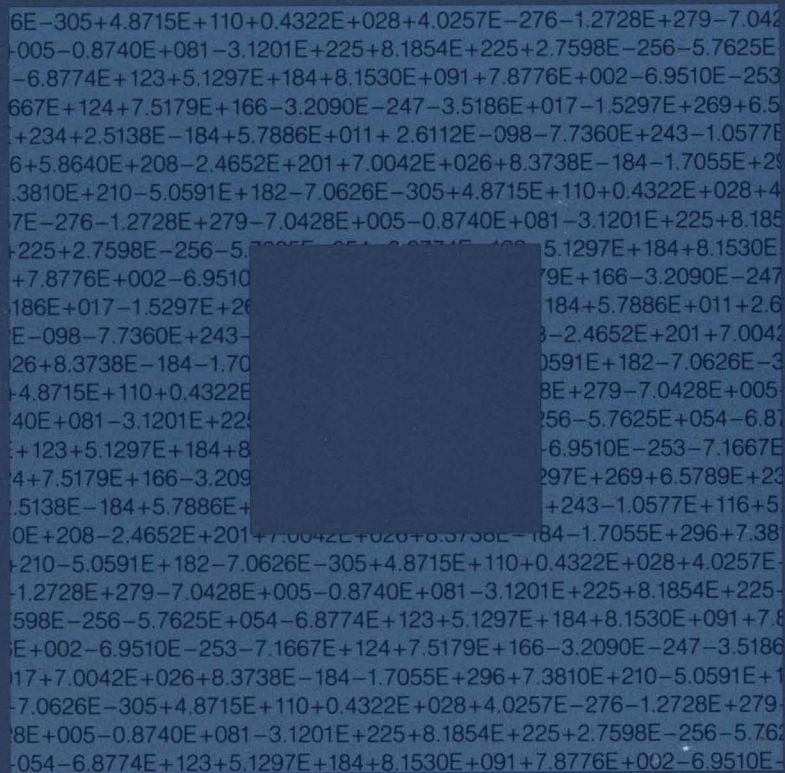
ADSP-21020/21010 User's Manual

ANALOG DEVICES



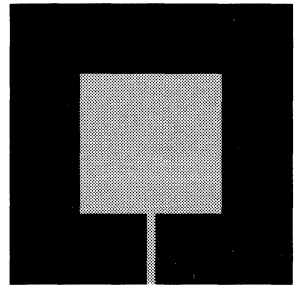
ANALOG DEVICES

ADSP-21020
ADSP-21010

User's Manual

ADSP-21020
ADSP-21010

User's Manual



ANALOG DEVICES

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

- By contacting your local Analog Devices Sales Representative
- For Marketing information, call (617) 461-3881 in Norwood, Massachusetts, USA
- For Applications Engineering information, call (617) 461-3672 in Norwood, Massachusetts, USA
- The Norwood office Fax number is (617) 461-3010
- The Norwood office may also be reached by
  Telex: 924491
  TWX: 710/394-6577
  Cables: ANALOG NORWOODMASS
- The DSP Group runs a Bulletin Board Service that can be reached at 300, 1200, 2400 or 9600 baud, no parity, 8 bits data, 1 stop bit by dialing:
  (617) 461-4258
- By writing to:
  Analog Devices
  DSP
  One Technology Way
  P.O. Box 9106
  Norwood, MA 02062-9106
  USA

ADSP-21020/21010 User's Manual

© 1993 Analog Devices, Inc.
ALL RIGHTS RESERVED

Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use; nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under the patent rights of Analog Devices.

PRINTED IN U.S.A. SECOND EDITION

INTRODUCTION 1

COMPUTATION UNITS 2

PROGRAM SEQUENCING 3

DATA ADDRESSING 4

TIMER 5

MEMORY INTERFACE 6

INSTRUCTION SUMMARY 7

ASSEMBLY PROGRAMMING TUTORIAL 8

HARDWARE SYSTEM DESIGN 9

INSTRUCTION SET REFERENCE A

COMPUTE OPERATION REFERENCE B

IEEE 1149.1 JTAG TEST ACCESS PORT C

NUMERIC FORMATS D

CONTROL/STATUS REGISTERS E

Contents

CHAPTER 1 INTRODUCTION

1.1 GENERAL DESCRIPTION 1–1
1.1.1 Key Enhancements 1–2
1.1.2 Why Floating-Point? 1–3
1.1.3 Future Product Migration Path 1–4
1.2 ARCHITECTURE OVERVIEW 1–4
1.2.1 Computation Units 1–4
1.2.2 Address Generators And Program Sequencer 1–6
1.2.3 Interrupts 1–7
1.2.4 Timer 1–7
1.2.5 Memory Buses And Interface 1–7
1.2.6 Internal Data Transfers 1–8
1.2.7 Context Switching 1–8
1.2.8 Instruction Set 1–8
1.3 DEVELOPMENT SYSTEM 1–9
1.4 MANUAL ORGANIZATION 1–11

CHAPTER 2 COMPUTATION UNITS

2.1 OVERVIEW 2–1
2.2 IEEE FLOATING-POINT OPERATIONS 2–2
2.2.1 Extended Floating-Point Precision (ADSP-21020 Only) 2–3
2.2.2 Floating-Point Exceptions 2–3
2.3 FIXED-POINT OPERATIONS 2–4
2.4 ROUNDING 2–4
2.5 ALU 2–5
2.5.1 ALU Operation 2–5
2.5.2 ALU Operating Modes 2–6
2.5.2.1 Saturation Mode 2–6
2.5.2.2 Floating-Point Rounding Modes 2–6
2.5.2.3 Floating-Point Rounding Boundary 2–7
2.5.3 ALU Status Flags 2–7
2.5.3.1 ALU Zero Flag (AZ) 2–8
2.5.3.2 ALU Underflow Flag (AZ, AUS) 2–8

Contents

2.5.3.3 ALU Negative Flag (AN) 2–8
2.5.3.4 ALU Overflow Flag (AV, AOS, AVS) 2–8
2.5.3.5 ALU Fixed-Point Carry Flag (AC) 2–8
2.5.3.6 ALU Sign Flag (AS) 2–9
2.5.3.7 ALU Invalid Flag (AI) 2–9
2.5.3.8 ALU Floating-Point Flag (AF) 2–9
2.5.3.9 Compare Accumulation 2–9
2.5.4 ALU Instruction Summary 2–10
2.6 MULTIPLIER 2–11
2.6.1 Multiplier Operation 2–11
2.6.2 Fixed-Point Results 2–12
2.6.2.1 MR Registers 2–12
2.6.3 Fixed-Point Operations 2–13
2.6.3.1 Clear MR Register 2–14
2.6.3.2 Round MR Register 2–14
2.6.3.3 Saturate MR Register On Overflow 2–14
2.6.4 Floating-Point Operating Modes 2–15
2.6.4.1 Floating-Point Rounding Modes 2–15
2.6.4.2 Floating-Point Rounding Boundary 2–15
2.6.5 Multiplier Status Flags 2–15
2.6.5.1 Multiplier Underflow Flag (MU) 2–16
2.6.5.2 Multiplier Negative Flag (MN) 2–17
2.6.5.3 Multiplier Overflow Flag (MV) 2–17
2.6.5.4 Multiplier Invalid Flag (MI) 2–17
2.6.6 Multiplier Instruction Summary 2–18
2.7 SHIFTER 2–19
2.7.1 Shifter Operation 2–19
2.7.2 Bit Field Deposit & Extract Instructions 2–20
2.7.3 Shifter Status Flags 2–24
2.7.3.1 Shifter Zero Flag (SZ) 2–24
2.7.3.2 Shifter Overflow Flag (SV) 2–24
2.7.3.3 Shifter Sign Flag (SS) 2–24
2.7.4 Shifter Instruction Summary 2–25
2.8 MULTIFUNCTION COMPUTATIONS 2–26
2.9 REGISTER FILE 2–27
2.9.1 Alternate (Secondary) Registers 2–28

Contents

CHAPTER 3 PROGRAM SEQUENCING

3.1 OVERVIEW 3–1
3.1.1 Instruction Cycle 3–2
3.1.2 Program Sequencer Architecture 3–3
3.1.2.1 Program Sequencer Registers & System Registers 3–5
3.2 PROGRAM SEQUENCER OPERATIONS 3–6
3.2.1 Sequential Instruction Flow 3–6
3.2.2 Program Memory Data Access 3–6
3.2.3 Branches 3–6
3.2.4 Loops 3–6
3.3 CONDITIONAL INSTRUCTION EXECUTION 3–7
3.4 BRANCHES (CALL, JUMP, RTS, RTI) 3–9
3.4.1 Delayed And Nondelayed Branches 3–10
3.4.2 PC Stack 3–12
3.5 LOOPS 3–13
3.5.1 Restrictions And Short Loops 3–14
3.5.1.1 General Restrictions 3–14
3.5.1.2 Counter-Based Loops 3–15
3.5.1.3 Non-Counter-Based Loops 3–17
3.5.2 Loop Address Stack 3–17
3.5.3 Loop Counters And Stack 3–18
3.5.3.1 CURLCNTR 3–18
3.5.3.2 LCNTR 3–19
3.6 INTERRUPTS 3–20
3.6.1 Interrupt Latency 3–21
3.6.2 Interrupt Latch 3–23
3.6.2.1 Interrupt Priority 3–24
3.6.2.2 Software Interrupts 3–25
3.6.3 Interrupt Masking And Control 3–25
3.6.3.1 Interrupt Mask 3–25
3.6.3.2 Interrupt Nesting & IMASKP 3–26
3.6.4 Status Stack 3–26
3.6.5 External Interrupt Timing & Sensitivity 3–27
3.6.5.1 Asynchronous External Interrupts 3–28
3.7 STACK FLAGS 3–28
3.8 IDLE 3–29
3.9 INSTRUCTION CACHE 3–30
3.9.1 Cache Architecture 3–30
3.9.2 Cache Efficiency 3–32
3.9.3 Cache Enable And Cache Freeze 3–32

Contents

CHAPTER 4 DATA ADDRESSING

4.1 OVERVIEW 4–1
4.2 DAG REGISTERS 4–1
4.2.1 Alternate DAG Registers 4–3
4.3 DAG OPERATION 4–4
4.3.1 Address Output And Modification 4–4
4.3.1.1 DAG Modify Instructions 4–4
4.3.1.2 Immediate Modifiers 4–6
4.3.2 Circular Buffer Addressing 4–6
4.3.2.1 Circular Buffer Operation 4–6
4.3.2.2 Circular Buffer Registers 4–8
4.3.2.3 Circular Buffer Overflow Interrupts 4–8
4.3.3 Bit-Reversal 4–9
4.3.3.1 Bit-Reverse Mode 4–9
4.3.3.2 Bit-Reverse Instruction 4–10
4.4 DAG REGISTER TRANSFERS 4–10
4.4.1 DAG Register Transfer Restrictions 4–11

CHAPTER 5 TIMER

5.1 OVERVIEW 5–1
5.2 TIMER OPERATION 5–1
5.2.1 Timer Enable And Disable 5–1
5.2.2 Timer Interrupt 5–4
5.3 TIMER REGISTERS 5–4

CHAPTER 6 MEMORY INTERFACE

6.1 MEMORY MANAGEMENT AND INTERFACE 6–1
6.2 MEMORY BUSES AND CONTROL PINS 6–2
6.3 MEMORY INTERFACE TIMING 6–4
6.3.1 Memory Read 6–4
6.3.2 Memory Write 6–5
6.3.3 Three-State Controls 6–6
6.4 MEMORY BANKS 6–8
6.5 WAIT STATES (EXTENDED MEMORY CYCLES) 6–9
6.5.1 Extended Data Memory Address Hold Time 6–10
6.6 MEMORY PAGE BOUNDARY DETECTION 6–12
6.6.1 Page Size 6–12
6.6.2 Wait States On Page Boundary Crossings 6–13
6.7 BUS REQUEST/BUS GRANT 6–13
6.8 BUS EXCHANGE (PX REGISTERS) 6–15

Contents

CHAPTER 7 INSTRUCTION SUMMARY

7.1 OVERVIEW 7–1
7.2 IMPORTANT PROGRAMMING REMINDERS 7–2
7.2.1 Extra Cycle Conditions 7–2
7.2.1.1 Nondelayed Branches 7–2
7.2.1.2 Program Memory Data Access With Cache Miss 7–2
7.2.1.3 Program Memory Data Access In Loops 7–2
7.2.1.4 One- And Two-Instruction Loops 7–3
7.2.1.5 DAG And Memory Control Register Writes 7–4
7.2.1.6 Wait States 7–5
7.2.1.7 Page Boundary Crossing 7–5
7.2.1.8 Three-State Enables 7–5
7.2.1.9 Bus Request/Bus Grant 7–5
7.2.2 Delayed Branch Restrictions 7–5
7.2.3 Loop Restrictions 7–6
7.2.4 Interrupts 7–6
7.2.5 IRPTL 7–7
7.2.6 Effect Latency And Read Latency 7–7
7.2.7 CURLCNTR Write & LCE 7–8
7.2.8 Circular Buffer Initialization 7–8
7.2.9 Bit-Reverse Mode And Data Memory Bank Select 7–8
7.2.10 Disallowed DAG Register Transfers 7–8
7.2.11 Two Writes To Register File 7–9
7.2.12 Stack Status Flags 7–9
7.2.13 Wait States And Three-State Enables 7–9
7.2.14 Computation Units 7–9

CHAPTER 8 ASSEMBLY PROGRAMMING TUTORIAL

8.1 INTRODUCTION 8–1
8.2 EXAMPLE #1: DATA IN MEMORY, NO INTERRUPTS 8–2
8.2.1 File Inventory 8–5
8.2.2 Architecture Description File (generic.ach) 8–5
8.2.3 External vs. Internal Address Decoding 8–7
8.2.4 Specifying The .ACH File 8–8
8.2.5 Main Program (iirmem.asm) 8–8
8.2.5.1 Initial Setups: Initialization Following Reset 8–11
8.2.5.2 Main Processing Loop 8–14
8.2.6 Creating The Executable Program 8–15
8.2.7 Simulation 8–15

Contents

8.3 EXAMPLE #2—INTERRUPT-DRIVEN, WITH PORT I/O 8–16
8.3.1 File Inventory 8–16
8.3.2 Architecture Description File (iirirq.ach) 8–16
8.3.3 Main Program (iirirq.asm) 8–19
8.3.3.1 Initialization Following Reset (Initial Setups) 8–23
8.3.3.2 Main Processing Loop 8–28
8.3.3.3 Terminating The Main Processing Loop 8–29
8.3.4 Creating The Executable Program 8–29
8.3.5 Simulation 8–29
8.4 CALLED SUBROUTINES (cascade.asm) 8–31
8.4.1 Writing Looped Code 8–31
8.4.2 Rolling Loops For More Efficient Code 8–32
8.4.3 Multifunction Instructions And Register Restrictions 8–34
8.5 DEVELOPING THE IIR FILTER AND COEFFICIENTS 8–35
8.5.1 Normalized b_0 Coefficient Biquad Filter Design Method 8–35
8.5.2 DSP Code Generation 8–36
8.5.3 Coefficient Formatting 8–36
8.6 PROGRAMMING HINTS 8–37
8.6.1 System Considerations For Scoping 8–37
8.6.2 Delayed Branches 8–39
8.6.3 Multifunction Instruction Coding 8–40
8.7 COMPLETE FFT EXAMPLE 8–40

CHAPTER 9 HARDWARE SYSTEM DESIGN

9.1 OVERVIEW 9–1
9.1.1 Basic System Configuration 9–1
9.1.2 More Complex Configurations 9–2
9.2 CLOCKS & SYNCHRONIZATION 9–3
9.2.1 Synchronization Delay 9–3
9.3 POWERUP & RESET 9–4
9.4 RCOMP PIN 9–6
9.5 FLAGS 9–7
9.5.1 Flag Direction 9–7
9.5.2 Flag Input 9–7
9.5.3 Flag Output 9–8
9.6 MEMORY CONFIGURATIONS 9–8
9.6.1 Single Processor Configurations 9–8
9.6.1.1 One Memory Bank 9–9
9.6.1.2 Several Memory Banks 9–10
9.6.1.3 Memory & I/O Devices 9–11
9.6.1.4 Hardware Acknowledge 9–12

Contents

9.6.1.5 Cache Memory 9–13
9.6.1.6 DRAM With Paging 9–14
9.6.1.7 Direct Memory Access (DMA) 9–16
9.6.2 Multiprocessor Configurations 9–18
9.6.2.1 Multiport Memory 9–18
9.6.2.2 Serial Data Flow 9–20
Buffer Latches 9–20
FIFOs 9–22
Dual-Port Memory 9–24
9.7 PROGRAM MEMORY BOOT AT RESET 9–25
9.8 MEMORY INTERFACE CAPACITIVE LOAD 9–27
9.8.1 Load Variations 9–27
9.8.2 Correcting The Timing 9–29
9.9 EZ-ICE EMULATOR CONSIDERATIONS 9–30
9.9.1 Target Board Connector For EZ-ICE Probe 9–30
9.9.2 Other Hardware Considerations 9–31
9.10 HOST PROCESSOR INTERFACE 9–32
9.10.1 Data Transfer Sequences 9–32
9.10.2 Host Interface Code Examples 9–37
9.10.2.1 Buffer Transfers 9–37
9.10.2.2 Interrupt-Driven Transfers 9–37
9.10.2.3 High Speed Transfers 9–38

APPENDIX A INSTRUCTION SET REFERENCE

A.1 OVERVIEW A–1
A.2 INSTRUCTION SYNTAX NOTATION A–2
A.3 OPCODE NOTATION A–3
A.4 CONDITION CODES A–8
A.5 UNIVERSAL REGISTERS A–9

GROUP I.
COMPUTE AND MOVE INSTRUCTIONS A–11

compute/dreg ↔ DM / dreg ↔ PM A–12
compute A–13
compute/ureg ↔ DM/PM, register modify A–14
compute/dreg ↔ DM/PM, immediate modify A–16
compute/ureg ↔ ureg A–18
immediate shift / dreg ↔ DM/PM A–20
compute/modify A–22

Contents

GROUP II.
PROGRAM FLOW CONTROL A–23

direct jump/call A–24
indirect jump/call / compute A–26
return from subroutine/interrupt / compute A–28
do until counter expired A–30
do until A–32

GROUP III.
IMMEDIATE MOVE A–33

ureg ↔ DM/PM (direct addressing) A–34
ureg ↔ DM/PM (indirect addressing) A–35
immediate data → DM/PM A–36
immediate data → ureg A–37

GROUP IV.
MISCELLANEOUS A–39

system register bit manipulation A–40
I register modify / bit-reverse A–42
push/pop stacks A–44
nop A–45
idle A–46

APPENDIX B COMPUTE OPERATION REFERENCE

B.1 OVERVIEW B–1
B.2 SINGLE-FUNCTION OPERATIONS B–1
B.2.1 ALU Operations B–2
Rn = Rx + Ry B–4
Rn = Rx – Ry B–5
Rn = Rx + Ry + CI B–6
Rn = Rx – Ry + CI – 1 ..... B–7
Rn = (Rx + Ry)/2 ..... B–8
COMP(Rx, Ry) B–9
Rn = Rx + CI B–10
Rn = Rx + CI – 1 ..... B–11
Rn = Rx + 1 ..... B–12

Contents

Rn = Rx – 1 ..... B–13
Rn = –Rx B–14
Rn = ABS Rx B–15
Rn = PASS Rx B–16
Rn = Rx AND Ry B–17
Rn = Rx OR Ry B–18
Rn = Rx XOR Ry B–19
Rn = NOT Rx B–20
Rn = MIN(Rx, Ry) B–21
Rn = MAX(Rx, Ry) B–22
Rn = CLIP Rx BY Ry B–23
Fn = Fx + Fy B–24
Fn = Fx – Fy B–25
Fn = ABS (Fx + Fy) B–26
Fn = ABS (Fx – Fy) B–27
Fn = (Fx + Fy)/2 ..... B–28
COMP(Fx, Fy) B–29
Fn = –Fx B–30
Fn = ABS Fx B–31
Fn = PASS Fx B–32
Fn = RND Fx B–33
Fn = SCALB Fx BY Ry B–34
Rn = MANT Fx B–35
Rn = LOGB Fx B–36
Rn = FIX Fx BY Ry/Rn = FIX Fx B–37
Fn = FLOAT Rx BY Ry/Fn = FLOAT Rx B–38
Fn = RECIPS Fx B–39
Fn = RSQRTS Fx B–40
Fn = Fx COPYSIGN Fy B–41
Fn = MIN(Fx, Fy) B–42
Fn = MAX(Fx, Fy) B–43
Fn = CLIP Fx BY Fy B–44

B.2.2 Multiplier Operations B–45
RnlMR = Rx * Ry B–47
RnlMR = MR + Rx * Ry B–48
RnlMR = MR – Rx * Ry B–49
RnlMR = SAT MR B–50
RnlMR = RND MR B–51
MR = 0 ..... B–52
MR=Rn / Rn=MR B–52
Fn = Fx * Fy B–53

Contents

B.2.3 Shifter Operations B–54
Rn = LSHIFT Rx BY Ryl<data8> B–55
Rn = Rn OR LSHIFT Rx BY Ryl<data8> B–56
Rn = ASHIFT Rx BY Ryl<data8> B–57
Rn = Rn OR ASHIFT Rx BY Ryl<data8> B–58
Rn = ROT Rx BY RYl<data8> B–59
Rn = BCLR Rx BY Ryl<data8> B–60
Rn = BSET Rx BY Ryl<data8> B–61
Rn = BTGL Rx BY Ryl<data8> B–62
BTST Rx BY Ryl<data8> B–63
Rn = FDEP Rx BY Ryl<bit6>:<len6> B–64
Rn = Rn OR FDEP Rx BY Ryl<bit6>:<len6> B–65
Rn = FDEP Rx BY Ryl<bit6>:<len6> (SE) B–66
Rn = Rn OR FDEP Rx BY Ryl<bit6>:<len6> (SE) B–67
Rn = FEXT Rx BY Ryl<bit6>:<len6> B–68
Rn = FEXT Rx BY Ryl<bit6>:<len6> (SE) B–69
Rn = EXP Rx B–70
Rn = EXP Rx (EX) B–71
Rn = LEFTZ Rx B–72
Rn = LEFTO Rx B–73
B.2.4 Multifunction Computations B–74
Dual Add/Subtract (Fixed-Pt.) B–75
Dual Add/Subtract (Floating-Pt) B–76
Parallel Multiplier & ALU (Fixed-Pt.) B–77
Parallel Multiplier & ALU (Floating-Pt.) B–78
Parallel Multiplier & Dual Add/Subtract B–80

APPENDIX C IEEE 1149.1 JTAG TEST ACCESS PORT

C.1 OVERVIEW C–1
C.2 TEST ACCESS PORT C–2
C.3 INSTRUCTION REGISTER C–2
C.4 BOUNDARY REGISTER C–4
C.5 DEVICE IDENTIFICATION REGISTER C–11
C.6 BUILT-IN SELF-TEST OPERATION (BIST) C–11
C.7 PRIVATE INSTRUCTIONS C–11
C.8 REFERENCES C–11

Contents

APPENDIX D NUMERIC FORMATS

D.1 OVERVIEW D–1
D.2 IEEE SINGLE-PRECISION FLOATING-POINT DATA FORMAT D–1
D.3 EXTENDED FLOATING-POINT FORMAT D–2
D.4 FIXED-POINT FORMATS D–3

APPENDIX E CONTROL/STATUS REGISTERS

E.1 OVERVIEW E–1
E.2 SYSTEM REGISTERS E–1
E.2.1 System Register Bit Operations E–2
E.2.1.1 Bit Test Flag E–2
E.2.2 User Registers E–2
E.3 MODE1 REGISTER E–3
E.4 MODE2 REGISTER E–4
E.5 ARITHMETIC STATUS REGISTER (ASTAT) E–5
E.6 STICKY ARITHMETIC STATUS REGISTER (STKY) E–6
E.7 INTERRUPT LATCH (IRPTL) & INTERRUPT MASK (IMASK) E–8
E.8 PROGRAM MEMORY INTERFACE CONTROL (PMWAIT) E–10
E.9 DATA MEMORY INTERFACE CONTROL (DMWAIT) E–11

REVISIONS FOR 2ND EDITION

INDEX

FIGURES

Figure 1.1 ADSP-21020 Block Diagram 1–5
Figure 1.2 Program Development 1–9

Figure 2.1 Computation Units 2–2
Figure 2.2 Multiplier Fixed-Point Result Placement 2–12
Figure 2.3 MR Transfer Formats 2–13
Figure 2.4 Register File Fields for Shifter Instructions 2–20
Figure 2.5 Register File Fields for FDEP, FEXT Instructions 2–20
Figure 2.6 Bit Field Deposit Instruction 2–21
Figure 2.7 Bit Field Deposit Example 2–22
Figure 2.8 Bit Field Extract Example 2–23

Contents

Figure 3.1 Program Flow Variations 3–2
Figure 3.2 Pipelined Execution Cycles 3–3
Figure 3.3 Program Sequencer Block Diagram 3–4
Figure 3.4 Nondelayed Branches 3–10
Figure 3.5 Delayed Branches 3–11
Figure 3.6 Loop Operation 3–14
Figure 3.7 One-Instruction Loops 3–15
Figure 3.8 Two-Instruction Loops 3–16
Figure 3.9 Pushing the Loop Counter Stack for Nested Loops 3–20
Figure 3.10 Interrupt Handling 3–22
Figure 3.11 Instruction Cache Architecture 3–31
Figure 3.12 Cache-Inefficient Code 3–33

Figure 4.1 Data Address Generator Block Diagram 4–2
Figure 4.2 Alternate DAG Registers 4–3
Figure 4.3 Pre-Modify and Post-Modify Operations 4–5
Figure 4.4 Circular Data Buffers 4–7
Figure 4.5 DAG Register Transfers 4–11

Figure 5.1 Timer Block Diagram 5–2
Figure 5.2 TIMEXP Signal 5–2
Figure 5.3 Timer Enable and Disable 5–3
Figure 5.4 Timer Interrupt Timing 5–4

Figure 6.1 Memory Read Cycle 6–5
Figure 6.2 Memory Write Cycle 6–6
Figure 6.3 Wait State Control Registers 6–11
Figure 6.4 Bus Request/Bus Grant Timing 6–14
Figure 6.5 PX Register Transfers 6–15

Figure 8.1 Program Flow for First Example 8–3
Figure 8.2 Program Flow for Second Example 8–4
Figure 8.3 Physical Memory Architecture Described in "generic.ach" 8–8
Figure 8.4 Memory Map Described in "generic.ach" 8–9
Figure 8.5 Physical Memory Architecture Described in "iirirq.ach" 8–17
Figure 8.6 Memory Map Described in "iirirq.ach" 8–18
Figure 8.7 Filling and Draining the Pipeline 8–33
Figure 8.8 Loop Code Before Rolling 8–33

Figure 9.1 Basic ADSP-21020 System Configuration 9–2
Figure 9.2 Program Memory Interface Timing at Reset 9–6
Figure 9.3 Flag Output Timing 9–8
Figure 9.4 Interface to Single Data Memory Bank 9–9
FIgure 9.5 Interface to Three Data Memory Banks 9–10

Contents

Figure 9.6 Interface to Three Data Memory Banks and Two I/O Devices 9–11
Figure 9.7 I/O Device Interface with Hardware Acknowledge 9–12
Figure 9.8 Cache Memory Interface 9–13
Figure 9.9 Timing on Cache Miss 9–14
Figure 9.10 Page-Mode DRAM Interface 9–15
Figure 9.11 DMA Controller Interface Using Bus Request 9–16
Figure 9.12 Bus Request Timing for DMA 9–17
Figure 9.13 Four-Port RAM Configuration 9–19
Figure 9.14 Serial Data Flow Using Buffers 9–20
Figure 9.15 Serial Data Flow Using FIFOs 9–22
Figure 9.16 Serial Data Flow Using Dual-Port RAM 9–24
Figure 9.17 Example Loader Program 9–26
Figure 9.18 Memory Configuration with Unequal Capacitive Loads 9–28
Figure 9.19 Effect of Unequal Capacitive Loads 9–29
Figure 9.20 Target Board Connector For EZ-ICE Probe (jumpers in place) 9–31
Figure 9.21 Host Interface Block Diagram 9–33
Figure 9.22 Host Interface Logic 9–34
Figure 9.23 Host Write Timing 9–35
Figure 9.24 Host Read Timing 9–36

Figure A.1 Map 1 Universal Register Addresses A–9
Figure A.2 Map 2 Universal Register Addresses A–10

Figure C.1 Serial Scan Paths C–3

Figure D.1 IEEE 32-Bit Single-Precision Floating-Point Format D–1
Figure D.2 40-Bit Extended-Precision Floating-Point Format D–2
Figure D.3 32-Bit Fixed-Point Formats D–3
Figure D.4 64-Bit Unsigned Fixed-Point Product D–4
Figure D.5 64-Bit Signed Fixed-Point Product D–5

TABLES

Table 3.1 Program Sequencer Registers & System Registers 3–5
Table 3.2 Condition Codes 3–8
Table 3.3 Interrupt Vectors and Priority 3–24
Table 3.4 States of Outputs During IDLE 3–29

Table 7.1 Syntax Notation Conventions 7–10
Table 7.2 Condition and Termination Codes 7–11
Table 7.3 Universal Registers and System Registers 7–12
Table 7.4 ALU Instructions 7–13

Contents

Table 7.5 Multiplier Instructions 7–14
Table 7.6 Shifter and Shifter Immediate Instructions 7–15
Table 7.7 Multifunction Instructions 7–16
Table 7.8 Interrupt Vectors and Priority 7–17

Table 8.1 Files Used for Memory-Based (No Interrupts) Program 8–5
Table 8.2 Files Used for Interrupt-Driven Program using Port I/O 8–16
Table 8.3 Interrupt-Related Registers 8–26

Table 9.1 ADSP-21020 Register Values After Reset 9–5
Table 9.2 ADSP-21020 Pin States During Reset (while $\overline{\text{RESET}}$=0) 9–6

Table B.1 Fixed-Point ALU Operations B–2
Table B.2 Floating-Point ALU Operations B–3
Table B.3 Multiplier Operations B–45
Table B.4 Multiplier Mod2 Options B–46
Table B.5 Multiplier Mod1 Options B–46
Table B.6 Shifter Operations B–54
Table B.7 Parallel Multiplier/ALU Computations B–79

Table C.1 Test Instructions C–3

Table D.1 IEEE Single-Precision Floating-Point Data Types D–2

Table E.1 System Registers E–1

LISTINGS

Listing 8.1 generic.ach 8–6
Listing 8.2 iirmem.asm 8–10
Listing 8.3 iirirq.ach 8–17
Listing 8.4 iirirq.asm 8–19
Listing 8.5 def21020.h 8–21
Listing 8.6 Input Data Read by Input Port (Normalized Unit Impulse) 8–30
Listing 8.7 Output Data Stored by Output Port (Impulse Response) 8–30
Listing 8.8 Final Rolled Loop Example in "cascade.asm" 8–34
Listing 8.9 FDAS File 8–35
Listing 8.10 "iircoefs.dat" File 8–36

Introduction 1

1.1 GENERAL DESCRIPTION

The ADSP-21020 and ADSP-21010 are the two members of Analog Devices' ADSP-21000 family of floating-point digital signal processors (DSPs). The ADSP-21000 family architecture further addresses the five central requirements for DSPs established in the ADSP-2100 family of 16-bit fixed-point DSPs:

- Fast, flexible arithmetic computation units
- Unconstrained data flow to and from the computation units
- Extended precision and dynamic range in the computation units
- Dual address generators
- Efficient program sequencing

Fast, Flexible Arithmetic. The ADSP-21020/21010 executes all instructions in a single cycle. It provides both one of the fastest cycle times available and the most complete set of arithmetic operations, including Seed 1/X, Seed $1/\sqrt{X}$, Min, Max, Clip, Shift and Rotate, in addition to the traditional multiplication, addition, subtraction and combined addition/subtraction. It is IEEE floating-point compatible and allows either interrupt on arithmetic exception or latched status exception handling.

Unconstrained Data Flow. The ADSP-21020/21010 has a Harvard architecture combined with a 10-port data register file. In every cycle:

- Two operands can be read or written off-chip to or from the register file,
- Two operands can be supplied to the ALU,
- Two operands can be supplied to the multiplier, and
- Two results can be received from the ALU and multiplier (three, if the ALU operation is a combined addition/subtraction).

The processors' 48-bit orthogonal instruction word supports fully parallel data transfer and arithmetic operations in the same instruction.

1 Introduction

40-Bit Extended Precision. The ADSP-21020 and ADSP-21010 handle 32-bit IEEE floating-point format as well as 32-bit integer and fractional formats (twos-complement and unsigned), while the ADSP-21020 also handles extended-precision 40-bit IEEE floating-point format. The ADSP-21020 carries extended precision throughout its computation units, limiting intermediate data truncation errors. When working with data on-chip, the extended-precision 32-bit mantissa can be transferred to and from all computation units. The 40-bit data bus may be extended off-chip as desired. The fixed-point formats have an 80-bit accumulator for true 32-bit fixed-point computations.

Dual Address Generators. The ADSP-21020/21010 has two data address generators (DAGs) that provide immediate or indirect (pre- and post-modify) addressing. Modulus and bit-reverse operations are supported with no constraints on buffer placement.

Efficient Program Sequencing. In addition to zero-overhead loops, the ADSP-21020/21010 supports single-cycle setup and exit for loops. Loops are both nestable (six levels in hardware) and interruptable. The processor supports both delayed and non-delayed branches.

1.1.1 Key Enhancements

The ADSP-21000 family enhances the core DSP architecture to enable easier system development. The enhancements occur in four key areas:

- Architectural features for high-level language and operating system support.
- Access to serial scan path (IEEE 1149.1 compatible) and on-chip emulation features.
- Support of IEEE floating-point formats.
- Open memory system.

High Level Languages. The ADSP-21000 family architecture has several features which directly support high-level language compilers and operating systems:

- General purpose data and address register files,
- 32-bit native data types,
- Large address spaces (16M words in program memory, 4G words in data memory),
- Pre- and post-modify addressing,
- Unconstrained circular buffer placement, and
- On-chip program, loop, and interrupt stacks.

Additionally, the ADSP-21000 family architecture is designed specifically to support ANSI standard Numerical C—the first compiled language to support vector data types and operators for numeric and signal processing.

Serial Scan and Emulation Features. The ADSP-21020/21010 supports the IEEE-standard P1149 Joint Test Action Group (JTAG) standard for system test. This standard defines a method for serially scanning the I/O status of each component in a system. This serial port is also used to gain access to the ADSP-21020/21010 on-chip emulation features.

IEEE Formats. The ADSP-21020/21010 supports IEEE floating-point data formats. This means that algorithms developed on IEEE-compatible processors and workstations are portable across processors without concern for possible instability introduced by biased rounding or inconsistent error handling.

Open Memory System. No on-chip memory is included on the ADSP-21020/21010 (aside from a high-performance cache) specifically to avoid artificially constraining the development and upgrade of floating-point signal processing applications. This approach also facilitates the use of high-level languages and multitasking operating systems.

1.1.2 Why Floating-Point?

A processor's data format determines its ability to handle signals of differing precision, dynamic range, and signal-to-noise ratios. However, ease-of-use and time-to-market considerations are often equally important.

Precision. The precision of converters has been increasing and will continue to increase. In the past several years, average precision requirements have increased by 3 bits. A 20-bit audio A/D converter is now available from Analog Devices, and the trend is for both precision and sampling rates to increase.

Dynamic Range. Compression and decompression algorithms have traditionally operated on signals of known bandwidth. These algorithms were developed to behave regularly, to keep costs down and implementations easy. Increasingly, however, the trend in algorithm development is not to constrain the regularity and dynamic range of intermediate results. Adaptive filtering and imaging are two applications requiring wide dynamic range.

Signal-to-Noise Ratio. Radar, sonar and even commercial applications like speech recognition require wide dynamic range in order to discern selected signals from noisy environments.

Ease-of-Use. In general, floating-point digital signal processors are easier to use and allow a quicker time-to-market than processors that do not support floating-point formats. The extent to which this is true depends on the floating-point processor's architecture. Consistency with IEEE workstation simulations and the elimination of scaling are two clear ease-of-use advantages. High-level language programmability, large address spaces, and wide dynamic range allow system development time to be spent on algorithms and signal processing problems rather than assembly coding, code paging, and error handling.

1.1.3 Future Product Migration Path

Analog Devices offers the ADSP-21000 family architecture as the highest performance for signal processing applications. Future processors based on this architecture will offer higher speed and feature integration, incorporating both internal memory and I/O peripherals on-chip.

1.2 ARCHITECTURE OVERVIEW

The following sections summarize the basic features of the ADSP-21020/21010 architecture. These features are described in more detail in succeeding chapters. Figure 1.1 shows a block diagram of the ADSP-21020 with it's 40-bit data paths.

1.2.1 Computation Units

The ADSP-21020/21010 contains three independent computation units: an ALU, a multiplier with fixed-point accumulator, and a shifter. For meeting a wide variety of processing needs, the computation units process data in three formats: 32-bit fixed-point, 32-bit floating-point and 40-bit floating-point (ADSP-21020 only). The floating-point operations are single-precision IEEE-compatible. The 32-bit floating-point format is the standard IEEE format, whereas the 40-bit IEEE extended-precision format has eight more LSBs of mantissa for additional accuracy.

