one to point to the next available status stack location; and pushes the arithmetic status (ASTAT), mode status (MSTAT), and interrupt mask register (IMASK) onto the processor's status stack. Note that the PUSH STS operation is executed automatically whenever an interrupt service routine is entered.

Any POP pops the value on the top of the designated stack and decrements the same stack pointer to point to the next lowest location in the stack. POP STS causes the arithmetic status (ASTAT), mode status (MSTAT), and interrupt mask (IMASK) to be popped into these same registers. This also happens automatically whenever a return from interrupt (RTI) is executed.

POP CNTR causes the counter stack to be popped into the down counter. When the loop stack or PC stack is popped (with POP LOOP or POP PC, respectively), the information is lost. Returning from an interrupt (RTI) or subroutine (RTS) also pops the PC stack automatically.

Status Generated:

SSTAT:	7	6	5	4	3	2	1	0
	LSO	LSE	SSO	SSE	CSO	CSE	PSO	PSE
	–	*	*	*	–	*	–	*

PSE PC Stack Empty: set if a pop results in an empty program counter stack; cleared otherwise.
CSE Counter Stack Empty: set if a pop results in an empty counter stack; cleared otherwise.
SSE Status Stack Empty: for PUSH STS, this bit is always cleared (indicating status stack not empty). For POP STS, SSE is set if the pop results in an empty status stack; cleared otherwise.
SSO Status Stack Overflow: for PUSH STS set if the status stack overflows; otherwise not affected.
LSE Loop Stack Empty: set if a pop results in an empty loop stack; cleared otherwise.

Note that once any Stack Overflow occurs, the corresponding stack overflow bit is set in SSTAT, and this bit stays set indicating there has been loss of information. Once set, the stack overflow bit can only be cleared by resetting the processor.

Instruction Format:

Stack Control, Instruction Type 26:

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

Pp: PC Stack Control
Lp: Loop Stack Control
Cp: Counter Stack Control
Spp: Status Stack Control

TOPPCSTACK

A special version of the Register-to-Register Move instruction, Type 17, is provided for reading (and popping) or writing (and pushing) the top value of the PC stack. The normal POP PC instruction does not save the value popped from the stack, so to save this value into a register you must use the following special instruction:

```
reg = TOPPCSTACK;       {pop PC stack into reg}
                        {"toppcstack" may also be lowercase}
```

The PC stack is also popped by this instruction, after a one-cycle delay. A NOP should usually be placed after the special instruction, to allow the pop to occur properly:

```
reg = TOPPCSTACK;
NOP;                    {allow pop to occur correctly}
```

There is no standard PUSH PC stack instruction. To push a specific value onto the PC stack, therefore, use the following special instruction:

```
TOPPCSTACK = reg;       {push reg contents onto PC stack}
```

The stack is pushed immediately, in the same cycle.

Note that "TOPPCSTACK" may not be used as a register in any other instruction type!

Examples:

```
AX0 = TOPPCSTACK;       {pop PC stack into AX0}
NOP;

TOPPCSTACK = I7;        {push contents of I7 onto PC stack}
```

Only the following registers may be used in the special TOPPCSTACK instructions:

ALU, MAC, & Shifter Registers			*DAG Registers*					
AX0	AR	SI	I0	I4	M0	M4	L0	L4
AX1	MR0	SE	I1	I5	M1	M5	L1	L5
MX0	MR1	SR0	I2	I6	M2	M6	L2	L6
MX1	MR	SR1	I3	I7	M3	M7	L3	L7
AY0								
AY1								
MY0								
MY1								

There are several restrictions on the use of the special TOPPCSTACK instructions; they are described in Chapter 3, Program Control.

(instruction continues on next page)

Instruction Format:

TOPPCSTACK=reg
Internal Data Move, Instruction Type 17:

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	1	1	SRC RGP		1	1	1	1	SOURCE REG			

SRC RGP (Source Register Group) and SOURCE REG (Source Register) select the source register according to the Register Selection Table (see Appendix A).

reg=TOPPCSTACK
Internal Data Move, Instruction Type 17:

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		1	1	DEST REG				1	1	1	1

DST RGP (Destination Register Group) and DEST REG (Destination Register) select the destination register according to the Register Selection Table (see Appendix A).

Syntax:

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

Example: DIS AR_SAT, ENA M_MODE;

Description: Enables (ENA) or disables (DIS) the designated processor mode. The corresponding mode status bit in the mode status register (MSTAT) is set for ENA mode and cleared for DIS mode. At reset, MSTAT is set to zero, meaning that all modes are disabled. Any number of modes can be changed in one cycle with this instruction. Multiple ENA or DIS clauses must be separated by commas.

MSTAT Bits:

0	SEC_REG	Alternate Register Data Bank
1	BIT_REV	Bit-Reverse Mode on Address Generator #1
2	AV_LATCH	ALU Overflow Status Latch Mode
3	AR_SAT	ALU AR Register Saturation Mode
4	M_MODE	MAC Result Placement Mode
5	TIMER	Timer Enable
6	G_MODE	Enables GO Mode

The data register bank select bit (SEC_REG) determines which set of data registers is currently active (0=primary, 1=secondary).

The bit-reverse mode bit (BIT_REV), when set to 1, causes addresses generated by Data Address Generator #1 to be output in bit reversed order.

The ALU overflow latch mode bit (AV_LATCH), when set to 1, causes the AV bit in the arithmetic status register to stay set once an ALU overflow occurs. In this mode, if an ALU overflow occurs, the AV bit will be set and will remain set even if subsequent ALU operations do not generate overflows. The AV bit can only be cleared by writing a zero into it directly over the DMD bus.

(instruction continues on next page)

The AR saturation mode bit, (AR_SAT), when set to 1, causes the AR register to saturate if an ALU operation causes an overflow, as described in Chapter 2, "Computation Units."

The MAC result placement mode (M_MODE) determines whether or not the left shift is made between the multiplier product and the MR register.

Setting the Timer Enable bit (TIMER) starts the timer decrementing logic. Clearing it halts the timer.

The GO mode (G_MODE) allows an ADSP-21xx processor to continue executing instructions from internal memory (if possible) during a bus grant. 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.

Instruction Format:

Mode Control, Instruction Type 18:

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

TI: Timer Enable
AS: AR Saturation Mode Control
BR: Bit Reverse Mode Control
GM: GO Mode
MM: Multiplier Placement
OL: ALU Overflow Latch Mode Control
SR: Secondary Register Bank Mode

Syntax: MODIFY (I0 / I1 / I2 / I3 / I4 / I5 / I6 / I7 , M0 / M1 / M2 / M3 / M4 / M5 / M6 / M7);

Example: MODIFY (I1, M1);

Description: Add the selected M register (M_n) to the selected I register (I_m), then process the modified address through the modulus logic with buffer length as determined by the L register corresponding to the selected I register (L_m), and store the resulting address pointer calculation in the selected I register. The I register is modified as if an indexed memory address were taking place, but no actual memory data transfer occurs. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.**

The selection of the I and M registers is constrained to registers within the same Data Address Generator: selection of I0-I3 in Data Address Generator #1 constrains selection of the M registers to M0-M3. Similarly, selection of I4-I7 constrains the M registers to M4-M7.

Status Generated: None affected.

Instruction Format:
Modify Address Register, Instruction Type 21:

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	

G specifies which Data Address Generator is selected. The I and M registers specified must be from the same DAG, separated by the gray bar above. I specifies the I register (depends on which DAG is selected by the G bit). M specifies the M register (depends on which DAG is selected by the G bit).

NOP

Syntax: NOP ;

Description: No operation occurs for one cycle. Execution continues with the instruction following the NOP instruction.

Status Generated: None affected.

Instruction Format:
No operation, Instruction Type 30 (see Appendix A), as shown below:

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

INTERRUPT ENABLE & DISABLE
(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Syntax:
```
ENA INTS ;
DIS INTS ;
```

Description: Interrupts are enabled by default at reset. Executing the DIS INTS instruction causes all interrupts (including the powerdown interrupt) to be masked, without changing the contents of the IMASK register.

Executing the ENA INTS instruction allows all unmasked interrupts to be serviced again.

Note: Disabling interrupts does not affect serial port autobuffering or ADSP-218x DMA transfers (IDMA or BDMA). These operations will continue normally whether or not interrupts are enabled.

Status Generated: None affected.

Instruction Format:
DIS INTS, Instruction Type 26:

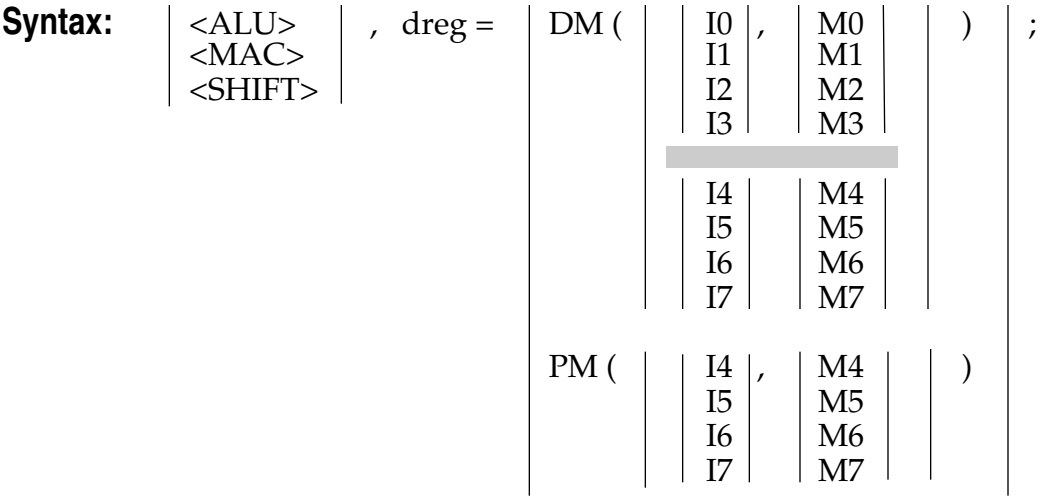
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

ENA INTS, Instruction Type 26:

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	1	1	0	0	0	0	0

15 MULTIFUNCTION COMPUTATION with MEMORY READ

Syntax:



Permissible dregs

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

Description: Perform the designated arithmetic operation and data transfer. The read operation moves the contents of the source to the destination register. The addressing mode when combining an arithmetic operation with a memory read is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, memory read second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (using the –s switch) the warning is not issued.

Because of the read-first, write-second characteristic of the processor, using the same register as source in one clause and a destination in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle.

For example,

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

is a legal version of this multifunction instruction and is not flagged by the assembler. Reversing the order of clauses, as in

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

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the data memory value is loaded into AX0 and then used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Using the same register as a destination in both clauses, however, produces an indeterminate result and should not be done. The assembler issues a warning unless semantics checking is turned off. Regardless of whether or not the warning is produced, however, this practice is not supported.

The following, therefore, is illegal and not supported, even though assembler semantics checking produces only a warning:

(3) AR = AX0 + AY0, AR = DM (I0, M0); *Illegal!*

(instruction continues on next page)

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV	Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS	Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:

ALU/MAC operation with Data Memory Read, Instruction Type 4:

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	0	Z	AMF					Yop			Xop			Dreg			I		M	

ALU/MAC operation with Program Memory Read, Instruction Type 5:

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	0	Z	AMF					Yop			Xop			Dreg			I		M	

Shift operation with Data Memory Read, Instruction Type 12:

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	0			SF			Xop			Dreg			I		M	

Shift operation with Program Memory Read, Instruction Type 13:

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	0			SF			Xop			Dreg			I		M	

Z:	Result register	Dreg:	Destination register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand
G:	Data Address Generator	I:	Indirect address register
M:	Modify register		

15 MULTIFUNCTION COMPUTATION with REGISTER to REGISTER MOVE

Syntax:

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

Permissible dregs

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

Description: Perform the designated arithmetic operation and data transfer. The contents of the source are always right-justified in the destination register after the read.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, register transfer second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (–s switch) the warning is not issued.

Because of the read-first, write-second characteristic of the processor, using the same register as source in one clause and a destination in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle.

For example,

(1) AR = AX0 + AY0, AX0 = MR1;

is a legal version of this multifunction instruction and is not flagged by the assembler. Reversing the order of clauses, as in

(2) AX0 = MR1, AR = AX0 + AY0;

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the MR1 register value is loaded into AX0 and then AX0 is used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Using the same register as a destination in both clauses, however, produces an indeterminate result and should not be done. The assembler issues a warning unless semantics checking is turned off. Regardless of whether or not the warning is produced, however, this practice is not supported.

The following, therefore, is illegal and not supported, even though assembler semantics checking produces only a warning:

(3) AR = AX0 + AY0, AR = MR1; *Illegal!*

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

(instruction continues on next page)

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV — Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS — Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:

ALU/MAC operation with Data Register Move, Instruction Type 8:

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			Dreg dest				Dreg source			

Shift operation with Data Register Move, Instruction Type 14:

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			Dreg dest				Dreg source			

Z:	Result register	Dreg:	Data register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand

Syntax:

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

Permissible dregs

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

Description: Perform the designated arithmetic operation and data transfer. The write operation moves the contents of the source to the specified memory location. The addressing mode when combining an arithmetic operation with a memory write is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register.

The computation must be unconditional. All ALU, MAC and Shifter operations are permitted except Shift Immediate and ALU DIVS and DIVQ instructions.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (memory write first, computation second) is intended to imply this. In fact, you may code this instruction with the order of clauses reversed. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (–s switch) the warning is not issued.

(instruction continues on next page)

Because of the read-first, write-second characteristic of the processor, using the same register as destination in one clause and a source in the other is legal. The register supplies the value present at the beginning of the cycle and is written with the new value at the end of the cycle.

For example,

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

is a legal version of this multifunction instruction and is not flagged by the assembler. Reversing the order of clauses, as in

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

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the result of the computation in AR is then written to memory, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Status Generated: All status bits are affected in the same way as for the single function versions of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV — Set if the accumulated product overflows the lower-order 32 bits of the MR register. Cleared otherwise.

<SHIFT> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	*	-	-	-	-	-	-	-

SS — Affected only when executing the EXP operation; set if the source operand is negative. Cleared if the number is positive.

Instruction Format:

ALU/MAC operation with Data Memory Write, Instruction Type 4:

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	1	Z	AMF	Yop	Xop	Dreg	I M

ALU/MAC operation with Program Memory Write, Instruction Type 5:

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	1	Z	AMF	Yop	Xop	Dreg	I M

(instruction continues on next page)

Shift operation with Data Memory Write, Instruction Type 12:

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	1	SF				Xop			Dreg				I		M	

Shift operation with Program Memory Write, Instruction Type 13:

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	1	SF				Xop			Dreg				I		M	

Z:	Result register	Dreg:	Destination register
SF:	Shifter operation	AMF:	ALU/MAC operation
Yop:	Y operand	Xop:	X operand
I:	Indirect address register	M:	Modify register
G:	Data Address Generator; I & M registers must be from the same DAG, as separated by the gray bar in the Syntax description.		

Syntax:

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

Description: Perform the designated memory reads, one from data memory and one from program memory. Each read operation moves the contents of the memory location to the destination register. For this double data fetch, the destinations for data memory reads are the X registers in the ALU and the MAC, and the destinations for program memory reads are the Y registers. The addressing mode for this memory read is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register.

A multifunction instruction requires three items to be fetched from memory: the instruction itself and two data words. No extra cycle is needed to execute the instruction as long as only one of the fetches is from external memory.

If two off-chip accesses are required, however—the instruction fetch and one data fetch, for example, or data fetches from both program and data memory—then one overhead cycle occurs. In this case the program memory access occurs first, then the data memory access. If three off-chip accesses are required—the instruction fetch as well as data fetches from both program and data memory—then two overhead cycles occur.

Status Generated: No status bits are affected.

Instruction Format:

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

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	0 0 0 0 0	PM I	PM M	DM I	DM M

AMF specifies the ALU or MAC function. In this case, AMF = 00000, designating a no-operation for the ALU or MAC function.

PD: Program Destination register
DD: Data Destination register
AMF: ALU/MAC operation
I: Indirect address register
M: Modify register

ALU / MAC with DATA & PROGRAM MEMORY READ

Syntax:

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

Description: This instruction combines an ALU or a MAC operation with a data memory read and a program memory read. The read operations move the contents of the memory location to the destination register. For this double data fetch, the destinations for data memory reads are the X registers in the ALU and the MAC, and the destinations for program memory reads are the Y registers. The addressing mode is register indirect with post-modify. **For linear (i.e. non-circular) indirect addressing, the L register corresponding to the I register used must be set to zero.** The contents of the source are always right-justified in the destination register after the read.

A multifunction instruction requires three items to be fetched from memory: the instruction itself and two data words. No extra cycle is needed to execute the instruction as long as only one of the fetches is from external memory.

If two off-chip accesses are required, however—the instruction fetch and one data fetch, for example, or data fetches from both program and data memory—then one overhead cycle occurs. In this case the program memory access occurs first, then the data memory access. If three off-chip accesses are required—the instruction fetch as well as data fetches from both program and data memory—then two overhead cycles occur.

The computation must be unconditional. All ALU and MAC operations are permitted except the DIVS and DIVQ instructions. The results of the computation must be written into the R register of the computational unit; ALU results to AR, MAC results to MR.

The fundamental principle governing multifunction instructions is that registers (and memory) are read at the beginning of the processor cycle and written at the end of the cycle. The normal left-to-right order of clauses (computation first, memory reads second) is intended to imply this. In fact, you may code this instruction with the order of clauses altered. The assembler produces a warning, but the results are identical at the opcode level. If you turn off semantics checking in the assembler (–s switch) the warning is not issued.

The same data register may be used as a source for the arithmetic operation and as a destination for the memory read. The register supplies the value present at the beginning of the cycle and is written with the value from memory at the end of the cycle.

For example,

(1) MR=MR+MX0*MY0(UU), MX0=DM(I0, M0), MY0=PM(I4,M4);

is a legal version of this multifunction instruction and is not flagged by the assembler. Changing the order of clauses, as in

(2) MX0=DM(I0, M0), MY0=PM(I4,M4), MR=MR+MX0*MY0(UU);

results in an assembler warning, but assembles and executes exactly as the first form of the instruction. Note that reading example (2) from left to right may suggest that the data memory value is loaded into MX0 and MY0 and subsequently used in the computation, all in the same cycle. In fact, this is not possible. The left-to-right logic of example (1) suggests the operation of the instruction more closely. Regardless of the apparent logic of reading the instruction from left to right, the read-first, write-second operation of the processor determines what actually happens.

Status Generated: All status bits are affected in the same way as for the single operation version of the selected arithmetic operation.

<ALU> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	-	-	*	*	*	*	*

AZ	Set if result equals zero. Cleared otherwise.
AN	Set if result is negative. Cleared otherwise.
AV	Set if an overflow is generated. Cleared otherwise.
AC	Set if a carry is generated. Cleared otherwise.
AS	Affected only when executing the Absolute Value operation (ABS). Set if the source operand is negative.

(instruction continues on next page)

ALU / MAC with DATA & PROGRAM MEMORY READ

<MAC> operation

ASTAT:	7	6	5	4	3	2	1	0
	SS	MV	AQ	AS	AC	AV	AN	AZ
	-	*	-	-	-	-	-	-

MV Set if the accumulated product overflows the lower-order 32-bits of the MR register. Cleared otherwise.