The ALU performs a standard set of arithmetic and logic operations in both fixed-point and floating-point formats. The multiplier performs floating-point and fixed-point multiplication as well as fixed-point multiply/add and multiply/subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, field deposit and extraction and exponent derivation operations on 32-bit operands.

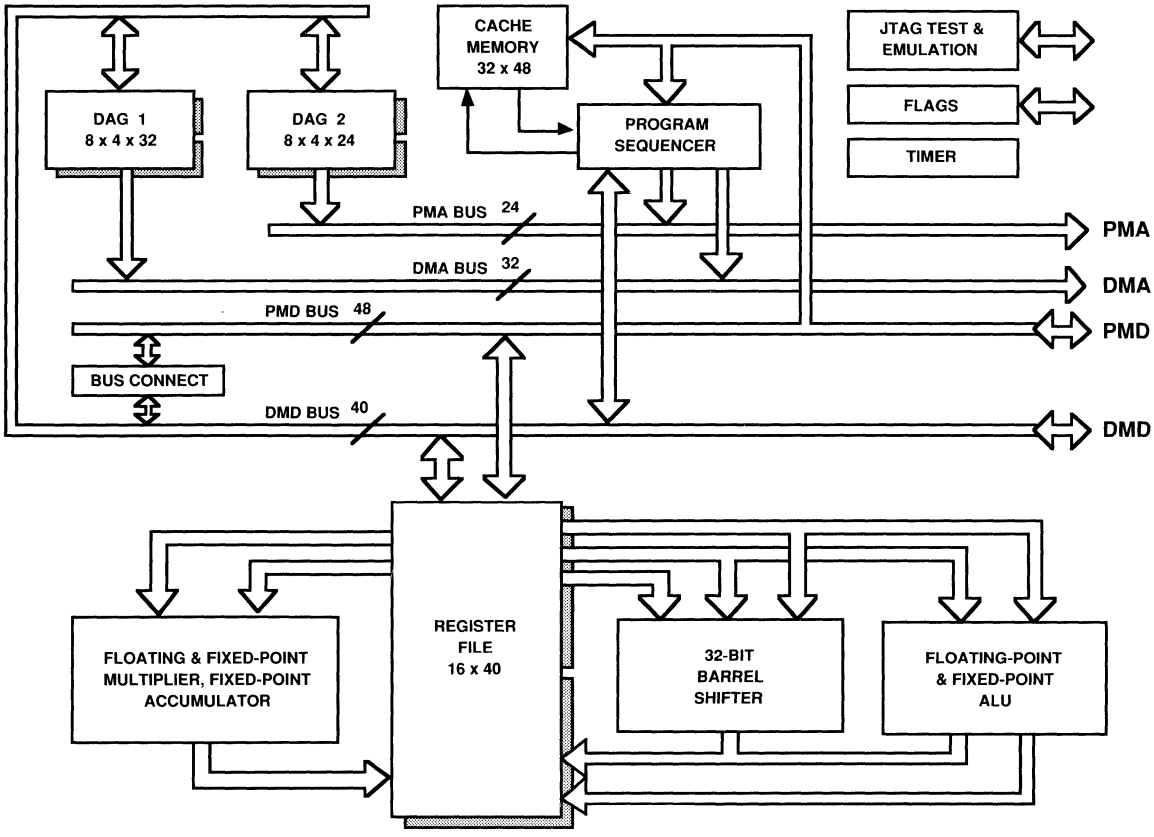


Figure 1.1 ADSP-21020 Block Diagram

The computation units perform single-cycle operations; there is no computation pipeline. The units are connected in parallel rather than serially. The output of any unit may be the input of any unit on the next cycle. In a *multifunction* computation, the ALU and multiplier perform independent simultaneous operations. A 10-port register file is used for transferring data between the computation units and the data buses, and for storing intermediate results. The register file has two sets (primary and alternate) of sixteen registers each, for fast context switching. The registers are 32 bits wide on the ADSP-21010 and 40 bits wide on the ADSP-21020.

1.2.2 Address Generators And Program Sequencer

Two dedicated address generators and a program sequencer supply addresses for memory accesses. Together the sequencer and data address generators allow computational operations to execute with maximum efficiency because the computation units can be devoted exclusively to processing data. Because of its instruction cache, the ADSP-21020/21010 can simultaneously fetch an instruction and access data in both program memory and data memory.

The data address generators (DAGs) provide memory addresses when external memory data is transferred over the parallel memory ports to or from internal registers. Dual data address generators enable the processor to output simultaneous addresses for dual operand reads and writes. DAG1 supplies 32-bit addresses to data memory. DAG2 supplies 24-bit addresses to program memory for program memory data accesses.

Each DAG keeps track of up to eight address pointers, eight modifiers and eight length values. A pointer used for indirect addressing can be modified by a value in a specified register, either before (pre-modify) or after (post-modify) the access. A length value may be associated with each pointer to implement automatic modulo addressing for circular buffers, which can be located on arbitrary boundaries. Each DAG register has an alternate register that can be activated for fast context switching.

The program sequencer supplies instruction addresses to the program memory. It controls loop iterations and evaluates conditional instructions. With an internal loop counter and loop stack, the ADSP-21020/21010 executes looped code with zero overhead. No explicit jump instructions are required to loop or to decrement and test the counter.

The ADSP-21020/21010 achieves its fast program execution rate by means of pipelined *fetch, decode* and *execute* cycles. External memories have more time to complete an access than if there were no decode cycle; consequently, ADSP-21020/21010 systems can be built using slower and therefore less expensive memories.

The program sequencer includes a 32-word instruction cache. The cache allows the ADSP-21020/21010 to perform a program memory data access and execute the corresponding instruction in the same cycle, without any delay. The program sequencer fetches the instruction from the cache instead of program memory so that the processor can simultaneously access data in program memory. Only the instructions whose fetches conflict with program memory data accesses are cached.

1.2.3 Interrupts

The ADSP-21020/21010 has five external hardware interrupts (four general-purpose interrupts and a special interrupt for reset), nine internally generated interrupts and eight software interrupts. For the general-purpose external interrupts and the internal timer interrupt, the processor automatically stacks the arithmetic status and mode (MODE1) registers in parallel with servicing the interrupt, allowing four nesting levels of very fast service for these interrupts.

1.2.4 Timer

The programmable interval timer provides periodic interrupt generation. When enabled, the timer decrements a 32-bit count register every cycle. When this count register reaches zero, the ADSP-21020/21010 generates an interrupt and asserts its TIMEXP output. The count register is automatically reloaded from a 32-bit period register and the count resumes immediately.

1.2.5 Memory Buses And Interface

The external memory interface supports memory-mapped peripherals and slower memories with a user-defined combination of programmable wait states and hardware acknowledge signals. Both program memory and data memory addressing support page mode addressing of static column DRAMs.

The processor has four internal buses: the program memory address (PMA) and data memory address (DMA) buses are used for the addresses associated with program and data memory. The program memory data (PMD) and data memory data (DMD) buses are used for the data associated with the memory spaces. These buses are extended off chip. The DMS and PMS signals select data memory and program memory, respectively.

The program memory address (PMA) bus is 24 bits wide allowing direct access of up to 16M words of mixed instruction code and data. The program memory data (PMD) bus is 48 bits wide to accommodate the 48-bit instruction width. Fixed-point and single-precision floating-point data is aligned to the upper 32 bits of the PMD bus.

The data memory address (DMA) bus is 32 bits wide allowing direct access of up to 4G words of data. The data memory data (DMD) bus is 40 bits wide on the ADSP-21020 and 32 bits wide on the ADSP-21010. On the ADSP-21020, fixed-point and single-precision floating-point data is aligned to the upper 32 bits of the DMD bus. 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 external data memory location in a single cycle. The data memory address comes from two sources: an absolute value specified in the instruction code (direct addressing) or the output of a data address generator (indirect addressing).

External devices can gain control of memory buses from the ADSP-21020/21010 with bus request/grant signals ($\overline{BR}$ and $\overline{BG}$). To grant its buses in response to a bus request, the ADSP-21020/21010 halts internal operations and places its program and data memory interfaces in a high-impedance state. In addition, three-state controls ($\overline{DMTS}$ and $\overline{PMTS}$) allow an external device to place either program or data memory interface in a high-impedance state without affecting the other interface and without halting the processor unless it requires a program memory access.

1.2.6 Internal Data Transfers

Nearly every internal register of the ADSP-21020/21010 is classified as a *universal register*. ADSP-21020/21010 instructions provide for transferring data between any two universal registers or between a universal register and external memory. This includes control registers and status registers, as well as the data registers in the register file.

The PX registers permit data to be passed between the 48-bit PMD bus and the 40-bit DMD bus or between the 40-bit register file and the PMD bus. These registers contain hardware to handle the 8-bit width difference.

1.2.7 Context Switching

Many of the processor's registers have alternate registers that can be activated during interrupt servicing to facilitate a fast context switch. The data registers in the register file, DAG registers and the multiplier result register all have alternates. Registers active at reset are called *primary* registers, and the others are *alternate* registers. Bits in a mode control register determine the registers that are active at any particular time.

1.2.8 Instruction Set

The ADSP-21000 family instruction set provides a wide variety programming capabilities. *Multifunction* instructions enable computations in parallel with data transfers, as well as simultaneous multiplier and ALU operations. The addressing power of the ADSP-21020/21010 gives you flexibility in moving data both internally and externally. Every instruction can be executed in a single processor cycle. The ADSP-21000 family assembly language uses an algebraic syntax for ease of coding and readability. A comprehensive set of development tools supports program development.

1.3 DEVELOPMENT SYSTEM

The ADSP-21020/21010 is supported with a complete set of software and hardware development tools. The ADSP-21000 Family Development System includes software tools for programming and debugging as well as in-circuit emulators for system integration and debugging.

Figure 1.2 shows the process of developing an application using the development tools. File name extensions (.ASM, .OBJ, etc.) at the input and output of each step signify different types of files.

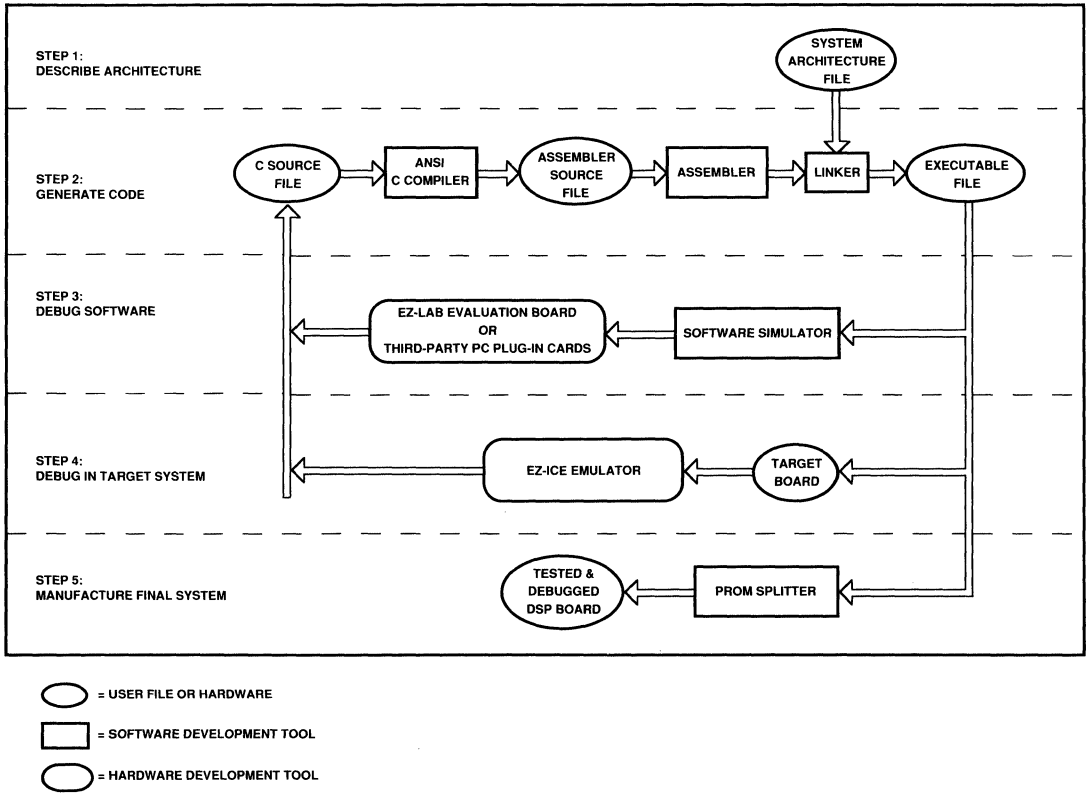


Figure 1.2 DSP System Development

1 Introduction

The development system includes the following:

C Compiler & Runtime C Library. The C Compiler reads source files written in ANSI-standard C language. The compiler outputs ADSP-21xxx assembly language files. It comes with a standard library of C-callable routines.

Numerical C Compiler. DSP/C™ is Analog Devices' implementation of ANSI-standard Numerical C—a set of extensions to C that allow matrix data types and operators. The compiler outputs ADSP-21xxx assembly language files. With DSP/C, signal processing algorithms are easier to program and the compiled code is more efficient because the compiler directly translates matrix operations in Numerical C to the matrix capabilities of the ADSP-21020/21010.

Assembler. The assembler inputs a file of ADSP-21xxx source code and assembler directives and outputs a relocatable object file. The assembler supports standard C preprocessor directives as well as its own directives.

Linker. The linker processes separately assembled object and library files to create a single executable program. It assigns memory locations to code and data in accordance with a user-defined architecture file, a text file that describes the memory configuration of the target system.

Assembly Library/Librarian. The assembly library contains standard arithmetic and DSP routines that can be called from your program, saving development time. You can add your own routines to this library using the librarian function.

Simulator. The simulator executes an ADSP-21020/21010 program in software in the same way that the processor would in hardware. The simulator also simulates the memory and I/O devices specified in the architecture file. The simulator's window-based user interface lets you interactively observe and alter data contained in the processor's registers and in memory.

PROM Splitter. The PROM splitter translates an ADSP-21xxx executable program into one of several formats (Motorola S2 and S3, Intel Hex Record, etc.) that can be used to configure a PROM or be downloaded to a target from a microcontroller.

In-Circuit Emulator. The EZ-ICE® emulator provides hardware debugging capabilities for ADSP-21020/21010 systems with stand-alone in-circuit emulation, running the target board processor in self-emulation mode. The emulator design allows program execution with little or no degradation in processor performance.

The emulator features the same window-based user interface as the simulator, for ease-of-use and faster development cycles. The emulator communicates with the target processor through the processor's JTAG test access port. This 7-wire interface allows for a probe that is smaller and less intrusive than a traditional full-pinout emulator-to-target connector.

1.4 MANUAL ORGANIZATION

The chapters of this manual are organized as follows:

Chapter 2, Computation Units. Describes the capabilities and operation of the ALU, multiplier and shifter.

Chapter 3, Program Sequencing. Describes the processor's features for executing various types of program structures: subroutines, loops and interrupt service routines. Also describes the operation of the instruction cache and the handling of interrupts.

Chapter 4, Data Addressing. Describes how to use the data address generators to address data in data memory and program memory.

Chapter 5, Timer. Describes the operation of the programmable interval timer.

Chapter 6, Memory Interface. Describes how the processor accesses external data and program memory. Also describes the memory management features of the processor.

Chapter 7, Instruction Summary. An overview of the ADSP-21000 family instruction set. Use this chapter as a reference when writing programs in assembly language. The chapter also contains a summary of programming reminders and restrictions.

Chapter 8, Assembly Programming Tutorial. Presents two examples of ADSP-21020/21010 programs and describes in detail how they were written. Describes many techniques to take advantage of the processors' architecture and instruction set.

Chapter 9, Hardware System Design. Presents numerous system diagrams based on the ADSP-21020. Hardware considerations such as clocking, reset, flags, capacitive loading and emulator access are also addressed. Examples of a program memory boot at reset and a host interface are shown.

Appendix A, Instruction Set Reference. Describes each instruction in detail. Also details the instruction opcodes. The compute portion of instructions are described in Appendix B.

Appendix B, Compute Operation Reference. Describes each compute operation and its opcode field in detail.

Appendix C, IEEE 1149.1 JTAG Test Access Port. Describes the features and operation of the IEEE 1149.1 (JTAG) test access port.

Appendix D, Numeric Formats. Shows all the floating-point and fixed-point data formats supported by the ADSP-21020/21010.

Appendix E, Control/Status Registers. Summarizes the contents of all ADSP-21020/21010 registers that contain control and/or status bits. Also describes bit manipulation operations available.

Computation Units 2

2.1 OVERVIEW

The computation units of the ADSP-21020 and ADSP-21010 provide the numeric processing power for performing DSP algorithms. The ADSP-21020/21010 contains three computation units: an arithmetic/logic unit (ALU), a multiplier and a shifter. Both fixed-point and floating-point operations are supported by the processor. Each computation unit executes instructions in a single cycle.

The ALU performs a standard set of arithmetic and logic operations in both fixed-point and floating-point formats. The multiplier performs floating-point and fixed-point multiplication as well as fixed-point multiply/add and multiply/subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, field deposit and extraction operations on 32-bit operands and can derive exponents as well.

The computation units are architecturally arranged in parallel, as shown in Figure 2.1 on the next page. The output of any computation unit may be the input of any computation unit on the next cycle. The computation units input data from and output data to a 10-port register file that consists of sixteen primary registers and sixteen alternate registers. The register file is accessible to the ADSP-21020/21010 program and data memory data buses for transferring data between the computation units and external memory or other parts of the processor.

The individual registers of the register file are prefixed with an "f" when used in floating-point computations (in assembly language source code). The registers are prefixed with an "r" when used in fixed-point computations. The following instructions, for example, use the same registers:

```
F0=F1 * F2;        floating-point multiply
R0=R1 * R2;        fixed-point multiply
```

The "f" and "r" prefixes do not affect the 40-bit (or 32-bit) data transfer; they only determine how the ALU, multiplier, or shifter treat the data.

2 Computation Units

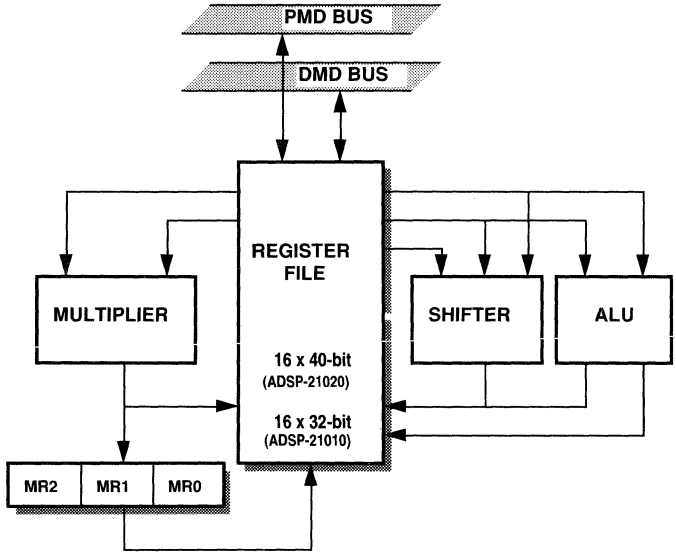


Figure 2.1 Computation Units

This chapter covers the following topics:

- Data Formats and Rounding
- ALU Architecture and Functions
- Multiplier Architecture and Functions
- Shifter Architecture and Functions
- Multifunction Computations
- Register File and Data Transfers

2.2 IEEE FLOATING-POINT OPERATIONS

The ADSP-21020/21010 multiplier and ALU support the single-precision floating-point format specified in the IEEE 754/854 standard. This standard is described in Appendix D. The ADSP-21020/21010 is IEEE 754/854 compatible for single-precision floating-point operations in all respects except that:

- The ADSP-21020/21010 does not provide inexact flags.
- NAN ("Not-A-Number") inputs generate an invalid exception and return a quiet NAN (all 1s).

- Denormal operands are flushed to zero when input to a computation unit and do not generate an underflow exception. Any denormal or underflow result from an arithmetic operation is flushed to zero and an underflow exception is generated.

- Round-to-nearest and round-toward-zero modes are supported. Rounding to +Infinity and rounding to -Infinity are not supported.

In addition, the ADSP-21020 supports a 40-bit extended precision floating-point mode, which has eight additional LSBs of the mantissa and is compliant with the 754/854 standards; however, results in this format are more precise than the IEEE single-precision standard specifies. The ADSP-21010 does not offer this 40-bit format.

2.2.1 Extended Floating-Point Precision (ADSP-21020 Only)

Floating-point data can be either 32 or 40 bits wide on the ADSP-21020. Extended precision floating-point format (8 bits of exponent and 32 bits of mantissa) is selected if the RND32 bit in the MODE1 register is cleared (0). If this bit is set (1), then normal IEEE precision is used (8 bits exponent and 24 bits of mantissa). In this case, the computation unit sets the eight LSBs of floating-point inputs to zeros before performing the operation. The mantissa of a result is rounded to 23 bits (not including the hidden bit) and the 8 LSBs of the 40-bit result are set to zeros to form a 32-bit number that is equivalent to the IEEE standard result.

On the ADSP-21010, the RND32 bit must be set to 1 at system powerup (at the beginning of your program).

2.2.2 Floating-Point Exceptions

The multiplier and ALU each provide exception information when executing floating-point operations. Each unit updates overflow, underflow and invalid operation flags in the arithmetic status (ASTAT) register and in the sticky status (STKY) register. An underflow, overflow or invalid operation from any unit also generates a maskable interrupt. Thus, there are three ways to handle floating-point exceptions:

- Interrupts. The exception condition is handled immediately in an interrupt service routine. You would use this method if it was important to correct all exceptions as they happen.

- ASTAT register. The exception flags in the ASTAT register pertaining to a particular arithmetic operation are tested after the operation is performed. You would use this method to monitor a particular floating-point operation.

- STKY register. Exception flags in the STKY register are examined at the end of a series of operations. If any flags are set, some of the results are incorrect. You would use this method if exception handling was not critical.

2.3 FIXED-POINT OPERATIONS

Fixed-point numbers are always represented in 32 bits and are left-justified (occupy the 32 MSBs) in the 40-bit data fields of the ADSP-21020. They may be treated as fractional or integer numbers and as unsigned or twos-complement. Each computation unit has its own limitations on how these formats may be mixed for a given operation. The computation units read 32-bit operands from 40-bit registers, ignoring the 8 LSBs, and write 32-bit results, zeroing the 8 LSBs.

2.4 ROUNDING

Two modes of rounding are supported in the ADSP-21020 and ADSP-21010: round-toward-zero and round-toward-nearest. The rounding modes follow the IEEE 754 standard definitions, which are briefly stated as follows:

Round-toward-Zero. If the result before rounding is not exactly representable in the destination format, the rounded result is that number which is nearer to zero. This is equivalent to truncation.

Round-toward-Nearest. If the result before rounding is not exactly representable in the destination format, the rounded result is that number which is nearer to the result before rounding. If the result before rounding is exactly halfway between two numbers in the destination format (differing by an LSB), the rounded result is that number which has an LSB equal to zero. Statistically, rounding up occurs as often as rounding down, so there is no large sample bias. Because the maximum floating-point value is one LSB less than the value that represents Infinity, a result that is halfway between the maximum floating-point value and Infinity rounds to Infinity in this mode.

The rounding mode for all ALU operations and for floating-point multiplier operations is determined by the TRUNC bit in the MODE1 register. If the TRUNC bit is set, the round-to-zero mode is selected; otherwise, the round-to-nearest mode is used.

For fixed-point multiplier operations on fractional data, the same two rounding modes are supported, but only the round-to-nearest operation is actually performed by the multiplier. Because the multiplier has a local result register for fixed-point operations, rounding-to-zero is accomplished implicitly by reading only the upper bits of the result and discarding the lower bits.

2.5 ALU

The ALU performs arithmetic operations on fixed-point or floating-point data and logical operations on fixed-point data. ALU fixed-point instructions operate on 32-bit fixed-point operands and output 32-bit fixed-point results. ALU floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results.

ALU instructions include:

- Floating-point addition, subtraction, add/subtract, average
- Fixed-point addition, subtraction, add/subtract, average
- Floating-point manipulation: binary log, scale, mantissa
- Fixed-point add with carry, subtract with borrow, increment, decrement
- Logical AND, OR, XOR, NOT
- Functions: Absolute value, pass, min, max, clip, compare
- Format conversion
- Reciprocal and reciprocal square root primitives

Dual add/subtract and parallel ALU and multiplier operations are described under "Multifunction Computations," later in this chapter.

2.5.1 ALU Operation

The ALU takes one or two input operands, called the X input and the Y input, which can be any data registers in the register file. It usually returns one result; in add/subtract operations it returns two results, and in compare operations it returns no result (only flags are updated). ALU results can be returned to any location in the register file.

Input operands are transferred from the register file during the first half of the cycle. Results are transferred to the register file during the second half of the cycle. Thus the ALU can read and write the same register file location in a single cycle.

If the ALU operation is fixed-point, the X input and Y input are each treated as a 32-bit fixed-point operand. The upper 32 bits from the source location in the register file are transferred. For fixed-point operations, the result(s) are always 32-bit fixed-point values. Some floating-point operations (LOGB, MANT and FIX) can also yield fixed-point results. Fixed-point results are transferred to the upper 32 bits of register file. The lower eight bits of the register file destination are cleared.

The format of fixed-point operands and results depends on the operation. In most arithmetic operations, there is no need to distinguish between integer and fractional formats. Fixed-point inputs to operations such as scaling a floating-point value are treated as integers. For purposes of determining status such as overflow, fixed-point arithmetic operands and results are treated as twos-complement numbers.

2.5.2 ALU Operating Modes

The ALU is affected by three mode status bits in the MODE1 register; the ALU saturation bit affects ALU operations that yield fixed-point results, and the rounding mode and rounding boundary bits affect floating-point operations in both the ALU and multiplier.

MODE1

Bit	*Name*	*Function*
13	ALUSAT	1=Enable ALU saturation (full scale in fixed-point); 0=No ALU saturation
15	TRUNC	1=Truncation; 0=Round to nearest
16	RND32	1=Round to 32 bits; 0=Round to 40 bits (RND32 must be set to 1 on ADSP-21010)

2.5.2.1 Saturation Mode

In saturation mode, all positive fixed-point overflows cause the maximum positive fixed-point number (0x7FFF FFFF) to be returned, and all negative overflows cause the maximum negative number (0x8000 0000) to be returned. If the ALUSAT bit is set, fixed-point results that overflow are saturated. If the ALUSAT bit is cleared, fixed-point results that overflow are not saturated; the upper 32 bits of the result are returned unaltered. The ALU overflow flag reflects the ALU result before saturation.

2.5.2.2 Floating-Point Rounding Modes

The ALU supports two IEEE rounding modes. If the TRUNC bit is set, the ALU rounds a result to zero (truncation). If the TRUNC bit is cleared, the ALU rounds to nearest.

2.5.2.3 Floating-Point Rounding Boundary

The results of floating-point ALU operations can be either 32-bit or 40-bit floating-point data on the ADSP-21020. If the RND32 bit is set, the eight LSBs of each input operand are flushed to zeros before the ALU operation is performed (except for the RND operation), and ALU floating-point results are output in the 32-bit IEEE format. The lower eight bits of the result are cleared. If the RND32 bit is cleared, the ALU inputs 40-bit operands unchanged and outputs 40-bit results from floating-point operations, and all 40 bits are written to the specified register file location.

In fixed-point to floating-point conversion, the rounding boundary is always 40 bits even if the RND32 bit is set.

2.5.3 ALU Status Flags

The ALU updates seven status flags in the ASTAT register, shown below, at the end of each operation. The states of these flags reflect the result of the most recent ALU operation. The ALU updates the Compare Accumulation bits in ASTAT at the end of every Compare operation. The ALU also updates four "sticky" status flags in the STKY register. Once set, a sticky flag remains high until explicitly cleared.

ASTAT

Bit	*Name*	*Definition*
0	AZ	ALU result zero or floating-point underflow
1	AV	ALU overflow
2	AN	ALU result negative
3	AC	ALU fixed-point carry
4	AS	ALU X input sign (ABS and MANT operations)
5	AI	ALU floating-point invalid operation
10	AF	last ALU operation was a floating-point operation
31-24	CACC	Compare Accumulation register (results of last 8 Compare operations)

STKY

Bit	*Name*	*Definition*
0	AUS	ALU floating-point underflow
1	AVS	ALU floating-point overflow
2	AOS	ALU fixed-point overflow
5	AIS	ALU floating-point invalid operation

Flag update occurs at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the ASTAT register or STKY register explicitly in the same cycle that the ALU is performing an operation, the explicit write to ASTAT or STKY supersedes any flag update from the ALU operation.

2.5.3.1 ALU Zero Flag (AZ)

The zero flag is determined for all fixed-point and floating-point ALU operations. AZ is set whenever the result of an ALU operation is zero. AZ also signifies floating-point underflow; see the next section. It is otherwise cleared.

2.5.3.2 ALU Underflow Flag (AZ, AUS)

Underflow is determined for all ALU operations that return a floating-point result and for floating-point to fixed-point conversion. AUS is set whenever the result of an ALU operation is smaller than the smallest number representable in the output format. AZ is set whenever a floating-point result is smaller than the smallest number representable in the output format.

2.5.3.3 ALU Negative Flag (AN)

The negative flag is determined for all ALU operations. It is set whenever the result of an ALU operation is negative. It is otherwise cleared.

2.5.3.4 ALU Overflow Flag (AV, AOS, AVS)

Overflow is determined for all fixed-point and floating-point ALU operations. For fixed-point results, AV and AOS are set whenever the XOR of the two most significant bits is a 1; otherwise AV is cleared. For floating-point results AV and AVS are set whenever the post-rounded result overflows (unbiased exponent > 127); otherwise AV is cleared.

2.5.3.5 ALU Fixed-Point Carry Flag (AC)

The carry flag is determined for all fixed-point ALU operations. For fixed-point arithmetic operations, AC is set if there is a carry out of most significant bit of the result, and is otherwise cleared. AC is cleared for fixed-point logic, PASS, MIN, MAX, COMP, ABS, and CLIP operations. The ALU reads the AC flag in fixed-point addition with carry and fixed-point subtraction with carry operations.

2.5.3.6 ALU Sign Flag (AS)

The sign flag is determined for only the fixed-point and floating-point ABS operations and the MANT operation. AS is set if the input operand is negative. It is otherwise cleared. The ALU clears AS for all operations other than ABS and MANT operations; this is different from the operation of ADSP-2100 family processors, which do not update the AS flag on operations other than ABS.

2.5.3.7 ALU Invalid Flag (AI)

The invalid flag is determined for all floating-point ALU operations. AI and AIS are set whenever

- an input operand is a NAN
- an addition of opposite-signed Infinities is attempted
- a subtraction of like-signed Infinities is attempted
- when saturation mode is not set, a floating-point to fixed-point conversion results in an overflow or operates on an Infinity.

AI is otherwise cleared.

2.5.3.8 ALU Floating-Point Flag (AF)

AF is determined for all fixed-point and floating-point ALU operations. It is set if the last operation was a floating-point operation; it is otherwise cleared.

2.5.3.9 Compare Accumulation

Bits 31-24 in the ASTAT register store the flag results of up to eight ALU compare operations. These bits form a right-shift register. When an ALU compare operation is executed, the eight bits are shifted toward the LSB (bit 24 is lost). The MSB, bit 31, is then written with the result of the compare operation. If the X operand is greater than the Y operand in the compare instruction, bit 31 is set; it is cleared otherwise. The accumulated compare flags can be used to implement 2- and 3-dimensional clipping operations for graphics applications.

2.5.4 ALU Instruction Summary

Instruction	*ASTAT Status Flags*								*STKY Status Flags*			
Fixed-point:	AZ	AV	AN	AC	AS	AI	AF	CACC	AUS	AVS	AOS	AIS
c Rn = Rx + Ry	*	*	*	*	0	0	0	–	–	–	**	–
c Rn = Rx – Ry	*	*	*	*	0	0	0	–	–	–	**	–
c Rn = Rx + Ry + CI	*	*	*	*	0	0	0	–	–	–	**	–
c Rn = Rx – Ry + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = (Rx + Ry)/2	*	0	*	*	0	0	0	–	–	–	–	–
COMP(Rx, Ry)	*	0	*	0	0	0	0	*	–	–	–	–
Rn = Rx + CI	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + CI – 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx + 1	*	*	*	*	0	0	0	–	–	–	**	–
Rn = Rx – 1	*	*	*	*	0	0	0	–	–	–	**	–
c Rn = –Rx	*	*	*	*	0	0	0	–	–	–	**	–
c Rn = ABS Rx	*	*	0	0	*	0	0	–	–	–	**	–
Rn = PASS Rx	*	0	*	0	0	0	0	–	–	–	–	–
c Rn = Rx AND Ry	*	0	*	0	0	0	0	–	–	–	–	–
c Rn = Rx OR Ry	*	0	*	0	0	0	0	–	–	–	–	–
c Rn = Rx XOR Ry	*	0	*	0	0	0	0	–	–	–	–	–
c Rn = NOT Rx	*	0	*	0	0	0	0	–	–	–	–	–
Rn = MIN(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn = MAX(Rx, Ry)	*	0	*	0	0	0	0	–	–	–	–	–
Rn = CLIP Rx BY Ry	*	0	*	0	0	0	0	–	–	–	–	–
Floating-point:												
Fn = Fx + Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn = Fx – Fy	*	*	*	0	0	*	1	–	**	**	–	**
Fn = ABS (Fx + Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn = ABS (Fx – Fy)	*	*	0	0	0	*	1	–	**	**	–	**
Fn = (Fx + Fy)/2	*	0	*	0	0	*	1	–	**	–	–	**
COMP(Fx, Fy)	*	0	*	0	0	*	1	*	–	–	–	**
Fn = –Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = ABS Fx	*	*	0	0	*	*	1	–	–	**	–	**
Fn = PASS Fx	*	0	*	0	0	*	1	–	–	–	–	**
Fn = RND Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = SCALB Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
Rn = MANT Fx	*	*	0	0	*	*	1	–	–	**	–	**
Rn = LOGB Fx	*	*	*	0	0	*	1	–	–	**	–	**
Rn = FIX Fx BY Ry	*	*	*	0	0	*	1	–	**	**	–	**
Rn = FIX Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn = FLOAT Rx BY Ry	*	*	*	0	0	0	1	–	**	**	–	–
Fn = FLOAT Rx	*	0	*	0	0	0	1	–	–	–	–	–
Fn = RECIPS Fx	*	*	*	0	0	*	1	–	**	**	–	**
Fn = RSQRTS Fx	*	*	*	0	0	*	1	–	–	**	–	**
Fn = Fx COPYSIGN Fy	*	0	*	0	0	*	1	–	–	–	–	**
Fn = MIN(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
Fn = MAX(Fx, Fy)	*	0	*	0	0	*	1	–	–	–	–	**
Fn = CLIP Fx BY Fy	*	0	*	0	0	*	1	–	–	–	–	**

Rn, Rx, Ry = Any register file location; treated as fixed-point
Fn, Fx, Fy = Any register file location; treated as floating-point
c = ADSP-21xx-compatible instruction

* set or cleared, depending on results of instruction
** may be set (but not cleared), depending on results of instruction
– no effect

2.6 MULTIPLIER

The multiplier performs fixed-point or floating-point multiplication and fixed-point multiply/accumulate operations. Fixed-point multiply/accumulates may be performed with either cumulative addition or cumulative subtraction. Floating-point multiply accumulates can be accomplished through parallel operation of the ALU and multiplier, using multifunction instructions. See "Multifunction Operations," later in this chapter.

Multiplier floating-point instructions operate on 32-bit or 40-bit floating-point operands and output 32-bit or 40-bit floating-point results. Multiplier fixed-point instructions operate on 32-bit fixed-point data and produce 80-bit results. Inputs are treated as fractional or integer, unsigned or twos-complement.

Multiplier instructions include:

- Floating-point multiplication
- Fixed-point multiplication
- Fixed-point multiply/accumulate with addition, rounding optional
- Fixed-point multiply/accumulate with subtraction, rounding optional
- Rounding result register
- Saturating result register
- Clearing result register

2.6.1 Multiplier Operation

The multiplier takes two input operands, called the X input and the Y input, which can be any data registers in the register file. Fixed-point operations can accumulate fixed-point results in either of two local multiplier result (MR) registers or write results back to the register file. Results stored in the MR registers can also be rounded or saturated in separate operations. Floating-point operations yield floating-point results, which are always written directly back to the register file.

Input operands are transferred during the first half of the cycle. Results are transferred during the second half of the cycle. Thus the multiplier can read and write the same register file location in a single cycle.

If the multiplier operation is fixed-point, inputs taken from the register file are read from the upper 32 bits of the source location. Fixed-point operands may be treated as both in integer format or both in fractional format. The format of the result is the same as the format of the inputs. Each fixed-point operand may be treated as either an unsigned or a twos-

complement number. If both inputs are fractional and signed, the multiplier automatically shifts the result left one bit to remove the redundant sign bit. The input data type is specified within the multiplier instruction.

2.6.2 Fixed-Point Results

Fixed-point operations yield 80-bit results. The location of a result in the 80-bit field depends on whether the result is in fractional or integer format, as shown in Figure 2.2. If the result is sent directly to the register file, the 32 bits that have the same format as the input data are transferred, i.e. bits 63-32 for a fractional result or bits 31-0 for an integer result. The eight LSBs of the 40-bit register file location are zero-filled. Fractional results can be rounded-to-nearest before being sent to the register file, as explained later in this chapter. If rounding is not specified, discarding bits 31-0 effectively truncates a fractional result (rounds to zero).

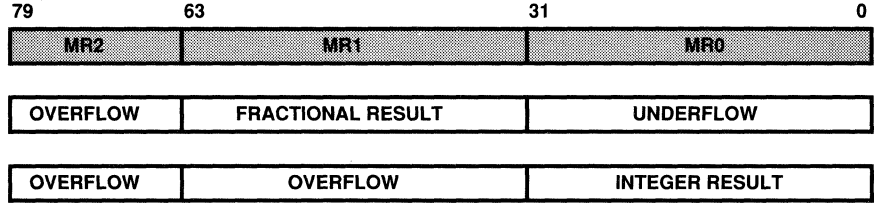


Figure 2.2 Multiplier Fixed-Point Result Placement

2.6.2.1 MR Registers

The entire result can be sent to one of two dedicated 80-bit result (MR) registers. The MR registers have identical format; each is divided into MR2, MR1 and MR0 registers that can be individually read from or written to the register file. When data is read from MR2, it is sign-extended to 32 bits (see Figure 2.3). The eight LSBs of the 40-bit register file location are zero-filled when data is read from MR2, MR1 or MR0 to the register file. Data is written into MR2, MR1 or MR0 from the 32 MSBs of a register file location; the eight LSBs are ignored. Data written to MR1 is sign-extended to MR2, i.e. the MSB of MR1 is repeated in the 16 bits of MR2. Data written to MR0, however, is not sign-extended.

The two MR registers are designated MRF (foreground) and MRB (background); *foreground* refers to those registers currently activated by the SRCU bit in the MODE1 register, and *background* refers to those that

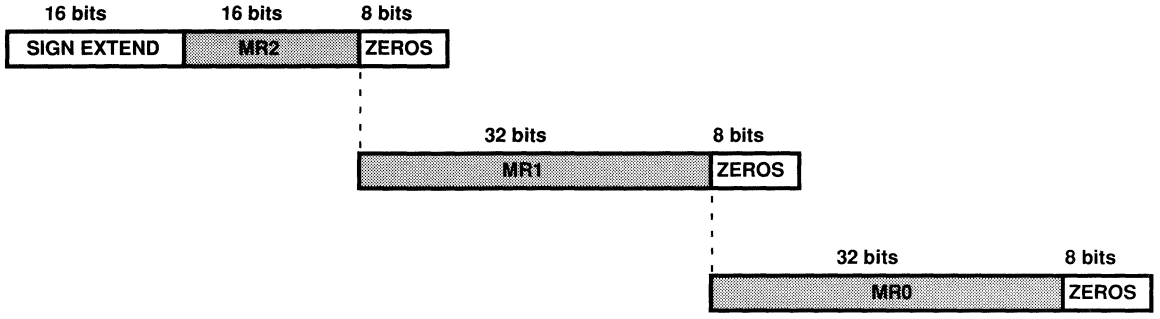


Figure 2.3 MR Transfer Formats

are not. In the case that only one MR register is used at a time, the SRCU bit activates one or the other to facilitate context switching. However, unlike other registers for which alternate sets exist, both MR register sets are accessible at the same time. All (fixed-point) accumulation instructions may specify either result register for accumulation, regardless of the state of the SRCU bit. Thus, instead of using the MR registers as a primary and an alternate, you can use them as two parallel accumulators. This feature facilitates complex math.

Transfers between MR registers and the register file are considered computation unit operations, since they involve the multiplier. Thus, although the syntax for the transfer is the same as for any other transfer to or from the register file, an MR transfer is placed in an instruction where a computation is normally specified. For example, the ADSP-21020 can perform a multiply/accumulate in parallel with a read of data memory, as in:

```
MRF=MRF-R5*R0, R6=DM(I1,M2);
```

or it can perform an MR transfer instead of the computation, as in:

```
R5=MR1F, R6=DM(I1,M2);
```

2.6.3 Fixed-Point Operations

In addition to multiplication, fixed-point operations include accumulation, rounding and saturation of fixed-point data. There are three MR register operations: Clear, Round and Saturate.

2.6.3.1 Clear MR Register

The clear operation resets the specified MR register to zero. This operation is performed at the start of a multiply/accumulate operation to remove results left over from the previous operation.

2.6.3.2 Round MR Register

Rounding of a fixed-point result occurs either as part of a multiply or multiply/accumulate operation or as an explicit operation on the MR register. The rounding operation applies only to fractional results (integer results are not affected) and rounds the 80-bit MR value to nearest at bit 32, i.e. at the MR1-MR0 boundary. The rounded result in MR1 can be sent either to the register file or back to the same MR register. To round a fractional result to zero (truncation) instead of to nearest, you would simply transfer the unrounded result from MR1, discarding the lower 32 bits in MR0.

2.6.3.3 Saturate MR Register On Overflow

The saturate operation sets MR to a maximum value if the MR value has overflowed. Overflow occurs when the MR value is greater than the maximum value for the data format (unsigned or twos-complement and integer or fractional) that is specified in the saturate instruction. There are six possible maximum values (shown in hexadecimal):

MR2	**MR1**	**MR0**	
Maximum twos-complement fractional number			
0000	7FFF FFFF	FFFF FFFF	positive
FFFF	8000 0000	0000 0000	negative
Maximum twos-complement integer number			
0000	0000 0000	7FFF FFFF	positive
FFFF	FFFF FFFF	8000 0000	negative
Maximum unsigned fractional number			
0000	FFFF FFFF	FFFF FFFF	
Maximum unsigned integer number			
0000	0000 0000	FFFF FFFF	

The result from MR saturation can be sent either to the register file or back to the same MR register.

2.6.4 Floating-Point Operating Modes

The multiplier is affected by two mode status bits in the MODE1 register: the rounding mode and rounding boundary bits, which affect operations in both the multiplier and the ALU.

MODE1

Bit	*Name*	*Function*
15	TRUNC	1=Truncation; 0=Round to nearest
16	RND32	1=Round to 32 bits; 0=Round to 40 bits (RND32 must be set to 1 on ADSP-21010)

2.6.4.1 Floating-Point Rounding Modes

The multiplier supports two IEEE rounding modes for floating-point operations. If the TRUNC bit is set, the multiplier rounds a floating-point result to zero (truncation). If the TRUNC bit is cleared, the multiplier rounds to nearest.

2.6.4.2 Floating-Point Rounding Boundary

Floating-point multiplier inputs and results can be either 32-bit or 40-bit floating-point data on the ADSP-21020. If the RND32 bit is set, the eight LSBs of each input operand are flushed to zeros before multiplication, and floating-point results are output in the 32-bit IEEE format, with the lower eight bits of the 40-bit register file location cleared. The mantissa of the result is rounded to 23 bits (not including the hidden bit). If the RND32 bit is cleared, the multiplier inputs full 40-bit values from the register file and outputs results in the 40-bit extended IEEE format, with the mantissa rounded to 31 bits not including the hidden bit.

2.6.5 Multiplier Status Flags

The multiplier updates four status flags at the end of each operation. All of these flags appear in the ASTAT register. The states of these flags reflect the result of the most recent multiplier operation. The multiplier also updates four "sticky" status flags in the STKY register. Once set, a sticky flag remains high until explicitly cleared.

ASTAT

Bit	*Name*	*Definition*
6	MN	Multiplier result negative
7	MV	Multiplier overflow
8	MU	Multiplier underflow
9	MI	Multiplier floating-point invalid operation

STKY

Bit	*Name*	*Definition*
6	MOS	Multiplier fixed-point overflow
7	MVS	Multiplier floating-point overflow
8	MUS	Multiplier underflow
9	MIS	Multiplier floating-point invalid operation

Flag update occurs at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the ASTAT register or STKY register explicitly in the same cycle that the multiplier is performing an operation, the explicit write to ASTAT or STKY supersedes any flag update from the multiplier operation.

2.6.5.1 Multiplier Underflow Flag (MU)

Underflow is determined for all fixed-point and floating-point multiplier operations. It is set whenever the result of a multiplier operation is smaller than the smallest number representable in the output format. It is otherwise cleared.

For floating-point results, MU and MUS are set whenever the post-rounded result underflows (unbiased exponent < –126). Denormal operands are treated as Zeros, therefore they never cause underflows.

For fixed-point results, MU and MUS depend on the data format and are set under the following conditions:

Twos-complement:
Fractional: upper 48 bits all zeros or all ones, lower 32 bits not all zeros
Integer: not possible

Unsigned:
Fractional: upper 48 bits all zeros, lower 32 bits not all zeros
Integer: not possible

If the fixed-point result is sent to an MR register, the underflowed portion of the result is available in MR0 (fractional result only).

2.6.5.2 Multiplier Negative Flag (MN)

The negative flag is determined for all multiplier operations. MN is set whenever the result of a multiplier operation is negative. It is otherwise cleared.

2.6.5.3 Multiplier Overflow Flag (MV)

Overflow is determined for all fixed-point and floating-point multiplier operations.

For floating-point results, MV and MVS are set whenever the post-rounded result overflows (unbiased exponent > 127).

For fixed-point results, MV and MOS depend on the data format and are set under the following conditions:

Twos-complement:

Fractional:	upper 17 bits of MR not all zeros or all ones
Integer:	upper 49 bits of MR not all zeros or all ones

Unsigned:

Fractional:	upper 16 bits of MR not all zeros
Integer:	upper 48 bits of MR not all zeros

If the fixed-point result is sent to an MR register, the overflowed portion of the result is available in MR1 and MR2 (integer result) or MR2 only (fractional result).

2.6.5.4 Multiplier Invalid Flag (MI)

The invalid flag is determined for floating-point multiplication. MI is set whenever:

- an input operand is a NAN.
- the inputs are Infinity and Zero. (Note: Denormal inputs are treated as Zeros.)

MI is otherwise cleared.

2 Computation Units

2.6.6 Multiplier Instruction Summary

Instruction	*ASTAT Flags* MU	MN	MV	MI	*STKY Flags* MUS	MOS	MVS	MIS
Fixed-point:								
Rn / MRF / MRB = Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = MRF / Rn = MRB / MRF = MRF / MRB = MRB + Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = MRF / Rn = MRB / MRF = MRF / MRB = MRB – Rx * Ry (S / U , S / U , F / I / FR)	*	*	*	0	–	**	–	–
Rn = SAT MRF / Rn = SAT MRB / MRF = SAT MRF / MRB = SAT MRB (SI) / (UI) / (SF) / (UF)	*	*	*	0	–	**	–	–
Rn = RND MRF / Rn = RND MRB / MRF = RND MRF / MRB = RND MRB (SF) / (UF)	*	*	*	0	–	**	–	–
MRF / MRB = 0	0	0	0	0	–	–	–	–
MRxF / MRxB = Rn	0	0	0	0	–	–	–	–
Rn = MRxF / MRxB	0	0	0	0	–	–	–	–
Floating-point:								
Fn = Fx * Fy	*	*	*	*	**	–	**	**

Note: For floating-point multiply/accumulates, see "Multifunction Instructions" on page 23.

* set or cleared, depending on results of instruction
** may be set (but not cleared), depending on results of instruction
– no effect

Rn, Rx, Ry –R15-R0; register file location, treated as fixed-point
Fn, Fx, Fy –F15-F0; register file location, treated as floating-point
MRxF –MR2F, MR1F, MR0F; multiplier result accumulators, foreground
MRxB –MR2B, MR1B, MR0B; multiplier result accumulators, background

Multiplier Instruction Summary, cont.

Optional Modifiers for Fixed-Point:

(❑ ❑ ❑)
X-input, Y-input, Data format, rounding

S	Signed input
U	Unsigned input
I	Integer input(s)
F	Fractional input(s)
FR	Fractional inputs, Rounded output
(SF)	Default format for 1-input operations
(SSF)	Default format for 2-input operations

2.7 SHIFTER

The shifter operates on 32-bit fixed-point operands. Shifter operations include:

- shifts and rotates from off-scale left to off-scale right
- bit manipulation operations, including bit set, clear, toggle, and test
- bit field manipulation operations including extract and deposit
- support for ADSP-2100 family compatible fixed-point/floating-point conversion operations (exponent extract, number of leading 1s or 0s)

2.7.1 Shifter Operation

The shifter takes from one to three input operands: the X-input, which is operated upon; the Y-input, which specifies shift magnitudes, bit field lengths or bit positions; and the Z-input, which is operated on and updated (as in, for example, Rn = Rn OR LSHIFT Rx BY Ry). The shifter returns one output to the register file.

Input operands are fetched from the upper 32 bits of a register file location (bits 39-8, as shown in Figure 2.4) or from an immediate value in the instruction. The operands are transferred during the first half of the cycle. The result is transferred to the upper 32 bits of a register (with the eight LSBs zero-filled) during the second half of the cycle. Thus the shifter can read and write the same register file location in a single cycle.

The X-input and Z-input are always 32-bit fixed-point values. The Y-input is a 32-bit fixed-point value or an 8-bit field (*shf8*), positioned in the register file as shown in Figure 2.4 below.

Some shifter operations produce 8-bit or 6-bit results. These results are placed in either the *shf8* field or the *bit6* field (see Figure 2.5) and are sign-extended to 32 bits. Thus the shifter always returns a 32-bit result.

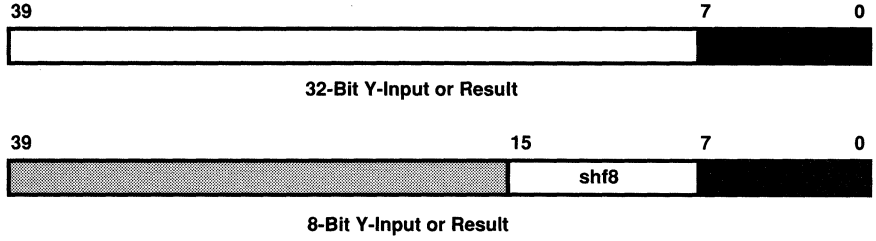


Figure 2.4 Register File Fields for Shifter Instructions

2.7.2 Bit Field Deposit & Extract Instructions

The shifter's bit field deposit and bit field extract instructions allow the manipulation of groups of bits within a 32-bit fixed-point integer word.

The Y-input for these instructions specifies two 6-bit values, *bit6* and *len6*, positioned in the Ry register as shown in Figure 2.5. Bit6 and len6 are interpreted as positive integers. Bit6 is the starting bit position for the deposit or extract. Len6 is the bit field length, which specifies how many bits are deposited or extracted.

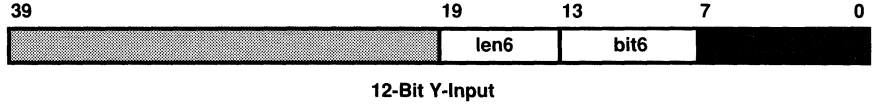


Figure 2.5 Register File Fields for FDEP, FEXT Instructions

The FDEP (field deposit) instructions take a group of bits from the input register Rx (starting at the LSB of the 32-bit integer field) and deposit them anywhere within the result register Rn. The bit6 value specifies the starting bit position for the deposit. See Figure 2.6.

The FEXT (field extract) instructions extract a group of bits from anywhere within the input register Rx and place them in the result register Rn (aligned with the LSB of the 32-bit integer field). The bit6 value specifies the starting bit position for the extract.

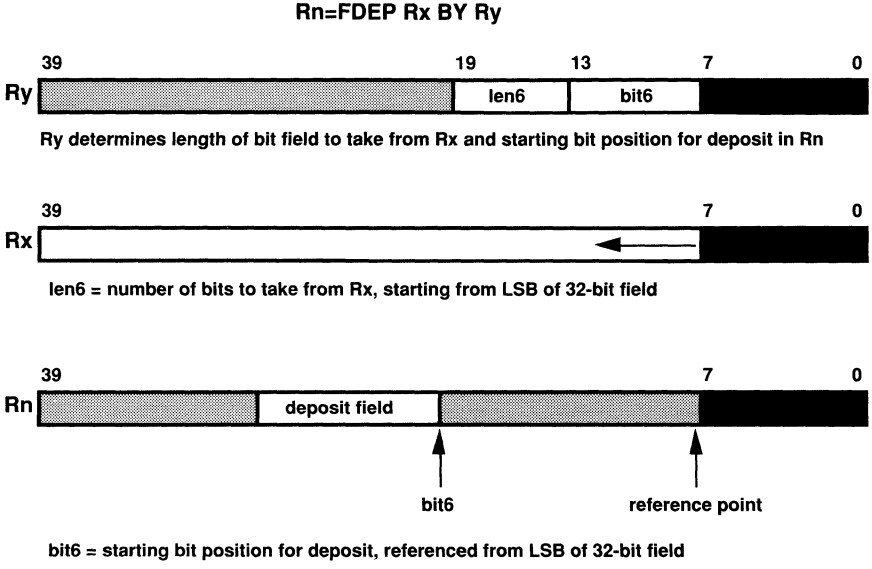
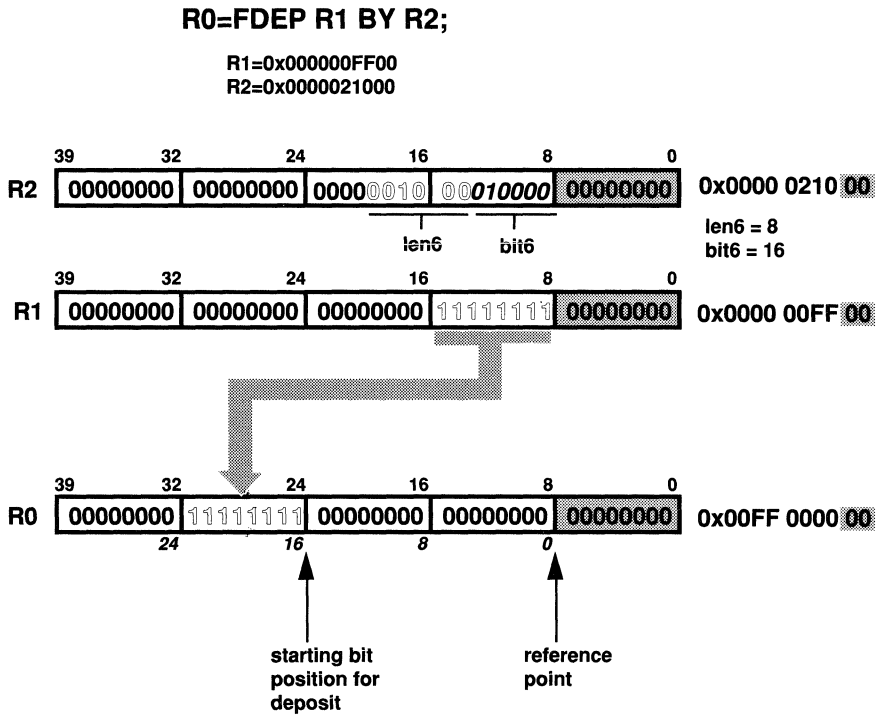


Figure 2.6 Bit Field Deposit Instruction

2 Computation Units

The following field deposit instruction example is pictured in Figure 2.7:

```
R0=FDEP R1 BY R2;
```

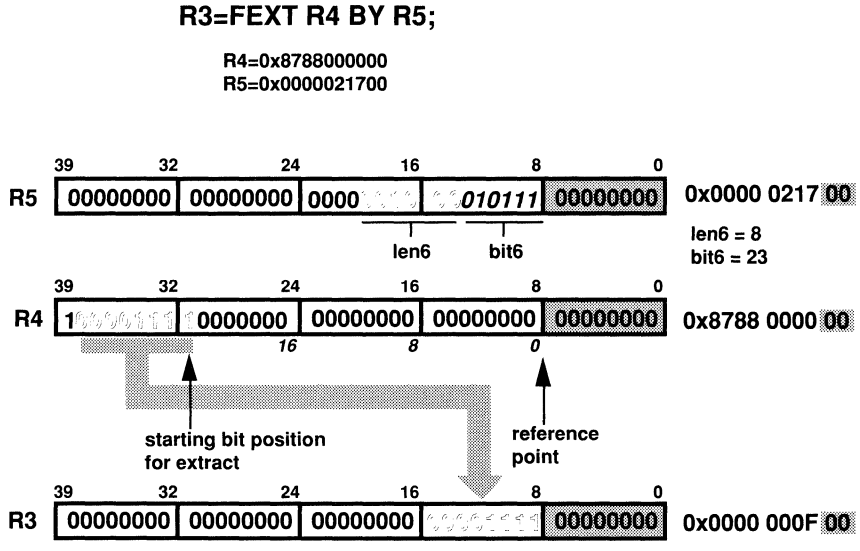


8 bits are taken from R1 and deposited in R0, starting at bit 16.
("Bit 16" is relative to reference point, the LSB of 32-bit integer field.)

Figure 2.7 Bit Field Deposit Example

The following field extract instruction example is pictured in Figure 2.8:

```
R3=FEXT R4 BY R5;
```



8 bits are extracted from R4 and placed in R3, aligned to the LSB of the 32-bit integer field.

Figure 2.8 Bit Field Extract Example

2.7.3 Shifter Status Flags

The shifter returns three status flags at the end of the operation. All of these flags appear in the ASTAT register. The SZ flag indicates if the output is zero, the SV flag indicates an overflow, and the SS flag indicates the sign bit in exponent extract operations.

ASTAT

Bit	*Name*	*Definition*
11	SV	Shifter overflow of bits to left of MSB
12	SZ	Shifter result zero
13	SS	Shifter input sign (for exponent extract only)

Flag update occurs at the end of the cycle in which the status is generated and is available on the next cycle. If a program writes the ASTAT register explicitly in the same cycle that the shifter is performing an operation, the explicit write to ASTAT supersedes any flag update caused by the shift operation.

2.7.3.1 Shifter Zero Flag (SZ)

SZ is affected by all shifter operations. It is set whenever:

- the result of a shifter operation is zero, or
- a bit test instruction specifies a bit outside of the 32-bit fixed-point field.

SZ is otherwise cleared.

2.7.3.2 Shifter Overflow Flag (SV)

SV is affected by all shifter operations. It is set whenever:

- significant bits are shifted to the left of the 32-bit fixed-point field,
- a bit outside of the 32-bit fixed-point field is tested, set or cleared,
- a field that is partially or wholly to the left of the 32-bit fixed-point field is extracted, or
- a LEFTZ or LEFTO operation returns a result of 32.

SV is otherwise cleared.

2.7.3.3 Shifter Sign Flag (SS)

SS is affected by all shifter operations. For the two EXP (exponent extract) operations, it is set if the fixed-point input operand is negative and cleared if it is positive. For all other shifter operations, SS is cleared.

2.7.4 Shifter Instruction Summary

	Instruction	*Flags* SZ	SV	SS
c	Rn = LSHIFT Rx BY Ry	*	*	0
c	Rn = LSHIFT Rx BY <data8>	*	*	0
c	Rn = Rn OR LSHIFT Rx BY Ry	*	*	0
c	Rn = Rn OR LSHIFT Rx BY <data8>	*	*	0
c	Rn = ASHIFT Rx BY Ry	*	*	0
c	Rn = ASHIFT Rx BY<data8>	*	*	0
c	Rn = Rn OR ASHIFT Rx BY Ry	*	*	0
c	Rn = Rn OR ASHIFT Rx BY <data8>	*	*	0
	Rn = ROT Rx BY RY	*	0	0
	Rn = ROT Rx BY <data8>	*	0	0
	Rn = BCLR Rx BY Ry	*	*	0
	Rn = BCLR Rx BY <data8>	*	*	0
	Rn = BSET Rx BY Ry	*	*	0
	Rn = BSET Rx BY <data8>	*	*	0
	Rn = BTGL Rx BY Ry	*	*	0
	Rn = BTGL Rx BY <data8>	*	*	0
	BTST Rx BY Ry	*	*	0
	BTST Rx BY <data8>	*	*	0
	Rn = FDEP Rx BY Ry	*	*	0
	Rn = FDEP Rx BY <bit6>:<len6>	*	*	0
	Rn = Rn OR FDEP Rx BY Ry	*	*	0
	Rn = Rn OR FDEP Rx BY <bit6>:<len6>	*	*	0
	Rn = FDEP Rx BY Ry (SE)	*	*	0
	Rn = FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
	Rn = Rn OR FDEP Rx BY Ry (SE)	*	*	0
	Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)	*	*	0
	Rn = FEXT Rx BY Ry	*	*	0
	Rn = FEXT Rx BY <bit6>:<len6>	*	*	0
	Rn = FEXT Rx BY Ry (SE)	*	*	0
	Rn = FEXT Rx BY <bit6>:<len6> (SE)	*	*	0
c	Rn = EXP Rx (EX)	*	0	*
c	Rn = EXP Rx	*	0	*
	Rn = LEFTZ Rx	*	*	0
	Rn = LEFTO Rx	*	*	0

* = Depends on data
Rn, Rx, Ry = Any register file location; bit fields used depend on instruction
c = ADSP-2100-compatible instruction

2 Computation Units

2.8 MULTIFUNCTION COMPUTATIONS

In addition to the computations performed by each computation unit, the ADSP-21020 also provides multifunction computations that combine parallel operation of the multiplier and the ALU, or dual functions in the ALU. The two operations are performed in the same way as they are in corresponding single-function computations. Flags are also determined in the same way as for the same single-function computations, except that in the dual add/subtract computation the ALU flags from the two operations are ORed together.

Each of the four input operands for computations that use both the ALU and multiplier are constrained to a different set of four register file locations, as summarized below. For example, the X-input to the multiplier can only be R8, R9, R10 or R11. In all other operations, the input operands may be any register file locations.

Dual Add/Subtract

Ra = Rx + Ry , Rs = Rx – Ry
Fa = Fx + Fy , Fs = Fx – Fy

Fixed-Point Multiply/Accumulate and Add, Subtract or Average

Rm=R3-0 * R7-4 (SSFR)	,	Ra=R11-8 + R15-12
MRF=MRF + R3-0 * R7-4 (SSF)	,	Ra=R11-8 – R15-12
Rm=MRF + R3-0 * R7-4 (SSFR)	,	Ra=(R11-8 + R15-12)/2
MRF=MRF – R3-0 * R7-4 (SSF)	,	
Rm=MRF – R3-0 * R7-4 (SSFR)	,	

Floating-Point Multiplication and ALU Operation

Fm=F3-0 * F7-4 ,	Fa=F11-8 + F15-12
	Fa=F11-8 – F15-12
	Fa=FLOAT R11-8 by R15-12
	Fa=FIX R11-8 by R15-12
	Fa=(F11-8 + F15-12)/2
	Fa=ABS F11-8
	Fa=MAX (F11-8, F15-12)
	Fa=MIN (F11-8, F15-12)

Multiplication and Dual Add/Subtract

Rm = R3-0 * R7-4 (SSFR) ,	Ra = R11-8 + R15-12 ,	Rs = R11-8 – R15-12
Fm = F3-0 * F7-4 ,	Fa = F11-8 + F15-12 ,	Fs = F11-8 – F15-12

Rm, Ra, Rs, Rx, Ry –Any register file location; fixed-point
Fm, Fa, Fs, Fx, Fy –Any register file location; floating-point

R3-0	–R3, R2, R1, R0	F3-0	–F3, F2, F1, F0
R7-4	–R7, R6, R5, R4	F7-4	–F7, F6, F5, F4
R11-8	–R11, R10, R9, R8	F11-8	–F11, F10, F9, F8
R15-12	–R15, R14, R13, R12	F15-12	–F15, F14, F13, F12

SSFR –X-input signed, Y-input signed, Fractional input, Rounded-to-nearest output
SSF –X-input signed, Y-input signed, Fractional input

2.9 REGISTER FILE

The register file provides the interface between the main processor buses (DMD and PMD) and the computation units. It also provides local storage for operands and results. The register file consists of 16 primary registers and 16 alternate (secondary) registers. All registers are 40 bits wide on the ADSP-21020 and 32 bits wide on the ADSP-21010. On the ADSP-21020, 32-bit data from the computation units is always left-justified; on register reads, the eight LSBs are ignored, and on writes, the eight LSBs are written with zeros.

Program memory accesses and data memory accesses to the register file occur on the PMD and DMD buses, respectively. One program memory and/or one data memory access can occur in one cycle. Transfers between the register file and the 40-bit DMD bus are always 40 bits wide on the ADSP-21020. The register file transfers data to and from the 48-bit PMD bus on the most significant 40 bits, writing zeros in the lower eight bits on transfers to the PMD bus.

If the same register file location is specified as both the source of an operand and the destination of a result or memory fetch, the read occurs in the first half of the cycle and the write in the second half. Thus the old data is used as the operand before the location is updated with the new result data. If writes to the same location take place in the same cycle, only the write with higher precedence actually occurs. Precedence is determined by the source of the data being written; from highest to lowest, the precedence is:

- Data memory or universal register
- Program memory
- ALU
- Multiplier
- Shifter

The individual registers of the register file are prefixed with an "f" when used in floating-point computations (in assembly language source code). The registers are prefixed with an "r" when used in fixed-point computations. The following instructions, for example, use the same three registers:

```
F0=F1 * F2;        floating-point multiply
R0=R1 * R2;        fixed-point multiply
```

The "f" and "r" prefixes do not affect the 40-bit (or 32-bit) data transfer; they only determine how the ALU, multiplier, or shifter treat the data.

2.9.1 Alternate (Secondary) Registers

To facilitate fast context switching, the register file has an alternate register set. Each half of the register file—the lower half, R0 through R7, and the upper half, R8 through R15—can independently activate its alternate register set. Two bits in the MODE1 register select the active sets. Data can be shared between contexts by placing the data to be shared in one half of the register file and activating the alternate register set of the other half.

MODE1

Bit	*Name*	*Definition*
7	SRRFH	Register file alternate select for R15-R8 (F15-F8)
10	SRRFL	Register file alternate select for R7-R0 (F7-F0)

Program Sequencing 3

3.1 OVERVIEW

Program flow in the ADSP-21020/21010 is most often linear; the processor executes program instructions sequentially. Variations in this linear flow are provided by the following program structures, illustrated in Figure 3.1 on the following page:

- *Loops*. One sequence of instructions is executed several times with zero overhead.
- *Subroutines*. The processor temporarily interrupts sequential flow to execute instructions from another part of program memory.
- *Jumps*. Program flow is permanently transferred to another part of program memory.
- *Interrupts*. A special case of subroutines in which the execution of the routine is triggered by an event that happens at run time, not by a program instruction.
- *Idle*. A special instruction that causes the processor to cease operations, holding its current state. When an interrupt occurs, the processor services the interrupt and continues normal execution.

Managing these program structures is the job of the ADSP-21020/21010's program sequencer. The program sequencer selects the address of the next instruction, generating most of those addresses itself. It also performs a wide range of related functions, such as

- incrementing the fetch address,
- maintaining stacks,
- evaluating conditions,
- decrementing the loop counter,
- calculating new addresses,
- maintaining an instruction cache, and
- handling interrupts.

3 Program Sequencing

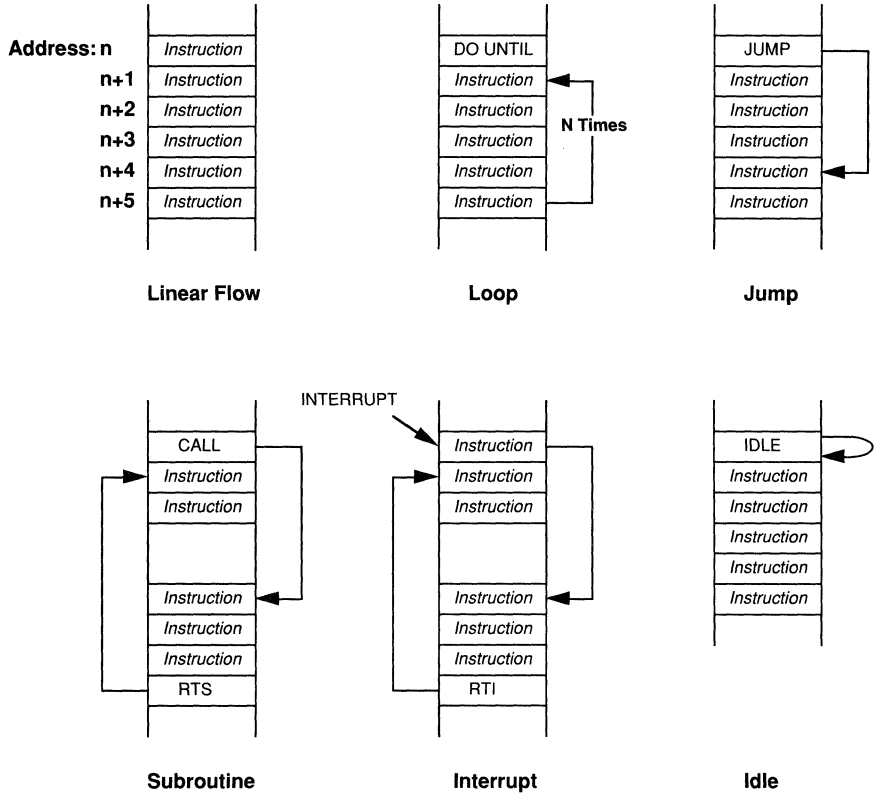


Figure 3.1 Program Flow Variations

3.1.1 Instruction Cycle

The ADSP-21020 processes instructions in three clock cycles:

- In the *fetch* cycle, the ADSP-21020 reads the instruction from either the internal instruction cache or program memory.
- During the *decode* cycle, the instruction is decoded, generating conditions that control instruction execution.
- In the *execute* cycle, the ADSP-21020 executes the instruction; the operations specified by the instruction are completed.

These cycles are overlapping, or pipelined, as shown in Figure 3.2. In sequential program flow, when one instruction is being fetched, the instruction fetched in the previous cycle is being decoded, and the instruction fetched two cycles before is being executed. Thus, the throughput is one instruction per cycle.

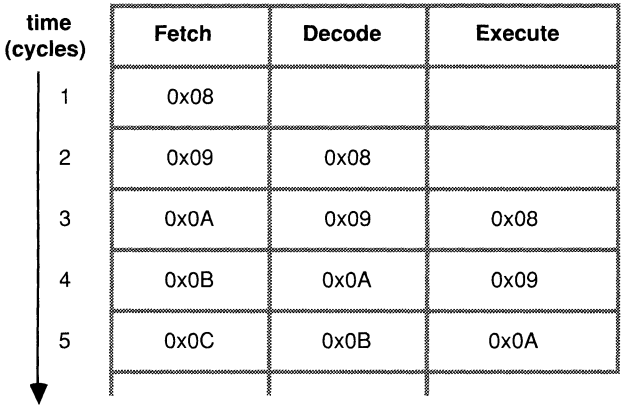


Figure 3.2 Pipelined Execution Cycles

Any non-sequential program flow can potentially decrease the ADSP-21020's instruction throughput. Non-sequential program operations include:

- Program memory data accesses that conflict with instruction fetches
- Jumps
- Subroutine Calls and Returns
- Interrupts and Returns
- Loops

3.1.2 Program Sequencer Architecture

Figure 3.3, on the next page, shows a block diagram of the program sequencer. The sequencer selects the value of the next fetch address from several possible sources.

The fetch address register, decode address register and program counter (PC) contain, respectively, the addresses of the instructions currently being fetched, decoded and executed. The PC is coupled with the PC stack, which is used to store return addresses and top-of-loop addresses.