Instruction Format:

ALU/MAC with Data and Program Memory Read, Instruction Type 1:

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	PMI	PMM	DMI	DMM

PD: Program Destination register
DD: Data Destination register
AMF: ALU/MAC operation
M: Modify register
Yop: Y operand
Xop: X operand
I: Indirect address register

Instruction Coding A

A.1 OPCODES

This appendix gives a summary of the complete instruction set of the ADSP-2100 family processors. Opcode field names are defined at the end of the appendix. 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			

Generate ALU Status (NONE = <ALU>) *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

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	0	AMF						Yop			Xop		1	0	1	0	1	0	1	0

ALU codes only

Type 9: Conditional ALU / MAC

xop * yop

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		

xop * xop *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

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						0	0		Xop		0	0	0	1	COND		

xop AND/OR/XOR constant *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

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						YY			Xop		CC		BO		COND		

BO, CC, and YY specify the constant according the table shown at the end of this appendix.

PASS constant (constant ≠ 0,1,–1) *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

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						YY			Xop		CC		BO		COND		

Instruction Coding A

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		

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 DREG				Source DREG			

Type 15: Shift Immediate

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

Type 16: Conditional Shift

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

A Instruction Coding

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

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:	GO Mode; enable means execute internal code if possible
TI:	Timer enable

11 = Enable Mode
10 = Disable Mode
0 1 = no change
0 0 = no change

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			

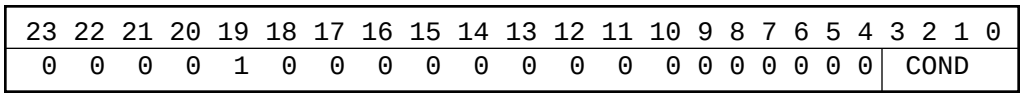
Type 20: Conditional Return

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			

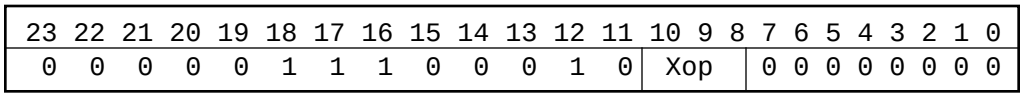
Type 21: Modify Address Register

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	

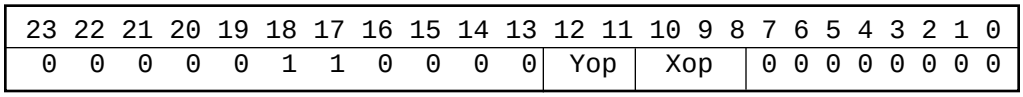
Type 22: Reserved



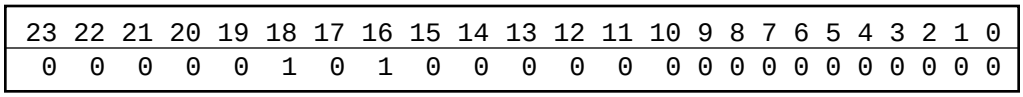
Type 23: DIVQ



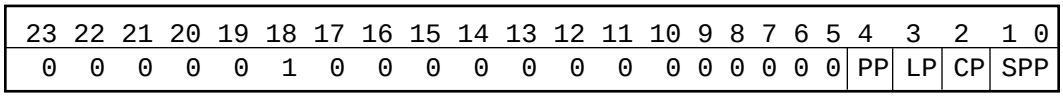
Type 24: DIVS



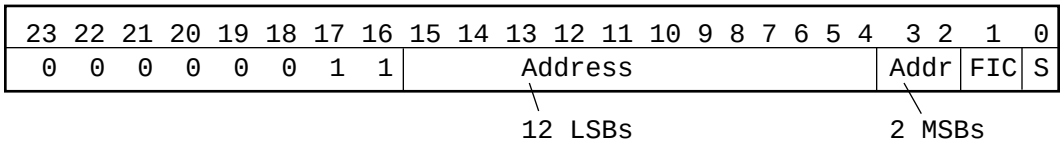
Type 25: Saturate MR



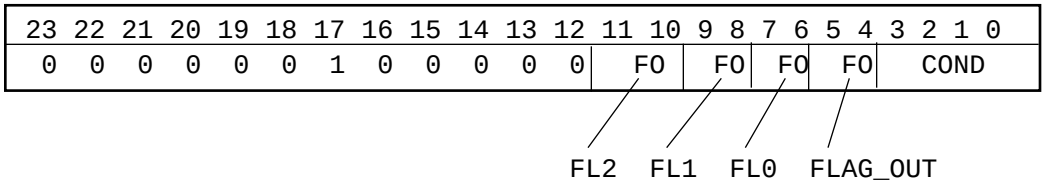
Type 26: Stack Control



Type 27: Call or Jump on Flag In



Type 28: Modify Flag Out



A Instruction Coding

Type 29: I/O Memory Space Read/Write *(ADSP-218x only)*

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	D	ADDR											DREG			

Type 30: No Operation (NOP)

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

Type 31: Idle (n) *(Slow 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	DV			

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	PASS 1 when y = 0
`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	X + C – 1 when y = 0
`1 0 1 1 1`	X – Y	
`1 1 0 0 0`	Y – 1	PASS –1 when y = 0
`1 1 0 0 1`	Y – X	– X when y = 0
`1 1 0 1 0`	Y – X + C – 1	–X + C – 1 when y = 0
`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	

A Instruction Coding

BO see **YY, CC, BO** at the end of this appendix

CC see **YY, CC, BO** at the end of this appendix

COND Status Condition codes

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

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

DV Divisor codes for Slow Idle instruction (*IDLE (n)*)

0 0 0 0	Normal Idle instruction (Divisor=0)
0 0 0 1	Divisor=16
0 0 1 0	Divisor=32
0 1 0 0	Divisor=64
1 0 0 0	Divisor=128

FIC FI condition code

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

FO Control codes for Flag Output Pins (FO, FL0, FL1, FL2)

.

0 0	No change
0 1	Toggle
1 0	Reset
1 1	Set

A Instruction Coding

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

PD Dual 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

REG Register codes

Codes not assigned are reserved.

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 (write only)
1 1 1 0	SR0	–	–	–
1 1 1 1	SR1	–	–	–

S Jump/Call codes

0	Jump
1	Call

A Instruction Coding

SF Shifter Function codes

0 0 0 0	LSHIFT	(HI)
0 0 0 1	LSHIFT	(HI, OR)
0 0 1 0	LSHIFT	(LO)
0 0 1 1	LSHIFT	(LO, OR)
0 1 0 0	ASHIFT	(HI)
0 1 0 1	ASHIFT	(HI, OR)
0 1 1 0	ASHIFT	(LO)
0 1 1 1	ASHIFT	(LO, OR)
1 0 0 0	NORM	(HI)
1 0 0 1	NORM	(HI, OR)
1 0 1 0	NORM	(LO)
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

TERM Termination codes for DO UNTIL

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

X X Operand codes

0 0 0	X0 (SI for shifter)
0 0 1	X1 (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

A Instruction Coding

YY see **YY, CC, BO** below

Z ALU/MAC Result Register codes

`0`	Result register
`1`	Feedback register

YY, CC, BO ALU / MAC Constant codes (Type 9)

(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Constant (hex)	*YY*	*CC*	*BO*	*Bit #*
0001	00	00	01	bit 0
0002	00	01	01	bit 1
0004	00	10	01	bit 2
0008	00	11	01	bit 3
0010	01	00	01	bit 4
0020	01	01	01	bit 5
0040	01	10	01	bit 6
0080	01	11	01	bit 7
0100	10	00	01	bit 8
0200	10	01	01	bit 9
0400	10	10	01	bit 10
0800	10	11	01	bit 11
1000	11	00	01	bit 12
2000	11	01	01	bit 13
4000	11	10	01	bit 14
8000	11	11	01	bit 15
FFFE	00	00	11	! bit 0
FFFD	00	01	11	! bit 1
FFFB	00	10	11	! bit 2
FFF7	00	11	11	! bit 3
FFEF	01	00	11	! bit 4
FFDF	01	01	11	! bit 5
FFBF	01	10	11	! bit 6
FF7F	01	11	11	! bit 7
FEFF	10	00	11	! bit 8
FDFF	10	01	11	! bit 9
FBFF	10	10	11	! bit 10
F7FF	10	11	11	! bit 11
EFFF	11	00	11	! bit 12
DFFF	11	01	11	! bit 13
BFFF	11	10	11	! bit 14
7FFF	11	11	11	! bit 15

Division Exceptions B

B.1 DIVISION FUNDAMENTALS

The ADSP-2100 family processors' 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.

B Division Exceptions

The DIVS primitive is executed once, with the proper operands (ex. DIVS AY1, AX0) to compute the sign of the quotient. The sign bit of the quotient is determined by XORing (exclusive-or) the sign bits of each operand. The entire 32-bit dividend is shifted left one bit. The lower 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 remainder 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 binary 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!

Division Exceptions B

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 (0x4000) as the dividend and .25 (0x2000) 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 CONDITIONS

Although the divide primitives for the ADSP-2100 family work correctly 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 0x7FFF. If the divisor in an unsigned division exceeds 0x7FFF, an invalid quotient will be generated.

B.2.1 Negative Divisor Error

The quotient produced by a divide with a negative divisor will generally be one LSB less than the correct result. The divide algorithm implemented on the ADSP-2100 family does not correctly compensate for the twos-complement format of a negative number, causing this inaccuracy.

B Division Exceptions

There is one case where this discrepancy does not occur. If the result of the division operation should equal 0x8000, then it will be correctly represented, and not be one LSB off.

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

One is to avoid division by negative numbers. 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, and 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 0x7FFF. 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.

Division Exceptions B

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 0x8000. Attempting to take the absolute value of 0x8000 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 0x8000, 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 0x8000, 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 0x8000, 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.

B Division Exceptions

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 (> 0x7FFF) 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-2100 family. The code is organized in sections. This entire module can
be used to handle all error conditions, or individual sections can be removed to
increase execution speed.

   Entry Points
   signed_div Computes 16-bit signed quotient
   unsigned_div Computes 16-bit unsigned quotient

   Calling Parameters
   AX0 = 16-bit divisor
   AY0 = Lower 16 bits of dividend
   AY1 = Upper 16 bits of dividend
```

```
   Return Values
   AR = 16-bit quotient
   MR0 = 16-bit remainder
   AV flag set if divide would overflow

   Altered Registers
   AX0, AX1, AR, AF, AY0, AY1, MR, MY0

   Computation Time: 30 cycles
}

.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=0x4;                  {Divide by zero, so overflow}
                RTS;                        {Return to calling program}

test_2:         IF NOT AV JUMP test_3;      {If divisor 0x8000, then the}
                AY0=AY1, AF=ABS AY1;        {quotient is simply -AY1}
                IF NOT AV JUMP recover_sign;
                ASTAT=0x4;                  {0x8000 divided by 0x8000,}
                RTS;                        {so overflow}

test_3:         AF=PASS AF;                 {Check for division overflow}
                IF NE JUMP test_4;          {Not equal, jump test 4}
                AY0=0x8000;                 {Quotient equals -1}
                ASTAT=0x0;                  {Clear AS bit of ASTAT}
                JUMP recover_sign;          {Compute remainder}

test_4:         AF=ABS MR1;                 {Get absolute of dividend}
                AR=ABS AX0;                 {Restore AS bit of ASTAT}
                AF=AF-AR;                   {Check for division overflow}
                IF LT JUMP do_divs;         {If Divisor>Dividend do divide}
                ASTAT=0x4;                  {Division overflow}
                RTS;
```

Listing B.1 Division Error Routine

(continues on next page)

B Division Exceptions

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

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

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

test_10:        IF GT JUMP test_11;         {No, so check overflow}
                ASTAT=0x4;                  {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=0x4;                  {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 MYO, Quotient in AR}
                MR=MR-AR*MY0 (UU);          {Determine remainder}
                RTS;                        {Return to calling program}

.ENDMOD;
```

Listing B.1 Division Error Routine

Numeric Formats C

C.1 OVERVIEW

ADSP-2100 family processors support 16-bit fixed-point data in hardware. Special features in the computation units allow you to support other formats in software. This appendix describes various aspects of the 16-bit data format. It also describes how to implement a block floating-point format in software.

C.2 UNSIGNED OR SIGNED: TWOS-COMPLEMENT FORMAT

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.

Signed numbers supported by the ADSP-2100 family are in twos-complement format. Signed-magnitude, ones-complement, BCD or excess-n formats are not supported.

C.3 INTEGER OR FRACTIONAL

The ADSP-2100 family supports both fractional and integer data formats, with the exception that the ADSP-2100 processor does not perform integer multiplication. In an integer, the radix point is assumed to lie to the right of the LSB, so that all magnitude bits have a weight of 1 or greater. This format is shown in Figure C.1, which can be found on the following page. Note that in twos-complement format, the sign bit has a negative weight.

C Numeric Formats

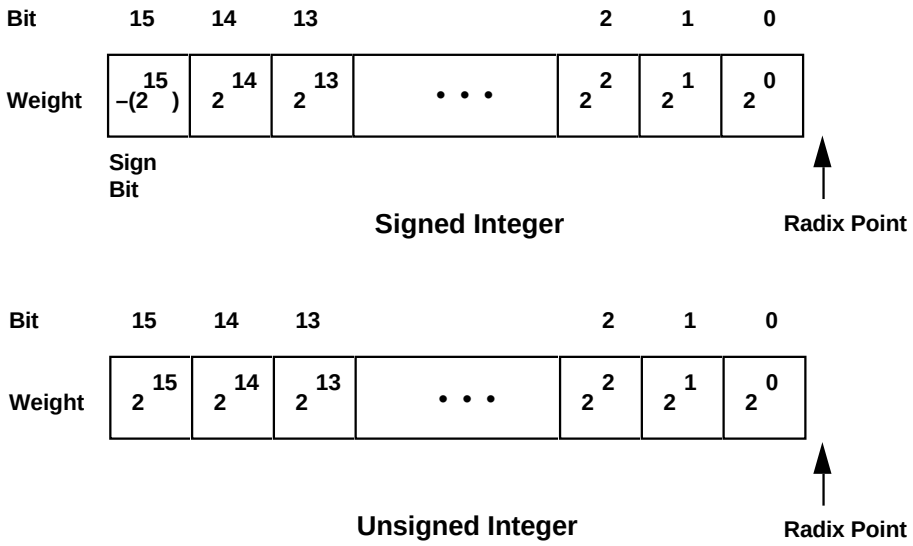


Figure C.1 Integer Format

In a fractional format, the assumed radix point lies within the number, so that some or all of the magnitude bits have a weight of less than 1. In the format shown in Figure C.2, the assumed radix point lies to the left of the 3 LSBs, and the bits have the weights indicated.

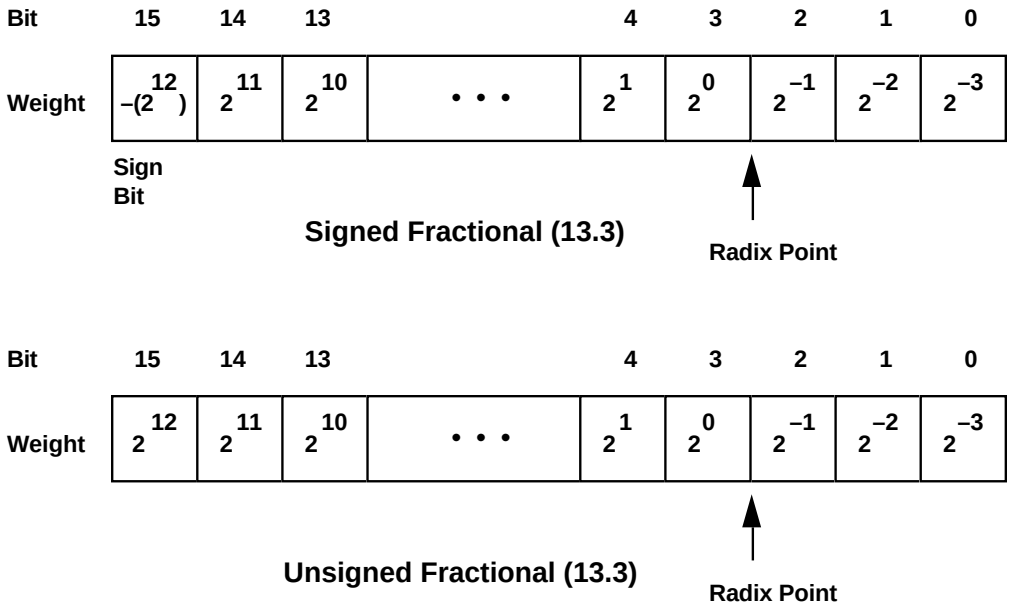


Figure C.2 Example Of Fractional Format

Numeric Formats C

The notation used to describe a format consists two numbers separated by a period (.); the first number is the number of bits to the left of radix point, the second is the number of bits to the right of the radix point. For example, 16.0 format is an integer format; all bits lie to the left of the radix point. The format in Figure C.2 is 13.3.

Table C.1 shows the ranges of numbers representable in the fractional formats that are possible with 16 bits.

Format	Number of Integer Bits	Number of Fractional Bits	Largest Positive Value (0x7FFF) In Decimal	Largest Negative Value (0x8000) In Decimal	Value of 1 LSB (0x0001) In Decimal
1.15	1	15	0.999969482421875	–1.0	0.000030517578125
2.14	2	14	1.999938964843750	–2.0	0.000061035156250
3.13	3	13	3.999877929687500	–4.0	0.000122070312500
4.12	4	12	7.999755859375000	–8.0	0.000244140625000
5.11	5	11	15.999511718750000	–16.0	0.000488281250000
6.10	6	10	31.999023437500000	–32.0	0.000976562500000
7.9	7	9	63.998046875000000	–64.0	0.001953125000000
8.8	8	8	127.996093750000000	–128.0	0.003906250000000
9.7	9	7	255.992187500000000	–256.0	0.007812500000000
10.6	10	6	511.984375000000000	–512.0	0.015625000000000
11.5	11	5	1023.968750000000000	–1024.0	0.031250000000000
12.4	12	4	2047.937500000000000	–2048.0	0.062500000000000
13.3	13	3	4095.875000000000000	–4096.0	0.125000000000000
14.2	14	2	8191.750000000000000	–8192.0	0.250000000000000
15.1	15	1	16383.500000000000000	–16384.0	0.500000000000000
16.0	16	0	32767.000000000000000	–32768.0	1.000000000000000

Table C.1 Fractional Formats And Their Ranges

C.4 BINARY MULTIPLICATION

In addition and subtraction, both operands must be in the same format (signed or unsigned, radix point in the same location) and the result format is the same as the input format. Addition and subtraction are performed the same way whether the inputs are signed or unsigned.

In multiplication, however, the inputs can have different formats, and the result depends on their formats. The ADSP-2100 family assembly language allows you to specify whether the inputs are both signed, both unsigned, or one of each (mixed-mode). The location of the radix point in the result can be derived from its location in each of the inputs. This is

shown in Figure C.3. The product of two 16-bit numbers is a 32-bit number. If the inputs' formats are M.N and P.Q, the product has the format (M+P).(N+Q). For example, the product of two 13.3 numbers is a 26.6 number. The product of two 1.15 numbers is a 2.30 number.