3 Program Sequencing

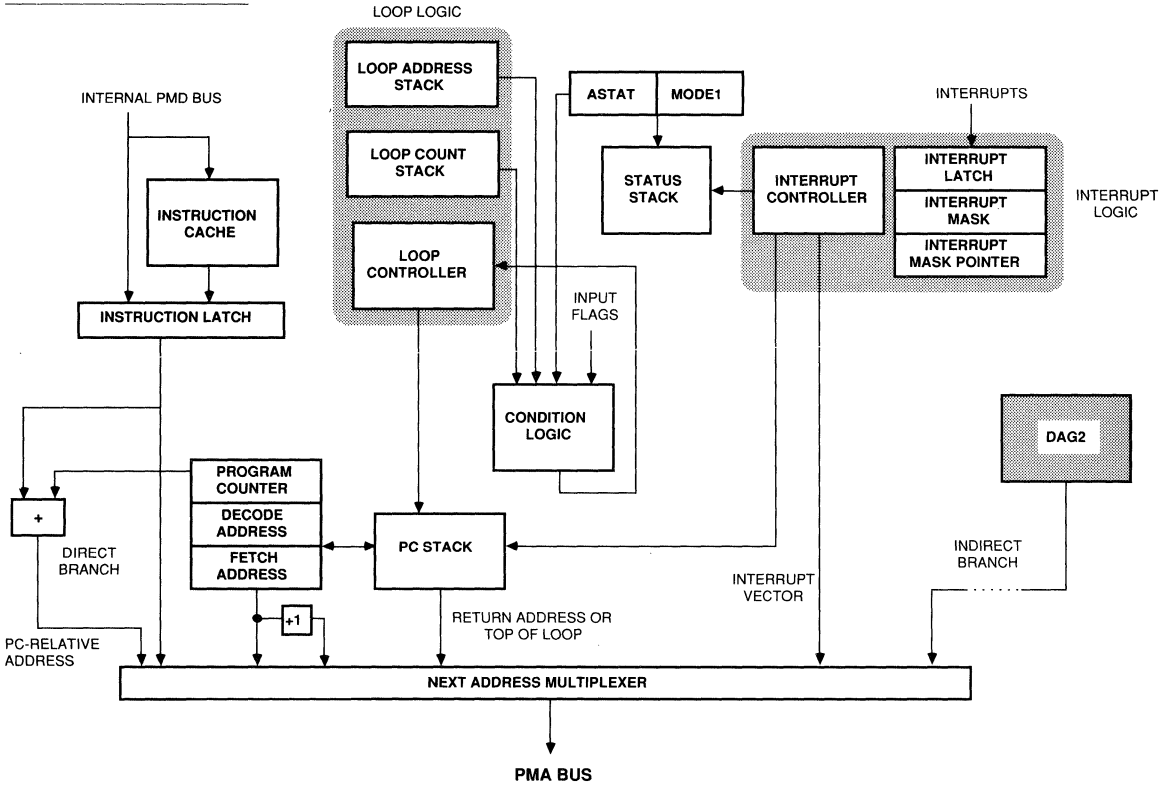


Figure 3.3 Program Sequencer Block Diagram

The interrupt controller performs all functions related to interrupt processing, such as determining whether an interrupt is masked and outputting the appropriate interrupt vector.

The instruction cache provides a means by which the ADSP-21020 can access data in program memory and fetch an instruction in the same cycle. The DAG2 data address generator (described in Chapter 4) outputs program memory data addresses.

The sequencer evaluates conditional instructions and loop termination conditions using information from the status registers. The loop address stack and loop counter stack support nested loops. The status stack stores status registers for implementing nested external interrupt routines.

3.1.2.1 Program Sequencer Registers & System Registers

Table 3.1 lists the registers located in the program sequencer. The functions of these registers are described in subsequent sections of this chapter. All registers in the program sequencer are universal registers and are thus accessible to other universal registers as well as external data memory. All registers and the tops of stacks are readable; all registers except the fetch address, decode address and PC are writeable. The PC stack can be pushed and popped by writing the PC stack pointer, which is readable and writeable. The loop address stack and status stack are pushed and popped by explicit instructions.

The *system register bit manipulation* instruction can be used to set, clear, toggle or test specific bits in the system registers. This instruction is described in Appendix A, Group IV–Miscellaneous instructions.

Due to pipelining, writes to some of these registers do not take effect on the next cycle; for example, if you write the MODE1 register to enable ALU saturation mode, the change will not occur until two cycles after the write. Also, some registers are not updated on the cycle immediately following a write; it takes an extra cycle before a read of the register yields the new value. Table 3.1 summarizes the number of extra cycles for a write to take effect (effect latency) and for a new value to appear in the register (read latency). A "0" indicates that the write takes effect or appears in the register on the next cycle after the write instruction is executed. A "1" indicates one extra cycle.

Program Sequencer Registers	*Contents*	*Bits*	*Read latency*	*Effect latency*
FADDR*	fetch address	24	-	-
DADDR*	decode address	24	-	-
PC*	execute address	24	-	-
PCSTK	top of PC stack	24	0	0
PCSTKP	PC stack pointer	5	1	1
LADDR	top of loop address stack	32	0	0
CURLCNTR	top of loop count stack (current loop count)	32	0	0
LCNTR	loop count for next DO UNTIL loop	32	0	0
System Registers				
MODE1	mode control bits	32	0	1
MODE2	mode control bits	32	0	1
IRPTL	interrupt latch	32	0	0
IMASK	interrupt mask	32	0	1
IMASKP	interrupt mask pointer (for nesting)	32	1	1
ASTAT	arithmetic status flags	32	0	1
STKY	sticky status flags	32	0	1
USTAT1	user-defined status flags	32	0	0
USTAT2	user-defined status flags	32	0	0

* read-only

Table 3.1 Program Sequencer Registers & System Registers

3 Program Sequencing

3.2 PROGRAM SEQUENCER OPERATIONS

This section is an overview of the operation of the program sequencer. The various kinds of program flow are defined here and described in detail in subsequent sections.

3.2.1 Sequential Instruction Flow

The program sequencer determines the next instruction address by examining both the current instruction being executed and the current state of the processor. If no conditions require otherwise, the ADSP-21020 executes instructions from program memory in sequential order by simply incrementing the fetch address.

3.2.2 Program Memory Data Access

Usually, the ADSP-21020 fetches an instruction from program memory on each cycle. When the ADSP-21020 executes an instruction which requires data to be read from or written to program memory, there is a conflict for that memory space. The ADSP-21020 has an instruction cache to reduce delays caused by this type of conflict.

The first time that the ADSP-21020 encounters an instruction fetch that conflicts with a program memory data access, it must fetch the instruction on the following cycle, causing a delay. The ADSP-21020 automatically writes the fetched instruction to the cache to avoid the overhead should the same instruction fetch occur again. The ADSP-21020 checks the instruction cache on every program memory data access. If the instruction needed is in the cache, the instruction fetch from the cache happens in parallel with the program memory data access, without incurring a delay.

3.2.3 Branches

A branch occurs when the fetch address is not the next sequential address following the previous fetch address. Jumps, calls and returns are the types of branches which the ADSP-21020 supports. In the program sequencer, the only difference between a jump and a call is that upon execution of a call, a return address is pushed onto the PC stack so that it is available when a return instruction is later executed. Jumps branch to a new location without allowing return.

3.2.4 Loops

The ADSP-21020 supports loop instructions through the DO UNTIL instruction. The DO UNTIL instruction causes the ADSP-21020 to repeat a sequence of instructions until a specified condition tests true.

3.3 CONDITIONAL INSTRUCTION EXECUTION

The program sequencer evaluates conditions to determine whether to execute a conditional instruction and when to terminate a loop. The conditions are based on information from the arithmetic status (ASTAT) register, mode control 1 (MODE1) register, flag inputs and loop counter. The arithmetic ASTAT bits are described in Chapter 2 in the description of each computation unit.

Each condition that the ADSP-21020 evaluates has an assembler mnemonic and a unique code (number) that is used in a conditional instruction's opcode. For most conditions, the program sequencer can test both true and false states, e.g., equal to zero and not equal to zero. Table 3.2, on the following page, defines the 32 status conditions.

The bit test flag (BTF) is bit 18 of the ASTAT register. This flag is set (or cleared) by the results of the BIT TST and BIT XOR forms of the *System Register Bit Manipulation* instruction, which can be used to test the contents of the ADSP-21020's system registers. This instruction is described in Appendix A, Group IV–Miscellaneous instructions. After BTF is set by this instruction, it can be used as the condition in a conditional instruction (with the mnemonic TF; see Table 3.2).

The two conditions that do not have complements are LCE/NOT LCE (loop counter expired/not expired) and TRUE/FOREVER. The interpretation of these condition codes is determined by context; TRUE and NOT LCE are used in conditional instructions, FOREVER and LCE in loop termination. The IF TRUE construct creates an unconditional instruction (the same effect as leaving out the condition entirely). A DO FOREVER instruction executes a loop indefinitely, until an interrupt or reset intervenes.

Because the LCE condition checks the value of the loop counter (CURLCNTR), an IF NOT LCE conditional instruction should not follow a write to CURLCNTR from memory. Otherwise, because the write occurs after the NOT LCE test, the condition is based on the old CURLCNTR value.

3 Program Sequencing

No.	*Mnemonic*	*Description*	*True If*
0	EQ	ALU equal zero	AZ = 1
1	LT	ALU less than zero	[$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 1
2	LE	ALU less than or equal zero	[$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or AZ = 1
3	AC	ALU carry	AC = 1
4	AV	ALU overflow	AV = 1
5	MV	Multiplier overflow	MV = 1
6	MS	Multiplier sign	MN = 1
7	SV	Shifter overflow	SV = 1
8	SZ	Shifter zero	SZ = 1
9	FLAG0_IN	Flag 0 input	FI0 = 1
10	FLAG1_IN	Flag 1 input	FI1 = 1
11	FLAG2_IN	Flag 2 input	FI2 = 1
12	FLAG3_IN	Flag 3 input	FI3 = 1
13	TF	Bit test flag	BTF = 1
14		Reserved	
15	LCE	Loop counter expired (DO UNTIL term)	CURLCNTR = 1
15	NOT LCE	Loop counter not expired (IF cond)	CURLCNTR ≠ 1
Bits 16-30 are the complements of bits 0-14			
16	NE	ALU not equal to zero	AZ = 0
17	GE	ALU greater than or equal zero	[$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN and $\overline{AZ}$)] = 0
18	GT	ALU greater than zero	[$\overline{AF}$ and (AN xor (AV and $\overline{ALUSAT}$)) or (AF and AN)] or AZ = 0
19	NOT AC	Not ALU carry	AC = 0
20	NOT AV	Not ALU overflow	AV = 0
21	NOT MV	Not multiplier overflow	MV = 0
22	NOT MS	Not multiplier sign	MN = 0
23	NOT SV	Not shifter overflow	SV = 0
24	NOT SZ	Not shifter zero	SZ = 0
25	NOT FLAG0_IN	Not Flag 0 input	FI0 = 0
26	NOT FLAG1_IN	Not Flag 1 input	FI1 = 0
27	NOT FLAG2_IN	Not Flag 2 input	FI2 = 0
28	NOT FLAG3_IN	Not Flag 3 input	FI3 = 0
29	NOT TF	Not bit test flag	BTF = 0
30		Reserved	
31	FOREVER	Always False (DO UNTIL)	always
31	TRUE	Always True (IF)	always

Table 3.2 Condition Codes

3.4 BRANCHES (CALL, JUMP, RTS, RTI)

The CALL instruction initiates a subroutine. Both jumps and calls transfer program flow to another part of program memory, but a call also pushes a return address onto the PC stack so that it is available when a return from subroutine instruction is later executed. Jumps branch to a new location without allowing return.

A return causes the processor to branch to the address stored at the top of the PC stack. There are two types of returns: return from subroutine (RTS) and return from interrupt (RTI). The difference between the two is that the RTI instruction not only pops the return address off the PC stack but also pops the status stack if status registers (ASTAT and MODE1) have been pushed as a result of an external interrupt.

There are a number of parameters you can specify for branches:

- Jumps, calls and returns can be conditional. The program sequencer can evaluate any one of several status conditions to decide whether the branch should be taken. If no condition is specified, the branch is always taken.

- Jumps and calls can be indirect, direct, or PC-relative. An *indirect* branch goes to an address supplied by one of the data address generators, DAG2. *Direct* branches go to the 24-bit address specified in an immediate field in the branch instruction. *PC-relative* branches also use a value specified in the instruction, but the sequencer adds this value to the current PC value to compute the address.

- Jumps, calls and returns can be delayed or nondelayed. In a *delayed* branch, the two instructions immediately after the branch instruction are executed; in a *nondelayed* branch, the program sequencer suppresses the execution of those two instructions (no-operations are performed instead).

- The JUMP (LA) instruction causes an automatic loop abort if it occurs inside a loop. When the loop is aborted, the PC and loop address stacks are popped once, so that if the loop was nested, the stacks still contain the correct values for the outer loop. (This is similar to the *break* instruction of the C programming language used to prematurely terminate execution of a loop.)

3.4.1 Delayed And Nondelayed Branches

An instruction modifier (DB) indicates that a branch is delayed; otherwise, it is nondelayed. If the branch is nondelayed, the two instructions after the branch, which are in the fetch and decode stages, are not executed (see Figure 3.4); for a call, the decode address (the address of the instruction after the call) is the return address. During the two no-operation cycles, the first instruction at the branch address is fetched and decoded.

NON-DELAYED JUMP OR CALL

CLOCK CYCLES ⟶

Execute Instruction	n	nop	nop	j
Decode Instruction	n+1->nop	n+2->nop	j	j+1
Fetch Instruction	n+2	j	j+1	j+2
	n+1 suppressed	n+2 suppressed; for call, n+1 pushed on PC stack		

NON-DELAYED RETURN

CLOCK CYCLES ⟶

Execute Instruction	n	nop	nop	r
Decode Instruction	n+1->nop	n+2->nop	r	r+1
Fetch Instruction	n+2	r	r+1	r+2
	n+1 suppressed	n+2 suppressed; r popped from PC stack		

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Figure 3.4 Nondelayed Branches

In a delayed branch, the processor continues to execute two more instructions while the instruction at the branch address is fetched and decoded (see Figure 3.5); in the case of a call, the return address is the third address after the branch instruction. A delayed branch is more efficient, but it makes the code harder to understand because of the instructions between the branch instruction and the actual branch.

DELAYED JUMP OR CALL

CLOCK CYCLES ⟶

Execute Instruction	n	n+1	n+2	j
Decode Instruction	n+1	n+2	j	j+1
Fetch Instruction	n+2	j	j+1	j+2
		for call, n+3 pushed on PC stack		

DELAYED RETURN

CLOCK CYCLES ⟶

Execute Instruction	n	n+1	n+2	r
Decode Instruction	n+1	n+2	r	r+1
Fetch Instruction	n+2	r	r+1	r+2
		r popped from PC stack		

n = Branch instruction

j = Instruction at Jump or Call address

r = Instruction at Return address

Figure 3.5 Delayed Branches

3 Program Sequencing

Because of the instruction pipeline, a delayed branch instruction and the two instructions that follow it in program memory must be executed sequentially. Instructions in the two program memory locations immediately following a delayed branch instruction can not be any of the following:

- Other Jumps, Calls or Returns
- Pushes or Pops of the PC stack
- Writes to the PC stack or PC stack pointer
- DO UNTIL instruction
- IDLE instruction

These exceptions are checked by the ADSP-21020 assembler.

The ADSP-21020 does not process an interrupt in between a delayed branch instruction and either of the two instructions that follow, since these three instructions must be executed sequentially. Any interrupt that occurs during these instructions is latched but not processed until the branch is complete.

A read of the PC stack or PC stack pointer immediately after a delayed call or return is permitted, but it will show that the return address on the PC stack has already been pushed or popped, even though the branch has not occurred yet.

3.4.2 PC Stack

The PC stack holds return addresses for subroutines and interrupt service routines and top-of-loop addresses for loops. The PC stack is 20 deep by 24 bits wide.

The PC stack is popped during returns from interrupts (RTI), returns from subroutines (RTS) and terminations of loops. The stack is full when all entries are occupied, empty when no entries are occupied, and overflowed if a call occurs when the stack is already full. The full and empty flags are stored in the sticky status register (STKY). The full flag causes a maskable interrupt.

A PC stack interrupt occurs when 19 locations of the PC stack are filled (the almost full state). Entering the interrupt service routine then immediately causes a push on the PC stack, making it full. Thus the interrupt is a *full* interrupt, even though the condition which triggers it is the *almost full* condition. The other stacks in the sequencer, the loop

address stack, loop counter stack and status stack, are provided with overflow interrupts that are activated when a push occurs while the stack is in a full state.

The program counter stack pointer (PCSTKP) is a readable and writeable register that contains the address of the top of the PC stack. The value of PCSTKP is zero when the PC stack is empty, 1, 2, ..., 20 when the stack contains data, and 31 when the stack is overflowed. A write to PCSTKP takes effect after a one-cycle delay. If the PC stack is overflowed, a write to PCSTKP has no effect.

3.5 LOOPS

The DO UNTIL instruction provides for efficient software loops, without the overhead of additional instructions to branch, test a condition, or decrement a counter. Here is a simple example of an ADSP-21020 loop:

```
LCNTR=30, DO label UNTIL LCE;
        R0=DM(I0,M0), F2=PM(I8,M8);
        R1=R0-R15;
label:  F4=F2+F3;
```

Chapter 8 contains more examples of DO UNTIL loops.

When the ADSP-21020 executes a DO UNTIL instruction, the program sequencer pushes the address of the last loop instruction and the termination condition for exiting the loop (both specified in the instruction) onto the loop address stack. It also pushes the top-of-loop address, which is the address of the instruction following the DO UNTIL instruction, on the PC stack.

Because of the instruction pipeline (fetch, decode and execute cycles), the processor tests the termination condition (and, if the loop is counter-based, decrements the counter) before the end of the loop so that the next fetch either exits the loop or returns to the top based on the test condition. Specifically, the condition is tested when the instruction two locations before the last instruction in the loop (at location e–2, where e is the end-of-loop address) is executed. If the termination condition is not satisfied, the processor fetches the instruction from the top-of-loop address stored on the top of the PC stack. If the termination condition is true, the sequencer fetches the next instruction after the end of the loop and pops the loop stack and PC stack. Loop operation is shown in Figure 3.6, on the next page.

3 Program Sequencing

LOOP-BACK

CLOCK CYCLES ⟶

Execute Instruction	e-2	e-1	e	b
Decode Instruction	e-1	e	b	b+1
Fetch Instruction	e	b	b+1	b+2
	termination condition tests false	loop start address is top of PC stack		

LOOP TERMINATION

CLOCK CYCLES ⟶

Execute Instruction	e-2	e-1	e	e+1
Decode Instruction	e-1	e	e+1	e+2
Fetch Instruction	e	e+1	e+2	e+3
	termination condition tests true	loop-back aborts; PC and loop stacks popped		

e = Loop end instruction
b = Loop start instruction

Figure 3.6 Loop Operation

3.5.1 Restrictions And Short Loops

This section describes several programming restrictions for loops. It also explains restrictions applying to short (one- and two-instruction) loops, which require special consideration because of the three-instruction fetch-decode-execute pipeline.

3.5.1.1 General Restrictions

The last three instructions of a loop cannot be any branch except a jump with loop abort (LA); otherwise, the loop may not be executed correctly.

Nested loops cannot terminate on the same instruction.

3.5.1.2 Counter-Based Loops

The third-to-last instruction of a counter-based loop (at e–2, where e is the end-of-loop address) cannot be a write to the counter from external memory.

Short loops terminate in a special way because of the instruction (fetch-decode-execute) pipeline. Counter-based loops of one or two instructions are not long enough for the sequencer to check the termination condition two instructions from the end of the loop. In these short loops, the sequencer has already looped back when the termination condition is tested. The sequencer provides special handling to avoid overhead (no-operation) cycles if the loop is iterated a minimum number of times. The detailed operation is shown in Figures 3.7 and 3.8 (on the following page). For no overhead, a loop of length one must be executed at least three times and a loop of length two must be executed at least twice.

ONE-INSTRUCTION LOOP, THREE ITERATIONS

CLOCK CYCLES ⟶

Execute Instruction	n	n+1 first iteration	n+1 second iteration	n+1 third iteration	n+2
Decode Instruction	n+1	n+1	n+1	n+2	n+3
Fetch Instruction	n+2	n+1	n+2	n+3	n+4
	LCNTR <– 3	opcode latch not updated; fetch address not updated; count expired tests true	loop-back aborts; PC & loop stacks popped		

ONE-INSTRUCTION LOOP, TWO ITERATIONS (Two Cycles of Overhead)

CLOCK CYCLES ⟶

Execute Instruction	n	n+1 first iteration	n+1 second iteration	nop	nop	n+2
Decode Instruction	n+1	n+1	n+1 -> nop	n+1 –> nop	n+2	n+3
Fetch Instruction	n+2	n+1	n+1	n+2	n+3	n+4
	LCNTR <– 2	opcode latch not updated; fetch address not updated	count expired tests true	loop-back aborts; PC & loop stacks popped		

n = DO UNTIL instruction

n+2 = Instruction after loop

Figure 3.7 One-Instruction Loops

3 Program Sequencing

Loops of length one that iterate only once or twice and loops of length two that iterate only once incur two cycles of overhead because there are two aborted instructions after the last iteration to clear the instruction pipeline.

TWO-INSTRUCTION LOOP, TWO ITERATIONS

CLOCK CYCLES ⟶

Execute Instruction	n	n+1 first iteration	n+2 first iteration	n+1 second iteration	n+2 second iteration	n+3
Decode Instruction	n+1	n+2	n+1	n+2	n+3	n+4
Fetch Instruction	n+2	n+1	n+2	n+3	n+4	n+5
	LCNTR <- 2	PC stack supplies loop start address	last instruction fetched, causes condition test; tests true	loop-back aborts; PC & loop stacks popped		

TWO-INSTRUCTION LOOP, ONE ITERATION (Two Cycles of Overhead)

CLOCK CYCLES ⟶

Execute Instruction	n	n+1 first iteration	n+2 first iteration	nop	nop	n+3
Decode Instruction	n+1	n+2	n+1->nop	n+2->nop	n+3	n+4
Fetch Instruction	n+2	n+1	n+2	n+3	n+4	n+5
	LCNTR <- 1	PC stack supplies loop start address	last instruction fetched, causes condition test; tests true	loop-back aborts; PC & loop stacks popped		

n = DO UNTIL instruction

n+3 = Instruction after loop

Figure 3.8 Two-Instruction Loops

Processing of an interrupt that occurs during the last iteration of a one-instruction loop that executes once or twice, a two-instruction loop that executes once, or the cycle following one of these loops (which is a no-operation) is delayed by one cycle. Similarly, in a one-instruction loop that iterates at least three times, processing is delayed by one cycle if the interrupt occurs during the third-to-last iteration.

3.5.1.3 Non-Counter-Based Loops

A non-counter-based loop is one in which the loop termination condition is something other than LCE. When a non-counter-based loop is the outer loop of a series of nested loops, the end address of the outer loop must be located at least two addresses after the end address of the inner loop.

The JUMP (LA) instruction is used to prematurely abort execution of a loop. When this instruction is located in the inner loop of a series of nested loops and the outer loop is non-counter-based, the address jumped to cannot be the last instruction of the outer loop. The address jumped to may, however, be the next-to-last instruction (or any earlier).

Non-counter-based short loops terminate in a special way because of the *fetch-decode-execute* instruction pipeline:

- In a three-instruction loop, the termination condition is tested when the top of loop instruction is executed. When the condition becomes true, the sequencer completes one full pass of the loop before exiting.

- In a two-instruction loop, the termination condition is checked during the last (second) instruction. If the condition becomes true when the first instruction is executed, it tests true during the second and one more full pass is completed before exiting. If the condition becomes true during the second instruction, however, two more full passes occur before the loop exit.

- In a one-instruction loop, the termination condition is checked every cycle. When the condition becomes true, the loop executes three more times before exiting.

3.5.2 Loop Address Stack

The loop address stack is six levels deep by 32 bits wide. The 32-bit word of each level consists of a 24-bit loop termination address, a 5-bit termination code, and a 2-bit loop type code:

Bits	*Value*
0-23	Loop termination address
24-28	Termination code
29	reserved (always reads 0)
30-31	Loop type code:

00	arithmetic condition-based (not LCE)
01	counter-based, length 1
10	counter-based, length 2
11	counter-based, length > 2

The loop termination address, termination code and loop type code are stacked when a DO UNTIL or PUSH LOOP instruction is executed. The stack is popped two instructions before the end of the last loop iteration or when a POP LOOP instruction is issued. A stack overflows if a push occurs when all entries in the loop stack are occupied. The stack is empty when no entries are occupied. The overflow and empty flags are in the sticky status register (STKY). Overflow causes a maskable interrupt.

The LADDR register contains the top of the loop address stack. It is readable and writeable over the DMD bus. Reading and writing LADDR does not move the loop address stack pointer; a stack push or pop, performed with explicit instructions, moves the stack pointer. LADDR contains the value 0xFFFF FFFF when the loop address stack is empty.

Because the termination condition is checked two instructions before the end of the loop, the loop stack is popped before the end of the loop on the final iteration. If LADDR is read at either of these instructions, the value will no longer be the termination address for the loop.

A jump out of a loop pops the loop address stack (and the loop count stack if the loop is counter-based) if the Loop Abort option is specified for the jump. This allows the loop mechanism to continue to function correctly. Only one pop is performed, however, so the Loop Abort cannot be used to jump more than one level of loop nesting.

3.5.3 Loop Counters And Stack

The loop counter stack is six levels deep by 32 bits wide. The loop counter stack works in synchronization with the loop address stack; both stacks always have the same number of locations occupied. Thus, the same empty and overflow status flags apply to both stacks.

The ADSP-21020 program sequencer operates two separate loop counters: the current loop counter (CURLCNTR), which tracks iterations for a loop being executed, and the loop counter (LCNTR), which holds the count value before the loop is executed. Two counters are needed to maintain the count for an outer loop while setting up the count for an inner loop.

3.5.3.1 CURLCNTR

The top entry in the loop counter stack always contains the loop count currently in effect. This entry is the CURLCNTR register, which is readable and writeable over the DMD bus. A read of CURLCNTR when the loop counter stack is empty gives the value 0xFFFF FFFF.

The program sequencer decrements the value of CURLCNTR for each loop iteration. Because the termination condition is checked two instruction cycles before the end of the loop, the loop counter is also decremented before the end of the loop. If CURLCNTR is read at either of the last two loop instructions, therefore, the value is already the count for the next iteration.

The loop counter stack is popped two instructions before the end of the last loop iteration. When the loop counter stack is popped, the new top entry of the stack becomes the CURLCNTR value, the count in effect for the executing loop. If there is no executing loop, the value of CURLCNTR is 0xFFFF FFFF after the pop.

Writing CURLCNTR does not cause a stack push. Thus, if you write a new value to CURLCNTR, you change the count value of the loop currently executing. A write to CURLCNTR when no DO UNTIL LCE loop is executing has no effect.

Because the processor must use CURLCNTR to perform counter-based loops, there are some restrictions on when you can write CURLCNTR. As mentioned under "Loop Restrictions," the third-to-last instruction of a DO UNTIL LCE loop cannot be a write to CURLCNTR from external memory. The instruction that follows a write to CURLCNTR from memory cannot be an IF NOT LCE instruction.

3.5.3.2 LCNTR

LCNTR is the value of the top of the loop counter stack *plus one*, i.e., it is the location on the stack which will take effect on the next loop stack push. To set up a count value for a nested loop without affecting the count value of the loop currently executing, you write the count value to LCNTR. A value of zero in LCNTR causes a loop to execute 2^{32} times.

The DO UNTIL LCE instruction pushes the value of LCNTR on the loop count stack, so that it becomes the new CURLCNTR value. This process is illustrated in Figure 3.9, on the next page. The previous CURLCNTR value is preserved one location down in the stack.

A read of LCNTR when the loop counter stack is full results in invalid data. When the loop counter stack is full, any data written to LCNTR is discarded.

If you read LCNTR during the last two instructions of a terminating loop, its value is the last CURLCNTR value for the loop.

3 Program Sequencing

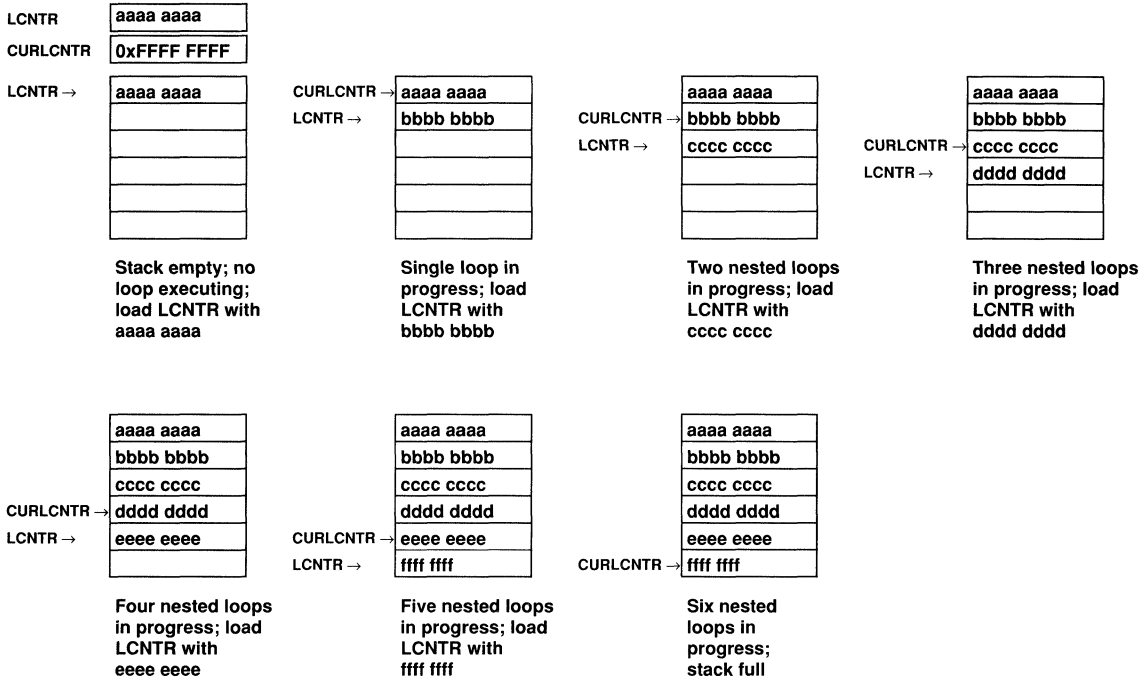


Figure 3.9 Pushing the Loop Counter Stack for Nested Loops

3.6 INTERRUPTS

An interrupt is caused by an external device asserting one of the ADSP-21020's interrupt inputs, by an internal exception such as a stack overflow, or by a user-defined software interrupt. An interrupt forces a a call to a predefined address, the interrupt vector. The ADSP-21020 assigns a unique vector to each type of interrupt it recognizes.

Externally, the ADSP-21020 supports four prioritized, individually maskable interrupts $\overline{\text{IRQ3-0}}$, each of which can be either level or edge-triggered. Among the internal interrupts are arithmetic, format, stack and timer interrupts, and reset.

An interrupt request is deemed valid if it is not masked, if interrupts are globally enabled (bit 12 in MODE1 is set), and if a higher priority request is not pending. Valid requests invoke an interrupt service sequence that

branches to the address reserved for that interrupt. Interrupt vectors are spaced at 8-instruction intervals; longer service routines can be accommodated by branching to another area of the memory space. Execution returns to normal sequencing when a RTI (return from interrupt) instruction is executed.

To process an interrupt, the program sequencer performs the following actions:

1. Outputs the appropriate interrupt vector on the program memory address.

2. Pushes the current PC value (return address) on the PC stack.

3. If the interrupt is either an external interrupt ($\overline{IRQ3\text{-}0}$) or the internal timer interrupt, the program sequencer pushes the current ASTAT and MODE1 registers on the status stack.

4. Alters the interrupt mask pointer (IMASKP) to reflect the current interrupt nesting state. The nesting mode (NESTM) bit in the MODE1 register determines whether all interrupts or only lower priority interrupts are masked during the service routine.

All interrupt service routines, except for reset, should end with a return-from-interrupt (RTI) instruction. After reset, the PC stack is empty, so there is no return address. The last instruction of a reset service routine should be a jump to the start of user code.

3.6.1 Interrupt Latency

The ADSP-21020 responds to interrupts in three stages: synchronization and latching (1 cycle), recognition (1 cycle), and branching to the interrupt vector (2 cycles). See Figure 3.10 on the next page. If an interrupt is forced in software by a write to a bit in IRPTL, it is recognized in the following cycle, and the two cycles of branching to the interrupt vector follow that. Chapter 9 contains a discussion of synchronization for external interrupts and other asynchronous signals.

3 Program Sequencing

Certain ADSP-21020 operations that span more than one cycle hold off interrupts. If an interrupt occurs during one of these operations, it is synchronized and latched, but its processing is delayed. The operations that delay interrupt processing are:

- a branch (call, jump or return) and the following cycle, whether it is an instruction (in a delayed branch) or no-operation (in a non-delayed branch)

INTERRUPT, SINGLE-CYCLE INSTRUCTION n = Single-cycle instruction

CLOCK CYCLES ⟶

Execute Instruction	n-1	n	nop	nop	v
Decode Instruction	n	n+1->nop	n+2->nop	v	v+1
Fetch Instruction	n+1	n+2	v	v+1	v+2
	interrupt occurs	interrupt recognized	n+1 pushed onto PC stack; interrupt vector output		

INTERRUPT, PROGRAM MEMORY DATA ACCESS WITH CACHE MISS n = Instruction coinciding with program memory data access, cache miss

CLOCK CYCLES ⟶

Execute Instruction	n-1	n	nop	nop	nop	v
Decode Instruction	n	n+1->nop	n+1->nop	n+2->nop	v	v+1
Fetch Instruction	n+1	-	n+2	v	v+1	v+2
	interrupt occurs	interrupt recognized, but not processed; program memory data access	interrupt processed	n+1 pushed onto PC stack; interrupt vector output		

INTERRUPT, DELAYED BRANCH n = Delayed branch instruction

CLOCK CYCLES ⟶

Execute Instruction	n-1	n	n+1	n+2	nop	nop	v
Decode Instruction	n	n+1	n+2	j->nop	j+1 ->nop	v	v+1
Fetch Instruction	n+1	n+2	j	j + 1	v	v+1	v+2
	interrupt occurs	interrupt recognized, but not processed		for a call, n+3 pushed onto PC stack; interrupt processed	j pushed onto PC stack; interrupt vector output		

v = Instruction at interrupt vector

j = Instruction at branch address

Figure 3.10 Interrupt Handling

- the first of the two cycles needed to perform a program memory data access and an instruction fetch (when there is an instruction cache miss).
- the third-to-last iteration of a one-instruction loop
- the last iteration of a one-instruction loop executed once or twice or of a two-instruction loop executed once, and the following cycle (which is a no-operation)
- the first of the two cycles needed to fetch and decode the first instruction of an interrupt routine
- waitstates for memory accesses
- bus grant

The ADSP-21020 cannot service an interrupt unless it is executing instructions or in the IDLE state. Interrupts are sampled, but not serviced, during bus grant and while the processor is waiting for memory acknowledge. IDLE is a special instruction that halts the processor until an external interrupt or timer interrupt occurs.

For most interrupts, internal and external, only one instruction is executed after the interrupt occurs and before the two instructions aborted while the processor fetches and decodes the first service routine instruction. Because of the one-cycle delay between an arithmetic exception and the STKY register update, however, there are two cycles after an arithmetic exception occurs before interrupt processing starts.

3.6.2 Interrupt Latch

The interrupt latch (IRPTL) register is a 32-bit register that latches interrupts generated by an external event (one of $\overline{IRQ}_{3\text{-}0}$) or an internal processor event (e.g., multiplier exception). This register contains any current interrupt or any pending interrupts. Because this register is readable and writeable, any interrupt except for reset can be set or cleared in software. Do not write to the reset bit (bit 1) in IRPTL because this puts the processor in an illegal state.

IRPTL is cleared by a processor reset.

Table 3.3 shows the bits in IRPTL. The second column lists the address (in hexadecimal) of the interrupt vector. Each interrupt vector is separated by eight memory locations. The third column lists an interrupt mnemonic. This name is provided for convenience; it is not required by the assembler.

Bit (IR#)	*Address*	*Name*	*Function*
0	0x00		Reserved for emulation*
1	0x08	RSTI	Reset (read-only)*
2	0x10		Reserved
3	0x18	SOVFI	Status stack or loop stack overflow or PC stack full
4	0x20	TMZHI	Timer =0 (high priority option)
5	0x28	IRQ3I	$\overline{IRQ_3}$ asserted
6	0x30	IRQ2I	$\overline{IRQ_2}$ asserted
7	0x38	IRQ1I	$\overline{IRQ_1}$ asserted
8	0x40	IRQ0I	$\overline{IRQ_0}$ asserted
9	0x48		Reserved
10	0x50		Reserved
11	0x58	CB7I	Circular buffer 7 overflow interrupt
12	0x60	CB15I	Circular buffer 15 overflow interrupt
13	0x68		Reserved
14	0x70	TMZLI	Timer=0 (low priority option)
15	0x78	FIXI	Fixed-point overflow
16	0x80	FLTOI	Floating-point overflow exception
17	0x88	FLTUI	Floating-point underflow exception
18	0x90	FLTII	Floating-point invalid exception
19-23	0x98-0xB8		Reserved
24	0xC0	SFT0I	User software interrupt 0
25	0xC8	SFT1I	User software interrupt 1
26	0xD0	SFT2I	User software interrupt 2
27	0xD8	SFT3I	User software interrupt 3
28	0xE0	SFT4I	User software interrupt 4
29	0xE8	SFT5I	User software interrupt 5
30	0xF0	SFT6I	User software interrupt 6
31	0xF8	SFT7I	User software interrupt 7

* Nonmaskable

Table 3.3 Interrupt Vectors and Priority

3.6.2.1 Interrupt Priority

The interrupt bits in IRPTL are ordered by priority. The interrupt priority is from 0 (highest) to 31 (lowest). Interrupt priority determines which interrupt is serviced first when two occur in the same cycle. It also determines which interrupts are nested when nesting is enabled (see "Interrupt Nesting and IMASKP," later in this chapter).

The arithmetic interrupts (fixed-point overflow and floating-point overflow, underflow and invalid operation) are determined from flags in the sticky status register (STKY). By reading these flags, the service routine for one of these interrupts can determine which condition caused the interrupt. The routine also has to clear the STKY bit so that the interrupt is not still active after the service routine is done.