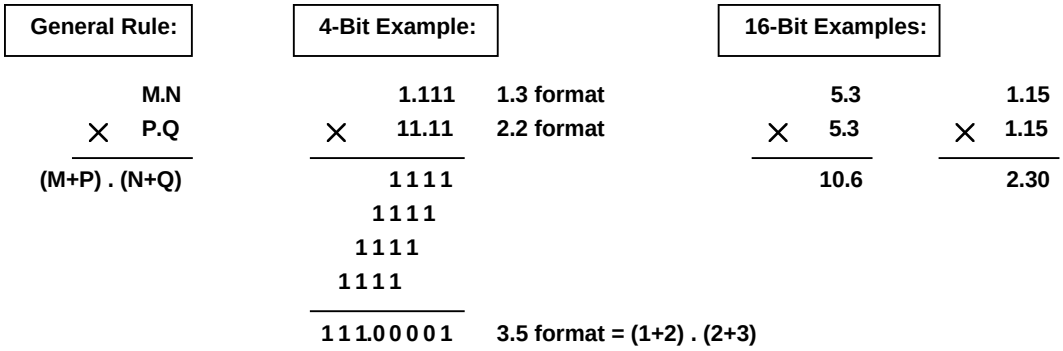


Figure C.3 Format Of Multiplier Result

C.4.1 Fractional Mode And Integer Mode

A product of 2 twos-complement numbers has two sign bits. Since one of these bits is redundant, you can shift the entire result left one bit. Additionally, if one of the inputs was a 1.15 number, the left shift causes the result to have the same format as the other input (with 16 bits of additional precision). For example, multiplying a 1.15 number by a 5.11 number yields a 6.26 number. When shifted left one bit, the result is a 5.27 number, or a 5.11 number plus 16 LSBs.

The ADSP-2100 family provides a mode (called the fractional mode) in which the multiplier result is always shifted left one bit before being written to the result register. (On the ADSP-2100 processor, this mode is always active; on other processors, the left shift can be omitted.) This left shift eliminates the extra sign bit when both operands are signed, yielding a correctly formatted result.

When both operands are in 1.15 format, the result is 2.30 (30 fractional bits). A left shift causes the multiplier result to be 1.31 which can be rounded to 1.15. Thus, if you use a fractional data format, it is most convenient to use the 1.15 format.

In the integer mode, the left shift does not occur. This is the mode to use if both operands are integers (in the 16.0 format). The 32-bit multiplier result is in 32.0 format, also an integer. On the ADSP-2100 only, the integer mode

is not available; the 32.0 result gets shifted to 31.1 format. Because the MSB is still available in the 40-bit accumulator, a right shift can correct the result.

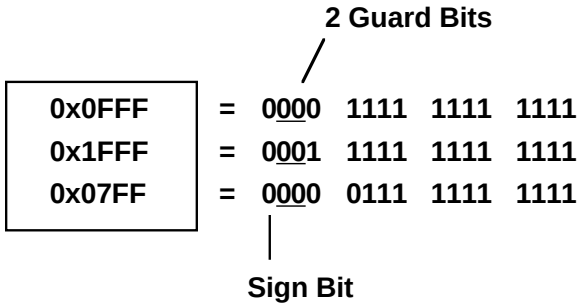
In all processors other than the ADSP-2100, fractional and integer modes are controlled by a bit in the MSTAT register. At reset, these processors default to the fractional mode, for compatibility with the ADSP-2100.

C.5 BLOCK FLOATING-POINT FORMAT

A block floating-point format enables a fixed-point processor to gain some of the increased dynamic range of a floating-point format without the overhead needed to do floating-point arithmetic. Some additional programming is required to maintain a block floating-point format, however.

A floating-point number has an exponent that indicates the position of the radix point in the actual value. In block floating-point format, a set (block) of data values share a common exponent. To convert a block of fixed-point values to block floating-point format, you would shift each value left by the same amount and store the shift value as the block exponent.

Typically, block floating-point format allows you to shift out non-significant MSBs, increasing the precision available in each value. You can also use block floating-point format to eliminate the possibility of a data value overflowing. Figure C.4 shows an example. The three data samples each have at least 2 non-significant, redundant sign bits. Each data value



To detect bit growth into 2 guard bits, set SB=–2

Figure C.4 Data With Guard Bits

can grow by these two bits (two orders of magnitude) before overflowing; thus, these bits are called *guard* bits. If it is known that a process will not cause any value to grow by more than these two bits, then the process can be run without loss of data. Afterward, however, the block must be adjusted to replace the guard bits before the next process.

Figure C.5 shows the data after processing but before adjustment. The block floating-point adjustment is performed as follows. Initially, the value of SB is –2, corresponding to the 2 guard bits. During processing, each resulting data value is inspected by the EXPADJ instruction, which counts the number of redundant sign bits and adjusts SB is if the number of redundant sign bits is less than 2. In this example, SB=–1 after processing, indicating that the block of data must be shifted right one bit to maintain the 2 guard bits. If SB were 0 after processing, the block would have to be shifted two bits right. In either case, the block exponent is updated to reflect the shift.

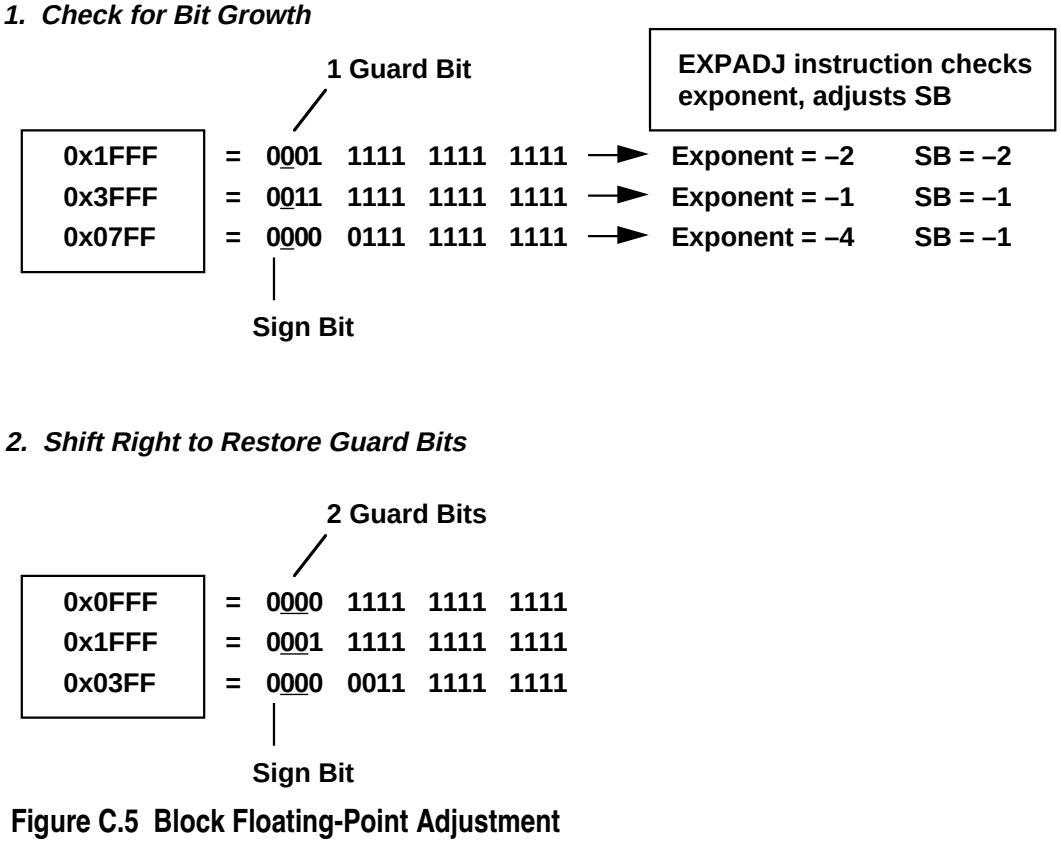


Figure C.5 Block Floating-Point Adjustment

Interrupt Vector Addresses D

D.1 INTERRUPT VECTOR ADDRESSES

Tables D.1–D.6 show the interrupts and associated vector addresses for each processor of the ADSP-2100 family. Note that SPORT1 can be configured as either a serial port or as a collection of control pins including two external interrupt inputs, $\overline{IRQ0}$ and $\overline{IRQ1}$.

The interrupt vector locations are spaced four program memory locations apart—this allows short interrupt service routines to be coded in place, with no jump to the service routine required. For interrupt service routines with more than four instructions, however, program control must be transferred to the service routine by means of a jump instruction placed at the interrupt vector location.

Interrupt Source	*Interrupt Vector Address*
$\overline{RESET}$ startup	0x0000
$\overline{IRQ2}$	0x0004 *(highest priority)*
SPORT0 Transmit	0x0008
SPORT0 Receive	0x000C
SPORT1 Transmit *or* $\overline{IRQ1}$	0x0010
SPORT1 Receive *or* $\overline{IRQ0}$	0x0014
Timer	0x0018 *(lowest priority)*

Table D.1 ADSP-2101/2115 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{RESET}$ startup	0x0000
$\overline{IRQ2}$	0x0004 *(highest priority)*
SPORT1 Transmit *or* $\overline{IRQ1}$	0x0010
SPORT1 Receive *or* $\overline{IRQ0}$	0x0014
Timer	0x0018 *(lowest priority)*

Table D.2 ADSP-2105 Interrupts & Interrupt Vector Addresses

D Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup	0x0000
$\overline{\text{IRQ2}}$	0x0004 *(highest priority)*
HIP Write (from Host)	0x0008
HIP Read (to Host)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0018
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x001C
Timer	0x0020 *(lowest priority)*

Table D.3 ADSP-2111 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup (or powerup w/PUCR=1)	0x0000 *(highest priority)*
Powerdown (non-maskable)	0x002C
$\overline{\text{IRQ2}}$	0x0004
HIP Write (from Host)	0x0008
HIP Read (to Host)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
Software Interrupt 1	0x0018
Software Interrupt 2	0x001C
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0020
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0024
Timer	0x0028 *(lowest priority)*

Table D.4 ADSP-2171 Interrupts & Interrupt Vector Addresses

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup (or powerup w/PUCR=1)	0x0000 *(highest priority)*
Powerdown (non-maskable)	0x002C
$\overline{\text{IRQ2}}$	0x0004
$\overline{\text{IRQL1}}$ (level-sensitive)	0x0008
$\overline{\text{IRQL0}}$ (level-sensitive)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
$\overline{\text{IRQE}}$ (edge-sensitive)	0x0018
Byte DMA (BDMA) Interrupt	0x001C
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0020
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0024
Timer	0x0028 *(lowest priority)*

Table D.5 ADSP-2181 Interrupts & Interrupt Vector Addresses

Interrupt Vector Addresses D

Interrupt Source	*Interrupt Vector Address*
$\overline{\text{RESET}}$ startup (or powerup w/PUCR=1)	0x0000 *(highest priority)*
Powerdown (non-maskable)	0x002C
$\overline{\text{IRQ2}}$	0x0004
HIP Write (from Host)	0x0008
HIP Read (to Host)	0x000C
SPORT0 Transmit	0x0010
SPORT0 Receive	0x0014
Analog (DAC) Transmit	0x0018
Analog (ADC) Receive	0x001C
SPORT1 Transmit *or* $\overline{\text{IRQ1}}$	0x0020
SPORT1 Receive *or* $\overline{\text{IRQ0}}$	0x0024
Timer	0x0028 *(lowest priority)*

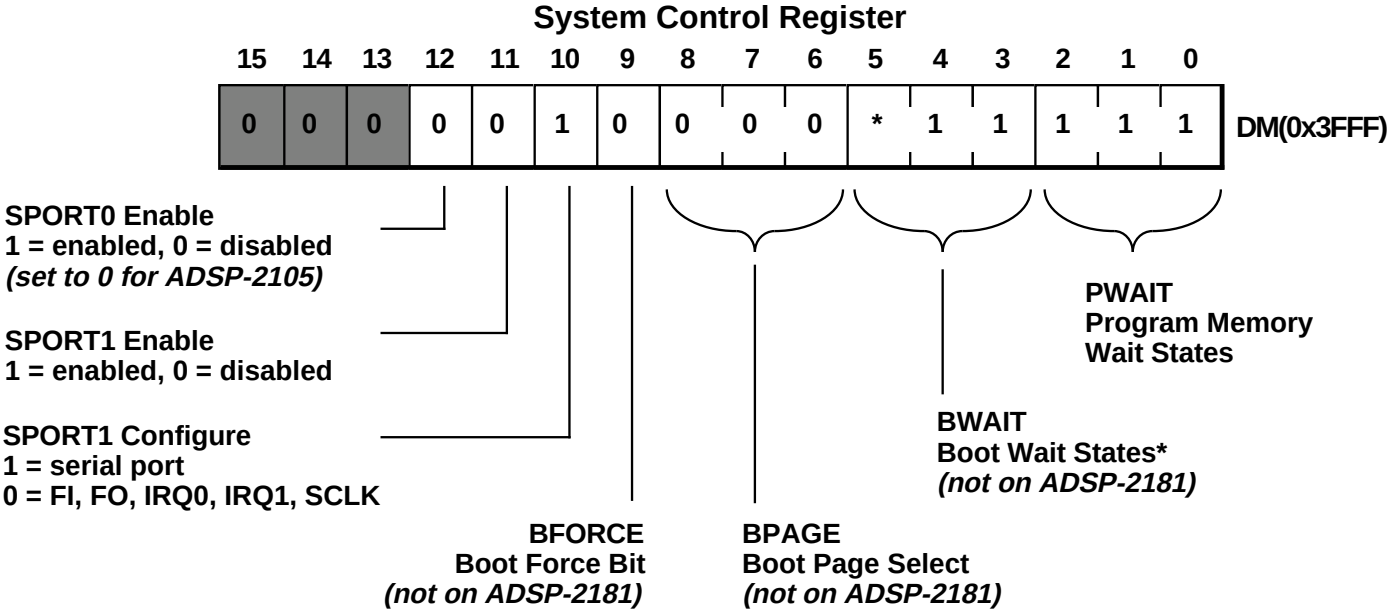
Table D.6 ADSP-21msp58/59 Interrupts & Interrupt Vector Addresses

Control/Status Registers E

E.1 OVERVIEW

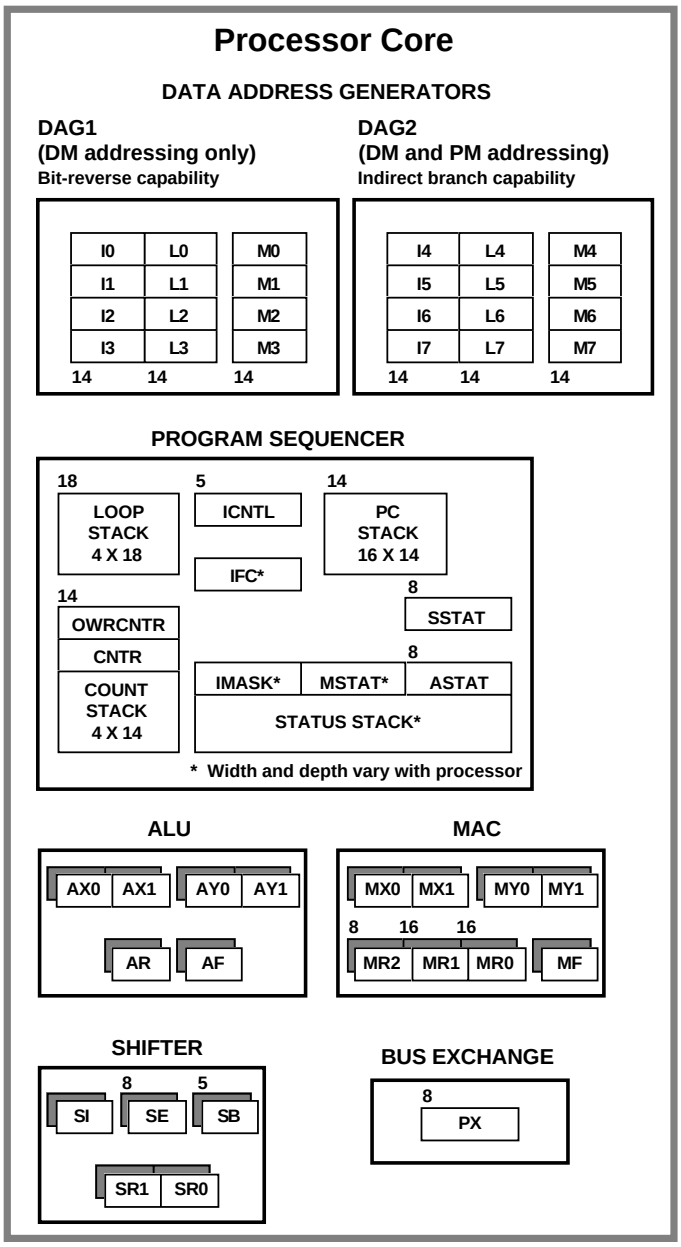
This appendix shows bit definitions for 1) the memory-mapped control registers and 2) other (non-memory-mapped) control and status registers of all ADSP-21xx processors. The memory-mapped registers are listed in descending address order. Default bit values at reset are shown; if no value is shown, the bit is undefined at reset. Reserved bits are shown on a gray field. These bits should always be written with zeros.