The timer reaching zero causes both interrupt 4 and interrupt 14. This feature allows you to choose the priority of the timer interrupt. Unmask the timer interrupt that has the priority you want, and leave the other one masked. Unmasking both interrupts results in two interrupts when the timer reaches zero. The processor would service the higher priority interrupt first, then the lower priority interrupt.

3.6.2.2 Software Interrupts

The ADSP-21020 provides software interrupts that emulate interrupt behavior but are activated through software instead of hardware. An instruction that sets one of bits 24-31 in IRPTL (either a BIT SET instruction or a write to IRPTL) activates a software interrupt. The ADSP-21020 branches to the corresponding interrupt routine if that interrupt is not masked and interrupts are enabled.

3.6.3 Interrupt Masking And Control

All interrupts except for reset can be enabled and disabled by the global interrupt enable bit, IRPTEN, bit 12 in the MODE1 register. This bit is cleared at reset. You must set this bit for interrupts to be enabled.

3.6.3.1 Interrupt Mask

All interrupts except for reset interrupt can be masked. Masked means the interrupt is disabled. Interrupts that are masked are still latched, so that if the interrupt is later unmasked, it is processed. Upon chip reset, all interrupts except reset are masked.

The IMASK register controls interrupt masking. The bits in the IMASK register correspond directly to the same bits in the IRPTL register; for example, bit 10 in the IMASK register masks or unmasks the same interrupt latched by bit 10 in the IRPTL register. If a bit is set, its interrupt is unmasked (enabled); if the bit is cleared, the interrupt is masked (disabled). The IMASK register prevents the interrupts from being serviced but not from being latched in IRPTL for future recognition.

3.6.3.2 Interrupt Nesting & IMASKP

The ADSP-21020 supports the nesting of one interrupt service routine inside another; that is, a service routine can be interrupted by a higher priority interrupt. This feature is controlled by the nesting mode bit (NESTM) in the MODE1 register. When the NESTM bit is a 0, an interrupt service routine cannot be interrupted; any interrupt that occurs will be processed only after the routine finishes. When NESTM is a 1, higher priority interrupts can interrupt if they are not masked; lower or equal priority interrupts cannot. The NESTM bit should only be changed outside of an interrupt service routine or during the reset service routine; otherwise, interrupt nesting may not work correctly.

In nesting mode, the ADSP-21020 uses the interrupt mask pointer (IMASKP) to create a temporary interrupt mask for each level of interrupt nesting; the IMASK value is not affected. The ADSP-21020 changes IMASKP each time a higher priority interrupt interrupts a lower priority service routine.

The bits in IMASKP correspond to the interrupts in order of priority. When an interrupt occurs, its bit is set in IMASKP. If nesting is enabled, a new temporary interrupt mask is generated by masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP (and keeping higher priority interrupts the same as in IMASK). When a return from an interrupt service routine is executed, the highest priority bit set in IMASKP is cleared, and again a new temporary interrupt mask is generated by masking all interrupts of equal or lower priority to the highest priority bit set in IMASKP. The bit set in IMASKP that has the highest priority always corresponds to the priority of the interrupt being serviced.

If nesting is not enabled, the processor masks out all interrupts and IMASKP is not used, although IMASKP is still updated to create a temporary interrupt mask.

An interrupt routine cannot be nested within itself. The ADSP-21020 ignores and does not latch an interrupt that occurs while its service routine is already executing.

3.6.4 Status Stack

For low-overhead interrupt servicing, the ADSP-21020 automatically saves and restores the status and mode contexts of the interrupted program. The four external interrupts and the timer interrupt

automatically push ASTAT and MODE1 onto the status stack, which is five levels deep. These registers are automatically popped from the status stack by the interrupt return (RTI) instruction. Other interrupts require explicit save and restore of the appropriate registers to data memory or program memory.

Pushing ASTAT and MODE1 preserves the status and control bit settings so that if the service routine alters these bits, the original settings are automatically restored upon the return from interrupt. Note, however, that the Flag bits in ASTAT are not affected by status stack pushes and pops; the values of these bits carry over from the main program to the service routine and from the service routine to the main program.

The top of the status stack contains the current values of ASTAT and MODE1. Reading and writing these registers does not move the stack pointer. The stack pointer is moved, however, by explicit PUSH and POP instructions.

3.6.5 External Interrupt Timing & Sensitivity

Each of the four ADSP-21020 external interrupts, $\overline{\text{IRQ3-0}}$, can be either level- or edge-triggered.

The ADSP-21020 samples interrupts once every CLKIN cycle. Level-sensitive interrupts are considered valid if sampled active (low). A level-sensitive interrupt must go inactive (high) before the processor returns from the interrupt service routine. If a level-sensitive interrupt is still active when the processor samples it, the processor treats it as a new request, repeating the same interrupt routine without returning to the main program (assuming no higher priority interrupts are active).

Edge-triggered interrupt requests are considered valid if sampled high in one cycle and low in the next. The interrupt can stay active indefinitely. To request another interrupt, the signal must go high, then low again.

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

3 Program Sequencing

A bit for each interrupt in the MODE2 register indicates the sensitivity mode of each interrupt.

MODE2

Bit	*Name*	*Definition*
0	IRQ0E	$\overline{\text{IRQ0}}$ 1=edge sensitive; 0=level-sensitive
1	IRQ1E	$\overline{\text{IRQ1}}$ 1=edge sensitive; 0=level-sensitive
2	IRQ2E	$\overline{\text{IRQ2}}$ 1=edge sensitive; 0=level-sensitive
3	IRQ3E	$\overline{\text{IRQ3}}$ 1=edge sensitive; 0=level-sensitive

Interrupts are sampled during a bus grant, but remain pending until the processor regains control of the bus and continues program execution. At that time pending interrupts are serviced in order of priority.

3.6.5.1 Asynchronous External Interrupts

The processor accepts interrupts that are asynchronous to the ADSP-21020 clock; that is, an interrupt signal may change at any time. An asynchronous interrupt must be held low at least one CLKIN cycle to guarantee that it gets sampled. The delay associated with synchronizing asynchronous signals is discussed in Chapter 9. Synchronous interrupts need only meet the setup and hold time requirements relative to the rising edge of CLKIN.

3.7 STACK FLAGS

The STKY register maintains stack full and stack empty flags for the PC stack as well as overflow and empty flags for the status stack and loop stack. Unlike other STKY bits, several of these flag bits are not "sticky." They are set by the occurrence of the condition they indicate and are cleared when the condition is changed (by a push, pop or processor reset).

Bit	*Name*	*Definition*	*Sticky/Not Sticky*	*Cleared By*
21	PCFL	PC stack full	Not sticky	Pop
22	PCEM	PC stack empty	Not sticky	Push
23	SSOV	Status stack overflow	Sticky	RESET
24	SSEM	Status stack empty	Not sticky	Push
25	LSOV	Loop stacks overflow*	Sticky	RESET
26	LSEM	Loop stacks empty*	Not sticky	Push

* Loop address stack and loop counter stack

The status stack flags are read-only. Writes to the STKY register have no effect on these bits.

The overflow and full flags are provided for diagnostic aid and are not intended to allow recovery from overflow. Status stack or loop stack overflow or PC stack full causes an interrupt.

The empty flags facilitate off-chip stack saves. You monitor the empty flag when saving a stack to the external memory to know when all values have been transferred. The empty flags do not cause interrupts because an empty stack is an acceptable condition.

3.8 IDLE

IDLE is a special instruction that halts the processor in a low-power state until an external interrupt or timer interrupt occurs. When the processor encounters an IDLE instruction, it fetches the instruction at the fetch address and then holds its outputs in the states shown in Table 3.4.

Output	*State*
PMA23-0	Next Fetch Address
PMD47-0	High Impedance
$\overline{\text{PMS1-0}}$	Driven; value depends on address (one is high, the other low)
$\overline{\text{PMRD}}$	High
$\overline{\text{PMWR}}$	High
PMPAGE	Driven; value depends on address
DMA31-0	Driven; value undefined but stable
DMD39-0	High Impedance
$\overline{\text{DMS3-0}}$	High
$\overline{\text{DMRD}}$	High
$\overline{\text{DMWR}}$	High
DMPAGE	Low
FLAG3-0	Depends on internal state
$\overline{\text{BG}}$	Depends on $\overline{\text{BR}}$
TIMEXP	Depends on internal state
TDO	Depends on $\overline{\text{TRST}}$, TCK and internal state

Table 3.4 States of Outputs During IDLE

The clock continues to run during IDLE, as well as the timer if enabled. When an interrupt occurs, either externally or from the timer, the processor responds as normal, outputting the interrupt vector. After two cycles needed to fetch and decode the first instruction of the interrupt routine, the processor continues executing instructions normally. On return from the interrupt, execution continues at the instruction after the IDLE instruction.

3.9 INSTRUCTION CACHE

The instruction cache is a 2-way, set-associative cache with entries for 32 instructions. The operation of the cache is transparent to the programmer. The ADSP-21020 caches only instructions that conflict with program memory data accesses. This feature makes the cache considerably more efficient than a cache that loads every instruction, because typically only a few instructions access data from program memory.

Because of the three-stage instruction pipeline, if the instruction at address n requires a program memory data access, there is a conflict with the instruction fetch at address n+2, assuming sequential execution. It is this fetched instruction (n+2) that is stored in the instruction cache, not the instruction requiring the program memory data access.

If the instruction needed is in the cache, a "cache hit" occurs—the cache provides the instruction while the program memory data access is performed. If the instruction needed is not in the cache, a "cache miss" occurs, and the external instruction fetch takes place in the cycle following the program memory data access, incurring one cycle of overhead. This instruction is loaded into the cache, if the cache is enabled and not frozen, so that it is available the next time the same instruction requiring program memory data is executed.

3.9.1 Cache Architecture

Figure 3.11 is a block diagram of the instruction cache. The instruction cache contains 32 entries. An entry consists of a register pair containing an instruction and its address. Each entry has a Valid bit that is set if the entry contains a valid instruction.

The entries are divided into 16 sets (set 15-set 0) of two entries each, entry 0 and entry 1. Each set has an LRU (Least Recently Used) bit whose value indicates which of the two entries contains the least recently used instruction (1=entry 1, 0=entry 0).

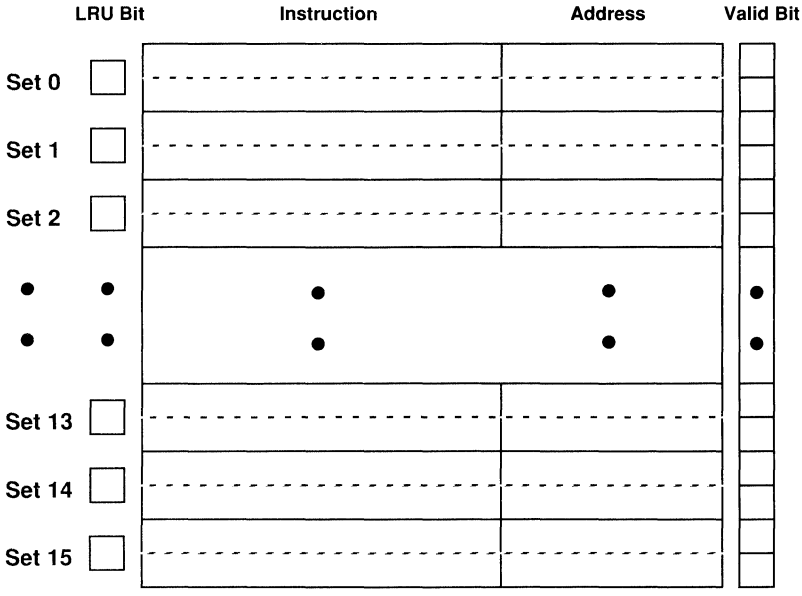


Figure 3.11 Instruction Cache Architecture

Every possible instruction address is mapped to a set in the cache by its 4 LSBs. When the processor needs to fetch an instruction from the cache, it uses the 4 address LSBs as an index to a particular set. Within that set, it checks the addresses of the two entries to see whether either contains the needed instruction. A cache hit occurs if the instruction is found, and the LRU bit is updated if necessary to indicate the entry that did not contain the needed instruction.

A cache miss occurs if neither entry in the set contains the needed instruction. In this case, a new instruction and its address are loaded into the least recently used entry of the set that matches the 4 LSBs of the address. The LRU bit is toggled to indicate that the other entry in the set is now the least recently used.

Because instructions are mapped to sets by their 4 address LSBs, there is no need to store these bits in the cache; the 4 LSBs are implied by the set in which the instruction has been stored. Only bits 23-4 are actually stored in a cache entry.

3.9.2 Cache Efficiency

Usually, the cache operation and its efficiency is not a concern. However, there are some situations that can degrade cache efficiency and can be remedied easily in the program.

When a cache miss occurs, the needed instruction is loaded into the cache so that if the same instruction is needed again, it will be there (a cache hit will occur). However, if another instruction whose address is mapped to the same set displaces this instruction, there will be a cache miss instead. The LRU bits help to reduce this possibility since at least two other instructions mapped to the same set must be needed before an instruction is displaced. If three instructions mapped to the same set are all needed repeatedly, cache efficiency (hit rate) can go to zero. The solution is to move one or more of the instructions to a new address, one that is mapped to a different set.

An example of some code that is cache-inefficient is shown in Figure 3.12. The program memory data access at address 0x101 in the *tight* loop causes the instruction at 0x103 to be cached (in set 3). Each time the subroutine *sub* is called, the program memory data accesses at 0x201 and 0x211 displace this instruction by loading the instructions at 0x203 and 0x213 into set 3. If the subroutine is called only rarely during the loop execution, the impact will be minimal. If the subroutine is called frequently, the effect will be noticeable. If the execution of the loop is time-critical, it would be advisable to move the subroutine up one location (starting at 0x201), so that the two cached instructions end up in set 4 instead of 3.

3.9.3 Cache Enable And Cache Freeze

Freezing the cache prevents any changes to its contents; i.e., a cache miss will not result in a new instruction being stored in the cache. Disabling the cache stops its operation completely; all instruction fetches conflicting with program memory data accesses are delayed by the access. These functions are selected by the CADIS and CAFRZ (cache enable/disable and cache freeze) bits in the MODE2 register. The cache is cleared (contains no instructions), unfrozen and enabled after a reset.

MODE2 Bit	*Name*	*Function*
4	CADIS	Cache disable
19	CAFRZ	Cache freeze

```
Address  Instruction
100              LCNTR=1024, DO tight UNTIL LCE;
101                 R0=DM(I0,M0), PM(I8,M8)=F3;
102                 R1=R0-R15;
103                 IF EQ CALL (sub);
104                 F2=FLOAT R1;
105                 F3=F2*F2;
106      tight:  F3=F3+F4;
107              PM(I8,M8)=F3;
•
•
•
200      sub:    R1=R13;
201              R14=PM(I9,M9);
•
•
•
211              PM(I9,M9)=R12;
•
•
•
21F              RTS;
```

Figure 3.12 Cache-Inefficient Code

Data Addressing 4

4.1 OVERVIEW

The ADSP-21020/21010's two data address generators (DAGs) simplify the task of organizing data by maintaining pointers into memory. The DAGs allow the processor to address memory *indirectly*; that is, an instruction specifies a DAG register containing an address instead of the address value itself.

Data address generator 1 (DAG1) produces 32-bit addresses for data memory. Data address generator 2 (DAG2) produces 24-bit addresses for program memory. The basic architecture for both DAGs is shown in Figure 4.1, which can be found on the following page.

The DAGs also support in hardware some functions commonly used in digital signal processing algorithms. Both DAGs support circular buffers, which require advancing a pointer repetitively through a range of memory. DAG1 can also perform a bit-reverse operation, which places the bits of an address in reverse order to form a new address.

4.2 DAG REGISTERS

Each DAG contains four types of registers: Index (I), Modify (M), Base (B) registers, and Length (L) registers.

An I register acts as a pointer to memory, and an M register contains the increment value for advancing the pointer. By modifying an I register with different M values, you can vary the increment as needed.

B registers and L registers are used only for circular data buffers. A B register holds the base (starting) address of a circular buffer. The same-numbered L register holds the number of locations in (i.e. the length of) the circular buffer.

4 Data Addressing

Each DAG contains eight of each type of register:

DAG1 registers (32-bit)	*DAG2 registers (24-bit)*
B0 - B7	B8 - B15
I0 - I7	I8 - I15
M0 - M7	M8 - M15
L0 - L7	L8 - L15

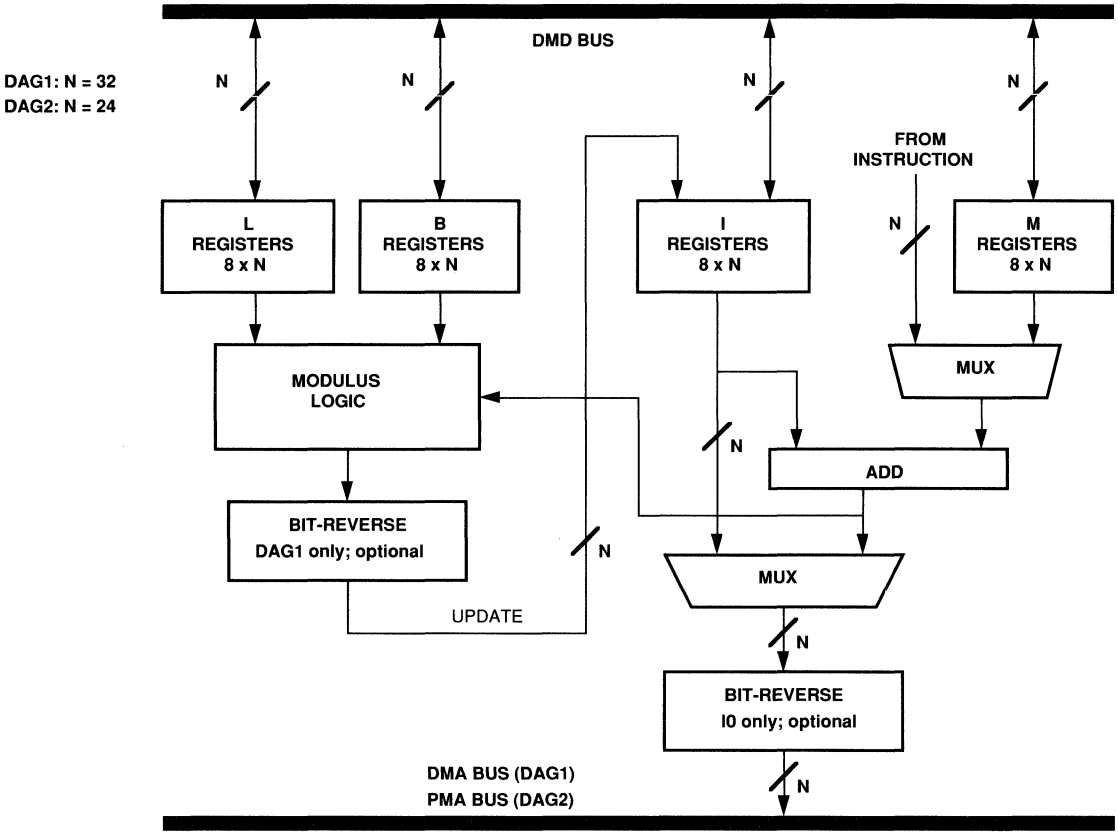


Figure 4.1 Data Address Generator Block Diagram

4.2.1 Alternate DAG Registers

Each DAG register has an alternate register for context switching. For activating alternate registers, each DAG is organized into high and low halves, as shown in Figure 4.2. The high half of DAG1 contains the I, M, B and L registers numbered 4-7, and the low half, the registers numbered 0-3. Likewise, the high half of DAG2 consists of registers 12-15, and the low half consists of registers 8-11.

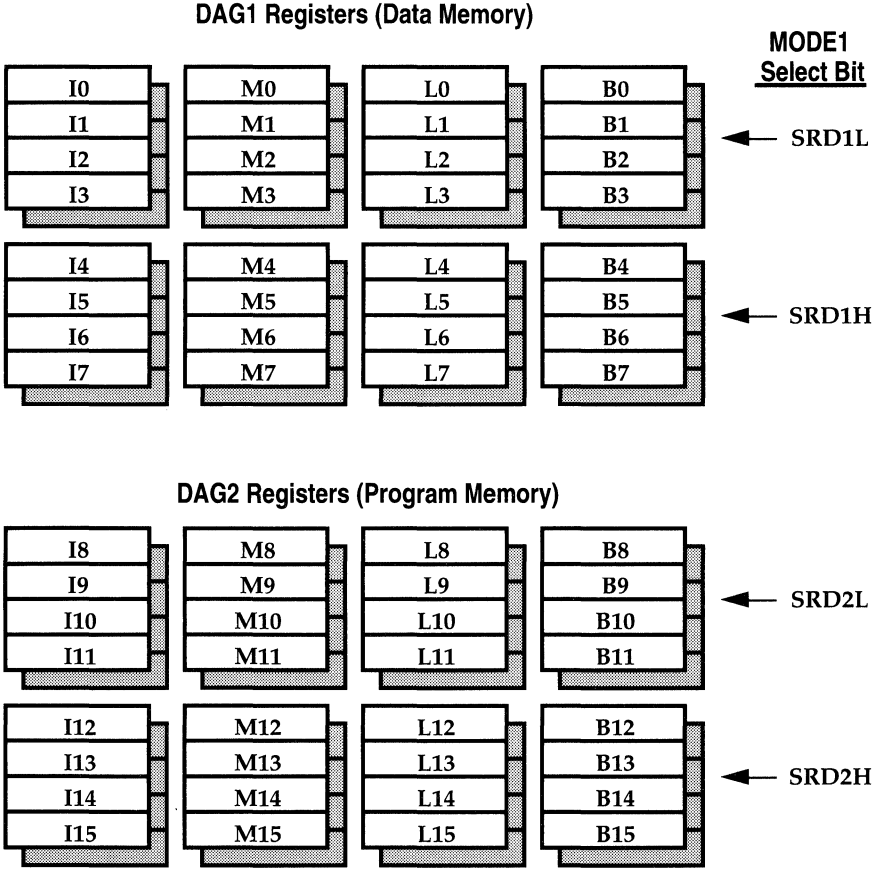


Figure 4.2 Alternate DAG Registers

4 Data Addressing

Bits in the MODE1 register determine for each half whether primary or alternate registers are active (0=primary registers active, 1=alternate registers active):

MODE1

Bit	*Name*	*Definition*
3	SRD1H	DAG1 alternate register select (4-7)
4	SRD1L	DAG1 alternate register select (0-3)
5	SRD2H	DAG2 alternate register select (12-15)
6	SRD2L	DAG2 alternate register select (8-11)

This grouping of alternate registers lets you pass pointers between contexts in each DAG.

4.3 DAG OPERATION

DAG operations include:

- address output and modification,
- modulo addressing (for circular buffers), and
- bit-reversed addressing

4.3.1 Address Output And Modification

The processor can add an offset (modifier), either an M register or an immediate value, to an I register and output the resulting address; this is called a *pre-modify without update* operation. Or it can output the I register value as it is, and then add an M register or immediate value to form a new I register value. This is a *post-modify* operation. These operations are compared in Figure 4.3. The pre-modify operation does not change the value of the I register. The width of an immediate modifier depends on the instruction; it can be as much as the width of the I register. The L register and modulo logic do not affect a pre-modified address—pre-modify addressing is always linear, not circular.

4.3.1.1 DAG Modify Instructions

In ADSP-21020/21010 assembly language, pre-modify and post-modify operations are distinguished by the positions of the index and modifier (M register or immediate value) in the instruction. The I register before the modifier indicates a post-modify operation. If the modifier comes first, a pre-modify without update operation is indicated. The following instruction, for example, accesses the program memory location with an

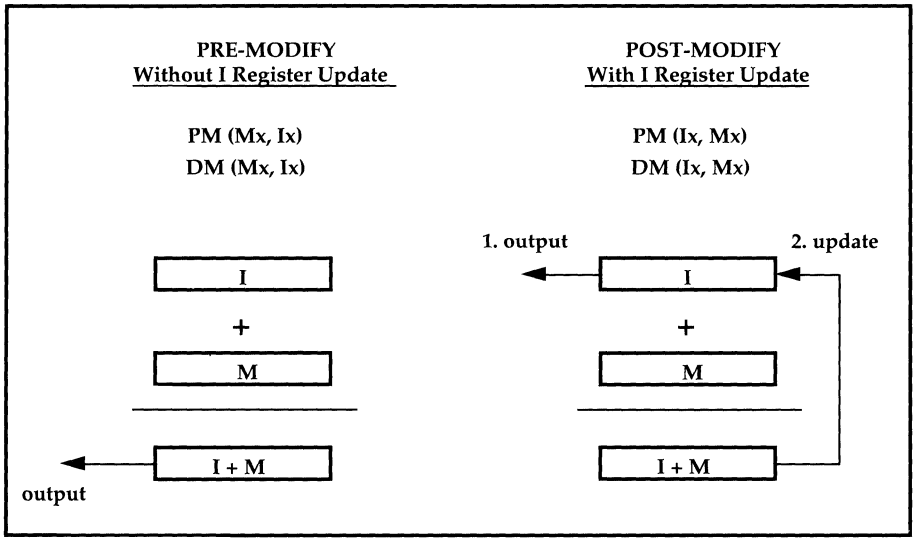


Figure 4.3 Pre-Modify and Post-Modify Operations

address equal to the value stored in I15, and the value I15 + M12 is written back to the I15 register:

```
R6 = PM(I15,M12);
```
Indirect addressing with post-modify

If the order of the I and M registers is switched, however,

```
R6 = PM(M12,I15);
```
Indirect addressing with pre-modify

the instruction accesses the location in program memory with an address equal to I15 + M12, but does not change the value of I15.

Any M register can modify any I register within the same DAG (DAG0 or DAG1). Thus,

```
DM(M0,I2) = TPERIOD;
```

is a legal instruction that accesses the data memory location M0 + I2; however,

```
DM(M0,I14) = TPERIOD;
```

is *not* a legal instruction because the I and M registers belong to different DAGs.

4.3.1.2 Immediate Modifiers

The magnitude of an immediate value that can modify an I register depends on the instruction type and whether the I register is in DAG1 or DAG2. DAG1 modify values can be up to 32 bits wide; DAG2 modify values can be up to 24 bits wide. Some instructions with parallel operations only allow modify values up to 6 bits wide. Here are two examples:

32-bit modifier:

```
R1=DM(0x40000000,I1);
```
DM address = I1 + 0x4000 0000

6-bit modifier:

```
F6=F1+F2,PM(I8,0x0B)=ASTAT;
```
PM address = I8, I8 = I8 + 0x0B

4.3.2 Circular Buffer Addressing

The DAGs provide for addressing of locations within a circular data buffer. A circular buffer is a set of memory locations that stores data. An index pointer steps through the buffer, being post-modified and updated by the addition of a specified value (positive or negative) for each step. If the modified address pointer falls outside the buffer, the length of the buffer is subtracted from or added to the value, as required to wrap the index pointer back to the start of the buffer (see Figure 4.4). There is no restriction on the value of the base address for a circular buffer.

Circular buffer addressing must use M registers for post-modify of I registers, not pre-modify; for example:

```
F1=DM(I0,M0);
F1=DM(M0,I0);
```
Use post-modify addressing for circular buffers, not pre-modify.

4.3.2.1 Circular Buffer Operation

You set up a circular buffer in assembly language by initializing an L register with a positive, nonzero value and loading the corresponding (same-numbered) B register with the base (starting) address of the buffer. The corresponding I register is automatically loaded with this same starting address.

On the first post-modify access using the I register, the DAG outputs the I register value on the address bus and then modifies it by adding the specified M register or immediate value to it. If the modified value is

Length = 11
Base address = 0
Modifier (step size) = 4

Sequence shows order in which locations are accessed in one pass.
Sequence repeats on subsequent passes.

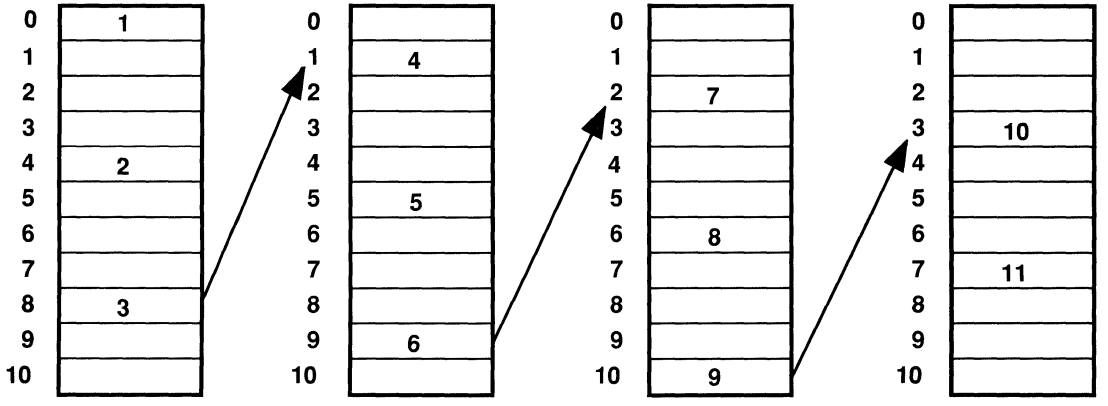


Figure 4.4 Circular Data Buffers

within the buffer range, it is written back to the I register. If the value is outside the buffer range, the L register value is subtracted (or, if the modify value is negative, added) first.

If M is positive,

$I_{new} = I_{old} + M$ *if* $I_{old} + M <$ *Buffer base + length (end of buffer)*
$I_{new} = I_{old} + M - L$ *if* $I_{old} + M \geq$ *Buffer base + length (end of buffer)*

If M is negative,

$I_{new} = I_{old} + M$ *if* $I_{old} + M \geq$ *Buffer base (start of buffer)*
$I_{new} = I_{old} + M + L$ *if* $I_{old} + M <$ *Buffer base (start of buffer)*

4.3.2.2 Circular Buffer Registers

All four types of DAG registers are involved in the operation of a circular buffer:

- The I register contains the value which is output on the address bus.
- The M register contains the post-modify amount (positive or negative) which is added to the I register at the end of each memory access. The M register can be any M register in the same DAG as the I register and does not have to have the same number. The modify value can also be an immediate number instead of an M register. The magnitude of the modify value, whether from an M register or immediate, must be less than the length (L register) of the circular buffer.
- The L register sets the size of the circular buffer and thus the address range that the I register is allowed to circulate through. L must be positive and cannot have a value greater than $2^{31} - 1$ *(L0-L7)* or $2^{23} - 1$ *(L8-L15)*. If an L register's value is zero, its circular buffer operation is disabled.
- The B register, or the B register plus the L register, is the value that the modified I value is compared to after each access. When the B register is loaded, the corresponding I register is simultaneously loaded with the same value. When I is loaded, B is not changed. B and I can be read independently.

4.3.2.3 Circular Buffer Overflow Interrupts

There is one set of DAG registers for each memory space that can generate an interrupt upon circular buffer overflow (i.e. address wraparound). For data memory, the registers are B7, I7, L7, and for program memory they are B15, I15, L15. Circular buffer overflow interrupts can be used to implement a ping-pong (swap I/O buffer pointers) routine, for example.

Whenever a circular buffer addressing operation using these registers causes the address in the I register to be incremented (or decremented) past the end (or start) of the circular buffer, an interrupt is generated. Depending on which register set was used, the interrupt is either:

Interrupt	*DAG Registers To Use*	*Vector Address*	*Symbolic Name**
DAG1 circular buffer 7 overflow	B7, I7, L7	0x58	CB7I
DAG2 circular buffer 15 overflow	B15, I15, L15	0x60	CB15I

* These symbols are defined in the `#include` file `def21020.h`. See Listing 8.5 in the section "Initialization Following Reset (Initial Setups)" of Chapter 8, Programmer's Tutorial, or the *ADSP-21020/21010 Programmer's Quick Reference*.

Specifically, an interrupt is generated during an instruction's address post-modify when:

(for M<0) $I + M < B$
(for M≥0) $I + M \geq B + L$

The interrupts can be masked by clearing the appropriate bit in IMASK.

There may be situations where you want to use I7 or I15 without circular buffering but with the circular buffer overflow interrupts unmasked. To disable the generation of these interrupts, set the B7/B15 and L7/L15 registers to values that assure that the conditions that generate interrupts (as specified above) never occur. For example, when accessing the address range 0x1000–0x2000, your program could set B=0x0000 and L=0xFFFF. Note that setting the L register to zero will not achieve the desired results.

If you are using either of the circular buffer overflow interrupts, you should avoid using the corresponding I register(s) (I7 and/or I15) in the rest of your program, or be careful to set the B and L registers as described above to prevent spurious interrupt branching.

The STKY status register includes two bits that also indicate the occurrence of a circular buffer overflow, bit 17 (DAG1 circular buffer 7 overflow) and bit 18 (DAG2 circular buffer 15 overflow). Rather then remaining set until explicitly cleared, however, these bits are cleared by the next subsequent memory access that uses the corresponding I register (I7, I15). Circular buffer interrupts, therefore, should be used instead of these STKY register bits.

4.3.3 Bit-Reversal

Bit-reversal of data memory addresses can be performed in two ways: by enabling the bit-reverse mode of DAG1 and using a specific I register (I0), or by executing the explicit bit-reverse instruction (BITREV).

4.3.3.1 Bit-Reverse Mode

In bit-reverse mode, DAG1 bit-reverses 32-bit address values output from I0. This mode is enabled by the BR0 bit in the MODE1 register. Only address values from I0 are bit-reversed. This mode affects both pre-modify and post-modify operations.

MODE1

Bit	*Name*	*Definition*
1	BR0	Bit-reverse for I0 (uses $\overline{DMS}_0$ only)

4 Data Addressing

Important: Due to timing constraints, addresses output in bit-reverse mode always activate $\overline{DMS}_0$ (Data Memory Select 0) and the number of wait states associated with it, regardless of the actual address value. In most systems, this means that a bit-reversed address must be within the lowest bank of data memory space. (See Chapter 6 for more information on memory banks.)

Bit-reversal occurs at the output of DAG1 and does not affect the value in I0. In the case of a post-modify operation, the update value is not bit-reversed. However, after a data memory access using I0, you can read the bit-reversed address from universal register DMADR, which holds the last data memory address output.

Example:

```
I0=0x80400000;
R1=DM(I0,3);          DM address = 0x201, I0 = 0x8040 0003
```

4.3.3.2 Bit-Reverse Instruction

The BITREV instruction modifies and bit-reverses addresses in any DAG1 index register (I0-I7) without accessing external data memory. This instruction is independent of the bit-reverse mode (BR0 bit in MODE1). The BITREV instruction adds a 32-bit immediate value to a DAG1 index register, bit-reverses the result and writes the result back to the same index register. The bit-reversed value appears on the data memory address bus, but no strobes are active.

Example:

```
BITREV(I1,4);         I1 = Bit-reverse of (I1 + 4)
```

4.4 DAG REGISTER TRANSFERS

DAG registers are part of the universal register set and may be written from data memory, another universal register or an immediate field in an instruction. Their contents may be written to data memory or a universal register.

Transfers between 32-bit DAG1 registers (7-0) and the 40-bit DMD bus are aligned to bits 39-8 of the DMD bus. When 24-bit DAG2 registers (15-8) are read to the 40-bit DMD bus, M register values are sign-extended to 32 bits, and I, L, and B register values are zero-filled to 32 bits. The results are

aligned to bits 39-8 of the DMD bus. When DAG2 registers are written from the DMD bus, bits 31-8 are transferred and the rest are ignored. Figure 4.5 illustrates these transfers.

4.4.1 DAG Register Transfer Restrictions

For certain instruction sequences involving transfers to and from DAG registers, an extra (NOP) cycle is either automatically inserted by the processor **(1, 2)** or must be inserted in code by the programmer **(3)**. Certain other sequences cause incorrect results and are not allowed by the ADSP-21020/21010 Assembler **(4)**.

1.) When an instruction that loads a DAG register is followed by an instruction that uses any register in the same DAG for data addressing, the ADSP-21020/21010 inserts an extra (NOP) cycle between the two instructions. This happens because the same bus is needed by both operations in the same cycle, therefore the second operation must be delayed. An example is:

```
L2=8;
DM(I0,M1)=R1;
```

Because L2 is in the same DAG as I0 (and M1), an extra cycle is inserted after the write to L2.