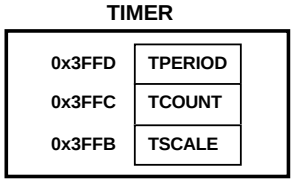
Memory-Mapped Registers

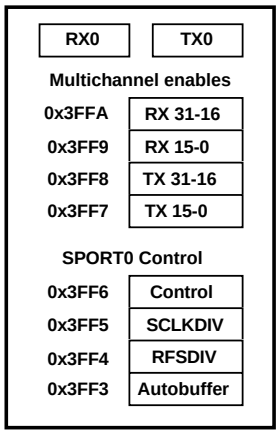


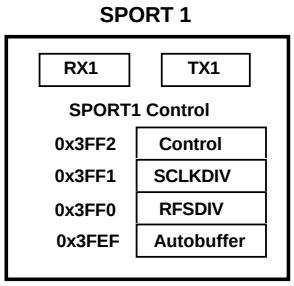
*** Bit 5 initialized to 1 on ADSP-2171, ADSP-21msp58/59**
Bit 5 initialized to 0 on ADSP-2101, ADSP-2105, ADSP-2115, ADSP-2111

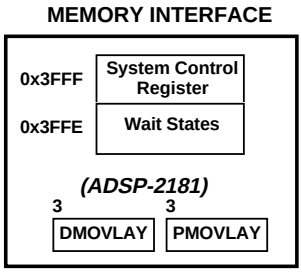
E Control/Status Registers

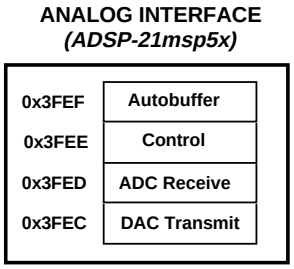


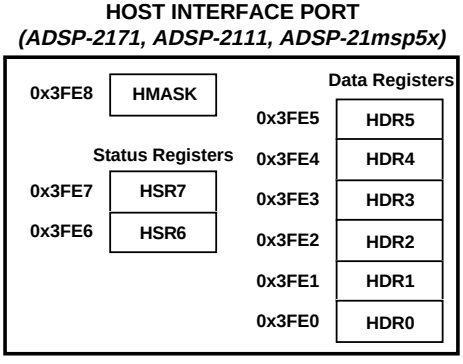


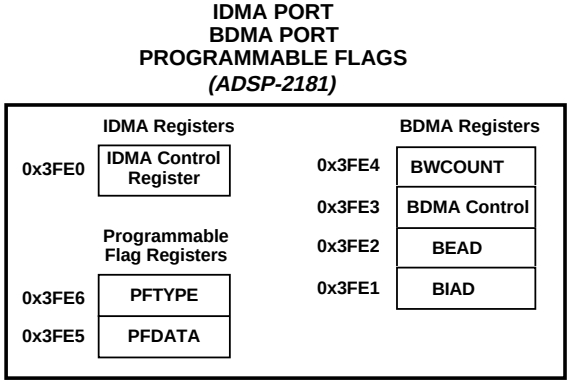








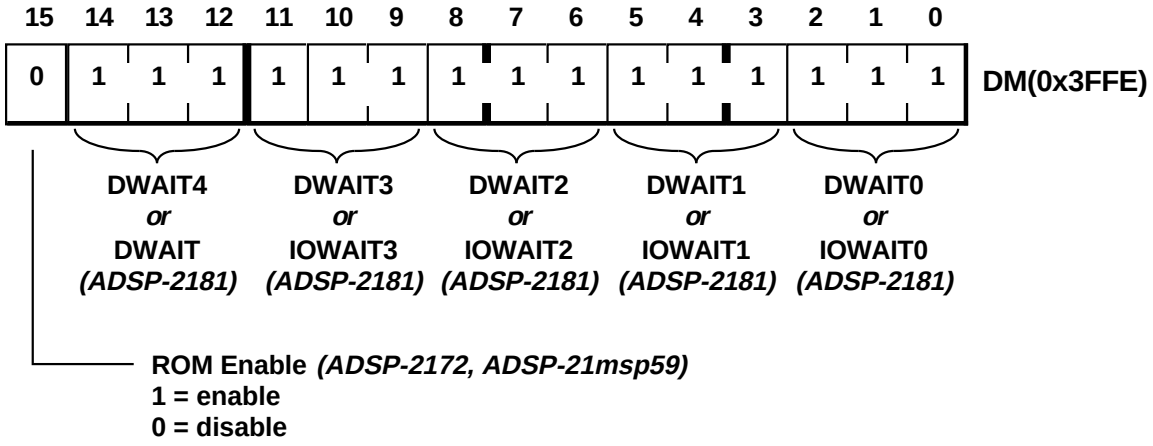




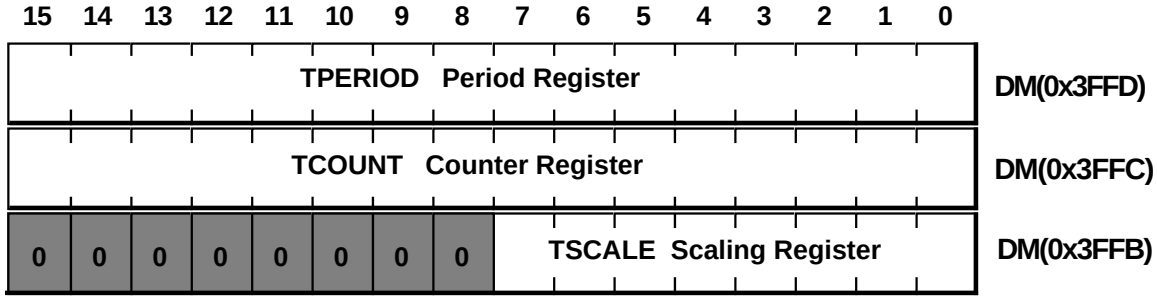
Control/Status Registers E

Memory-Mapped Registers

Waitstate Control Register



Timer Registers

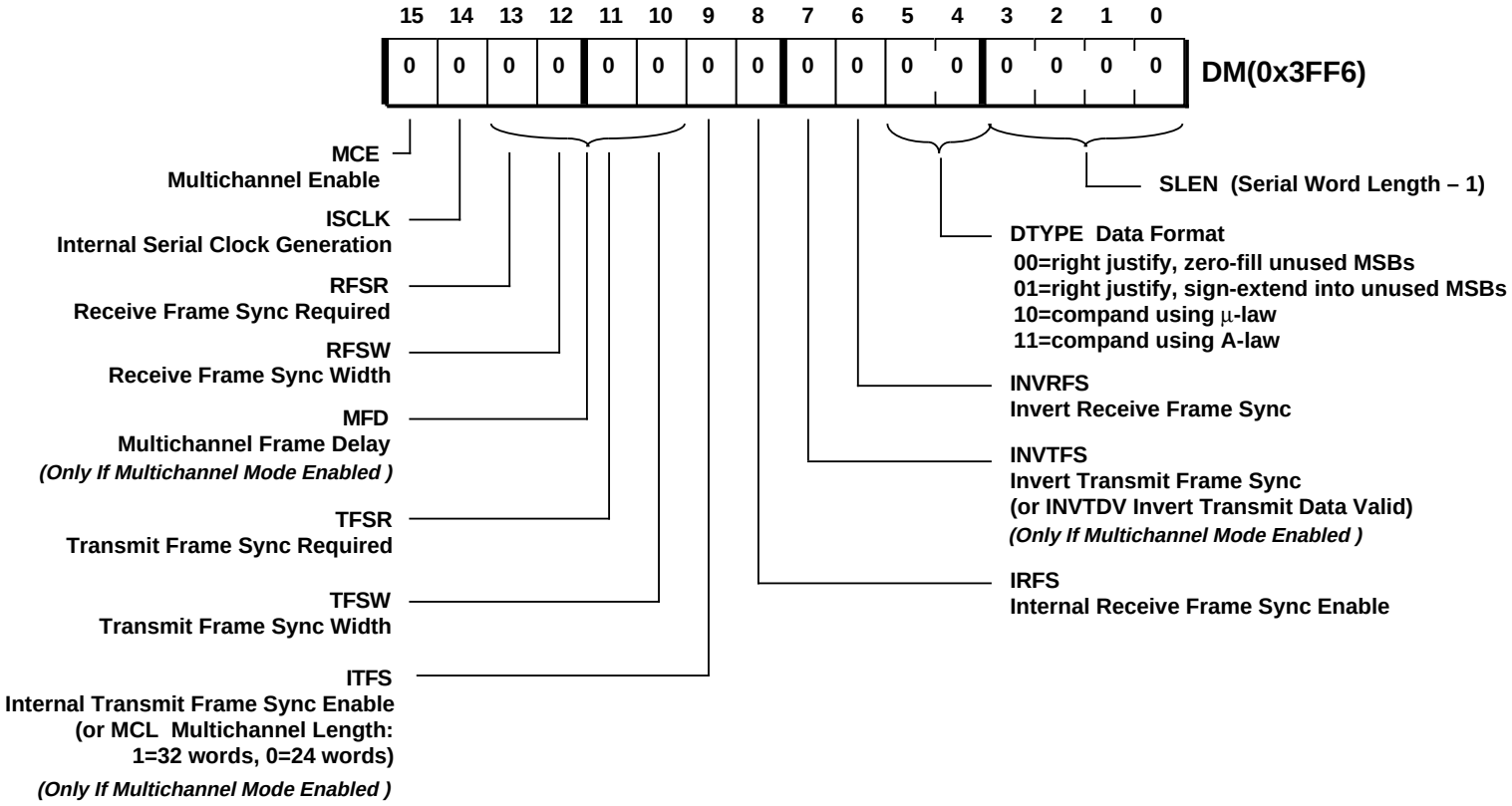


Default bit values at reset are shown; if no value is shown, the bit is undefined at reset.
Reserved bits are shown on a gray field—these bits should always be written with zeros.

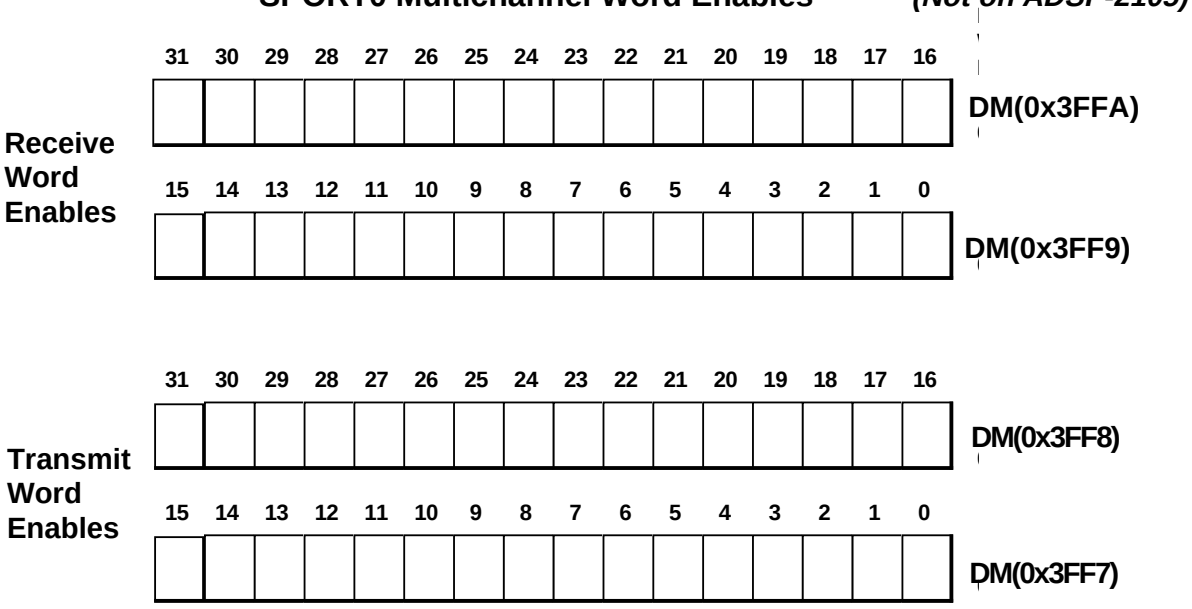
E Control/Status Registers

Memory-Mapped Registers

SPORT0 Control Register ***(Not on ADSP-2105)***



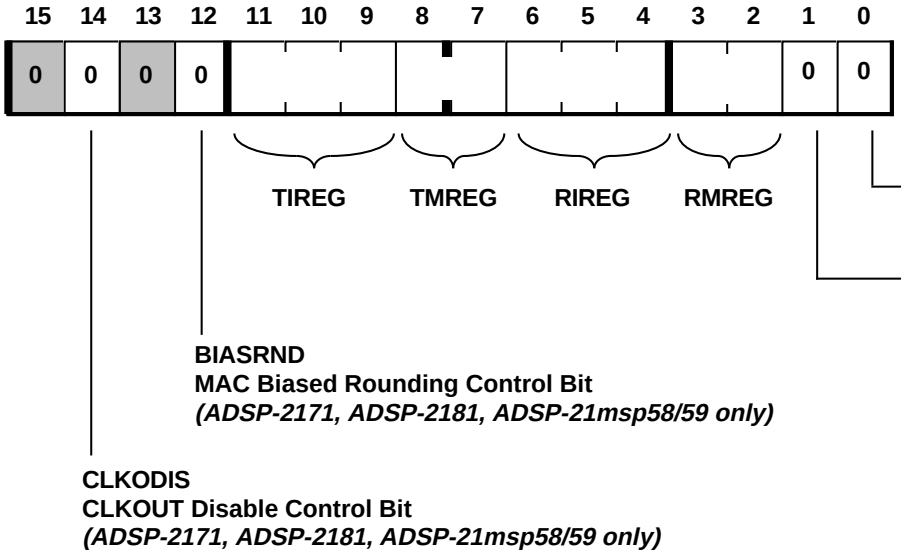
SPORT0 Multichannel Word Enables ***(Not on ADSP-2105)***



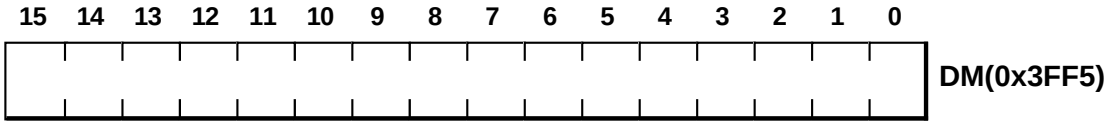
1 = channel enabled
0 = channel ignored

Memory-Mapped Registers

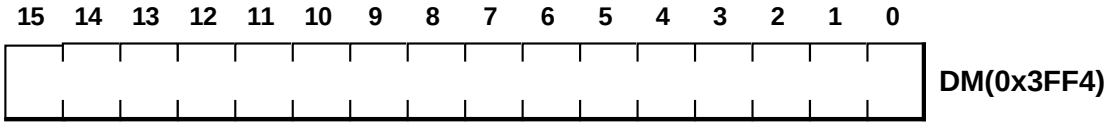
SPORT0 Autobuffer Control Register *(Not on ADSP-2105)*



SPORT0 SCLKDIV
Serial Clock Divide Modulus *(Not on ADSP-2105)*



SPORT0 RFSDIV
Receive Frame Sync Divide Modulus



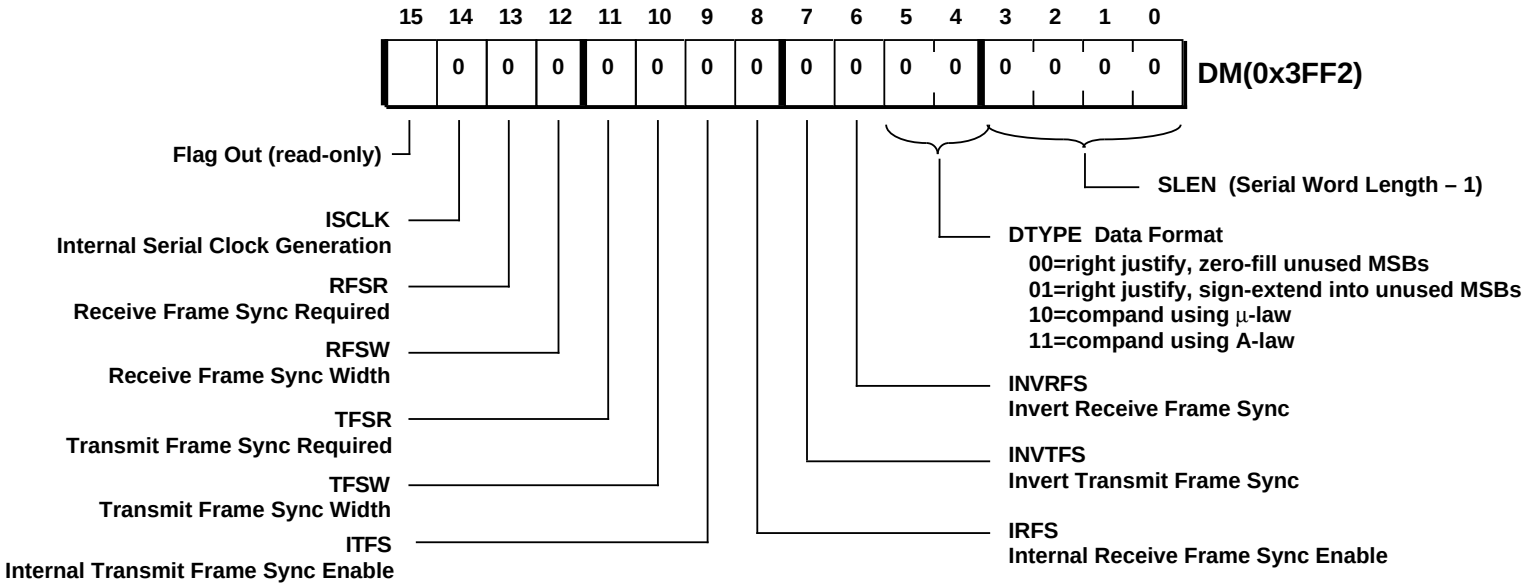
$$\text{SCLKDIV} = \frac{\text{CLKOUT frequency}}{2 * (\text{SCLK frequency})} - 1 \qquad \text{RFSDIV} = \frac{\text{SCLK frequency}}{\text{RFS frequency}} - 1$$

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.

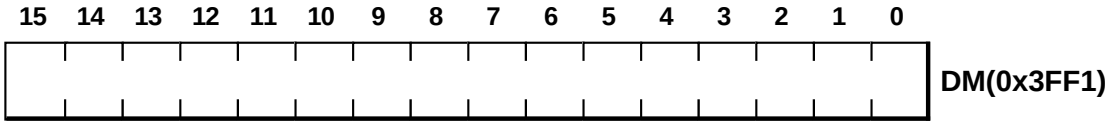
E Control/Status Registers

Memory-Mapped Registers

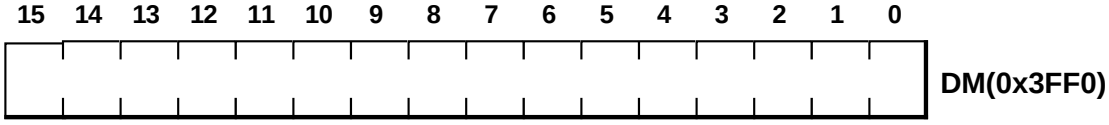
SPORT1 Control Register



SPORT1 SCLKDIV
Serial Clock Divide Modulus



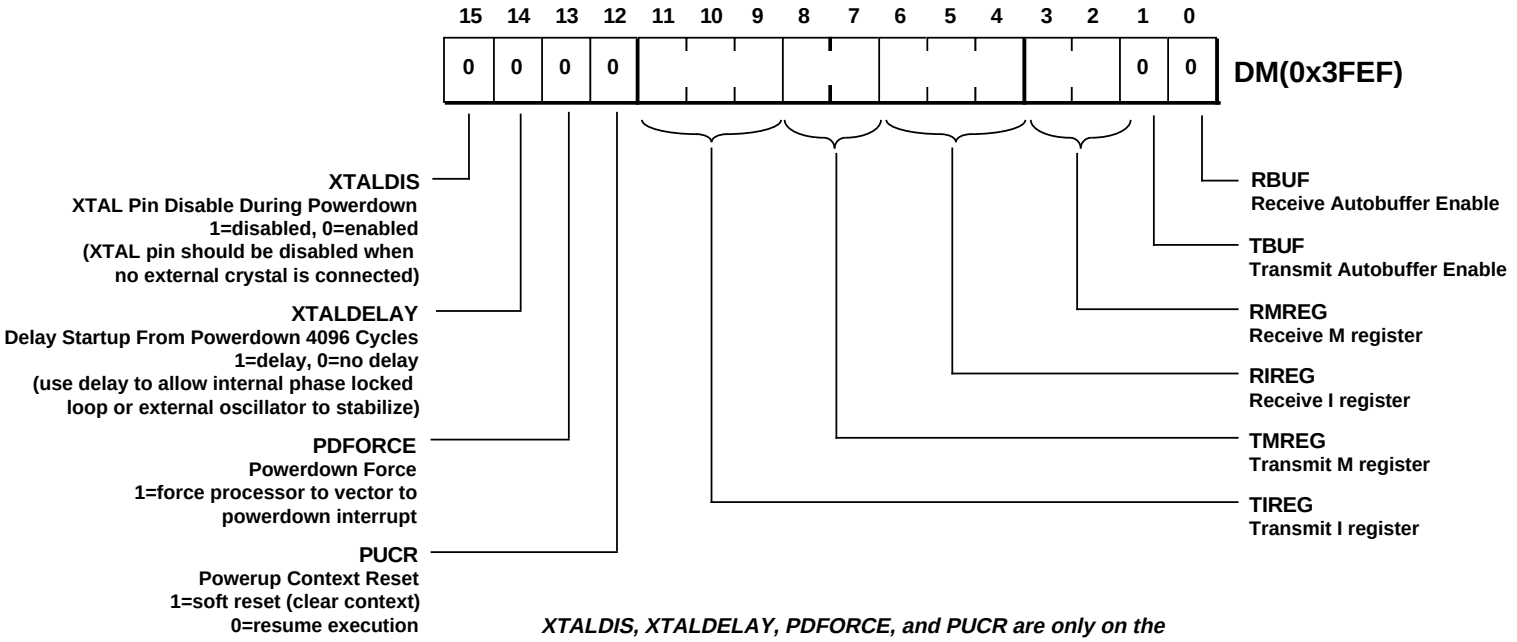
SPORT1 RFSDIV
Receive Frame Sync Divide Modulus



$$\text{SCLKDIV} = \frac{\text{CLKOUT frequency}}{2 * (\text{SCLK frequency})} - 1 \qquad \text{RFSDIV} = \frac{\text{SCLK frequency}}{\text{RFS frequency}} - 1$$

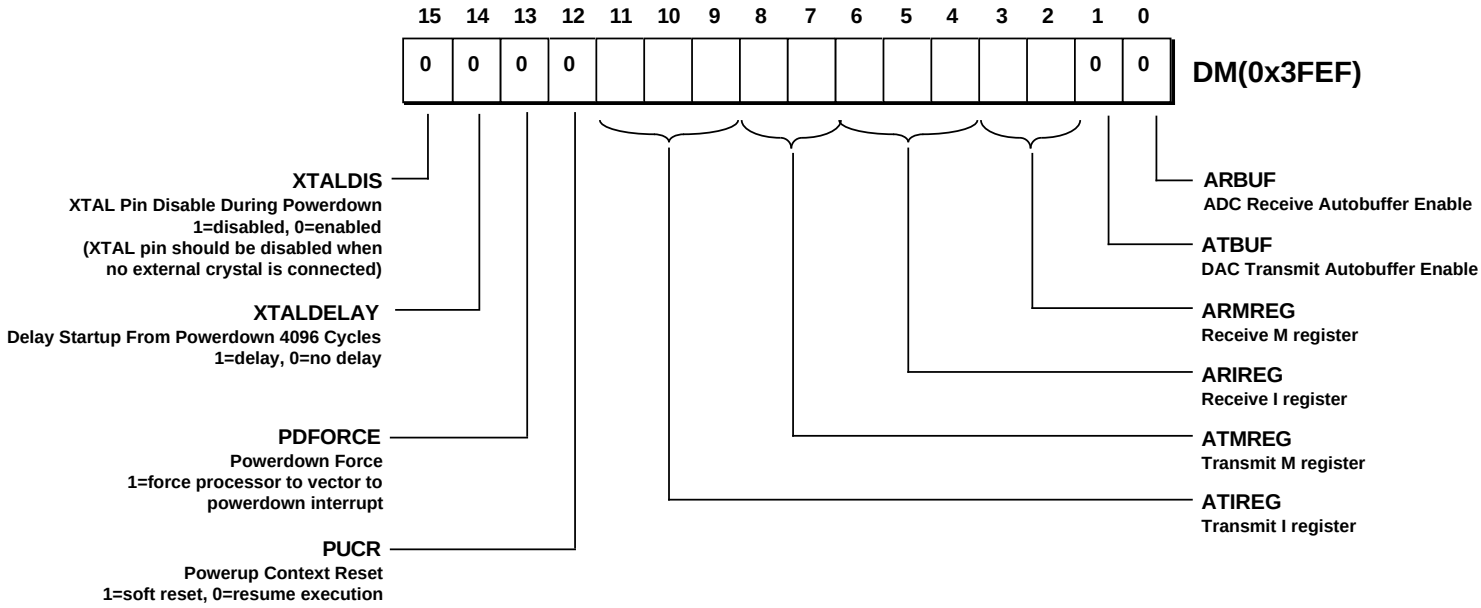
Memory-Mapped Registers

SPORT1 Autobuffer Control Register *(Not on ADSP-21msp5x)*



XTALDIS, XTALDELAY, PDFORCE, and PUCR are only on the ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors.

Analog Autobuffer Control Register *(ADSP-21msp5x only)*

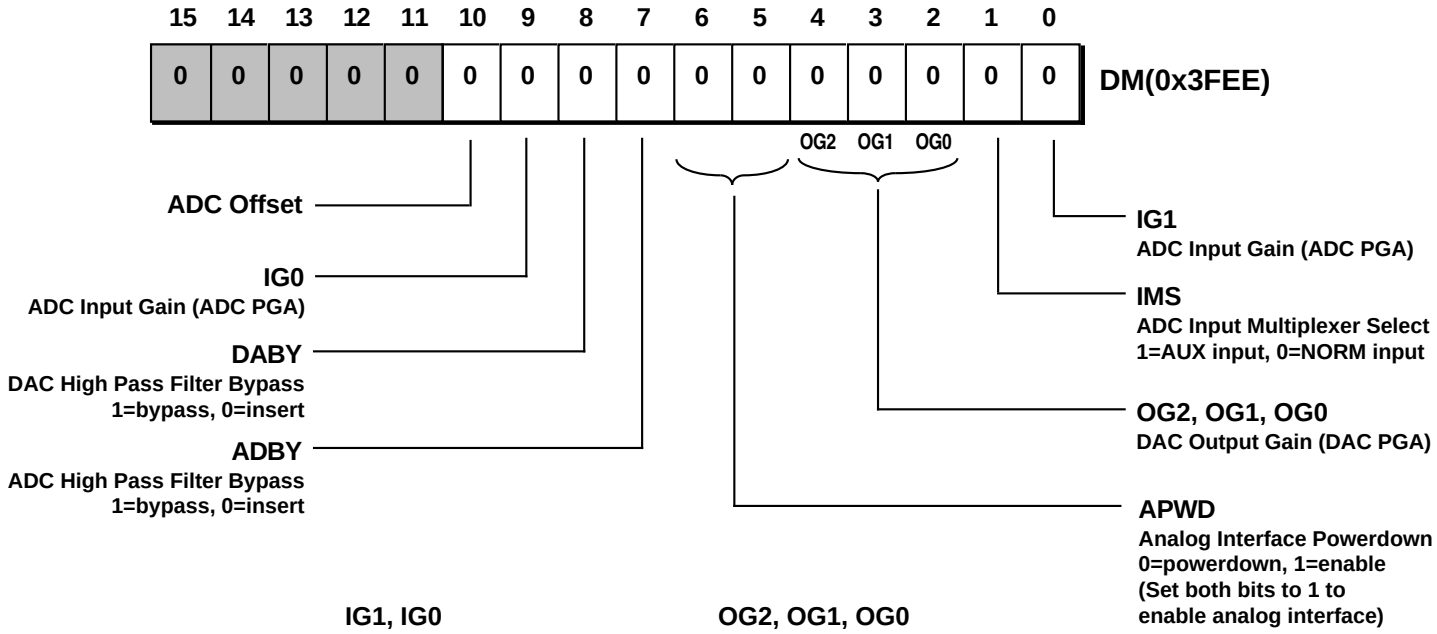


Default bit values at reset are shown; if no value is shown, the bit is undefined at reset.
Reserved bits are shown on a gray field—these bits should always be written with zeros.

E Control/Status Registers

Memory-Mapped Registers

Analog Control Register *(ADSP-21msp5x only)*



IG1, IG0
ADC Input Gain (ADC PGA)

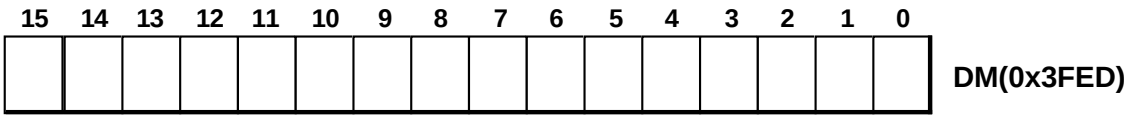
Gain	IG1	IG0
0 dB	0	0
+6 dB	0	1
+20 dB	1	0
+26 dB	1	1

OG2, OG1, OG0
DAC Output Gain (DAC PGA)

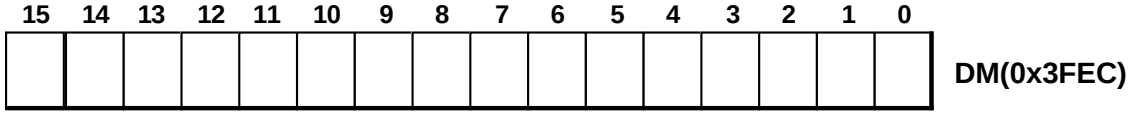
Gain	OG2	OG1	OG0
+6 dB	0	0	0
+3 dB	0	0	1
0 dB	0	1	0
–3 dB	0	1	1
–6 dB	1	0	0
–9 dB	1	0	1
–12 dB	1	1	0
–15 dB	1	1	1

Analog Data Registers *(ADSP-21msp5x only)*

ADC Receive Data

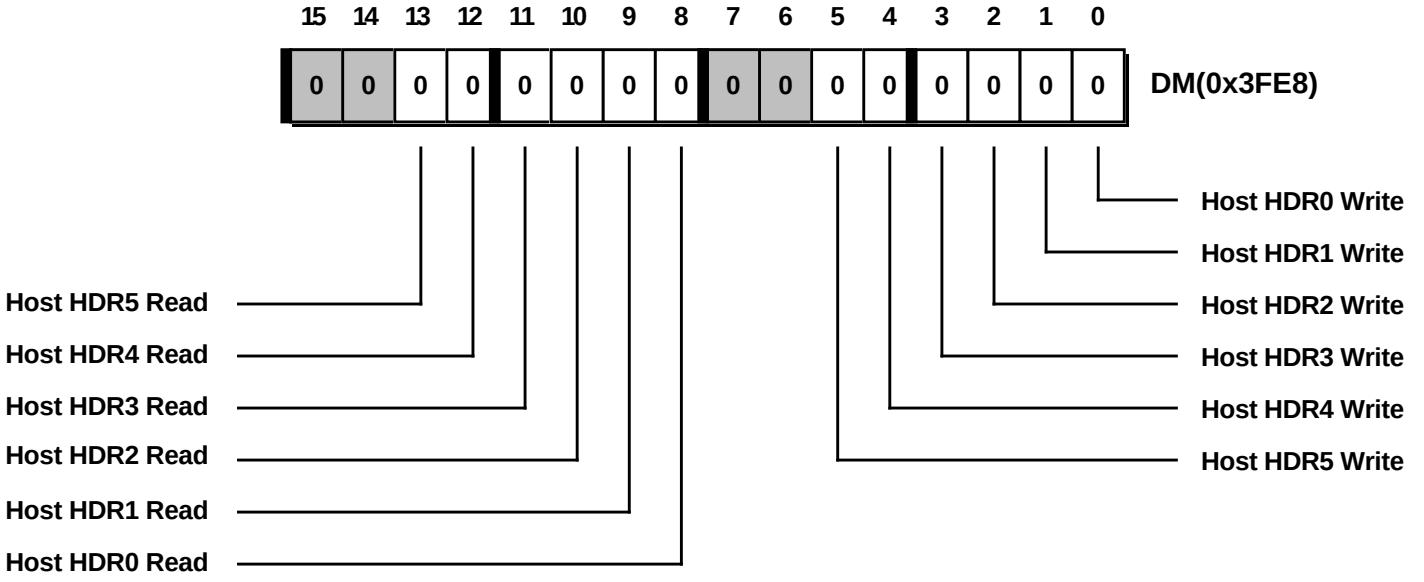


DAC Transmit Data

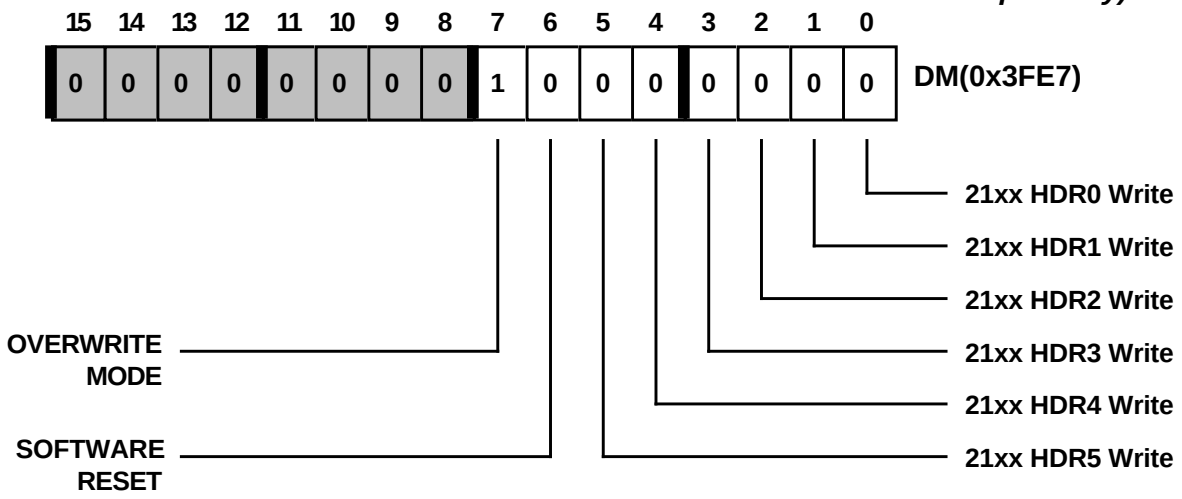


Memory-Mapped Registers

HMASK Interrupt Mask Register *(ADSP-2171, ADSP-2111, ADSP-21msp5x only)*



HSR7 Status Register *(ADSP-2171, ADSP-2111, ADSP-21msp5x only)*

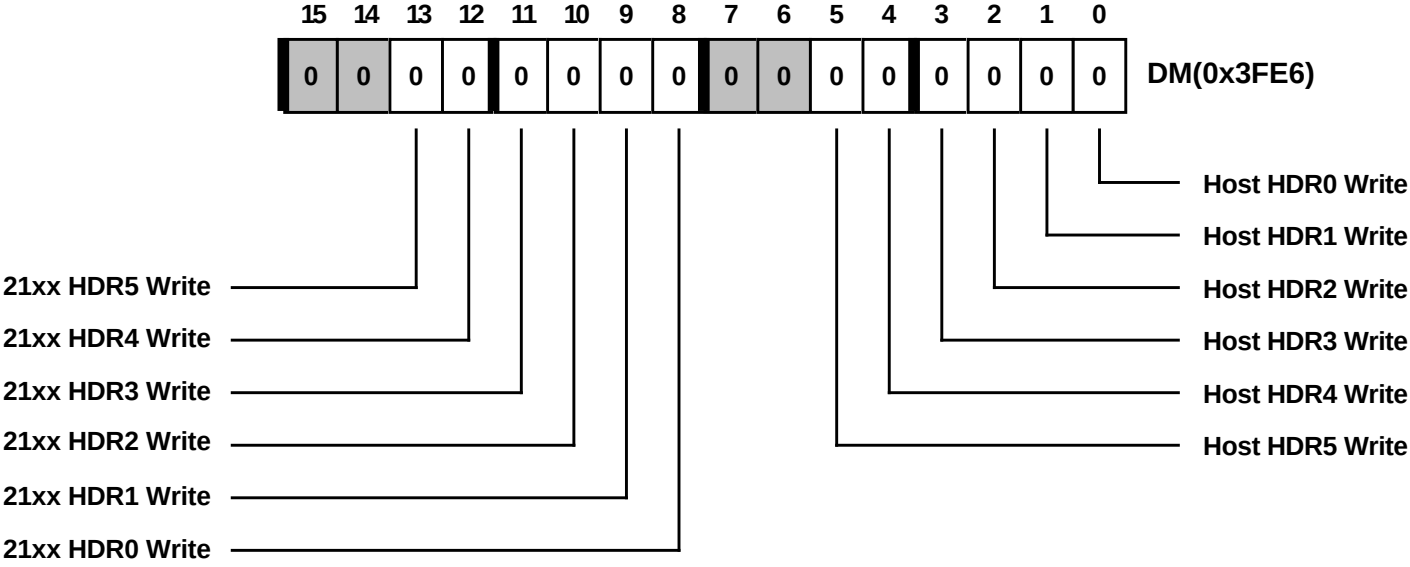


Default bit values at reset are shown; if no value is shown, the bit is undefined at reset.
Reserved bits are shown on a gray field—these bits should always be written with zeros.

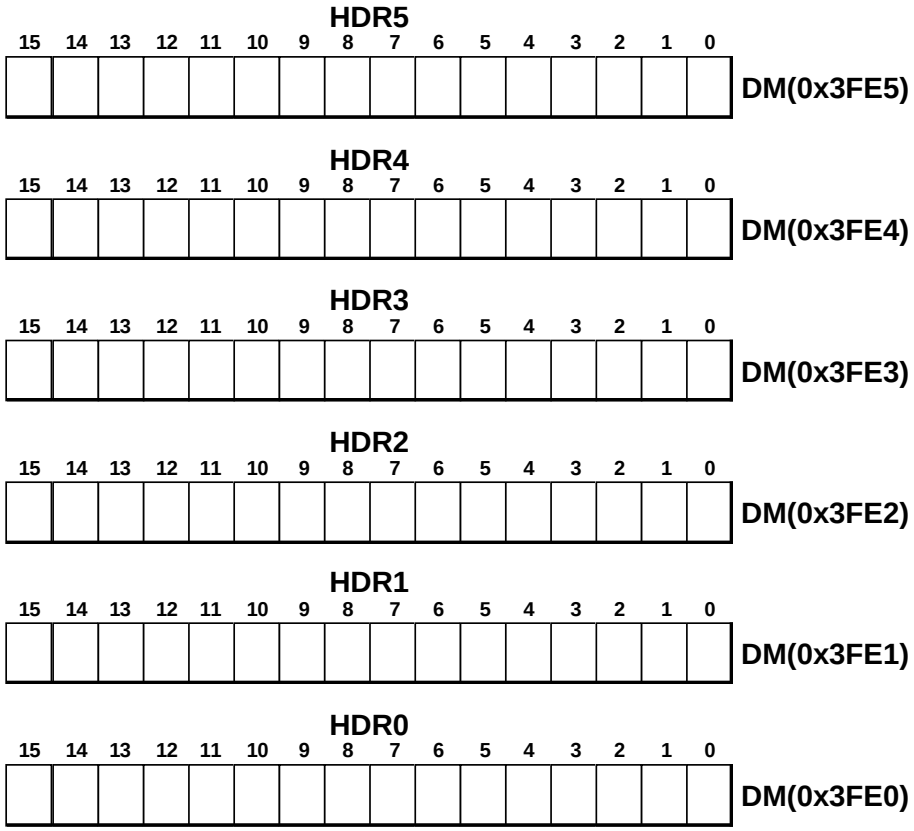
E Control/Status Registers

Memory-Mapped Registers

HSR6 Status Register *(ADSP-2171, ADSP-2111, ADSP-21msp5x only)*



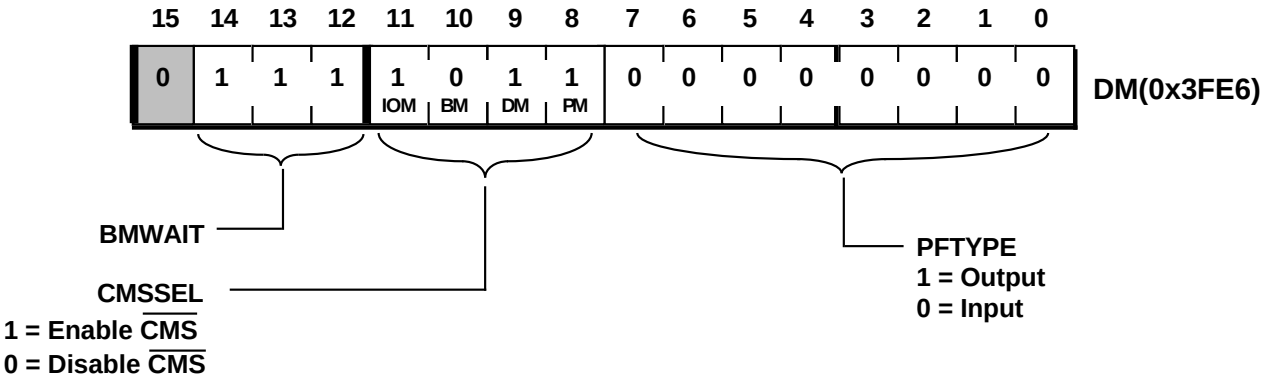
HIP Data Registers *(ADSP-2171, ADSP-2111, ADSP-21msp5x only)*