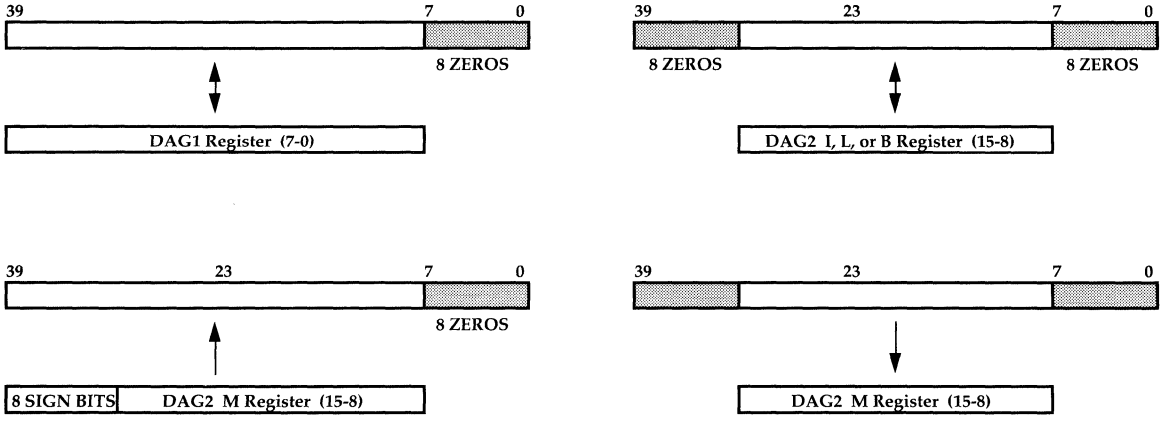


Figure 4.5 DAG Register Transfers

2.) For the same reason, the ADSP-21020/21010 also inserts an extra cycle after an instruction that writes a memory control register if it is followed by an instruction that uses a register in the corresponding DAG (DAG1 for data memory control registers, DAG2 for program memory control registers). Data memory control registers are DMWAIT, DMBANK1-3 and DMADR. Program memory control registers are PMWAIT, PMBANK1 and PMADR. (Note that because the DAG2 registers are used to fetch instructions or access data in every cycle, a write to a program memory control register will *always* require an extra cycle to be inserted.)

Each of the following instruction sequences, for example,

```
PMWAIT=0x080000;        or        DMBANK1=0x10000000;
NOP;                              R15=DM(I0,M1);
```

cause the ADSP-21020 to insert an extra cycle between the two instructions.

3.) An instruction that writes any L or M register of DAG2 (L8-L15, M8-M15), immediately followed by an instruction that reads the corresponding I register will result in incorrect data being read from the I register. The following instruction sequence, for example,

```
L8=24;
R0=I8;
```

will cause incorrect data to be read from I8. To prevent this, add a NOP to your program between the two instructions (i.e. the L or M register write and the I register read):

```
L8=24;
NOP;
R0=I8;
```

4.) The following kinds of instructions can execute on the processor, but cause incorrect results; these instructions are disallowed by the ADSP-21020/21010 Assembler:

- An instruction that stores a DAG register in memory using indirect addressing from the same DAG, with or without update of the index register. The instruction writes the wrong data to memory or updates the wrong index register.

  Examples:

  ```
  DM(M2,I1)=I0;      or      DM(I1,M2)=I0;
  ```

- An instruction that loads a DAG register from memory using indirect addressing from the same DAG, with update of the index register. The instruction will either load the DAG register or update the index register, but not both.

  Example:

  ```
  L2=DM(I1,M0);
  ```

Timer 5

5.1 OVERVIEW

The ADSP-21020/21010 has a programmable interval timer that can generate periodic interrupts. You program the timer by writing two universal registers, and you control timer operation through a bit in the MODE2 register. An external output, TIMEXP, signals to other devices that the timer count has expired.

5.2 TIMER OPERATION

Figure 5.1, on the next page, shows a block diagram of the timer. Two universal registers, TPERIOD and TCOUNT, control the timer interval.

Register	*Function*	*Bits*
TPERIOD	Timer Period Register	32
TCOUNT	Timer Counter Register	32

The TCOUNT register contains the timer counter. The timer decrements the TCOUNT register each clock cycle. When the TCOUNT value reaches zero, the timer generates an interrupt and asserts the TIMEXP output high for 4 cycles (see Figure 5.2, also on the next page). On the next clock cycle after TCOUNT reaches zero, the timer automatically reloads TCOUNT from the TPERIOD register.

The TPERIOD value specifies the frequency of timer interrupts. The number of cycles between interrupts is TPERIOD + 1. The maximum value of TPERIOD is $2^{32} - 1$, so if the clock cycle is 50 ns, the maximum interval between interrupts is 214.75 seconds.

5.2.1 Timer Enable And Disable

To start and stop the timer, you enable and disable it through a bit in the MODE2 register. With the timer disabled, you load TCOUNT with an initial count value and TPERIOD with the number of cycles for the interval you want. Then you enable the timer when you want to begin the count.

5 Timer

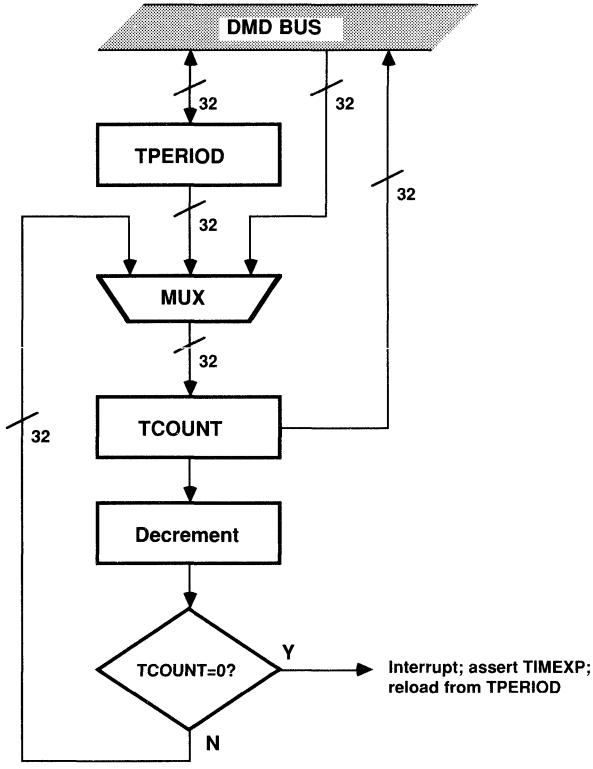


Figure 5.1 Timer Block Diagram

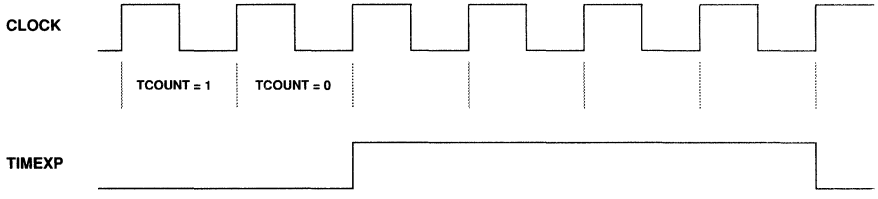


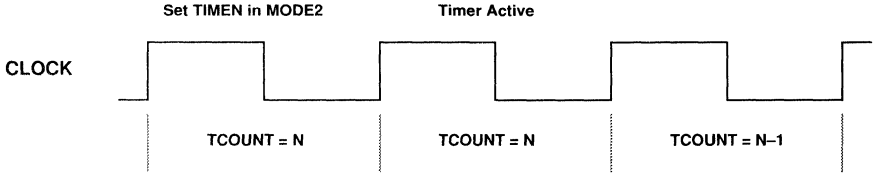
Figure 5.2 TIMEXP Signal

At reset, the timer enable bit in the MODE2 register is cleared, so the timer is disabled. When the timer is disabled, it does not decrement the TCOUNT register and it generates no interrupts. When the timer enable bit is set, the timer starts decrementing the TCOUNT register at the end of the next clock cycle. If the bit is subsequently cleared, the timer is disabled and stops decrementing TCOUNT after the next clock cycle (see Figure 5.3).

MODE2

Bit	*Name*	*Definition*
5	TIMEN	Timer enable

TIMER ENABLE



TIMER DISABLE

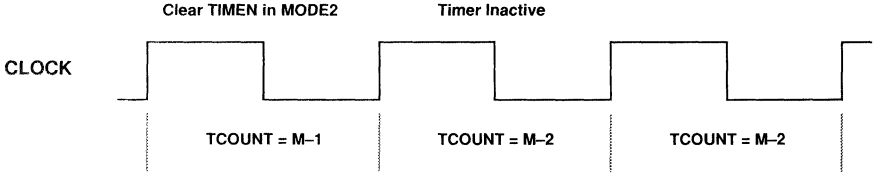


Figure 5.3 Timer Enable and Disable

5 Timer

5.2.2 Timer Interrupt

When the value of TCOUNT reaches zero, the timer generates two interrupts, one with a relatively high priority, the other with a relatively low priority. At reset, both are masked. You should unmask only the timer interrupt that has the priority you want, and leave the other masked.

IRPTL bit	*Name*	*Vector*	*Function*
4	TMZHI	0x20	Timer =0 (high priority option)
14	TMZLI	0x70	Timer=0 (low priority option)

Interrupt priority determines which interrupt is serviced first when two occur in the same cycle. It also affects interrupt nesting; when nesting is enabled, only higher priority interrupts can interrupt a service routine.

Like other interrupts, the timer interrupt requires two cycles to fetch and decode the first service routine instruction. The service routine begins executing four cycles after the timer count goes to zero, as shown in Figure 5.4.

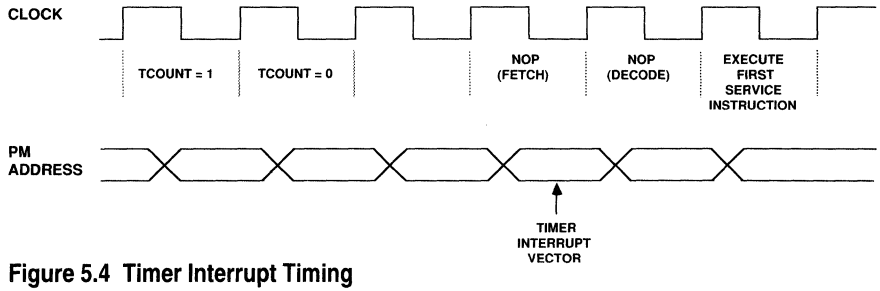


Figure 5.4 Timer Interrupt Timing

5.3 TIMER REGISTERS

Both the TPERIOD and TCOUNT registers can be read and written through universal register transfers. Reading the registers has no effect on the timer function. An explicit write to TCOUNT has priority over both the loading of TCOUNT from TPERIOD and the decrementing of TCOUNT.

Neither TCOUNT nor TPERIOD are affected by a reset, so you should initialize both registers after reset before enabling the timer.

Memory Interface 6

6.1 MEMORY MANAGEMENT AND INTERFACE

This chapter describes the memory management and interface capabilities of the ADSP-21020/21010. In addition, Chapter 9 shows several example systems with different memory configurations.

The ADSP-21020/21010 has two distinct but similar memory interfaces: one for program memory, which contains both instructions and data, and one for data memory, which contains data only. The processor is capable of connecting to a number of different memory devices and memory-mapped peripherals. Minimal external hardware is required for a variety of configurations.

The ADSP-21020/21010 provides on-chip memory management. Program and data memory spaces are user-configurable into banks (two for program memory, four for data memory). Wait states for each bank are independently programmable. The processor also detects page boundaries to facilitate memory paging.

The bus request/bus grant protocol allows an external device to take control of the processor's memory buses. This is useful for transferring data to its memory, for example. The ADSP-21020/21010 also has an internal bus exchange path for transferring data between the program memory and data memory spaces.

6 Memory Interface

6.2 MEMORY BUSES AND CONTROL PINS

The ADSP-21020/21010 accesses program memory through its program memory interface. Two types of accesses occur across the program memory buses: instruction fetches and program memory data accesses. The program memory interface consists of the following pins:

Pin	*Type*	*Definition*
PMA_{23-0}	O	Program Memory Address. The ADSP-21020/21010 outputs an address in program memory on these pins.
PMD_{47-0}	I/O	Program Memory Data. The ADSP-21020/21010 inputs and outputs data and instructions on these pins. 32-bit fixed-point data and 32-bit single-precision floating-point data is transferred over bits 47-16 of the PMD bus.
$\overline{PMS}_{1-0}$	O	Program Memory Select lines 1 & 0. These pins are asserted as chip selects for the corresponding banks of program memory. Memory banks must be defined in the processor's memory control registers. These pins are decoded program memory address lines and provide an early indication of a possible bus cycle.
$\overline{PMRD}$	O	Program Memory Read strobe. This pin is asserted when the ADSP-21020/21010 reads from program memory.
$\overline{PMWR}$	O	Program Memory Write strobe. This pin is asserted when the ADSP-21020/21010 writes to program memory.
PMACK	I	Program Memory Acknowledge. An external device deasserts this input to add wait states to a memory access.
PMPAGE	O	Program Memory Page Boundary. The ADSP-21020/21010 asserts this pin to signal that a program memory page boundary has been crossed. Memory pages must be defined in the processor's memory control registers.
$\overline{PMTS}$	I	Program Memory Three-State Control. $\overline{PMTS}$ places the program memory address, data, selects, and strobes in a high-impedance state. If $\overline{PMTS}$ is asserted while a PM access is in progress, the processor will halt and the memory access will not be completed. PMACK must be asserted for at least one cycle when $\overline{PMTS}$ is deasserted to allow any pending memory access to complete properly. $\overline{PMTS}$ should only be asserted (low) during an active memory access cycle.

O=Output, I=Input. When groups of pins are identified with subscripts, e.g. PMD_{47-0}, the highest numbered pin is the MSB (in this case, PMD_{47}).

Memory Interface 6

The ADSP-21020/21010 accesses data memory through its data memory interface. The data memory interface consists of the following pins:

Pin	*Type*	*Definition*
DMA_{31-0}	O	Data Memory Address. The ADSP-21020/21010 outputs an address in data memory on these pins.
DMD_{39-0}	I/O	Data Memory Data (ADSP-21020). The ADSP-21020 inputs and outputs data on these pins. 32-bit fixed-point data and 32-bit single-precision floating-point data is transferred over bits 39-8 of the DMD bus.
DMD_{31-0}	I/O	Data Memory Data (ADSP-21010). The ADSP-21010 inputs and outputs data on these pins. (DMD_{31-0} on the ADSP-21010 corresponds to DMD_{39-8} on the ADSP-21020. This should be taken into account if upgrading is planned.)
$\overline{DMS}_{3-0}$	O	Data Memory Select lines 0, 1, 2, & 3. These pins are asserted as chip selects for the corresponding banks of data memory. Memory banks must be defined in the processor's memory control registers. These pins are decoded data memory address lines and provide an early indication of a possible bus cycle.
$\overline{DMRD}$	O	Data Memory Read strobe. This pin is asserted when the ADSP-21020/21010 reads from data memory.
$\overline{DMWR}$	O	Data Memory Write strobe. This pin is asserted when the ADSP-21020/21010 writes to data memory.
DMACK	I	Data Memory Acknowledge. An external device deasserts this input to add wait states to a memory access.
DMPAGE	O	Data Memory Page Boundary. The ADSP-21020/21010 asserts this pin to signal that a data memory page boundary has been crossed. Memory pages must be defined in the processor's memory control registers.
$\overline{DMTS}$	I	Data Memory Three-State Control. $\overline{DMTS}$ places the data memory address, data, selects, and strobes in a high-impedance state. If $\overline{DMTS}$ is asserted while a DM access is in progress, the processor will halt and the memory access will not be completed. DMACK must be asserted for at least one cycle when $\overline{DMTS}$ is deasserted to allow any pending memory access to complete properly. $\overline{DMTS}$ should only be asserted (low) during an active memory access cycle.

O=Output, I=Input. When groups of pins are identified with subscripts, e.g. DMD_{39-0}, the highest numbered pin is the MSB (in this case, DMD_{39}).

6 Memory Interface

6.3 MEMORY INTERFACE TIMING

This section describes the relative timing of memory interface signals during memory accesses. The descriptions apply to both program memory and data memory accesses. The following generic signal names represent the memory control signals; the signals actually used in a particular access depend on whether program memory or data memory is being accessed and which bank contains the address.

Signal Name	*Program Memory*	*Data Memory*
Address	PMA_{23-0}	DMA_{31-0}
Data	PMD_{47-0}	DMD_{39-0}
Memory select	$\overline{PMS}_{1-0}$	$\overline{DMS}_{3-0}$
Read strobe	$\overline{PMRD}$	$\overline{DMRD}$
Write strobe	$\overline{PMWR}$	$\overline{DMWR}$
Acknowledge	PMACK	DMACK

6.3.1 Memory Read

Memory reads occur with the following sequence of events (see Figure 6.1):

1. The ADSP-21020 drives the read address and asserts a memory select signal to indicate the selected bank. A memory select signal is not deasserted between successive accesses of the same memory bank.

2. The ADSP-21020 asserts the read strobe (unless the memory access is aborted because of a conditional instruction).

3. The ADSP-21020 checks whether wait states are needed. If so, the memory select and read strobe remain active for additional cycle(s). Wait states are determined by the state of the external acknowledge signal, the internally programmed wait state count, or a combination of the two (see "Wait States," later in this chapter).

4. The ADSP-21020 latches in the data.

5. The ADSP-21020 deasserts the read strobe.

6. If initiating another memory access, the ADSP-21020 drives the address and memory select for the next cycle.

Note that if a memory read is part of a conditional instruction that is not executed because the condition is false, the ADSP-21020 still drives the address and memory select for the read, but does not assert the read strobe or read any data.

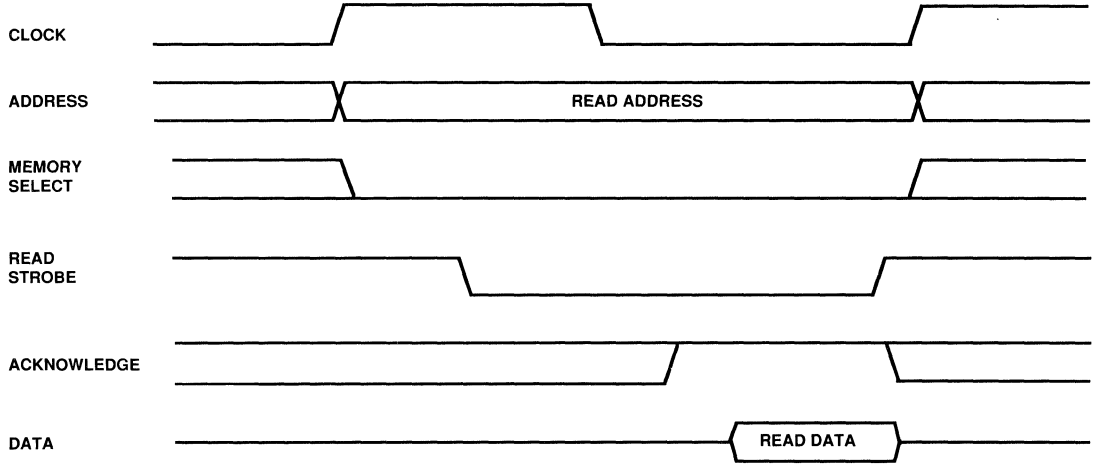


Figure 6.1 Memory Read Cycle

6.3.2 Memory Write

Memory writes occur with the following sequence of events (see Figure 6.2, on the following page):

1. The ADSP-21020 drives the write address and asserts a memory select signal to indicate the selected bank. A memory select signal is not deasserted between successive accesses of the same memory bank.

2. The ADSP-21020 asserts the write strobe and drives the data (unless the memory access is aborted because of a conditional instruction).

3. The ADSP-21020 checks whether wait states are needed. If so, the memory select and read strobe remain active for additional cycle(s). Wait states are determined by the state of the external acknowledge signal, the internally programmed wait state count, or a combination of the two (see "Wait States," later in this chapter).

4. The ADSP-21020 deasserts the write strobe near the end of the cycle.

5. The ADSP-21020 tristates its data outputs.

6. If initiating another memory access, the ADSP-21020 drives the address and memory select for the next cycle.

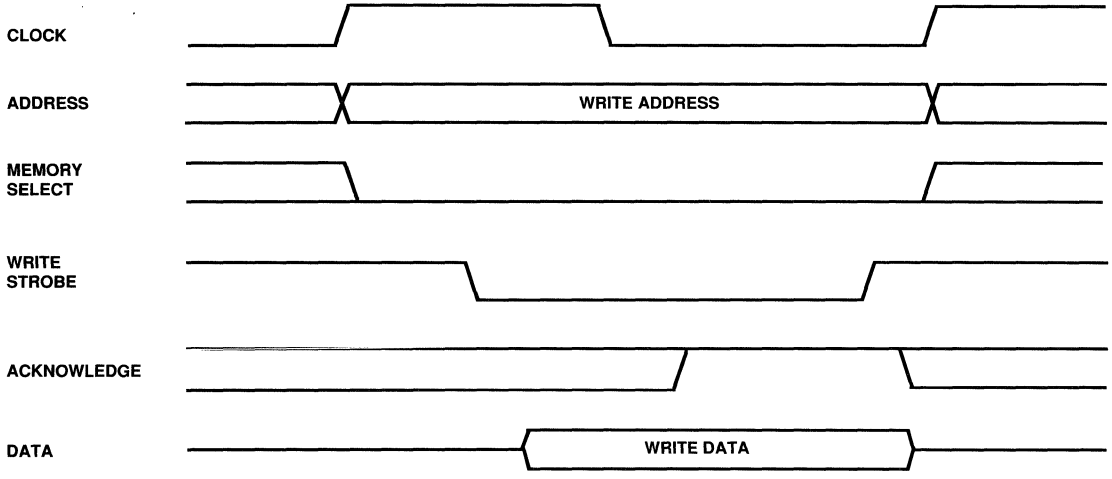


Figure 6.2 Memory Write Cycle

Note that if a memory write is part of a conditional instruction that is not executed because the condition is false, the ADSP-21020 still drives the address and memory select for the write, but does not assert the write strobe or drive any data.

6.3.3 Three-State Controls

The memory bus three-state enables, $\overline{\text{DMTS}}$ and $\overline{\text{PMTS}}$, prevent the ADSP-21020 from driving its external data memory port and program memory port, respectively. The corresponding acknowledge signal ($\overline{\text{DMACK}}$ or $\overline{\text{PMACK}}$) is not sampled when a three-state enable is active. $\overline{\text{DMTS}}$ and $\overline{\text{PMTS}}$ allow an external device, such as a cache controller, to take control of the memory interface by first asserting a three-state enable to keep the ADSP-21020 from driving the bus. When the external device deasserts the three-state enable, the processor resumes driving its memory port if it was executing a memory access.

These controls facilitate the implementation of an external cache system. When the the processor tries to access data that is not in the cache, the controller asserts the three-state control, writes the needed data into cache, then releases the three-state control so that the ADSP-21020 can finish the data access. Unlike with bus request ($\overline{\text{BR}}$), the ADSP-21020 does not finish its current instruction before it three-states the memory port.

You must use memory acknowledge (DMACK or PMACK) in conjunction with $\overline{\text{DMTS}}$ or $\overline{\text{PMTS}}$, whether or not the memory requires wait states. The first reason is that there must be an extra cycle after the three-state enable is deasserted for the memory cycle to complete; the acknowledge should be deasserted (low) in the same cycle that the three-state enable is deasserted in. The second reason is that the ADSP-21020 counts wait states whether or not the memory outputs are enabled, so internally programmed wait states cannot be used. If memory requires wait states, use DMACK or PMACK to insert them after the extra cycle that follows the deassertion of the three-state enable.

$\overline{\text{DMTS}}$ and $\overline{\text{PMTS}}$ do not halt the ADSP-21020 if the processor does not require a memory access from the three-stated port. In practice, $\overline{\text{PMTS}}$ will halt the ADSP-21020 because either an instruction fetch or program memory data access needs to occur every cycle. When only $\overline{\text{DMTS}}$ is asserted, however, the processor can continue running until it reaches a data memory access.

$\overline{\text{DMTS}}$ controls the following pins:

DMA_{31-0}
DMD_{39-0} (DMD_{31-0} on the ADSP-21010)
$\overline{\text{DMS0}}$
$\overline{\text{DMS1}}$
$\overline{\text{DMS2}}$
$\overline{\text{DMS3}}$
$\overline{\text{DMRD}}$
$\overline{\text{DMWR}}$
DMPAGE

$\overline{\text{PMTS}}$ controls the following pins:

PMA_{23-0}
PMD_{47-0}
$\overline{\text{PMS0}}$
$\overline{\text{PMS1}}$
$\overline{\text{PMRD}}$
$\overline{\text{PMWR}}$
PMPAGE

6 Memory Interface

6.4 MEMORY BANKS

Each address space on the ADSP-21020 can be divided into banks for selection. The program memory address space is divided into two banks. The data memory address space is divided into four banks. The relative size of these banks is under user control through the registers PMBANK1, DMBANK1, DMBANK2 and DMBANK3.

Bank 0 of program memory spans address 0 up to but not including the value in the 24-bit PMBANK1 register. Program memory bank 1 starts at the value in PMBANK1 and runs to the end of program memory space. Each bank has a separate memory select pin ($\overline{PMS}_0$ and $\overline{PMS}_1$) that is asserted when the ADSP-21020 outputs an address in the corresponding memory bank. Wait states for the two banks are independently controlled, as described under "Extending Memory Cycles with Wait States."

Similarly, bank 0 of data memory spans address 0 up to but not including the value in the 32-bit DMBANK1 register. Bank 1 runs from the value in DMBANK1 up to the value in DMBANK2, bank 2 from DMBANK2 up to DMBANK3, and bank 3 from DMBANK3 to the end of data memory space. For proper operation, the address in DMBANK2 should be greater than or equal to that in DMBANK1, and the address in DMBANK3 should be greater than or equal to that in DMBANK2. As in program memory, each data memory bank has its own memory select pin and independent wait state control.

Note: When bit-reverse mode is enabled (bit 1 in MODE1 is set), data memory accesses that use I0 will activate $\overline{DMS}_0$ and insert the number of wait states programmed for bank 0, **regardless of the value of the bit-reversed address**. In most systems, this means that bit-reversed mode can only be used to access bank 0.

If a memory access is aborted (because of a non-delayed branch, for example), the memory select signal may be asserted even though there is no memory access.

At reset, the memory bank address registers contain values as follows:

Register	*Value at Reset*
PMBANK1	0x800000
DMBANK1	0x20000000
DMBANK2	0x40000000
DMBANK3	0x80000000

6.5 WAIT STATES (EXTENDED MEMORY CYCLES)

To simplify the interface to slow off-chip peripherals and slow memories, the ADSP-21020 allows a variety of methods for extending off-chip memory accesses.

- *External.* The ADSP-21020 samples its acknowledge input (DMACK or PMACK) in each clock cycle. If it latches a low value, it inserts a wait state by holding strobes and address on the interface valid an additional cycle. If the value is high, the ADSP-21020 completes the cycle.

- *Internal.* The ADSP-21020 ignores the acknowledge input. Three bits in a control register specify the number of wait states (zero to seven) for the access. You can specify a different number for each bank of memory.

- *Both.* The ADSP-21020 samples its acknowledge input in each clock cycle. If it latches a low value, the ADSP-21020 inserts a wait state. If the value is high, the ADSP-21020 completes the cycle only if the number of wait states specified internally have expired. In this mode, the internal wait states specify a minimum number of cycles per access, and an external device can use the acknowledge pin to extend the access as necessary.

- *Either.* The ADSP-21020 completes the cycle as soon as it samples the acknowledge input high or the internally specified number of wait states have expired, whichever occurs first. In this mode, a system with two types of peripherals could shorten the cycle for the faster peripheral using the acknowledge but use the internal wait states for the slower peripheral.

The method selected for each bank of memory is independent of the other banks. Thus, you can map different speed devices into different memory banks for the appropriate wait state control.

Two bits specify the wait state method and three bits specify the number of wait states (zero to seven) for each bank of each memory space. These control bits are located in the DMWAIT and PMWAIT registers, shown in Figure 6.3. The mode bits are decoded as follows:

Mode	*Description*
0 0	External acknowledge only
0 1	Internal wait states only
1 0	Both internal and external acknowledge
1 1	Either internal or external acknowledge

Wait state control for the program memory interface is determined by the following bits in the PMWAIT register:

PMWAIT

Bits	*Function*
9-7	Number (in binary) of program memory bank 1 wait states
6-5	Wait state mode for program memory bank 1
4-2	Number (in binary) of program memory bank 0 wait states
1-0	Wait state mode for program memory bank 0

Wait state control for the data memory interface is determined by the following bits in the DMWAIT register:

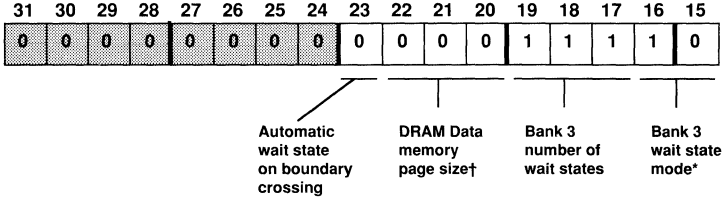
DMWAIT

Bits	*Function*
19-17	Number (in binary) of data memory bank 3 wait states
16-15	Wait state mode for data memory bank 3
14-12	Number (in binary) of data memory bank 2 wait states
11-10	Wait state mode for data memory bank 2
9-7	Number (in binary) of data memory bank 1 wait states
6-5	Wait state mode for data memory bank 1
4-2	Number (in binary) of data memory bank 0 wait states
1-0	Wait state mode for data memory bank 0

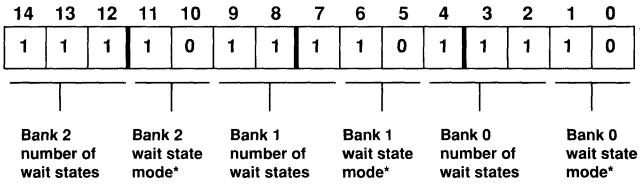
6.5.1 Extended Data Memory Address Hold Time

The ADSP-21020 holds its data memory address outputs from one cycle to the next until the next data memory access or BITREV instruction causes the address to change. This feature simplifies the interface to peripherals requiring long address hold times. The programmer ensures that the next data memory access or BITREV instruction occurs after the address hold requirement has been met. For example, inserting a NOP after a data memory access instruction guarantees that the address will be held for two cycles.

DMWAIT Register

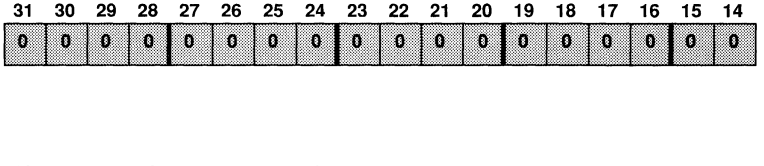


† DRAM Memory Page Size Codes	
000	256 Words
001	512 Words
010	1024 Words
011	2048 Words
100	4096 Words
101	8192 Words
110	16384 Words
111	32768 Words



* Wait State Mode Codes	
00	External acknowledge only
01	Internal wait states only
10	Both external and internal required
11	Either external or internal sufficient

PMWAIT Register



† DRAM Memory Page Size Codes	
000	256 Words
001	512 Words
010	1024 Words
011	2048 Words
100	4096 Words
101	8192 Words
110	16384 Words
111	32768 Words

* Wait State Mode Codes	
00	External acknowledge only
01	Internal wait states only
10	Both external and internal required
11	Either external or internal sufficient

Figure 6.3 Wait State Control Registers

6 Memory Interface

6.6 MEMORY PAGE BOUNDARY DETECTION

Applications that use large amounts of data may need to use dynamic RAMs (DRAMs) for bulk storage. To simplify its interface to page-mode DRAMs, the ADSP-21020 detects page boundary crossings and outputs a signal to a DRAM controller. Page boundaries are user-defined and are determined by page size fields in the wait state control registers.

The ADSP-21020 detects boundary crossings by comparing each address it outputs to the one just previously output (in the same memory space). For instance, if a page is 1024 words long, any of the lower ten address bits can change from one address to the next without addressing a different page. If the upper 22 (for data memory) or 14 (for program memory) address bits are the same, the addresses are in the same memory page. If any of the upper bits change, however, the ADSP-21020 signals a page boundary crossing by asserting the PMPAGE pin (for program memory) or the DMPAGE pin (for data memory).

If the memory access that crosses a page boundary is aborted (because of a non-delayed branch, for example), the PAGE pin may be asserted even though there is no memory access. In this case, the ADSP-21020 retains the address from the previous access to use for comparison; that is, the ADSP-21020 recognizes that the boundary was not actually crossed.

The PAGE pin is asserted only on the first access to a page. It is always asserted on the first memory access after a reset. The PAGE pin has the same timing as the address pins.

It is important to remember that memory pages and memory banks are handled independently by the ADSP-21020. Page boundary checking works the same way whether the same bank or a different bank is being accessed.

6.6.1 Page Size

Three bits in PMWAIT and three in DMWAIT specify the memory page size in program memory and data memory, respectively:

PMWAIT

Bits	*Function*
12-10	Program memory page size

DMWAIT

Bits	*Function*
22-20	Data memory page size

The page boundary detection logic interprets these bits according to the following table:

Bit Values	*Page Size (Words)*
000	256
001	512
010	1024
011	2048
100	4096
101	8192
110	16384
111	32768

6.6.2 Wait States On Page Boundary Crossings

One bit each in the DMWAIT and PMWAIT registers controls automatic wait state generation for page boundary crossings. If this bit is a 1, the ADSP-21020 inserts one wait state if the access crosses a page boundary *and* if the access would not otherwise include a wait state. If the access already includes at least one wait state, it is not affected.

PMWAIT

Bit	*Function*
13	1=automatic wait state for access across page boundary; 0=no automatic wait state

DMWAIT

Bit	*Function*
23	1=automatic wait state for access across page boundary; 0=no automatic wait state

At reset, this bit is a 0 in both PMWAIT and DMWAIT, disabling automatic wait states. The ADSP-21020 always performs page boundary detection, whether or not the automatic wait states are enabled.

6.7 BUS REQUEST/BUS GRANT

The bus request ($\overline{BR}$) and bus grant ($\overline{BG}$) signals on the ADSP-21020 allow an external processor to gain control of the program memory and data memory buses in order to, for example, transfer data in or out of the ADSP-21020's external memory.

6 Memory Interface

$\overline{BR}$ is an input pin that requests access to the buses. When $\overline{BR}$ is asserted, the ADSP-21020 completes the current instruction and then places both data buses, both address buses, read and write strobes, PMPAGE, DMPAGE and all memory select pins in a high-impedance state. The ADSP-21020 then asserts $\overline{BG}$ to indicate to the requesting device that it is no longer driving its memory buses. The ADSP-21020 remains halted until $\overline{BR}$ is deasserted, signalling the release of the buses. The ADSP-21020 then continues from the instruction at which it halted.

The bus grant operation is shown in Figure 6.4. Detailed timing requirements and characteristics are given in the *ADSP-21020 Data Sheet*. If $\overline{BR}$ is asserted and then deasserted (in the next cycle) before $\overline{BG}$ is asserted, the bus grant may or may not occur. For proper operation, $\overline{BR}$ should be held at least until $\overline{BG}$ goes low.

Interrupts are sampled while the bus is granted, but remain pending until the bus request is released. When the processor continues program execution, pending interrupts are serviced in order of priority.

While the ADSP-21020 is in reset, a bus request is recognized immediately because no instructions are being executed. The bus is granted just as in normal operation. The clock must be active for bus request to be recognized.

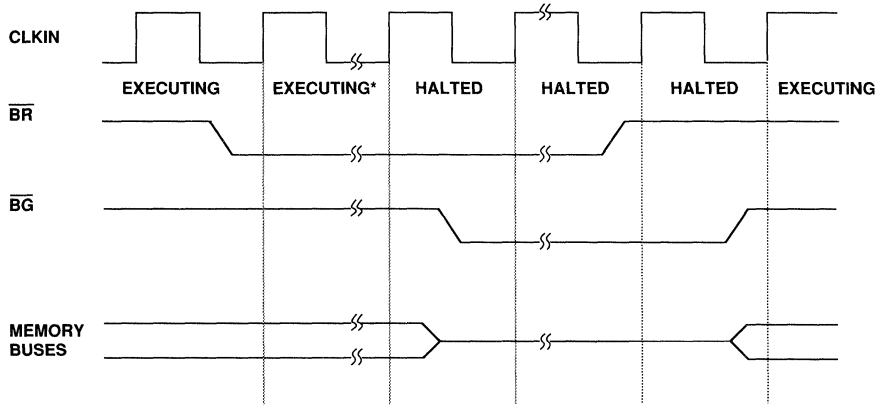


Figure 6.4 Bus Request/Bus Grant Timing

* The ADSP-21020 will complete execution of this instruction before granting its buses. Most instructions require only a single cycle to complete, but additional cycles may be needed for memory waitstates, DMACK/PMACK, multicycle instructions such as delayed branches (DB), etc.

6.8 BUS EXCHANGE (PX REGISTERS)

PX1 and PX2 are two registers used for transferring data between the 48-bit PMD bus and 40-bit register file locations or the 40-bit DMD bus. PX1 is 16 bits wide, and PX2 is 32 bits wide. Either register can be read from or written to the PMD bus, the DMD bus or the register file.

Data is aligned in PX register transfers as shown in Figure 6.5. When data is transferred between PX2 and the PMD bus, the upper 32 bits of the PMD bus are used. On transfers from PX2, the 16 LSBs of the PMD bus are filled with zeros. When data is transferred between PX1 and the PMD bus, the middle 16 bits of the PMD bus are used. On transfers from PX1, bits 15-0 and bits 47-32 are filled with zeros.