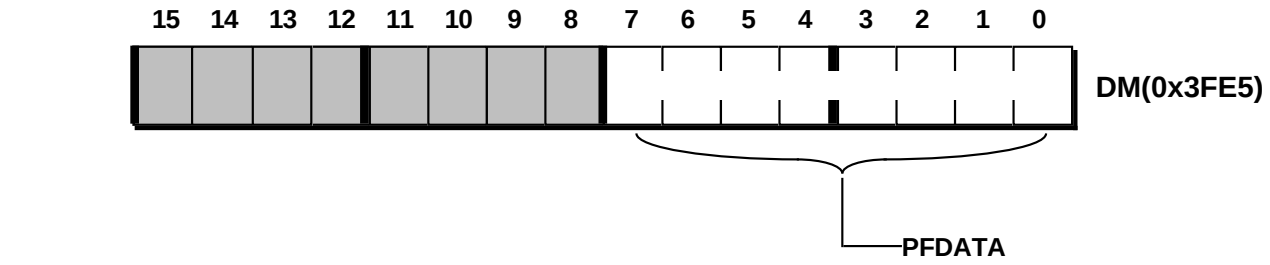
Control/Status Registers E

Memory-Mapped Registers

Programmable Flag & Composite Select Control ***(ADSP-2181 only)***



Programmable Flag Data ***(ADSP-2181 only)***

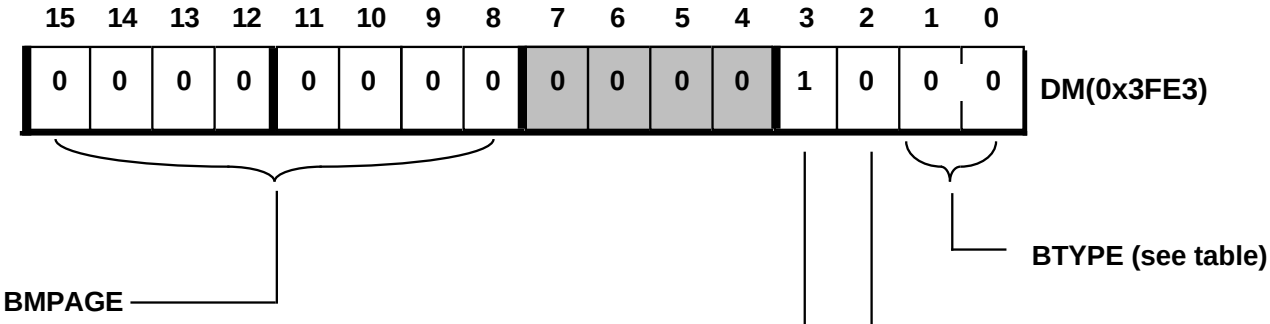


Default bit values at reset are shown; if no value is shown, the bit is undefined at reset.
Reserved bits are shown on a gray field—these bits should always be written with zeros.

E Control/Status Registers

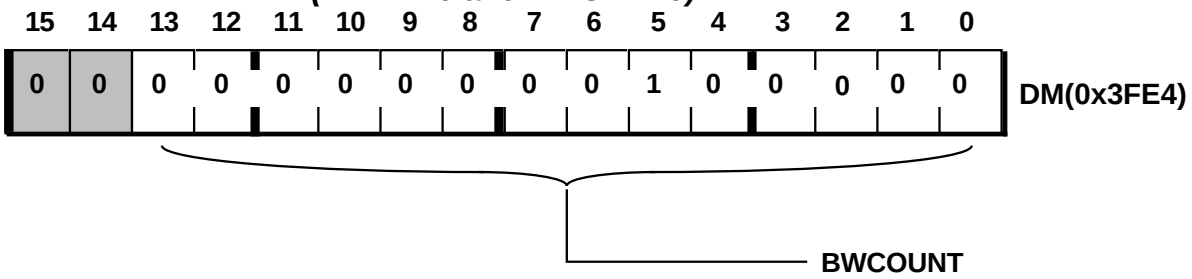
Memory-Mapped Registers

BDMA Control *(ADSP-2181 only)*



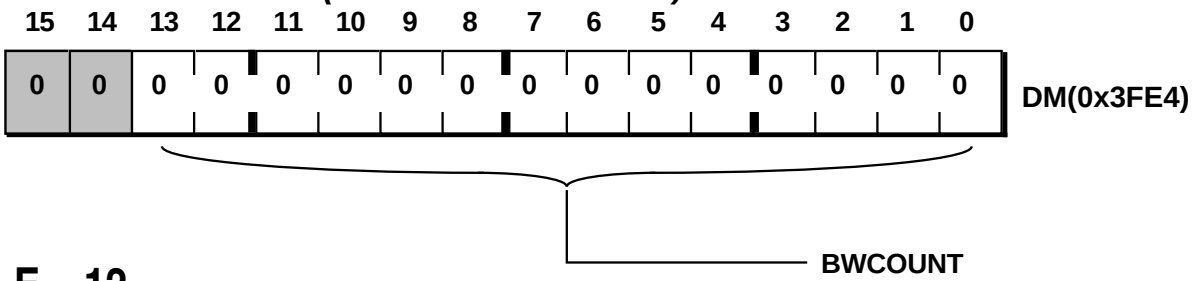
BTYPE	00	01	10	11
Internal Memory Space	PM	DM	DM	DM
Word Size	24	16	8	8
Alignment	full word	full word	MSB	LSB

BDMA Word Count *(ADSP-2181 only)*
(MMAP=0 and BMODE=0)



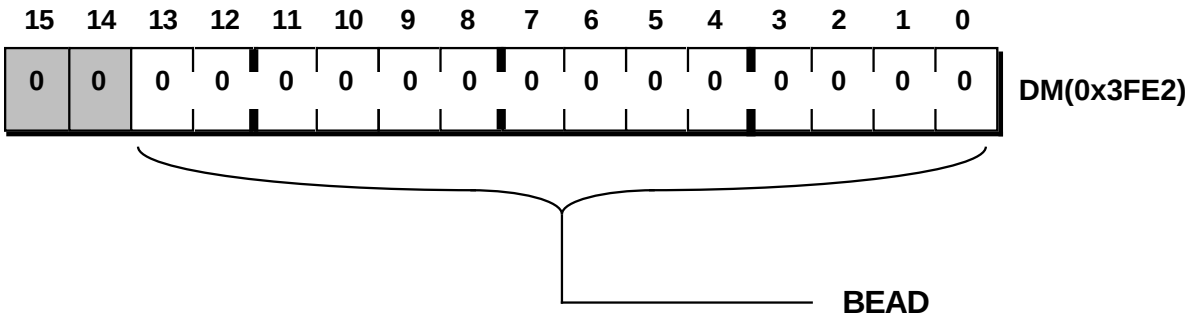
or

BDMA Word Count
(MMAP=1 or BMODE=1)

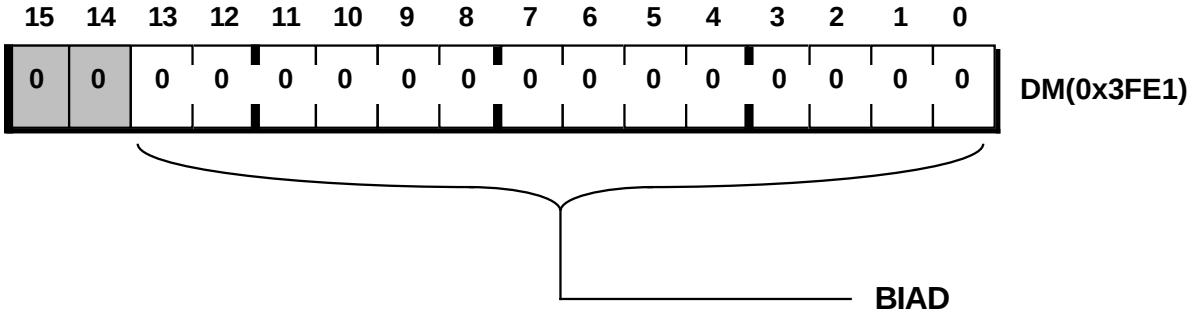


Memory-Mapped Registers

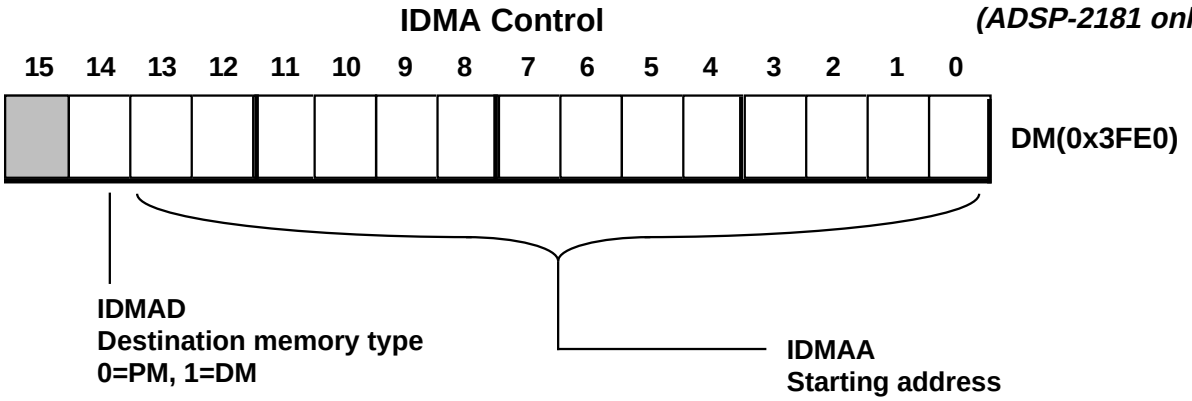
BDMA External Address *(ADSP-2181 only)*



BDMA Internal Address *(ADSP-2181 only)*



IDMA Control *(ADSP-2181 only)*

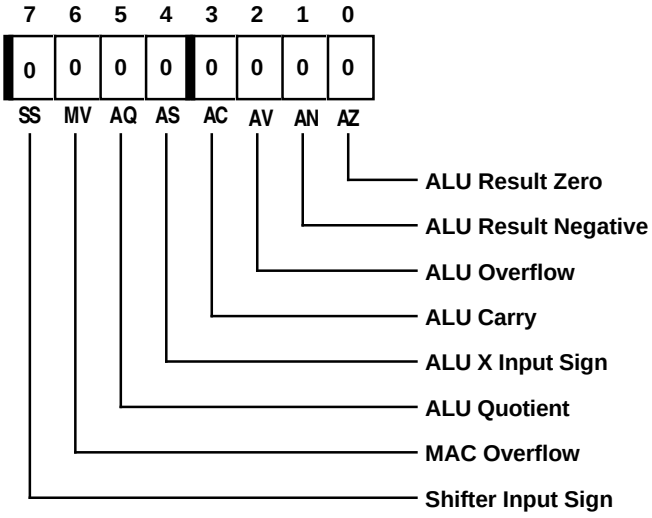


Default bit values at reset are shown; if no value is shown, the bit is undefined at reset.
Reserved bits are shown on a gray field—these bits should always be written with zeros.

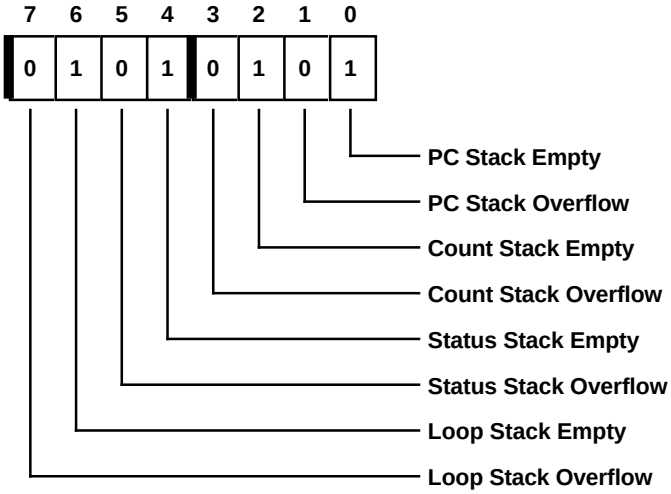
E Control/Status Registers

Non-Memory-Mapped Registers

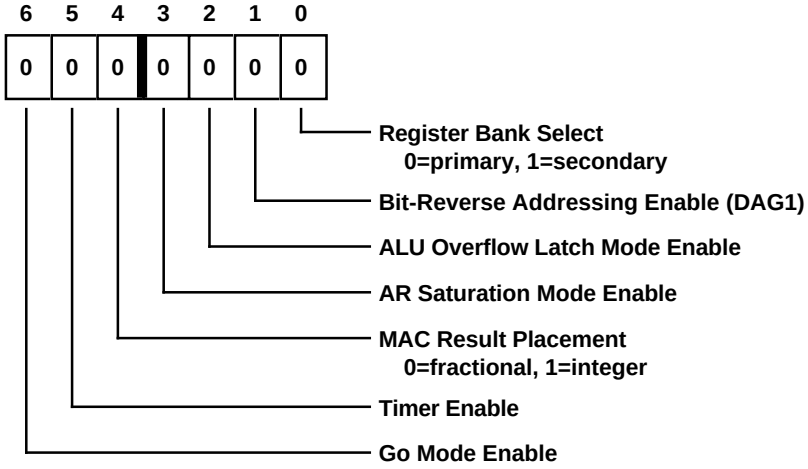
ASTAT



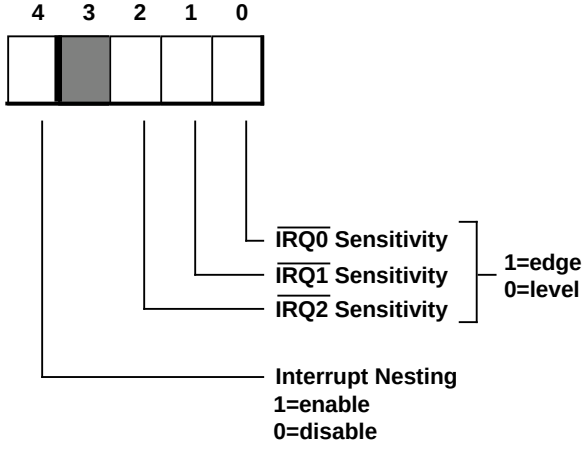
SSTAT *(read-only)*



MSTAT



ICNTL

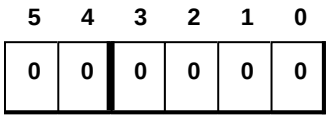


Control/Status Registers E

Non-Memory-Mapped Registers

ADSP-2101
ADSP-2105
ADSP-2115

IMASK



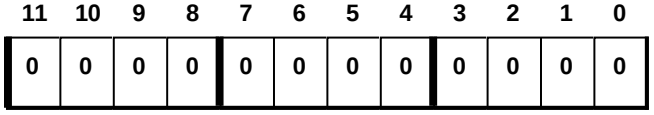
INTERRUPT ENABLES

1 = enable
0 = disable (mask)

Timer
SPORT1 Receive or $\overline{\text{IRQ0}}$
SPORT1 Transmit or $\overline{\text{IRQ1}}$
SPORT0 Receive ***(must be set to 0 for ADSP-2105)***
SPORT0 Transmit ***(must be set to 0 for ADSP-2105)***
$\overline{\text{IRQ2}}$

ADSP-2101
ADSP-2105
ADSP-2115
ADSP-2111

IFC ***(write-only)***



INTERRUPT FORCE BITS

$\overline{\text{IRQ2}}$
SPORT0 Transmit
(must be set to 0 for ADSP-2105)
SPORT0 Receive
(must be set to 0 for ADSP-2105)
SPORT1 Transmit or $\overline{\text{IRQ1}}$
SPORT1 Receive or $\overline{\text{IRQ0}}$
Timer

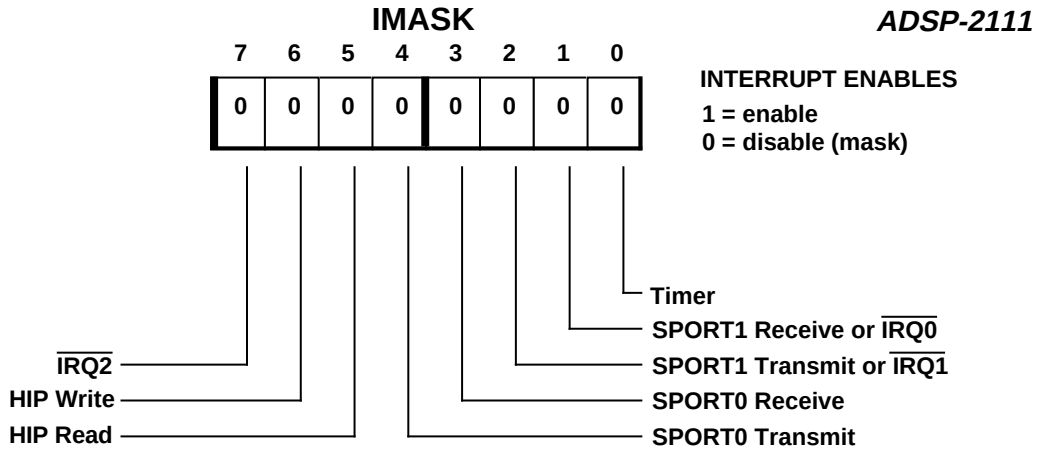
INTERRUPT CLEAR BITS

Timer
SPORT1 Receive or $\overline{\text{IRQ0}}$
SPORT1 Transmit or $\overline{\text{IRQ1}}$
SPORT0 Receive
(must be set to 0 for ADSP-2105)
SPORT0 Transmit
(must be set to 0 for ADSP-2105)
$\overline{\text{IRQ2}}$

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.

E Control/Status Registers

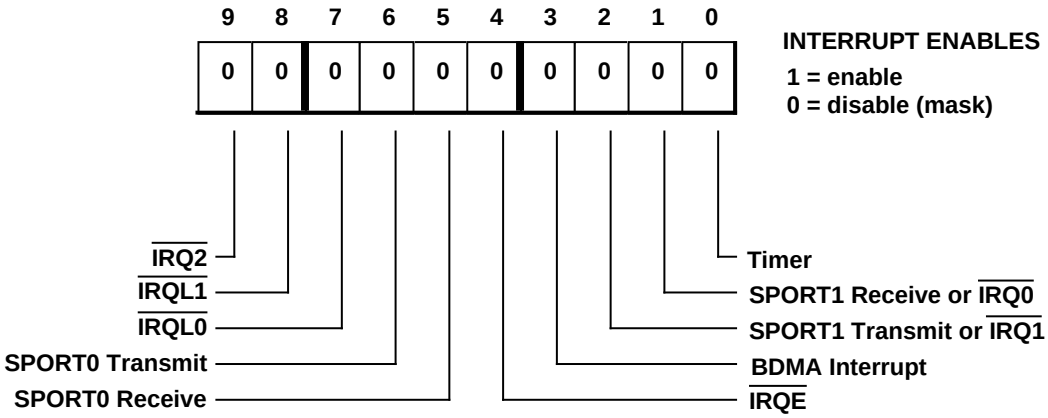
Non-Memory-Mapped Registers



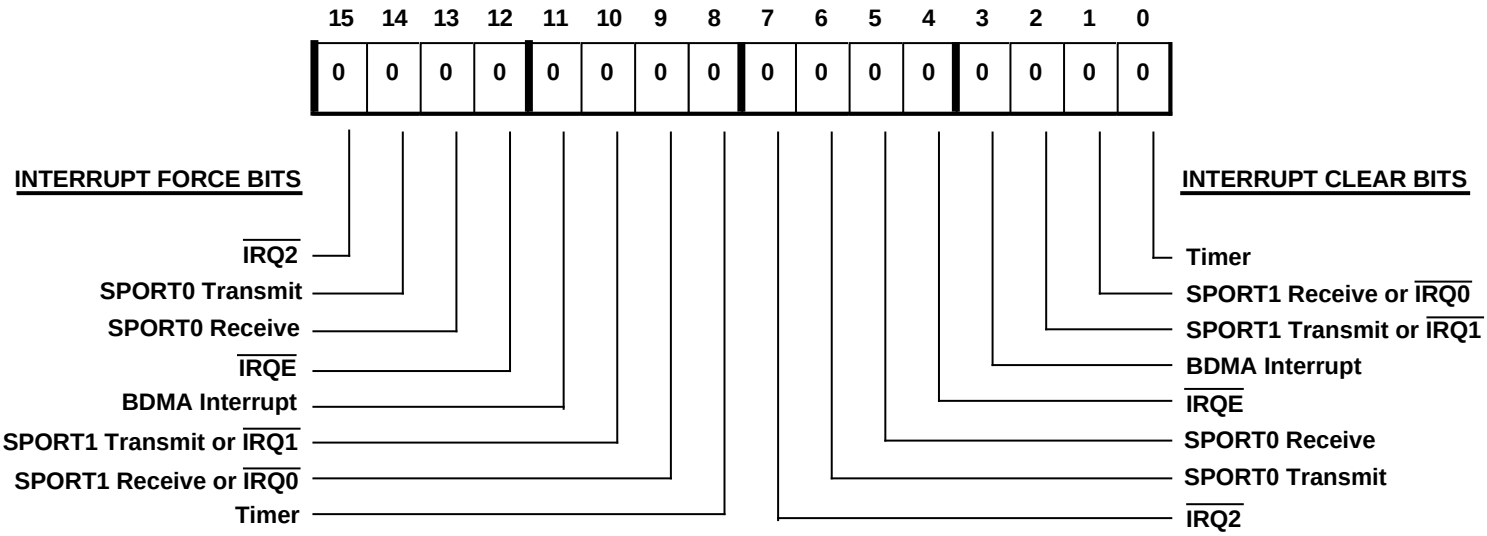
Control/Status Registers E

Non-Memory-Mapped Registers

IMASK ***ADSP-2181***



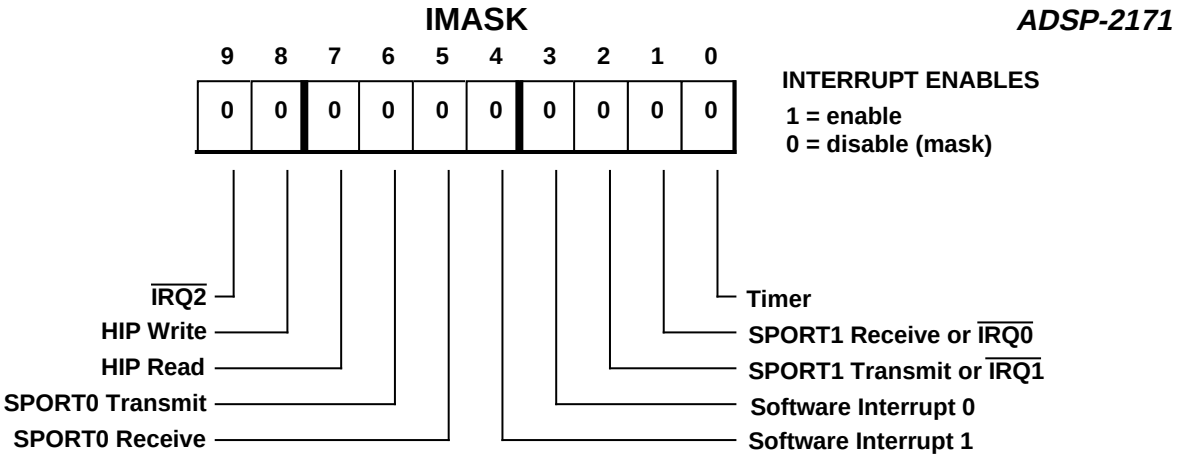
IFC *(write-only)* ***ADSP-2181***

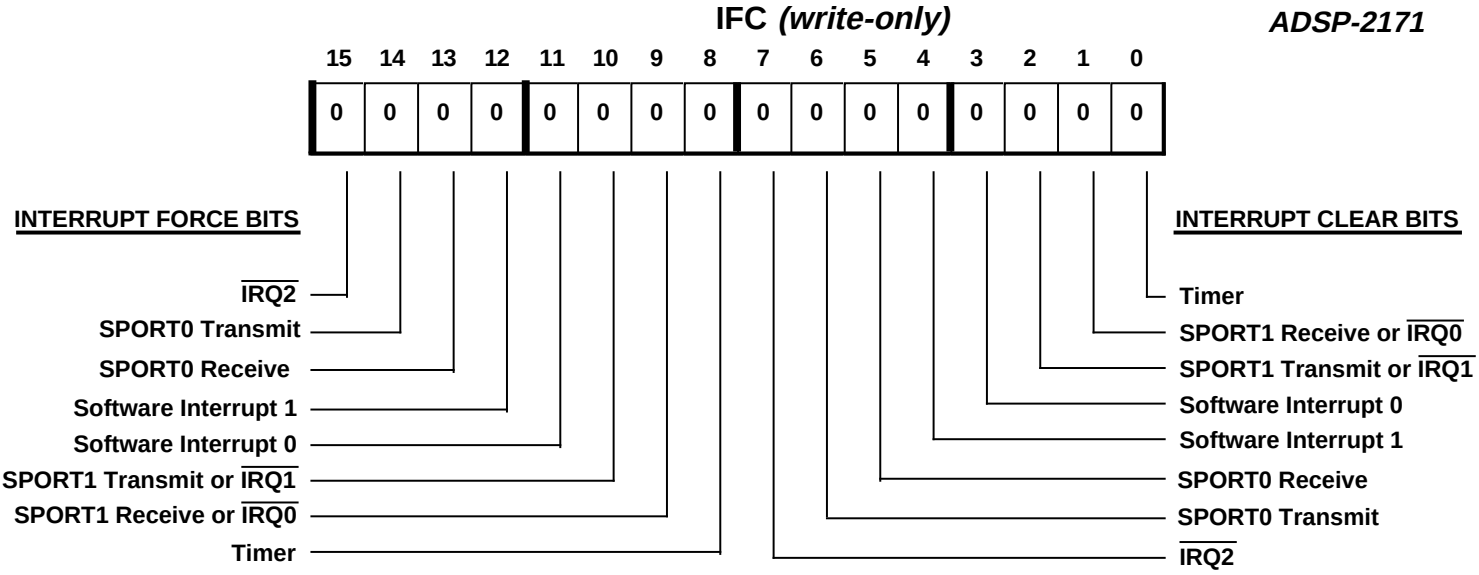


Default bit values at reset are shown; if no value is shown, the bit is undefined at reset.
Reserved bits are shown on a gray field—these bits should always be written with zeros.

E Control/Status Registers

Non-Memory-Mapped Registers

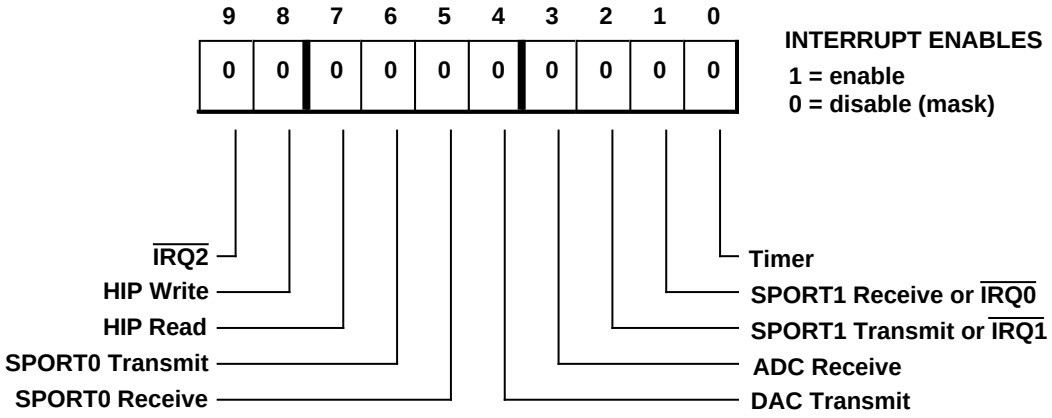




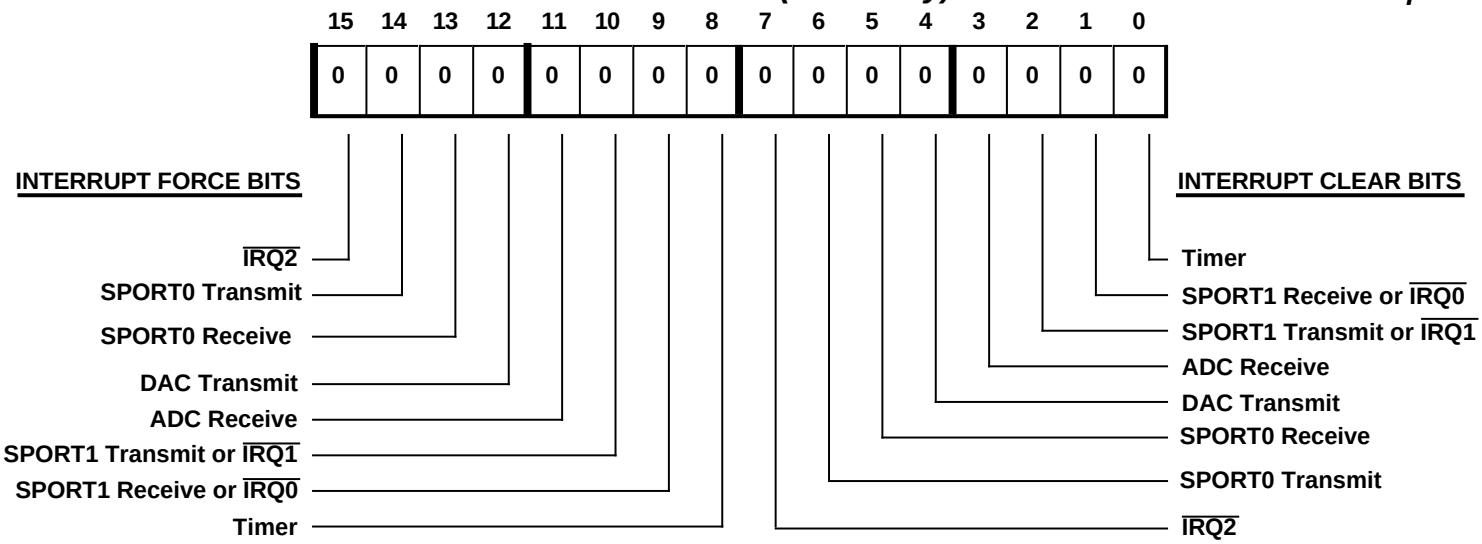
Control/Status Registers E

Non-Memory-Mapped Registers

IMASK *ADSP-21msp5x*

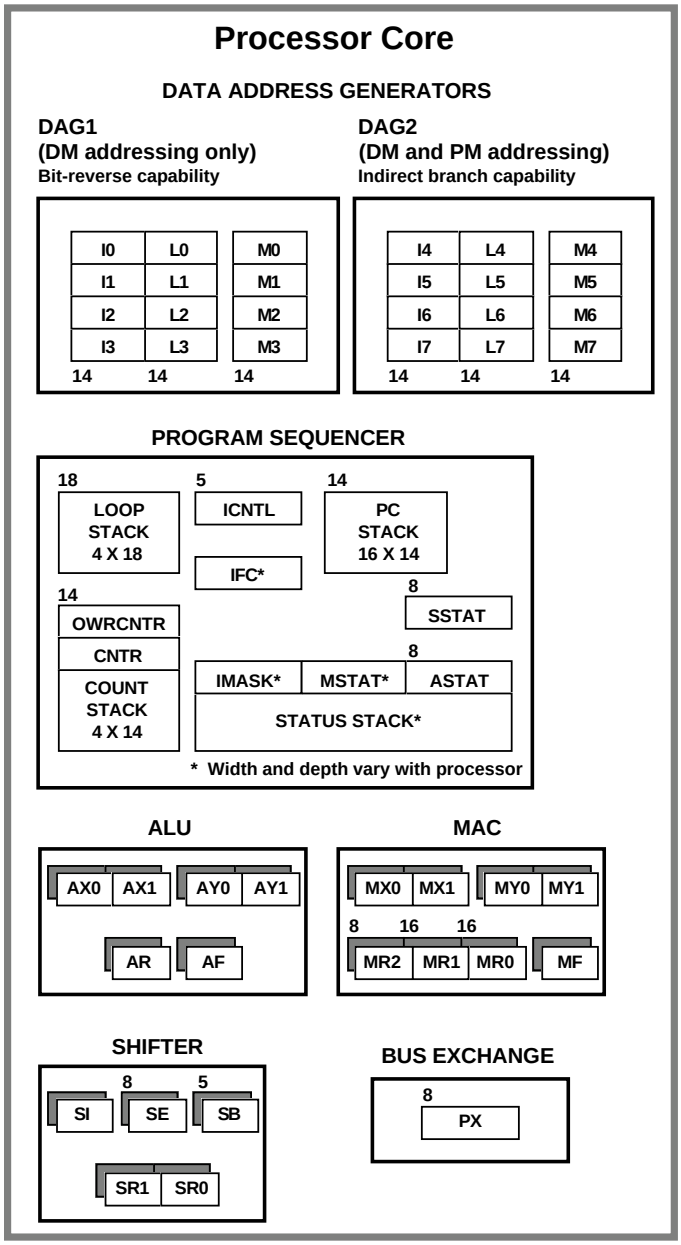


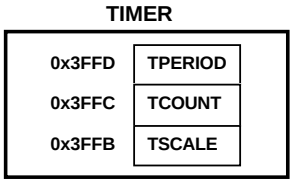
IFC *(write-only)* *ADSP-21msp5x*

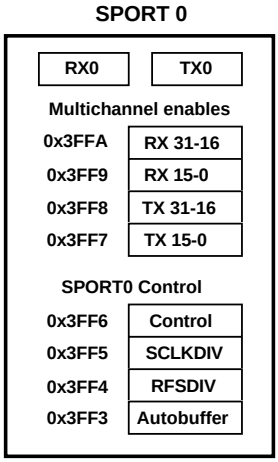


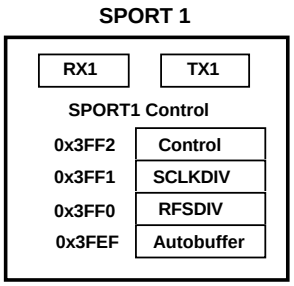
Default bit values at reset are shown; if no value is shown, the bit is undefined at reset.
Reserved bits are shown on a gray field—these bits should always be written with zeros.

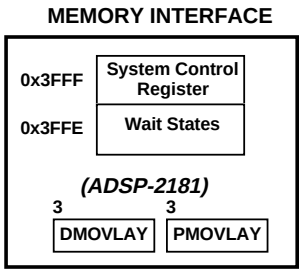
E Control/Status Registers

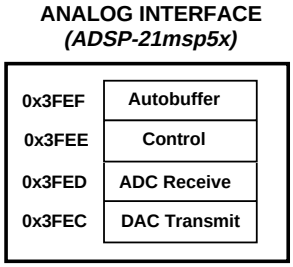


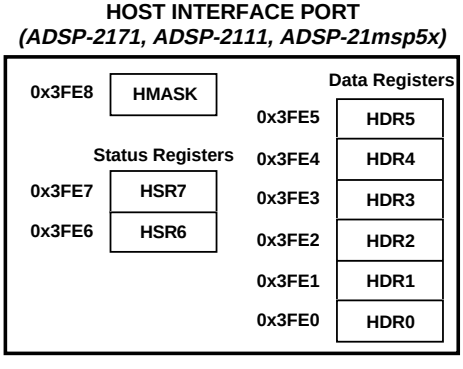


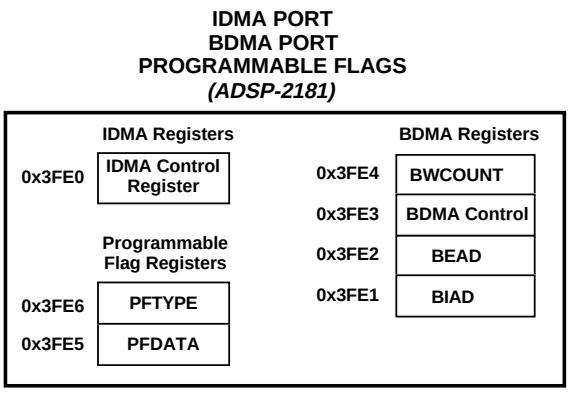












Index

1.15 format 2-2
μ-law 1-2, 5-15, 5-23

A

A-law 1-2, 5-23
A/D converter 1-3, 1-10, 3-18, 8-1, 8-2, 8-3, 8-4, 8-9, 10-14
AC (carry) 2-2, 2-5, 2-8, 2-13, 2-36
AC coupling 8-18
Accumulator 1-5
ADC, DAC interrupts 8-12
ADC 1-3, 1-10, 3-18, 8-1, 8-2, 8-3, 8-4, 8-9
ADC interface 13-10
Add with carry 2-8
Address generators 1-6, 1-7
Address pointers 1-7
ADSP-2181 ..... 9-13, 15-18
AF register 2-5
Alternate framing mode 5-13
ALU 1-1, 1-6, 2-5, 2-8, 3-21, 3-24, 12-2, 12-6, 15-9, B-1
ALU arithmetic 2-2
ALU carry (AC) 2-26
ALU overflow (AV) 3-23, 3-24
ALU overflow latch mode 2-8, 2-9
ALU saturation mode 2-8, 2-9
ALU status 2-13
AN (negative) 2-2, 2-5, 2-13
Analog control register 8-5, 8-6, 8-8, 8-9, 8-10
Analog interface 1-3, 1-10
Analog loopback program 8-12
Analog-to-digital conversion 13-5
AQ (quotient) 2-5, 2-9, 2-10, 2-13
AR register 2-8, 2-22, 2-36
AR saturation 3-23, 3-24
Arithmetic formats 2-4
Arithmetic operation 3-24
Arithmetic shift 2-3, 2-22, 2-28
AS (sign) 2-5, 2-13
ASHIFT 2-31
Assembler 1-10
Assembler directives 12-10, 14-1
ASTAT 2-10, 2-13, 2-19, 2-24, 2-26, 2-36, 3-21, 3-24, 12-5
Autobuffer service 5-39
Autobuffer timing 5-37
Autobuffering 5-3, 5-4, 5-26, 5-32, 5-38, 5-40, 5-41, 8-1, 8-9, 8-10, 8-11, 8-12, 8-13, 8-14, 8-15, 9-5, 9-23, 15-18
Autobuffer control register 5-27
AV (overflow) ... 2-2, 2-5, 2-8, 2-9, 2-13, 2-26, 2-36
AX0 register 2-5
AX1 register 2-5
AY0 register 2-5
AY1 register 2-5
AZ (zero) 2-5, 2-13