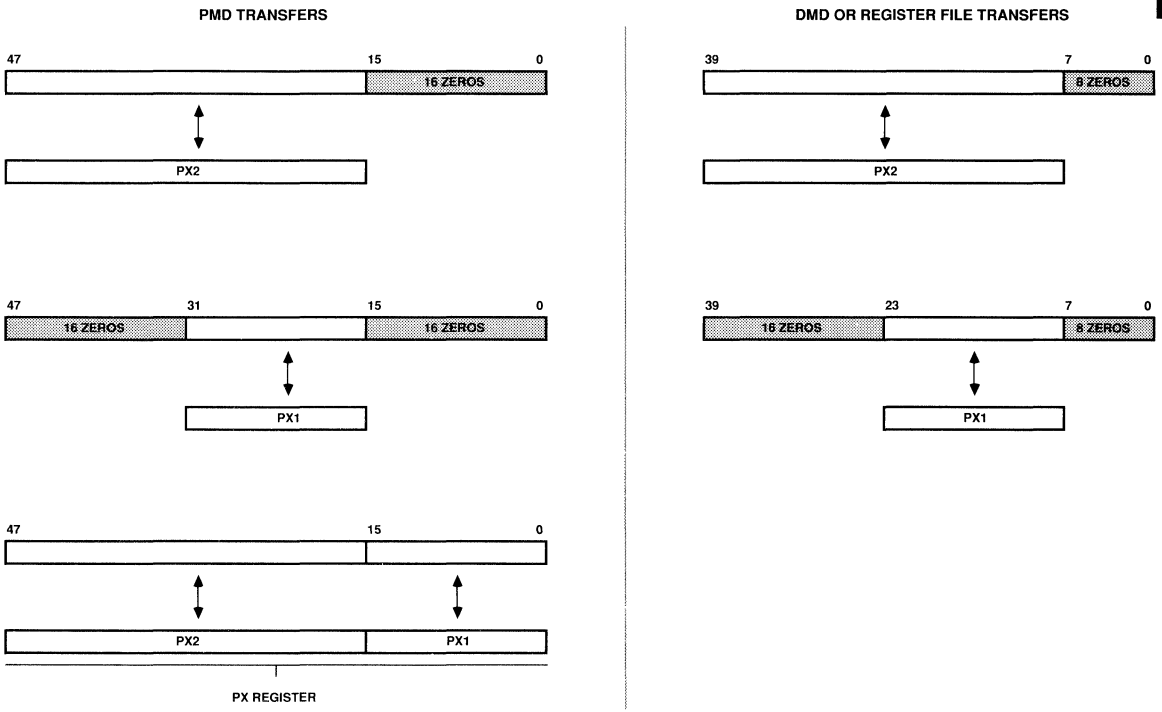


Figure 6.5 PX Register Transfers

6 Memory Interface

When data is transferred between PX2 and the DMD bus or the register file, the upper 32 bits of the DMD bus or the register file are used. On transfers from PX2, the eight LSBs are filled with zeros. When data is transferred between PX1 and the DMD bus or the register file, bits 23-8 of the DMD bus or the register file are used. On transfers from PX1, bits 7-0 and bits 39-24 are filled with zeros.

PX1 and PX2 can also be treated as a single PX register, but only for reads from and writes to program memory via the PMD bus. This allows the PX pair to contain the entire 48 bits coming from or going to program memory. PX2 contains the 32 MSBs of the 48-bit word while PX1 contains the 16 LSBs. (Program memory data is 40 bits wide and left-justified in the 48-bit word.)

To write a 48-bit word to the program memory location named *Port1*, for example, the following instructions would be used:

```
R0=0x9A00;        /* load R0 with 16 LSBs */
R1=0x12345678;    /* load R1 with 32 MSBs */
PX1=R0;
PX2=R1;
PM(Port1)=PX;     /* write 16 LSBs to PM bits 15-0 */
                  /*  and 32 MSBs to PM bits 47-16 */
```

Instruction Summary 7

7.1 OVERVIEW

This section describes the ADSP-21000 Family instruction set in brief. For more information, see Appendix A, Instruction Set Reference.

The instructions are grouped into four categories:

I. Compute and Move or Modify
II. Program Flow Control
III. Immediate Move
IV. Miscellaneous

The instructions are numbered; there are 22. Some instructions have more than one syntactical form; for example, Instruction 4 has four distinct forms. The instruction number has no bearing on programming, but corresponds to the opcode recognized by the ADSP-21020/21010 device.

This section also contains several reference tables for using the instruction set.

- Table 7.1 describes the notation and abbreviations used in this section.
- Table 7.2 lists all condition and termination code mnemonics.
- Table 7.3 lists all register mnemonics.
- Tables 7.4 through 7.7 list the syntax for all compute operations (ALU, multiplier, shifter or multifunction).
- Table 7.8 lists interrupts and their vectors.

7 Instruction Summary

7.2 IMPORTANT PROGRAMMING REMINDERS

This section summarizes information about the operation of the ADSP-21020/21010 that you should keep in mind when writing programs. Use it as a checklist for verifying that your program will execute as you intend.

7.2.1 Extra Cycle Conditions

All instructions can execute in a single clock period but may take longer in some cases. These cases are described in the following sections.

7.2.1.1 Nondelayed Branches

A nondelayed branch instruction (JUMP, CALL, RTS or RTI) fetches but does not execute the two instructions that follow it in program memory. Instead, these operations are aborted and the processor executes two NOPs.

This two-cycle delay can be avoided by using delayed branches, which execute the two instructions following the branch instruction. The tradeoff is that the actual program flow does not match the apparent order of operations in the program; you must remember that the two extra instructions are executed before the branch is taken.

7.2.1.2 Program Memory Data Access With Cache Miss

The ADSP-21020 checks the instruction cache on every program memory data access. If the instruction needed is in the cache, the instruction fetch from the cache happens in parallel with the program memory data access and the instruction executes in a single cycle. However, if the instruction is not in the cache, the ADSP-21020 must wait for the program memory data access to complete before it can fetch the next instruction. This results in a minimum one-cycle delay, more if the program memory data access uses wait states.

7.2.1.3 Program Memory Data Access In Loops

The ADSP-21020 caches an instruction that it needs to fetch during the execution of a program memory data access. Because of the execution pipeline, this instruction is usually two memory locations after the program memory data access. If the program memory data access is in a loop, there will usually be a cache miss on the first iteration of the loop and cache hits on subsequent iterations, for a total of one extra cycle during the loop execution.

However, there are certain cases in which different instructions are needed from the cache at different iterations. In these cases the number of cache misses, and therefore extra cycles, increases. These situations are summarized below. Note that this table is based on the worst-case scenario; the actual performance of the cache for a given program may be better.

Cache Misses	*Loop Length (Instructions)*	*Location of Program Memory Data Access*
1	> 2	Not at e or $(e-1)$
2	≥ 2	At e or $(e-1)$
3	1	At the single loop location

e = loop end address

Two Misses: If the program memory data access occurs in the last two instructions of a loop, there will usually be cache misses on the first and the last loop iteration, for a total of two extra cycles. On the first iteration, the ADSP-21020 needs to fetch from the top of the loop (the first or second instruction). On the last iteration, the ADSP-21020 needs to fetch one of the two instructions following the loop. At each of these points there will be a cache miss the first time the code containing the loop is executed.

Three Misses: If a loop contains only one instruction, and that instruction requires a program memory data access, there are potentially three cache misses. On the first iteration, the processor needs to fetch the loop instruction again (if the loop iterates three times or more). On the next-to-last iteration, the processor needs to fetch the instruction following the loop. On the last instruction, the processor needs to fetch the second instruction following the loop. In each case, there will be a cache miss the first time the code containing the loop is executed.

7.2.1.4 One- And Two-Instruction Loops

Counter-based loops that have only one or two instructions can cause delays if not executed a minimum number of times. The ADSP-21020 checks the termination condition two cycles before it exits the loop. In these short loops, the ADSP-21020 has already looped back when the termination condition is tested. Thus, if the termination condition tests true, the two instructions in the pipeline must be aborted and NOPs executed instead.

Specifically, a loop of length one executed one or two times or a loop of length two executed only once incurs two cycles of overhead because there are two aborted instructions after the last iteration. Note that these

overhead cycles are in addition to any extra cycles caused by a program memory data access inside the loop (see previous section). To avoid overhead, use straight-line code instead of loops in these cases.

7.2.1.5 DAG And Memory Control Register Writes

When an instruction that loads a DAG register is followed by an instruction that uses any register in the same DAG for data addressing, the ADSP-21020 inserts an extra (NOP) cycle between the two instructions. This happens because the same bus is needed by both operations in the same cycle, therefore the second operation must be delayed. An example is:

```
L2=8;
DM(I0,M1)=R1;
```

For the same reason, the ADSP-21020 also inserts an extra cycle after an instruction that writes a memory control register if it is followed by an instruction that uses a register in the corresponding DAG (DAG1 for data memory control registers, DAG2 for program memory control registers). Data memory control registers are DMWAIT, DMBANK1-3 and DMADR. Program memory control registers are PMWAIT, PMBANK1 and PMADR. (Note that because the DAG2 registers are used to fetch instructions or access data in every cycle, a write to a program memory control register will *always* require an extra cycle to be inserted.)

Each of the following instruction sequences, for example, cause the ADSP-21020 to insert an extra cycle between the two instructions:

```
PMWAIT=0x080000;          or          DMBANK1=0x10000000;
NOP;                                  R15=DM(I0,M1);
```

An instruction that writes any L or M register of DAG2 (L8-L15, M8-M15), immediately followed by an instruction that reads the corresponding I register will result in incorrect data being read from the I register. The following instruction sequence, for example, will cause incorrect data to be read from I8:

```
L8=24;
R0=I8;
```

To prevent this, add a NOP between the two instructions:

```
L8=24;
NOP;
R0=I8;
```

7.2.1.6 Wait States

A memory access can be programmed to include a specific number of wait states and/or to wait for an external acknowledge signal before completing. If only internally programmed wait states are used, the delay is the number of wait states (1 wait state = 1 cycle). If the external acknowledge is used, either alone or in combination with programmed wait states, the delay depends on the external system and can vary.

7.2.1.7 Page Boundary Crossing

The page detection logic in the ADSP-21020 can be configured to add a wait state to any memory access that crosses a page boundary. This feature facilitates the external memory page control, which may require extra time on a change of page. If it does not, or if paging is not implemented, the extra wait state does not need to be configured.

7.2.1.8 Three-State Enables

Both the program memory port and the data memory port include a three-state enable input that an external device can assert to hold the ADSP-21020 off the particular memory bus. See Chapter 6 for complete information on these controls. The ADSP-21020 continues to execute instructions while the three-state enable is active until it requires access to the memory bus. At that point, the ADSP-21020 must wait, executing NOPs, until the three-state enable is deasserted. The delay depends on the external system and can vary.

If the ADSP-21020 is accessing the memory when the corresponding three-state enable is asserted, it holds off completion of the memory cycle until the three-state enable is deasserted. DMACK or PMACK must be used to insert an extra cycle to complete the memory access.

7.2.1.9 Bus Request/Bus Grant

As with the three-state enables, an external device can assert the ADSP-21020 bus request ($\overline{BR}$) to gain control of the memory buses (in this case, both buses at once). The ADSP-21020 responds to a bus request by completing the current instruction, placing both memory ports in a high-impedance state, and asserting its bus grant ($\overline{BG}$) output. It executes NOPs until the bus request is deasserted. As with the three-state enables, the delay depends on the external system and can vary.

7.2.2 Delayed Branch Restrictions

A delayed branch instruction and the two instructions that follow it in program memory must be executed sequentially. Any interrupt that occurs in between a delayed branch instruction and either of the two instructions that follow is not processed until the branch is complete.

Instructions in the two program memory locations immediately following a delayed branch instruction can not be any of the following:

- Other Jumps, Calls or Returns
- Pushes or Pops of the PC stack
- Writes to the PC stack or PC stack pointer
- DO UNTIL instruction
- IDLE instruction

These exceptions are checked by the assembler.

7.2.3 Loop Restrictions

If any of the final three instructions of a loop are a jump without loop abort, a call or a return, the loop may not be executed correctly. If an interrupt occurs during the execution of the last three instructions of a loop, its processing is delayed until after the last instruction is executed.

The third-to-last instruction of a counter-based loop cannot be a write to CURLCNTR from external memory.

A non-counter-based loop three instructions long completes one full iteration after the termination condition becomes true. If the loop has two instructions, one or two full iterations occur after the condition becomes true. If the loop has only one instruction, three more passes are executed after the termination condition becomes true.

For no overhead, a counter-based loop of length one must be executed at least three times and a counter-based loop of length two must be executed at least twice. Loops of length one that iterate only once or twice and loops of length two that iterate only once incur two cycles of overhead because there are two aborted instructions after the last iteration.

Nested loops cannot terminate on the same instruction. For nested loops in which the outer loop's termination condition is not LCE, the end address of the outer loop must be at least two locations after the end address of the inner loop.

7.2.4 Interrupts

ADSP-21020 operations that span more than one cycle are not allowed to be interrupted. If an interrupt occurs during one of these operations, it is synchronized and latched, but its recognition is delayed:

- a branch (call, jump or return) and the following cycle, whether it is an instruction (in a delayed branch) or no-operation (in a non-delayed branch)

- the first of the two cycles needed to perform a program memory data access and an instruction fetch (when there is an instruction cache miss).
- the third-to-last iteration of a one-instruction loop
- the last iteration of a one-instruction loop executed twice and the following cycle (which is a no-operation)
- the last iteration of a two-instruction loop executed only once and the following cycle (which is a no-operation)
- the first of the two cycles needed to fetch and decode the first instruction of an interrupt routine

For most interrupts, internal and external, only one instruction is executed after the interrupt occurs and before the two instructions aborted while the processor fetches and decodes the first service routine instruction. Because of the one-cycle delay between an arithmetic exception and the STKY register update, however, there are two cycles after an arithmetic exception occurs before interrupt processing starts.

7.2.5 IRPTL

IRPTL is in an indeterminate state at reset. You should clear IRPTL by writing zeros to it before enabling interrupts or unmasking any interrupt.

7.2.6 Effect Latency And Read Latency

Writes to some registers require an extra cycle before taking effect. This delay is called effect latency. Some registers require an extra cycle after a write before a read of the register yields the new value. This delay is called read latency. Effect latency and read latency for registers are listed below:

Register Name	*Read Latency*	*Effect Latency*
PCSTK	0	0
PCSTKP	1	1
LADDR	0	0
CURLCNTR	0	0
LCNTR	0	0
MODE1	0	1
MODE2	0	1
IRPTL	0	0
IMASK	0	1
IMASKP	1	1
ASTAT	0	1
STKY	0	1
USTAT1	0	0
USTAT2	0	0

7.2.7 CURLCNTR Write & LCE

If an LCE instruction follows a write to CURLCNTR, the condition tested will be based on the old CURLCNTR value. This is because the write of CURLCNTR has a delay of one cycle.

7.2.8 Circular Buffer Initialization

You set up a circular buffer by initializing an L register with a positive, nonzero value and loading the corresponding (same-numbered) B register with the base (lowest) address of the buffer. The corresponding I register is automatically loaded with this same starting address.

7.2.9 Bit-Reverse Mode And Data Memory Bank Select

Due to timing constraints, addresses output in bit-reverse mode always activate $\overline{DMS}_0$ (Data Memory Select 0) and the number of wait states associated with it, regardless of the actual address value. In most systems, this means that a bit-reversed address must be within the lowest bank of data memory space.

7.2.10 Disallowed DAG Register Transfers

The following instructions execute on the ADSP-21020, but cause incorrect results. These instructions are disallowed by the assembler:

- An instruction that stores a DAG register in memory using indirect addressing from the same DAG, with or without update of the index register. The instruction writes the wrong data to memory or updates the wrong index register.

```
DM(M2, I1) = I0;   or    DM(I1, M2) = I0;
```

- An instruction that loads a DAG register from memory using indirect addressing from the same DAG, with update of the index register. The instruction will either load the DAG register or update the index register, but not both.

```
L2 = DM(I1, M0);
```

7.2.11 Two Writes To Register File

If two writes to the same register file location take place in the same cycle, only the write with higher precedence actually occurs. Precedence is determined by the source of the data being written; from highest to lowest, the precedence is:

- Data memory or universal register
- Program memory
- ALU
- Multiplier
- Shifter

7.2.12 Stack Status Flags

The stack overflow/full and underflow flags in the STKY register are not "sticky." Writes to the STKY register have no effect on these bits.

7.2.13 Wait States And Three-State Enables

$\overline{\text{DMTS}}$ and $\overline{\text{PMTS}}$ should not be used in conjunction with internally programmed wait states alone. The ADSP-21020 counts wait states whether or not the memory outputs are enabled, so three-stating during a memory access could cause the access to have too few wait states. You should use memory acknowledge ($\overline{\text{DMACK}}$ or $\overline{\text{PMACK}}$) to implement wait states, if needed. The acknowledge can be conditioned by the three-state enable so that the number of wait states depends on how long $\overline{\text{DMTS}}$ or $\overline{\text{PMTS}}$ is asserted. $\overline{\text{DMACK}}$ and $\overline{\text{PMACK}}$ are not sampled when the corresponding three-state enable is active.

7.2.14 Computation Units

In fixed-point to floating-point conversion, the rounding boundary is always 40 bits even if the RND32 bit is set.

The ALU Zero flag (AZ) signifies floating-point underflow as well as a zero result.

Transfers between MR registers and the register file are considered multiplier operations.

7 Instruction Summary

Notation	*Meaning*
UPPERCASE	explicit syntax; assembler keyword
;	instruction terminator
,	separates parallel operations in an instruction
italics	optional part of instruction
| between lines |	list of options (choose one)
<data*n*>	*n*-bit immediate data value
<addr*n*>	*n*-bit immediate address value
<reladdr*n*>	*n*-bit immediate PC-relative address value
<bit6>:<len6>	6-bit immediate bit position and length values (for shifter immediate operations)
compute	ALU, multiplier, shifter or multifunction operation (from Tables 7.4-7.7)
shiftimm	shifter immediate operation (from Table 7.6)
condition	status condition (from Table 7.2)
termination	termination condition (from Table7. 2)
ureg	universal register (from Table 7.3)
sreg	system register (from Table 7.3)
dreg	R15-R0, F15-F0; register file location
Rn, Rx, Ry, Ra, Rm, Rs	R15-R0; register file location, fixed-point
Fn, Fx, Fy, Fa, Fm, Fs	F15-F0; register file location, floating-point
R3-0	R3, R2, R1, R0
R7-4	R7, R6, R5, R4
R11-8	R11, R10, R9, R8
R15-12	R15, R14, R13, R12
F3-0	F3, F2, F1, F0
F7-4	F7, F6, F5, F4
F11-8	F11, F10, F9, F8
F15-12	F15, F14, F13, F12
Ia	I7-I0; DAG1 index register
Mb	M7-M0; DAG1 modify register
Ic	I15-I8; DAG2 index register
Md	M15-M8; DAG2 modify register
(DB)	Delayed branch
(LA)	Loop abort (pop loop, PC stacks on branch)
MR0F	Multiplier result accumulator 0, foreground
MR1F	Multiplier result accumulator 1, foreground
MR2F	Multiplier result accumulator 2, foreground
MR0B	Multiplier result accumulator 0, background
MR1B	Multiplier result accumulator 1, background
MR2B	Multiplier result accumulator 2, background

Table 7.1 Syntax Notation Conventions

Name	*Description*
EQ	ALU equal zero
NE	ALU not equal to zero
GE	ALU greater than or equal zero
LT	ALU less than zero
LE	ALU less than or equal zero
GT	ALU greater than zero
AC	ALU carry
NOT AC	Not ALU carry
AV	ALU overflow
NOT AV	Not ALU overflow
MV	Multiplier overflow
NOT MV	Not multiplier overflow
MS	Multiplier sign
NOT MS	Not multiplier sign
SV	Shifter overflow
NOT SV	Not shifter overflow
SZ	Shifter zero
NOT SZ	Not shifter zero
FLAG0_IN	Flag 0
NOT FLAG0_IN	Not Flag 0
FLAG1_IN	Flag 1
NOT FLAG1_IN	Not Flag 1
FLAG2_IN	Flag 2
NOT FLAG2_IN	Not Flag 2
FLAG3_IN	Flag 3
NOT FLAG3_IN	Not Flag 3
TF	Bit test flag
NOT TF	Not bit test flag
LCE	Loop counter expired (DO UNTIL)
NOT LCE	Loop counter not expired (IF)
FOREVER	Always False (DO UNTIL)
TRUE	Always True (IF)

In a conditional instruction, the execution of the entire instruction is based on the specified condition.

Table 7.2 Condition and Termination Codes

7 Instruction Summary

Mnemonic	*Contents*
PC*	program counter
PCSTK	top of PC stack
PCSTKP	PC stack pointer
FADDR*	fetch address
DADDR*	decode address
LADDR	top of loop address stack
CURLCNTR	top of loop count stack
LCNTR	loop count for next loop
R15-R0	register file locations (fixed-point data)
F15-F0	register file locations (floating-point data)
I15-I8	DAG2 index registers
I7-I0	DAG1 index registers
M15-M8	DAG2 modify registers
M7-M0	DAG1 modify registers
L15-L8	DAG2 length registers
L7-L0	DAG1 length registers
B15-B8	DAG2 base registers
B7-B0	DAG1 base registers
DMWAIT	wait state and page size control for data memory
DMBANK1	data memory bank 1 lower boundary
DMBANK2	data memory bank 2 lower boundary
DMBANK3	data memory bank 3 lower boundary
DMADR	copy of last data memory address
PMWAIT	wait state and page size control for program memory
PMBANK1	program memory bank 1 lower boundary
PMADR	copy of last program memory address
PX	48-bit PX1 and PX2 combination
PX1	bus exchange 1 (16 bits)
PX2	bus exchange 2 (32 bits)
TPERIOD	timer period
TCOUNT	timer counter

System Registers (these are also Universal Registers):

MODE1	mode control 1
MODE2	mode control 2
IRPTL	interrupt latch
IMASK	interrupt mask
IMASKP	interrupt mask pointer
ASTAT	arithmetic status
STKY	sticky status
USTAT1	user status reg 1
USTAT2	user status reg 2

* read-only

Table 7.3 Universal Registers and System Registers

The *system register bit manipulation* instruction can be used to set, clear, toggle or test specific bits in the system registers. This instruction is described in Appendix A, Group IV-Miscellaneous instructions.

Examples:

```
BIT SET MODE2 0x00000070;

BIT TST ASTAT 0x00002000;
```

Fixed-point	*Floating-point*
Rn = Rx + Ry	Fn = Fx + Fy
Rn = Rx – Ry	Fn = Fx – Fy
Rn = Rx + Ry, Rm = Rx – Ry	Fn = Fx + Fy, Fm = Fx – Fy
Rn = Rx + Ry + CI	Fn = ABS (Fx + Fy)
Rn = Rx – Ry + CI – 1	Fn = ABS (Fx – Fy)
Rn = (Rx + Ry)/2	Fn = (Fx + Fy)/2
COMP(Rx, Ry)	COMP(Fx, Fy)
Rn = –Rx	Fn = –Fx
Rn = ABS Rx	Fn = ABS Fx
Rn = PASS Rx	Fn = PASS Fx
Rn = MIN(Rx, Ry)	Fn = MIN(Fx, Fy)
Rn = MAX(Rx, Ry)	Fn = MAX(Fx, Fy)
Rn = CLIP Rx BY Ry	Fn = CLIP Fx BY Fy
Rn = Rx + CI	Fn = RND Fx
Rn = Rx + CI – 1	Fn = SCALB Fx BY Ry
Rn = Rx + 1	Rn = MANT Fx
Rn = Rx – 1	Rn = LOGB Fx
Rn = Rx AND Ry	Rn = FIX Fx BY Ry
Rn = Rx OR Ry	Rn = FIX Fx
Rn = Rx XOR Ry	Fn = FLOAT Rx BY Ry
Rn = NOT Rx	Fn = FLOAT Rx
	Fn = RECIPS Fx
	Fn = RSQRTS Fx
	Fn = Fx COPYSIGN Fy

Table 7.4 ALU Instructions

7 Instruction Summary

$$\begin{vmatrix}Rn\\MRF\\MRB\end{vmatrix} = Rx * Ry \left(\begin{vmatrix}S\\U\end{vmatrix} \begin{vmatrix}S\\U\end{vmatrix} \begin{vmatrix}F\\I\\FR\end{vmatrix} \right)$$

$$Fn = Fx * Fy$$

$$\begin{vmatrix}Rn = MRF\\Rn = MRB\\MRF = MRF\\MRB = MRB\end{vmatrix} + Rx * Ry \left(\begin{vmatrix}S\\U\end{vmatrix} \begin{vmatrix}S\\U\end{vmatrix} \begin{vmatrix}F\\I\\FR\end{vmatrix} \right)$$

$$\begin{vmatrix}Rn = MRF\\Rn = MRB\\MRF = MRF\\MRB = MRB\end{vmatrix} - Rx * Ry \left(\begin{vmatrix}S\\U\end{vmatrix} \begin{vmatrix}S\\U\end{vmatrix} \begin{vmatrix}F\\I\\FR\end{vmatrix} \right)$$

$$\begin{vmatrix}Rn = SAT\ MRF\\Rn = SAT\ MRB\\MRF = SAT\ MRF\\MRB = SAT\ MRB\end{vmatrix} \begin{vmatrix}(SI)\\(UI)\\(SF)\\(UF)\end{vmatrix}$$

$$\begin{vmatrix}Rn = RND\ MRF\\Rn = RND\ MRB\\MRF = RND\ MRF\\MRB = RND\ MRB\end{vmatrix} \begin{vmatrix}(SF)\\(UF)\end{vmatrix}$$

$$\begin{vmatrix}MRF\\MRB\end{vmatrix} = 0$$

$$\begin{vmatrix}MRxF\\MRxB\end{vmatrix} = Rn$$

$$Rn = \begin{vmatrix}MRxF\\MRxB\end{vmatrix}$$

(X-input ❑ Y-input ❑ Data format, rounding ❑)

S	Signed input
U	Unsigned input
I	Integer input(s)
F	Fractional input(s)
FR	Fractional inputs, Rounded output

Rn, Rx, Ry	R15-R0; register file location, fixed-point
Fn, Fx, Fy	F15-F0; register file location, floating-point
MRxF	MR2F, MR1F, MR0F; multiplier result accumulators, foreground
MRxB	MR2B, MR1B, MR0B; multiplier result accumulators, background
(SF)	Default format for 1-input operations
(SSF)	Default format for 2-input operations

Table 7.5 Multiplier Instructions

Shifter	*Shifter Immediate*
Rn = LSHIFT Rx BY Ry	Rn = LSHIFT Rx BY <data8>
Rn = Rn OR LSHIFT Rx BY Ry	Rn = Rn OR LSHIFT Rx BY <data8>
Rn = ASHIFT Rx BY Ry	Rn = ASHIFT Rx BY <data8>
Rn = Rn OR ASHIFT Rx BY Ry	Rn = Rn OR ASHIFT Rx BY <data8>
Rn = ROT Rx BY RY	Rn = ROT Rx BY <data8>
Rn = BCLR Rx BY Ry	Rn = BCLR Rx BY <data8>
Rn = BSET Rx BY Ry	Rn = BSET Rx BY <data8>
Rn = BTGL Rx BY Ry	Rn = BTGL Rx BY <data8>
BTST Rx BY Ry	BTST Rx BY <data8>
Rn = FDEP Rx BY Ry	Rn = FDEP Rx BY <bit6>:<len6>
Rn = Rn OR FDEP Rx BY Ry	Rn = Rn OR FDEP Rx BY <bit6>:<len6>
Rn = FDEP Rx BY Ry (SE)	Rn = FDEP Rx BY <bit6>:<len6> (SE)
Rn = Rn OR FDEP Rx BY Ry (SE)	Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)
Rn = FEXT Rx BY Ry	Rn = FEXT Rx BY <bit6>:<len6>
Rn = FEXT Rx BY Ry (SE)	Rn = FEXT Rx BY <bit6>:<len6> (SE)
Rn = EXP Rx	
Rn = EXP Rx (EX)	
Rn = LEFTZ Rx	
Rn = LEFTO Rx	

Table 7.6 Shifter and Shifter Immediate Instructions

7 Instruction Summary

Fixed-point

Rm=R3-0 * R7-4 (SSFR),	Ra=R11-8 + R15-12
Rm=R3-0 * R7-4 (SSFR),	Ra=R11-8 – R15-12
Rm=R3-0 * R7-4 (SSFR),	Ra=(R11-8 + R15-12)/2
MRF=MRF + R3-0 * R7-4 (SSF),	Ra=R11-8 + R15-12
MRF=MRF + R3-0 * R7-4 (SSF),	Ra=R11-8 – R15-12
MRF=MRF + R3-0 * R7-4 (SSF),	Ra=(R11-8 + R15-12)/2
Rm=MRF + R3-0 * R7-4 (SSFR),	Ra=R11-8 + R15-12
Rm=MRF + R3-0 * R7-4 (SSFR),	Ra=R11-8 – R15-12
Rm=MRF + R3-0 * R7-4 (SSFR),	Ra=(R11-8 + R15-12)/2
MRF=MRF – R3-0 * R7-4 (SSF),	Ra=R11-8 + R15-12
MRF=MRF – R3-0 * R7-4 (SSF),	Ra=R11-8 – R15-12
MRF=MRF – R3-0 * R7-4 (SSF),	Ra=(R11-8 + R15-12)/2
Rm=MRF – R3-0 * R7-4 (SSFR),	Ra=R11-8 + R15-12
Rm=MRF – R3-0 * R7-4 (SSFR),	Ra=R11-8 – R15-12
Rm=MRF – R3-0 * R7-4 (SSFR),	Ra=(R11-8 + R15-12)/2
Rm=R3-0 * R7-4 (SSFR),	Ra=R11-8 + R15-12, Rs=R11-8 – R15-12

Floating-point

Fm=F3-0 * F7-4,	Fa=F11-8 + F15-12
Fm=F3-0 * F7-4,	Fa=F11-8 – F15-12
Fm=F3-0 * F7-4,	Fa=FLOAT R11-8 by R15-12
Fm=F3-0 * F7-4,	Fa=FIX R11-8 by R15-12
Fm=F3-0 * F7-4,	Fa=(F11-8 + F15-12)/2
Fm=F3-0 * F7-4,	Fa=ABS F11-8
Fm=F3-0 * F7-4,	Fa=MAX (F11-8, F15-12)
Fm=F3-0 * F7-4,	Fa=MIN (F11-8, F15-12)
Fm=F3-0 * F7-4,	Fa=F11-8 + F15-12, Fs=F11-8 – F15-12

Ra, Rm	Any register file location (fixed-point)
R3-0	R3, R2, R1, R0
R7-4	R7, R6, R5, R4
R11-8	R11, R10, R9, R8
R15-12	R15, R14, R13, R12
Fa, Fm	Any register file location (floating-point)
F3-0	F3, F2, F1, F0
F7-4	F7, F6, F5, F4
F11-8	F11, F10, F9, F8
F15-12	F15, F14, F13, F12
(SSF)	X-input signed, Y-input signed, fractional inputs
(SSFR)	X-input signed, Y-input signed, fractional inputs, rounded output

Table 7.7 Multifunction Instructions

No.	*Vector*	*Function*
0	0x00	Reserved
1*	0x08	Reset
2	0x10	Reserved
3	0x18	Status stack or loop stack overflow or PC stack full
4	0x20	Timer=0 (high priority option)
5	0x28	$\overline{IRQ}_3$ asserted
6	0x30	$\overline{IRQ}_2$ asserted
7	0x38	$\overline{IRQ}_1$ asserted
8	0x40	$\overline{IRQ}_0$ asserted
9	0x48	Reserved
10	0x50	Reserved
11	0x58	Circular buffer 7 overflow
12	0x60	Circular buffer 15 overflow
13	0x68	Reserved
14	0x70	Timer=0 (low priority option)
15	0x78	Fixed-point overflow
16	0x80	Floating-point overflow
17	0x88	Floating-point underflow
18	0x90	Floating-point invalid operation
19-23	0x98-B8	Reserved
24-31	0xC0-F8	User software interrupts

* Nonmaskable

Table 7.8 Interrupt Vectors and Priority

7 Instruction Summary

I. Compute and Move or Modify

1. *compute,* | DM(Ia, Mb) = dreg1 | , | PM(Ic, Md) = dreg2 | ;
| dreg1 = DM(Ia, Mb) | | dreg2 = PM(Ic, Md) |

2. *IF condition* compute ;

3. a. *IF condition* *compute,* | DM(Ia, Mb) | = ureg ;
| PM(Ic, Md) |

b. *IF condition* *compute,* | DM(Mb, Ia) | = ureg ;
| PM(Md, Ic) |

c. *IF condition* *compute,* ureg = | DM(Ia, Mb) | ;
| PM(Ic, Md) |

d. *IF condition* *compute,* ureg = | DM(Mb, Ia) | ;
| PM(Md, Ic) |

4. a. *IF condition* *compute,* | DM(Ia, <data6>) | = dreg ;
| PM(Ic, <data6>) |

b. *IF condition* *compute,* | DM(<data6>, Ia) | = dreg ;
| PM(<data6>, Ic) |

c. *IF condition* *compute,* dreg = | DM(Ia, <data6>) | ;
| PM(Ic, <data6>) |

d. *IF condition* *compute,* dreg = | DM(<data6>, Ia) | ;
| PM(<data6>, Ic) |

5. *IF condition* *compute,* ureg1 = ureg2 ;

6. a. *IF condition* shiftimm , | *DM(Ia, Mb)* | = *dreg* ;
| *PM(Ic, Md)* |

b. *IF condition* shiftimm , *dreg* = | *DM(Ia, Mb)* | ;
| *PM(Ic, Md)* |

7. *IF condition* *compute,* MODIFY | (Ia, Mb) | ;
| (Ic, Md) |

Instruction Summary 7

II. Program Flow Control

8. *IF condition* | JUMP / CALL | | <addr24> / (PC, <reladdr24>) | (| *DB* / *LA* / *DB, LA* |) ;

9. *IF condition* | JUMP / CALL | | (Md, Ic) / (PC, <reladdr6>) | (| *DB* / *LA* / *DB, LA* |) , *compute* ;

11. *IF condition* | RTS / RTI | (| *DB* / *LA* / *DB, LA* |) , *compute* ;

12. LCNTR = | <data16> / ureg | , DO | <addr24> / (<PC, reladdr24>) | UNTIL LCE ;

13. DO | <addr24> / (PC, <reladdr24>) | UNTIL termination ;

7 Instruction Summary

III. Immediate Move

14. a. | DM(<addr32>) | = ureg ;
       | PM(<addr24>) |

    b. ureg = | DM(<addr32>) | ;
              | PM(<addr24>) |

15. a. | DM(<data32>, Ia) | = ureg ;
       | PM(<data24>, Ic) |

    b. ureg = | DM(<data32>, Ia) | ;
              | PM(<data24>, Ic) |

16. | DM(Ia, Mb) | = <data32> ;
    | PM(Ic, Md) |

17. ureg = <data32> ;

IV. Miscellaneous

18. BIT | SET
CLR
TGL
TST
XOR | sreg <data32> ;

19. a. MODIFY | (Ia, <data32>)
(Ic, <data32>) | ;

b. BITREV (Ia, <data32>) ;

20. | *PUSH*
POP | *LOOP* , | *PUSH*
POP | *STS* ;

21. NOP ;

22. IDLE ;

Assembly Programming Tutorial

8

8.1 INTRODUCTION

This tutorial is for first-time ADSP-21020/21010 assembly-language programmers. It explains basic techniques and conventions for good programming. The approach described here can be applied to ADSP-21020/21010 programming in general.

The tutorial makes use of the ADSP-21000 Family Development Software (Assembler, Linker and Simulator programs). You must have this software to complete the parts of the tutorial that create the executable programs.

This tutorial demonstrates ADSP-21020/21010 programming by example. It presents two completely working DSP systems based on the ADSP-21020, highlighting:

- Architecture file
- Assembler preprocessor directives
- Calling subroutines in another file from the main program
- General processor initialization procedure
- Interrupt vector table placement and usage
- Programming memory wait states and bank selects
- Looping code, including:
  - –Rolling a loop for more compact code
- Multifunction instructions, focusing on:
  - –Usage and restrictions
- Simulating I/O ports
- Using the interval timer
- How to use interrupts, including:
  - –Vector table code
  - –Which registers are used
  - –What to do after processor reset
  - –Fast context switching
- Special features of the system registers
- Useable code examples
- Two turnkey, complete system examples
- An additional, efficient FFT code example

8 Programming Tutorial

Both examples filter digital data using infinite impulse response (IIR) filters. The examples consist of a main calling shell and called subroutines. Technical information on the filters themselves is presented after the examples.