B

Bank select 3-23
Barrel shifter 1-6, 2-22
Base architecture 1-1, 1-6
BCR 9-13, 11-6, 11-7
BDMA 15-18
BDMA booting 9-13, 11-9
BDMA context reset 11-6, 11-7
BDMA interrupt 11-7
BEAD 11-4
BFORCE bit 10-16
BIAD 11-4
Binary multiplication C-3
Binary string 2-1
Bit-reverse addressing 1-7, 3-23, 4-2, 4-6, 14-18
Block exponent 2-26
Block floating-point 1-5, 14-19, C-5
BMODE pin 7-3, 7-16, 10-15, 11-9, 11-12
BMS 1-8, 7-16, 10-17
Boot address 10-18
Boot loading 13-2, 13-5
Boot loading sequence 10-17
Boot loading through the HIP 7-16
Boot memory 10-1, 10-15, 10-17
Boot pages 10-15
Booting operation 9-4
BPAGE 10-16

Index

Branching 3-1
BTYPE 11-9
Buffer length 4-5
Bus exchange 1-5, 1-8, 2-15, 4-1, 4-9, 12-6
Bus grant (BG) 3-18, 9-15
Bus request (BR) 5-38, 10-15, 10-21, 13-2
Buses 1-3, 1-8
BWAIT 10-17
BWCOUNT 9-13, 11-7
Byte memory 11-9

C

C Compiler 1-10
C language 14-3
CALL 3-4, 3-8, 3-9, 3-24
Carry (AC) 2-2, 2-5, 2-8, 2-13, 2-36
Carry-in (CI) 2-5
Chip enable 10-3
Circular buffer addressing ... 1-5, 1-7, 4-1, 4-3, 4-8
CLKIN 9-1, 9-3, 9-4
CLKOUT 9-2, 9-3, 9-4, 10-2
Clock frequency 8-9
Clock signals 9-1
Clock synchronization (SPORT) 5-35
Clock synchronization delay (SPORT) 5-34
CNTR register 3-4, 3-9, 12-4
Codec interface 13-5
Companding 1-2, 5-5, 5-15, 5-23, 5-24, 5-25, 5-32, 5-36, 5-37, 5-42
Computation with data register move 15-7
Computation with memory read 15-6
Computation with memory write 15-6
Computational units 1-6, 3-23, 12-1, 12-6
Condition 3-20
Condition logic 3-3
Conditional instructions 3-24
Configuring interrupts 3-14
Context reset 11-7
Context switching 3-12
Continuous transmission 5-20
Control/status registers 1-12, E-1
Core architecture 1-5
Count stack 3-4, 3-5, 12-5
Counter expired (CE) 3-4, 3-6, 3-9
Cycle stealing 11-25

D

D/A 10-14
DAC 1-3
DAC interface 13-8
DAG1 ..... 1-7, 4-2, 12-2
DAG2 ..... 1-7, 3-3, 3-4, 3-8, 4-2, 12-2
Data address generators 4-1, 12-2
Data bus 4-1
Data memory 1-7, 1-8, 10-1
Data memory address bus 1-3, 1-8
Data memory data bus 1-3, 1-8
Data memory interface 10-10
Data memory read 15-6
Data structures 4-7
Data transfer 4-1
Denormalization 1-6
Denormalize 2-31
Derive block exponent 2-29
Derive exponent 2-22, 2-26
Development tools 14-2
Digital-to-analog conversion 13-5
Direct addressing 1-8
Divide primitives 2-9
Division exceptions B-1
Division 2-9
DIVQ 2-10, 2-11, B-1
DIVS 2-2, 2-9, 2-10, B-1
DMA 15-18
DMA bus 1-8, 10-1
DMD bus 1-8, 2-19, 2-22, 3-4, 3-5, 3-20, 4-10, 10-1
DMD-PMD bus exchange 2-15, 4-1, 4-9, 4-10
DMOVLAY 10-31
DMS 1-8
DO UNTIL 3-4, 3-5, 3-6, 3-8, 3-9, 3-12, 12-5
DR input 5-2, 5-36
dreg 15-12, 15-14
DT output 5-2
Dual operand fetches 1-2, 1-5, 1-7

E

Edge-sensitive 3-15, 3-16, 3-18
Edge-sensitive interrupts 9-14
End-of-loop 3-7, 3-10
EPROM 1-4, 10-17

Index

EXP 2-33
EXPADJ 2-29
Exponent compare logic 2-22
Exponent detector 2-22, 2-26, 2-27
External address bus 1-3, 1-8
External clock 5-8, 9-21
External data bus 1-3, 1-8
External interrupts 9-14
External memory 5-38, 10-2
External SCLK 5-8
Extra cycles 15-18
EZ-ICE emulator 11-26

F

Fast fourier transform (FFT) 14-1, 14-11
Fast start-up 9-22
FIR filter 14-1, 14-4
Flag In (FI) 1-9, 3-24, 9-1
Flag Out (FO) 1-9, 9-1
Flag pins 9-15
Floating-point 2-33
Fractional mode 2-2, 2-3, 3-23, 3-24, C-1, C-4
Frame synchronization .. 5-2, 5-5, 5-10, 5-11, 5-12, ... 5-14, 5-15, 5-30, 5-34, 9-23
Framing 5-3, 5-16
Full duplex operation 5-34

G

GO mode 3-23, 3-24, 5-38

H

HACK 7-3, 7-7, 7-9
Harvard architecture 1-2, 10-1
HDR overwrite mode 7-7, 7-9
HDR registers 7-4, 7-5, 7-6, 7-7
HI-extend (HIX) 2-26, 2-36
HI/LO reference signal 2-22, 2-24
HIP configuration modes 7-3
HIP data registers 3-15, 7-4, 7-5, 7-6, 7-7
HIP during powerdown 9-24
HIP interrupt 3-18, 7-9, 7-10, 7-11
HIP pin summary 7-2
HIP read interrupt 7-4
HIP status registers 7-6
HIP status synchronization 7-8
HIP timing 7-12, 7-13, 7-14, 7-15
HIP write interrupt 7-4
HMASK register 3-15, 7-4, 7-10, 7-11, 12-8
HMD0 ..... 7-3, 7-4
HMD1 7-4
Hold offs 11-25
Host 1-2
Host data bus 7-4
Host handshaking 7-7
Host interface port (HIP) ... 1-2, 1-9, 3-15, 7-1, 7-4, ... 9-4, 10-15, 12-2, 13-13
Host interface timing 7-11
Host read strobe 7-4
Host write strobe 7-4
HSEL 7-3
HSIZE 7-3, 7-11
HSR registers 7-4, 7-5, 9-25

I

I registers 4-2, 4-3, 5-26, 5-28, 8-14, 12-2
IACK 11-12, 11-13, 11-25
ICNTL register 3-14, 3-15, 3-16, 3-20, 9-14, 12-4
IDLE instruction ..... 3-7, 3-10, 5-26, 9-15, ... 9-19, 9-26, 9-30
IDMA 11-12, 15-18
IDMA booting 11-24
IDMA control register 11-14, 11-15, 11-16
IDMA hold offs 11-25
IFC register 3-14, 3-18, 3-20, 12-4
IIR filter 14-1, 14-6
IMASK register ..... 3-14, 3-15, 3-16, 3-19, ... 3-20, 7-10, 8-12, 9-14, 12-4
Immediate shifts 2-30
Indirect addressing 1-8, 4-3, 12-2
Indirect jumps 12-2
Input formats 2-18
Input registers 1-7
Instruction completion latencies 5-38, 15-18
Instruction set 15-1, A-1
Integer 3-23, 3-24, C-1
Integer mode 2-3, C-4
Internal buses 1-8
Internal memory 1-2
Internal oscillator 9-22
Interrupt control register 3-15
Interrupt controller 3-1, 3-3, 3-11
Interrupt force & clear register 3-15, 3-18
Interrupt latencies 3-19, 5-42
Interrupt mask register 3-16
Interrupt nesting 3-16
Interrupt request 5-40, 9-1, 9-14
Interrupt sensitivity 9-14

Index

Interrupt service 3-12, 3-14, 3-16, 3-18, 3-20, 5-39
Interrupt vector 3-11, 3-12, 3-13, 3-14, D-1, D-2, D-3
Interrupts 1-9, 3-4, 3-16, 9-24, 9-30, 12-4, 12-11
Interrupts pending 3-18
Interval timer 3-18, 3-23, 3-24
INVRFS 5-14, 5-16, 5-32
INVTDV 5-32
INVTFS 5-14, 5-16, 5-32
IRFS 5-11, 5-32
IRQ0 ..... 1-9, 3-11, 5-3, 9-14
IRQ1 ..... 1-9, 3-11, 5-3, 9-14
IRQ2 ..... 5-38, 9-14
ITFS 5-11

J

JUMP 3-8, 3-9, 3-24

L

Length (L) registers 4-1, 4-3, 4-4, 12-2
Level-sensitive interrupts 3-15, 9-14
Linker 14-3
Logical shift 2-3, 2-22, 2-28
Loop 3-5, 3-6, 3-7, 3-9
Loop comparator 3-3, 3-5, 3-6, 3-10
Loop counter 1-7, 3-4, 3-5, 12-4
Loop stack 3-4, 3-5, 3-6, 3-7, 3-22, 12-5
LSHIFT 2-31

M

M registers 4-1, 4-2, 4-3, 5-26, 5-28, 8-14, 12-2
MAC 1-6, 2-13, 2-15, 3-21, 12-2, 12-6, 15-9, 15-10
MAC arithmetic 2-3
MAC input/output registers 2-18
MAC operations 2-16
MAC overflow 2-19
MAC saturation 2-19
Matrix multiply 14-1, 14-9
Memory address 3-3
Memory interface 10-2, 10-37, 12-2, 12-7
Memory read 10-3
Memory wait states 3-18
Memory write 10-3, 10-24
Memory-mapped registers 12-1
MF register 2-15, 2-20
Miscellaneous instructions 15-16
MMAP 7-16, 10-5, 10-6, 10-15
Mode bits 12-5
Mode control 12-5, 15-16
Mode status register (MSTAT) 3-12, 3-22
Modify (M) registers 4-1, 4-2, 4-3, 5-26, 5-28
Modulo addressing 4-1, 4-4
Modulus logic 4-3
MOVE instructions 15-13
MR register 2-13, 2-15, 2-16, 2-18, 2-20, 2-22
MR0 register 2-13, 2-15, 2-18
MR1 register 2-13, 2-15, 2-19, 2-20
MR2 register 2-13, 2-15, 2-19, 2-20
MSTAT 1-7, 2-8, 2-9, 2-16, 2-24, 3-12, 3-14, 3-20, 3-22, 4-2, 6-1, 12-5
Multichannel 1-2, 1-8, 5-3, 5-5, 5-30
Multichannel frame delay 5-32
Multichannel length bit 5-31
Multichannel mode 5-30, 5-31, 5-32
Multichannel operation 5-32, 5-33
Multichannel setup 5-30
Multichannel transfer 5-32, 5-33
Multifunction instructions 1-4, 15-7, 15-19
Multiplication 2-13
Multiplier 2-18
Multiplier result format 2-17
Multiplier/accumulator 1-6, 2-13, 2-15, 3-21
Multiply/add 1-6
Multiply/subtract 1-6
Multiprecision capability 2-8
Multiprocessing 9-4
MV 2-13, 2-19
MX register file 2-15
MX0 register 2-15
MX1 register 2-15
MY register file 2-15
MY0 register 2-15
MY1 register 2-15

N

Nested loops 3-8
Nesting 3-15, 3-16
Next instruction address 3-3, 3-4, 3-7, 3-10
Normal framing mode 5-13
Normalize 2-22, 2-28, 2-33
Normalize modifiers 2-35
Numeric formats C-1

O

Off-chip memory accesses 15-18
On-chip memory 1-2
On-chip peripherals 1-2, 1-8
Opcodes A-1
Operating mode 3-22

Operation during powerdown 9-23
OR/PASS logic 2-22, 2-24
Output enable 10-3, 10-24
Output registers 1-7
Overflow (AV) ... 2-2, 2-5, 2-8, 2-9, 2-13, 2-26, 2-36
Overwrite bit 9-24
Overwrite mode 7-7, 7-9
OWRCNTR 3-5

P

Page length 10-15, 10-18
PASS 2-32, 2-33
PC 3-4
PC incrementer 3-3
PC stack ..... 3-3, 3-4, 3-6, 3-10, 3-11, ... 3-22, 12-5, 15-84, 15-85
PDFORCE 9-18
Period register 6-1
Periodic interrupts 6-1
PMA bus 1-8, 3-3, 3-4, 3-8, 10-1, 10-3
PMD bus 1-8, 10-1, 10-3
PMD-DMD bus exchange ... 1-5, 1-8, 2-15, 4-1, 4-9
PMOVLAY 10-26
PMS 1-8, 10-3, 10-24
Polled operation 7-7
PORT directive 10-14
Powerdown 8-9, 9-17, 9-20, 9-30
Powerdown acknowledge 9-29
Powerdown control register 9-18, 9-20
Powerdown force control bit 9-19
Powerup 9-20
Powerup boot 10-16
Primary registers 2-7
Processor states 9-2
Program counter 3-4
Program flow control 15-14, 15-15
Program memory 1-8, 10-1
Program memory configuration 10-5
Program memory data bus 4-9
Program memory interface 10-3
Program memory map 10-5, 10-6
Program memory read 10-3, 15-6
Program memory write 10-3
Program sequencer 3-1, 3-5, 12-2, 12-4
Programming model 12-1
PUCR 9-18
PWAIT 10-5
PWD pin 9-19
PWDACK pin 9-29
PX registers 1-5, 4-9, 4-10

Q

Quotient format 2-12

R

R bus 1-6, 2-15, 2-18, 2-22
RAM 10-6
Read operation 15-12
Receive companding latency 5-40
Receive frame sync 5-11, 5-12, 5-30
Receive interrupt (SPORT) 5-4, 5-36
Receive register 5-6
Receive word enables 5-31
Receiving data 5-6
reg 15-12, 15-14
Register indirect JUMPs 3-8
Register notation 15-20
Registers 12-1, 12-3
RESET 9-1, 9-4, 9-21, 9-25
Result bus 1-6
Return 3-7, 3-10
RFS 5-2, 5-11, 5-12, 5-30
RFSDIV 5-12
RFSDIV 12-7
RFSR 5-10, 5-11
RFSW 5-13
Rounding mode 2-20
RTI instruction 3-11, 9-15, 9-30
RX register 5-2, 5-4, 5-32, 5-36
RX0 register 5-6, 5-32
RX1 register 5-6

S

Saturation 2-13
SB register 2-22, 2-29
SCLK 5-1, 5-12, 5-13, 5-15, 5-16
SCLK frequencies 5-9
SCLK pin 5-9
SCLK0 5-34
SCLK1 5-34
SCLKDIV 5-9, 12-7
SE register .. 2-22, 2-24, 2-26, 2-30, 2-31, 2-33, 2-34
Secondary register 3-12
Serial clock 5-1, 5-5, 5-8, 5-34, 5-38
Serial clock frequencies 5-8
Serial port autobuffering 5-42
Serial ports ..... 1-8, 5-1, 5-38, 9-5, 9-23, 12-7, ... 13-5, 13-6, 13-7, 13-8, 13-10
Serial word length 5-5, 5-9
Shifter 1-6, 3-21, 12-2, 15-9, 15-11

Index

Shifter arithmetic 2-3
Shifter array 2-22
Shifter input/output registers 2-28
Shifter operations 2-28
Shifter sign 2-26
SI register 2-22, 2-23
Signed numbers 2-1
Sine approximation 14-7
SLEN 5-10, 5-16, 5-31, 5-32
Software examples 14-1
Software reboot 10-16
SPORT 1-8, 3-18, 5-1, 5-3, 5-6, 9-23
SPORT control register ..... 5-8, 5-10, 5-11, 5-13, ... 5-14, 5-16, 5-23, 5-31
SPORT enable 5-7
SPORT interrupts 5-3, 5-34, 5-41
SPORT multichannel frame delay 5-31
SPORT programming 5-4
SPORT timing 5-34
SPORT configuration 5-5
SPORT0 ..... 5-1, 5-5, 5-15, 5-16, 5-30, ... 8-13, 12-7, 13-6, 13-7
SPORT0 configuration registers 5-5
SPORT0 control register 5-30
SPORT0 multichannel word enable registers . 5-32
SPORT1 ..... 5-5, 5-30, 8-9, ... 8-13, 9-14, 9-15, 12-2
SPORT1 alternate configuration 5-8
SPORT1 configuration registers 5-5
SR register 2-22, 2-23
SR0 register 2-22, 2-24
SR1 register 2-22, 2-24
SSTAT 12-5
Stacks 3-4, 12-5
Start-up delay 9-20
Start-up time 9-21
Startup timing 5-38
Status bits 3-21, 12-5
Status condition 3-6, 3-25
Status logic 3-4
Status registers 3-15, 3-20
Status stack 3-16, 3-20, 3-22, 12-5
Stolen cycles 15-18
Subroutine 3-9
Subtract with borrow 2-8
Synchronization delay 9-3
Synchronization (serial clk to processor clk) ... 5-38
System Builder 1-10
System Control Register ..... 9-14, 9-15, ... 10-17, 15-12, E-1
System interface 9-1

T

T1 interface 5-31
TCOUNT 6-1, 6-2, 6-3, 6-4, 12-6
TDV 5-32
Termination condition 3-6, 3-10
TFS 5-2, 5-11, 5-12, 5-30
TFS0 5-30
TFSR 5-10, 5-11
TFSW 5-13
Time-division multiplexed 5-30
Timer 3-18, 3-23, 3-24, 6-1, 9-5, 12-2, 12-6
Timer interrupt 3-19, 6-1
Timer operation 6-3
Timer registers 6-1, 6-2
TOPPCSTACK ..3-4, 3-25, 3-26, 3-27, 15-84, 15-85
TPERIOD 6-1, 6-2, 6-3, 6-4, 12-6
Transmit data valid 5-32
Transmit frame sync 5-11, 5-12
Transmit interrupt (SPORT) 5-4, 5-36
Transmit register 5-6
Transmit word enables 5-32
Transmitting data 5-6
TSCALE 6-1, 6-2, 6-3, 6-4, 12-6
Twos-complement 2-1, 2-18, 2-33, C-1, C-4
TX register 5-2, 5-4, 5-35
TX0 register 5-6
TX1 register 5-6

U

Underflows 2-8
Unsigned 2-1, 2-18, C-1

W

Word length 5-3, 5-9, 5-32
Write enable 10-3
Write operation 15-12

X

XTAL pin 9-1, 9-21, 9-22
XTALDELAY 9-18, 9-20
XTALDIS 9-18

Z

Zero-overhead looping 1-5, 3-1