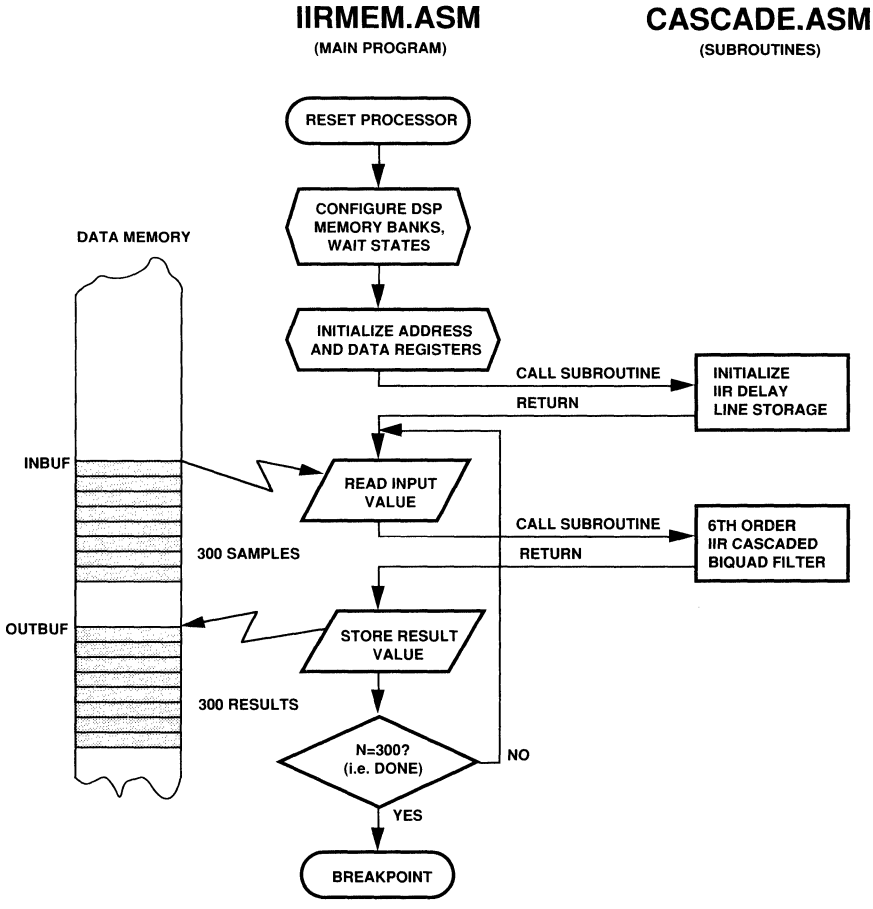
The source files for the examples presented in this chapter can be obtained by:

- Downloading the files from the DSP Applications Bulletin Board Service (see © page at the front of this manual for BBS contact information),
- Purchasing the ADSP-21000 Family Development Software, which includes the example files, or
- Contacting Analog Devices DSP Applications Engineering directly (see © page at the front of this manual for phone/FAX numbers).

The first example, called "iirmem", (see Figure 8.1) reads a buffer of input values stored in memory, passes the data through the filter, and writes the results to another buffer in memory. The second example, called "iirirq", (see Figure 8.2, on page 8-4) reads input data via a memory-mapped I/O port and writes the filtered values to another port. The second program also serves to demonstrate how the interval timer can be used to generate processor interrupts at a desired sampling rate and how interrupts are handled on the ADSP-21020/21010. The software aspects of I/O port hardware are also described.

8.2 EXAMPLE #1: DATA IN MEMORY, NO INTERRUPTS

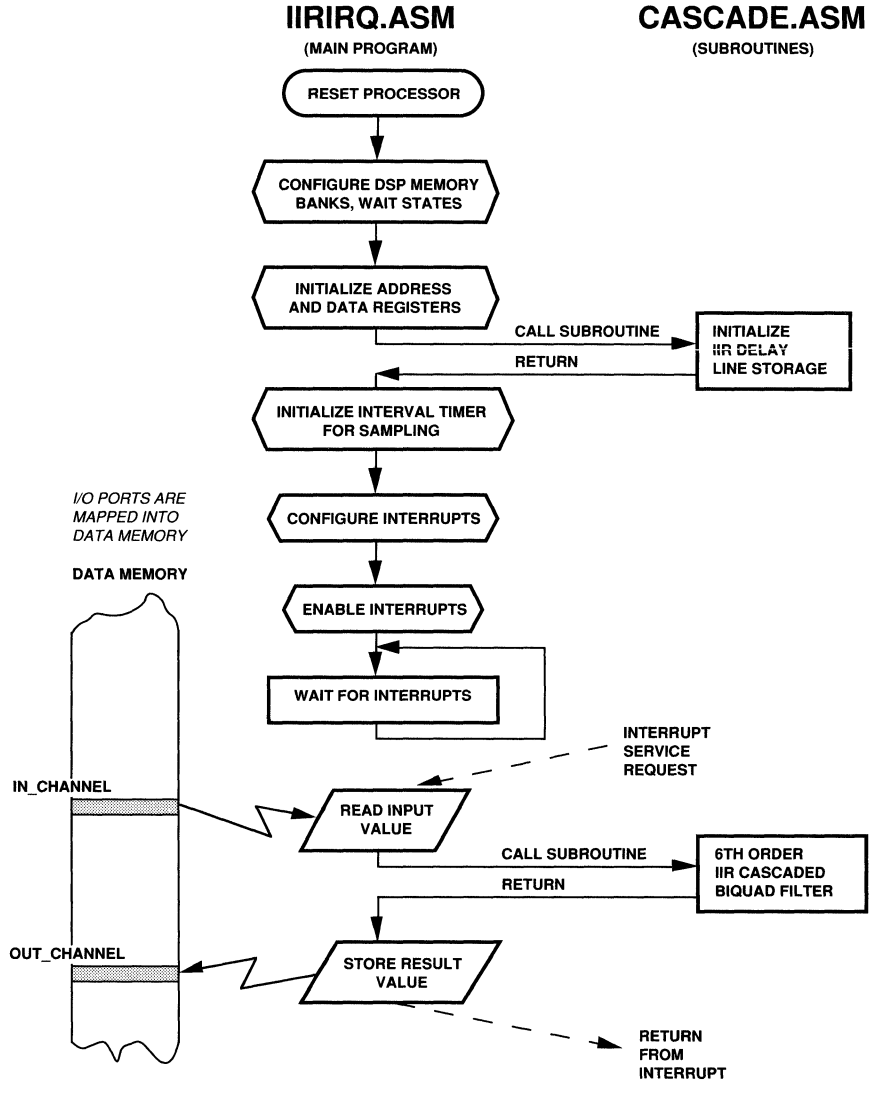
The first example (program flowchart shown in Figure 8.1) reads a buffer of input values stored in memory, passes the data through the filter, and writes the results to another buffer in memory. Because no interrupts are used, execution progresses at the processor instruction rate until all input values have been filtered. After filtering the input data set, execution would normally continue, so in this example, the IDLE instruction is used to halt execution. The example also shows how the input data is initialized in memory.



Memory-resident input sample vector ("INBUF") is filtered, storing the result vector in another data buffer ("OUTBUF"). No interrupts are used. Execution ceases when done.

Figure 8.1 Program Flow for First Example

8 Programming Tutorial



Sampled data is read from a memory-mapped I/O port during execution under interrupt control. Results are written out to another port. The DSP is interrupted at the sampling rate. The DSP's interval timer is the sampling clock.

Figure 8.2 Program Flow for Second Example

8.2.1 File Inventory

Let's start by taking inventory of the files used in this system. Table 8.1 enumerates the files and gives a brief description of their functions:

Filename	*Function*
`generic.ach`	architecture description file
`iirmem.asm`	main assembly program
`cascade.asm`	cascaded biquad filter subroutine (called by main program)
`iircoefs.dat`	filter coefficients
`makefile.mem`	MS-DOS "make" file used to create executable program

Table 8.1 Files Used for Memory-Based (No Interrupts) Program

8.2.2 Architecture Description File (generic.ach)

Figure 8.3, on page 8-8, shows how actual external memory could be connected, in hardware, to the ADSP-21020 in this example. Notice that there are three physical memory banks, each 2048 (0x800) words in length. Figure 8.4 shows the total addressable program and data memory map of the ADSP-21020 and highlights the portions used in Figure 8.3. Listing 8.1, on the following page, contains the description of the architecture found in Figure 8.3. This file, called the *architecture description file*, is used by both the linker and the simulator during the code development process. The architecture description file guides the linker in placing code and data in the ADSP-21020 memory map. This file is also used by the simulator in order to simulate not only the processor itself, but also the memory connected to it.

If you are familiar with the ADSP-2100 Family (16-bit fixed-point DSP) development tools, you may be surprised that the ADSP-21000 Family Development Software has no System Builder program. Instead, the ADSP-21020/21010 software tools read the .ach text file directly for the necessary information. In other words, you create the .ach file with a text editor and that's it—no other actions are needed.

8 Programming Tutorial

```
.SYSTEM          generic;
.PROCESSOR =     ADSP21020;

                 memory                                          memory   segment
  directive      type      start address      end address        space    name
.SEGMENT /RAM   /BEGIN=0x000000 /END=0x000007 /PM      resrvd0;
.SEGMENT /RAM   /BEGIN=0x000008 /END=0x00000F /PM      rst_svc;
.SEGMENT /RAM   /BEGIN=0x000010 /END=0x000017 /PM      resrvd1;
.SEGMENT /RAM   /BEGIN=0x000018 /END=0x00001F /PM      sovf_svc;
.SEGMENT /RAM   /BEGIN=0x000020 /END=0x000027 /PM      tmzh_svc;
.SEGMENT /RAM   /BEGIN=0x000028 /END=0x00002F /PM      irq3_svc;
.SEGMENT /RAM   /BEGIN=0x000030 /END=0x000037 /PM      irq2_svc;
.SEGMENT /RAM   /BEGIN=0x000038 /END=0x00003F /PM      irq1_svc;
.SEGMENT /RAM   /BEGIN=0x000040 /END=0x000047 /PM      irq0_svc;
.SEGMENT /RAM   /BEGIN=0x000048 /END=0x00004F /PM      resrvd2;
.SEGMENT /RAM   /BEGIN=0x000050 /END=0x000057 /PM      resrvd3;
.SEGMENT /RAM   /BEGIN=0x000058 /END=0x00005F /PM      cb7_svc;
.SEGMENT /RAM   /BEGIN=0x000060 /END=0x000067 /PM      cb15_svc;
.SEGMENT /RAM   /BEGIN=0x000068 /END=0x00006F /PM      resrvd4;
.SEGMENT /RAM   /BEGIN=0x000070 /END=0x000077 /PM      tmzl_svc;
.SEGMENT /RAM   /BEGIN=0x000078 /END=0x00007F /PM      fix_svc;
.SEGMENT /RAM   /BEGIN=0x000080 /END=0x000087 /PM      flto_svc;
.SEGMENT /RAM   /BEGIN=0x000088 /END=0x00008F /PM      fltu_svc;
.SEGMENT /RAM   /BEGIN=0x000090 /END=0x000097 /PM      flti_svc;
.SEGMENT /RAM   /BEGIN=0x000098 /END=0x00009F /PM      resrvd5;
.SEGMENT /RAM   /BEGIN=0x0000A0 /END=0x0000A7 /PM      resrvd6;
.SEGMENT /RAM   /BEGIN=0x0000A8 /END=0x0000AF /PM      resrvd7;
.SEGMENT /RAM   /BEGIN=0x0000B0 /END=0x0000B7 /PM      resrvd8;
.SEGMENT /RAM   /BEGIN=0x0000B8 /END=0x0000BF /PM      resrvd9;
.SEGMENT /RAM   /BEGIN=0x0000C0 /END=0x0000C7 /PM      sft0_svc;
.SEGMENT /RAM   /BEGIN=0x0000C8 /END=0x0000CF /PM      sft1_svc;
.SEGMENT /RAM   /BEGIN=0x0000D0 /END=0x0000D7 /PM      sft2_svc;
.SEGMENT /RAM   /BEGIN=0x0000D8 /END=0x0000DF /PM      sft3_svc;
.SEGMENT /RAM   /BEGIN=0x0000E0 /END=0x0000E7 /PM      sft4_svc;
.SEGMENT /RAM   /BEGIN=0x0000E8 /END=0x0000EF /PM      sft5_svc;
.SEGMENT /RAM   /BEGIN=0x0000F0 /END=0x0000F7 /PM      sft6_svc;
.SEGMENT /RAM   /BEGIN=0x0000F8 /END=0x0000FF /PM      sft7_svc;

.SEGMENT /RAM   /BEGIN=0x000100 /END=0x0007FF /PM      pm_code;
.SEGMENT /RAM   /BEGIN=0x000800 /END=0x000FFF /PM      pm_data;

.SEGMENT /RAM   /BEGIN=0x000000   /END=0x0007FF   /DM    dm_data;
.SEGMENT /PORT  /BEGIN=0xF0000000 /END=0xF000001F /DM    ports;

.ENDSYS;
```

Listing 8.1 generic.ach

The .SYSTEM and .ENDSYS directives indicate the start and end of the architecture description. Although the segment names are arbitrary, they are chosen here to be self-documenting.

The first of the three physical 2K-word memory blocks is divided into 33 segments. The first 32 segments are mapped to the 256 (0x100) locations reserved for the interrupt vector table, and the other segment named *pm_code* contains the balance (0x700) for general instruction code storage. This memory is connected to the program memory interface.

The second of the three physical 2K-word memory blocks is set aside for data storage in program memory space. This block only contains one segment named *pm_data*. This memory is also connected to the program memory interface. See the following section for a description of how the processor differentiates between this program memory block and the previous one.

The third of the three physical 2K-word memory blocks is general-purpose data memory. This block only contains one segment named *dm_data*. This memory is connected to the data memory interface.

8.2.3 External vs. Internal Address Decoding

Figure 8.3, on the following page, shows how **external** hardware address decoding logic arbitrates to select which program memory block drives the program memory interface at a given time. The block titled "address decode" is an address comparator. Depending on the value on the PMA bus, the address decoder enables either one memory block or the other.

To avoid the need for this external logic, the ADSP-21020 incorporates **internal** address decoding logic. Both the program memory and the data memory spaces are divided into several banks, each of which has it own select line ($\overline{\text{PMS0}}$ and $\overline{\text{PMS1}}$ for program memory, $\overline{\text{DMS0}}$, $\overline{\text{DMS1}}$, $\overline{\text{DMS2}}$ and $\overline{\text{DMS3}}$ for data memory). When accessing memory, only the appropriate select line becomes active, depending on the address value. The boundary addresses are stored in special registers on the ADSP-21020 (PMBANK1, and DMBANK1, 2, 3). Example #2 "iirirq" makes use of these bank selects. Compare Figure 8.4 for example #1 and Figure 8.6 for example #2.

8 Programming Tutorial

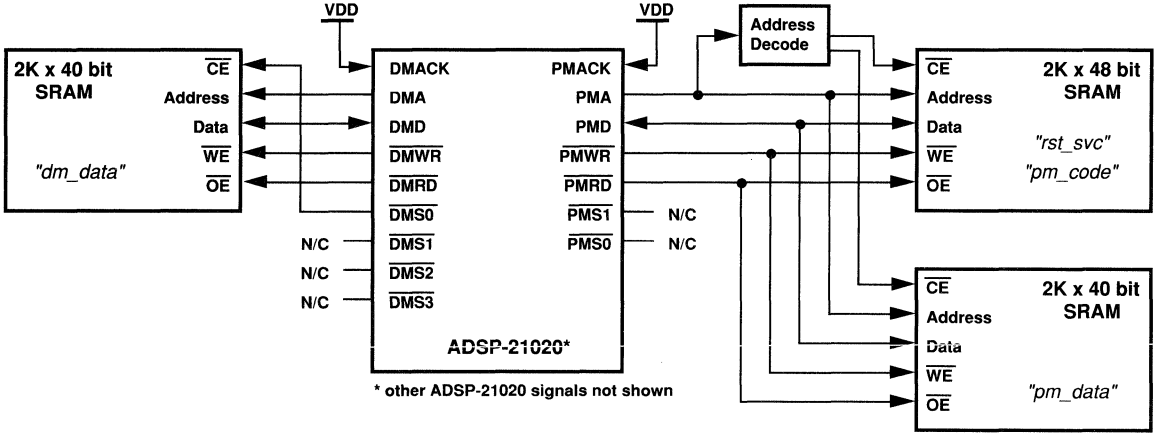


Figure 8.3 Physical Memory Architecture Described in "generic.ach"

8.2.4 Specifying The .ACH File

The architecture file can be specified to the linker (ld21k) and the simulator (sim21k) with the –a <filename> switch at invocation. For example:

```
ld21k iirmem cascade -a generic -m
```

runs the linker using "generic.ach" as the architecture file, and

```
sim21k -e iirmem -a generic
```

invokes the simulator loading "generic.ach" as the architecture file.

Although the filename extension ".ach" is explicitly specified in this example, the linker and simulator assume this extension by default and it need not be typed.

8.2.5 Main Program (iirmem.asm)

The main assembly program, called "iirmem.asm," is shown in Listing 8.2. This program performs the ADSP-21020's **initial setups** after reset, then proceeds into the **main processing** loop.

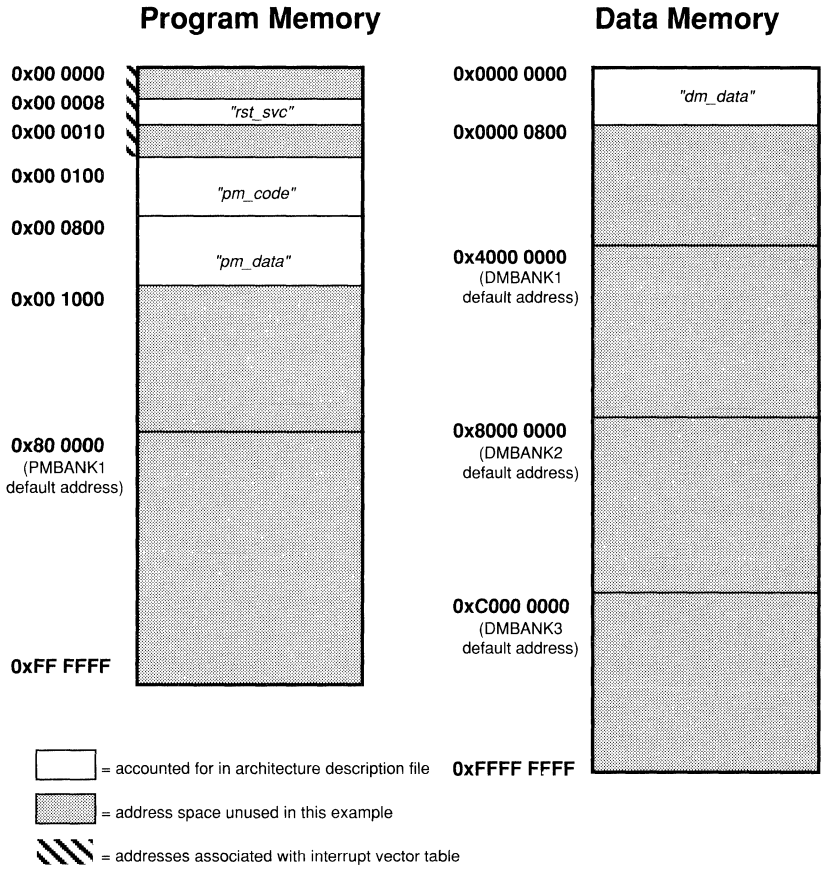


Figure 8.4 Memory Map Described in "generic.ach"

The typical **initial setups** include:

- Disabling interrupts, (default at processor reset)
- Initializing the interrupt vector table,
- Altering the values in the memory hardware configuration registers: DMBANK1, 2, 3, PMBANK1, DMWAIT, PMWAIT
- Initializing address and data registers,
- Initializing memory locations and buffers,
- Configuring and initializing on-chip peripherals such as the timer,
- Configuring interrupts, and
- Enabling interrupts (done last).

8 Programming Tutorial

The **main processing** loop is either:

- A list of tasks which eventually terminates, possibly with interrupts, or
- An endless loop, most often interrupted with interrupt-driven tasks.

```
.EXTERN cascaded_biquad, cascaded_biquad_init;
.GLOBAL coefs, dline;
.PRECISION=40;
.ROUND_NEAREST;

#define SAMPLES 300
#define SECTIONS 3

.SEGMENT /DM    dm_data;
.VAR    inbuf[SAMPLES] = 1.0, 0.0;      { input = unit impulse }
.VAR    outbuf[SAMPLES];                { ends up holding impulse response }
.VAR    dline[SECTIONS*2];              { w", w', NEXT w", NEXT w', ... }
.ENDSEG;

.SEGMENT /PM    pm_data;
.VAR    coefs[SECTIONS*4]="iircoefs.dat";   { a12,a11,b12,b11,a22,a21,... }
.ENDSEG;

.SEGMENT /PM    rst_svc;
        jump begin;
.ENDSEG;

.SEGMENT /PM    pm_code;
```

initial setups :

```
begin:  pmwait=0x0021;                  { zero wait states for all of PM }
        dmwait=0x8421;                  { zero wait states for all of DM }
        b3=inbuf;   l3=0;
        b4=outbuf;  l4=0;
        l0=0; l1=0; l8=0;
        m1=1; m8=1;
        call cascaded_biquad_init (db);     { zero the delay line }
         r0=SECTIONS;
         b0=dline;
```

main processing loop :

```
        lcntr=SAMPLES, do filtering until lce;
            f8=dm(i3,1);
            call cascaded_biquad (db);      { input=F8, output=F8 }
             b0=dline;
             b8=coefs;
filtering:  dm(i4,1)=f8;
done:   idle;
.ENDSEG;
```

Listing 8.2 iirmem.asm

8.2.5.1 Initial Setups: Initialization Following Reset

This section describes each of the initial setups.

Disabling interrupts. Resetting the processor automatically disables interrupts by setting the IMASK register to zero. IRPTL, the interrupt latch register, is not affected by reset and therefore should be cleared before enabling any interrupts. However, no interrupts are ever enabled in this example. Therefore you do not need to do anything with these registers in this case.

Initializing the interrupt vector table. At the beginning of instruction memory space is the non-relocatable **interrupt vector table** as required by the ADSP-21020 (see Figure 8.4). For any given interrupt signal, there is a predetermined instruction address (called the interrupt vector) which is branched to when the interrupt occurs. The instruction at this address is often a jump to an interrupt service routine which resides outside the interrupt vector table (see Listing 8.2).

Interrupt vector table addresses are spaced by eight locations. This allows eight cycles of code to be executed for a given interrupt without a branch and the associated overhead, providing quick interrupt servicing for short routines. The only requirement is to terminate the instruction sequence with a return from interrupt (RTI) instruction.

Interrupt service code within the interrupt vector table which **extends beyond the allotted eight locations** is not advised. For example, it is possible to extend into the space reserved for the next interrupt if that interrupt is not being used, but this is a poor programming practice.

In addition, **certain interrupt vector entries should not be used** at all. The first vector location PM[0x00] is reserved and should never be used by the programmer. The remaining vectors labeled "reserved" could be used for code, but this would be a poor programming practice. If those vector locations are utilized by future members of the ADSP-21000 family, using those locations on the ADSP-21020 today may result in code incompatibility in the future.

By sectioning the interrupt table space (PM[0] - PM[0xFF]) into 32 segments of 8 locations each, both of these poor programming practices are effectively discouraged.

This particular vector table simply causes the processor to branch after reset to the beginning of the executable code, at the label *begin*, which immediately follows the end of the vector table (PM[0x100]). In this example, the only interrupt vector table instructions are stored in the "rst_svc" segment:

```
.SEGMENT /PM       rst_svc;
                   jump begin;
.ENDSEG;
```

All other vectors are left undefined.

Example #2 ("iirirq") shows how other interrupts are incorporated into the vector table.

Altering the values in the memory hardware configuration registers. The first instructions executed after reset alter the ADSP-21020 memory configuration registers, the DMWAIT and PMWAIT registers. These wait state configuration registers contain default values at reset. In this example, the registers are altered with values that set all software wait states to zero. The wait state mode is set to software-programmed wait only (i.e., the DMACK and PMACK hardware inputs are not used).

```
begin:   pmwait=0x0021;      { zero wait states for all of PM }
         dmwait=0x8421;      { zero wait states for all of DM }
```

The programmable memory bank boundaries are defined with PMBANK1, DMBANK1, DMBANK2 and DMBANK3 registers. These memory bank registers contain default address values at reset (see Figure 8.4). Since bank 0 of both program and data memory always starts at address zero, there are no PMBANK0 or DMBANK0 registers. In this example, the bank default addresses are left unchanged. Example #2 shows a more extensive memory configuration procedure.

Initializing address and data registers. Registers which reside in the data address generators (DAGs) are used for addressing purposes. These registers may be initialized during the initial setup period, especially in short code examples such as this one, where DAG registers serve unchanging functions. The instructions that initialize these registers are:

```
b3=inbuf;   l3=0;
b4=outbuf;  l4=0;
l0=0;       l1=0;       l8=0;
            m1=1;       m8=1;
```

Notice that setting a B (base) register automatically also sets the associated I (index) register to the same value. Although the B register is only necessary in circular buffering applications, during initial setup it is a good practice to set the B register instead of the I register even if the buffer is not circular. That way, if you later decides to make the buffer a circular buffer by setting the associated L (length) register to a certain value, there will be no problems with having not initialized the B register.

It is especially important to preset the length (L) registers. Neglecting to set the L registers may cause circular buffer addressing when nonzero data appears in the L registers upon powerup. Where circular addressing is NOT desired, set the appropriate L registers to zero. Where circular addressing IS desired, set the appropriate L registers to the circular buffer length.

The initialization of one of the B registers (B0) occurs in the program after a delayed call instruction but it is actually executed before the call, with the other initializations, since the call is delayed. This is shown in the next section.

Initializing memory locations and buffers. The delay line storage memory for the IIR filter is cleared to zero by calling the subroutine *cascaded_biquad_init*. This **memory initialization** code resides in another file called "cascade.asm." Notice that delayed branching is used for execution efficiency.

```
call cascaded_biquad_init (db);  { zero the delay line }
 r0=SECTIONS;                    { executed on the way }
 b0=dline;                       { into the subroutine }
```

Configuring and initializing on-chip peripherals. No on-chip peripherals are used. No setups are required in this case.

Configuring and enabling interrupts. Since interrupts are not used in this example, they are neither configured nor enabled. Upon reset, the default register values are such that if you are not planning on using interrupts, you do not have to worry about interrupt configuration at all.

8.2.5.2 Main Processing Loop

Once the initial setups are complete, the filtering operation can begin. Three hundred input samples reside in an input data buffer, and the program must process each one via the filtering operation and store the results in the output data buffer. The 300 iterations are automatically managed in hardware by a zero-overhead DO UNTIL loop construct. The loop is set up in conjunction with assigning the value 300 to the loop counter register (LCNTR). Unlike in the ADSP-2100 family, these two operations can be done in the same instruction cycle. For example:

```
#define SAMPLES 300

lcntr=SAMPLES, do filtering until lce;
```

Five instructions are executed within the loop. First, an input data value is read from the input buffer. Then the *cascaded_biquad* subroutine is called. The subroutine code resides in a separate file called "cascade.asm." Because the call is a delayed branch, the two instructions that follow the call instruction actually get executed before the call. This is indicated here by indenting these two instructions. The next instruction after returning from the subroutine stores the returned value from the subroutine in the output buffer. For example:

```
            f8=dm(i3,1);
            call cascaded_biquad (db);     { input=F8, output=F8 }
             b0=dline;
             b8=coefs;
filtering:  dm(i4,1)=f8;
```

Upon termination of the loop, the ADSP-21020 continues by executing subsequent code. In this example, that code is an IDLE instruction, which halts the processor. When using the simulator, you could set a breakpoint at the instruction labeled *done* to halt execution. A breakpoint will not only halt simulation when reached, but it will also display a message. For example:

```
done:       idle;                    { set breakpoint here! }
```

8.2.6 Creating The Executable Program

The executable program for this example is created using the following commands to invoke the ADSP-21000 Family Assembler and Linker:

```
asm21k iirmem
asm21k cascade
ld21k iirmem cascade -a generic -m
```

The memory map file, created by linking the files using the –m switch, shows how the linker loads the interrupt vector instructions, the main program, the called subroutines, and all the associated data spaces into the memory segments defined by the architecture file in Listing 8.1.

8.2.7 Simulation

This example system can be simulated by entering the following command to invoke the ADSP-21020/21010 Simulator:

```
sim21k -e iirmem -a generic
```

Look at the input buffer and the output buffer. Notice that the output buffer does not contain the results yet. Set a breakpoint at the program memory location labelled *done*. Allow the simulator to run (execute the processor code). It will halt upon fetching the breakpoint instruction. Look at the output buffer, which should now contain the result values. These results may be dumped into a file using the memory dump command.

8 Programming Tutorial

8.3 EXAMPLE #2—INTERRUPT-DRIVEN, WITH PORT I/O

This second example (program flowchart shown in Figure 8.2) reads one input value from a memory-mapped I/O port, passes the data through the filter, and writes the result value to another port. This transaction occurs whenever requested by an interrupt signal, in this case the timer interrupt. This example uses the interval timer to generate periodic interrupts. Specific topics which will be explored here are:

- programming the interrupt vector table
- setting up the ADSP-21020 to react to interrupts and configuring how they are used
- using memory-mapped I/O ports
- simulating operation with the software development tools
- benchmarking the execution efficiency of interrupt-driven code
- interrupt debugging hints

Some information relevant to this example is presented in example #1 (iirmem). Instead of duplicating this information, example #2 highlights new or different information only.

8.3.1 File Inventory

Table 8.2 lists the files and gives a brief description of their functions:

Filename	*Function*
`iirirq.ach`	architecture description file
`def21020.h`	bit position definitions for ADSP-21020 system registers
`iirirq.asm`	main assembly program
`cascade.asm`	cascaded biquad filter subroutine
`iircoefs.dat`	filter coefficients
`makefile.irq`	MS-DOS "make" file used to create executable program
`input.dat`	example input file for I/O port simulation
`out300.dat`	example output file from I/O port simulation

Table 8.2 Files Used for Interrupt-Driven Program using Port I/O

8.3.2 Architecture Description File (iirirq.ach)

Figure 8.5 shows how actual external memory could be connected, in hardware, to the ADSP-21020 in this example. Notice three physical RAM memory banks, each 2048 (0x800) words in length, in conjunction with two memory-mapped I/O ports. Figure 8.6 shows the total addressable program and data memory maps of the ADSP-21020 and highlights which portions are used by the system in Figure 8.5. Listing 8.3 contains the description of the architecture found in Figure 8.5.

```
.SYSTEM          IIRIRQ_example_arch_file;
.PROCESSOR =     ADSP21020;

                 memory                                          memory   segment
  directive      type      start address          end address    space    name
.SEGMENT /RAM   /BEGIN=0x000008   /END=0x00000F    /PM    rst_svc;
.SEGMENT /RAM   /BEGIN=0x000020   /END=0x000027    /PM    tmzh_svc;
.SEGMENT /RAM   /BEGIN=0x000100   /END=0x0007FF    /PM    pm_code;
.SEGMENT /RAM   /BEGIN=0x000800   /END=0x000FFF    /PM    pm_bank1;

.SEGMENT /RAM   /BEGIN=0x00000000 /END=0x00000FFF /DM    dm_bank0;
.SEGMENT /PORT  /BEGIN=0x00001000 /END=0x00001000 /DM    dm_bank1;
.SEGMENT /PORT  /BEGIN=0x00002000 /END=0x00002000 /DM    dm_bank2;

.ENDSYS;
```

Listing 8.3 iirirq.ach

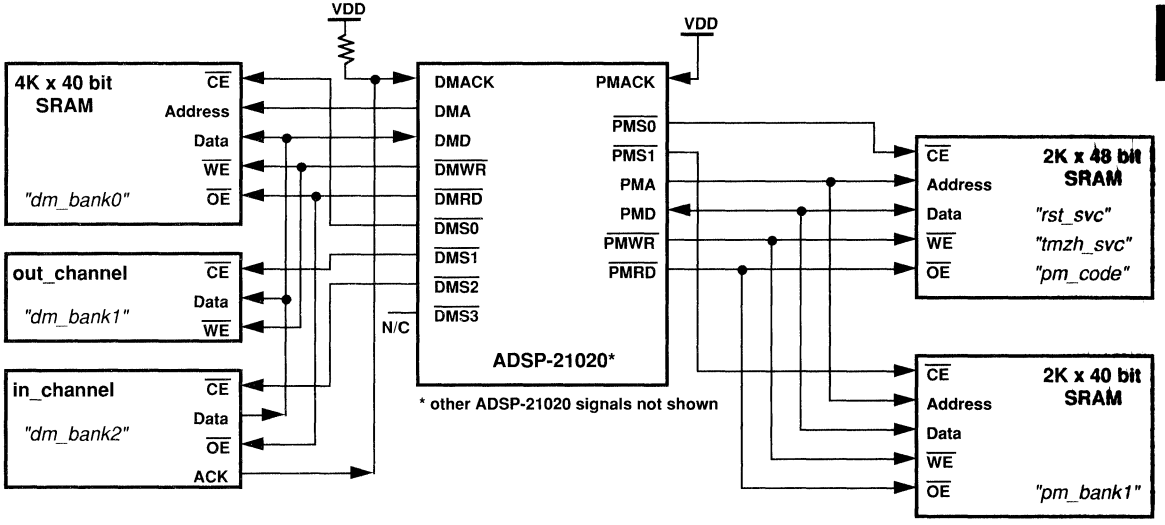


Figure 8.5 Physical Memory Architecture Described in "iirirq.ach"

8 Programming Tutorial

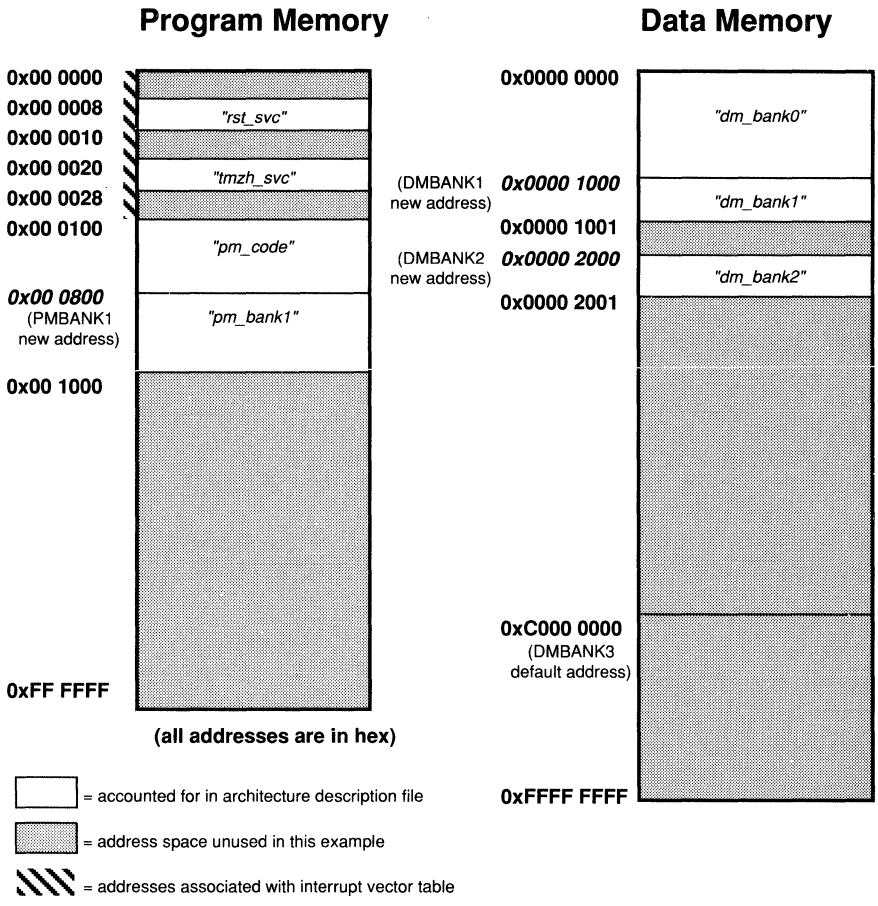


Figure 8.6 Memory Map Described in "iirirq.ach"

8.3.3 Main Program (iirirq.asm)

The main assembly program, called "iirirq.asm," is shown in Listing 8.4. Following the typical structure of any ADSP-21020 program as shown in example #1, this program also performs the ADSP-21020's **initial setups** after reset, then proceeds into the **main processing** loop. Because **interrupts**, the interval timer, and memory bank selects will be used, there are more setup tasks to be done in this example than in example #1.

```
#include "def21020.h"                    { bit definitions }
#define SAMPLES 300
#define SECTIONS 3

.EXTERN cascaded_biquad, cascaded_biquad_init;
.GLOBAL coefs, dline;
.PRECISION = 40;
.ROUND_NEAREST;

.SEGMENT /DM    dm_bank0;                { selected by DMS0~ }
.VAR    dline[SECTIONS*2];               { filter delay line: }
.ENDSEG;                                 { w", w', NEXT w", NEXT w', ... }

.SEGMENT /DM    dm_bank1;                { selected by DMS1~ }
.VAR    in_channel;                      { memory-mapped I/O port }
.ENDSEG;

.SEGMENT /DM    dm_bank2;                { selected by DMS2~ }
.VAR    out_channel;                     { memory-mapped I/O port }
.ENDSEG;

.SEGMENT /PM    pm_bank1;                { selected by PMS1~ }
.VAR    coefs[SECTIONS*4]="iircoefs.dat";
                                         { a12,a11,b12,b11,a22,a21,... }
.ENDSEG;

.SEGMENT /PM    rst_svc;                 { processor RESET service }
        jump begin;
.ENDSEG;

.SEGMENT /PM    tmzh_svc;                { timer interrupt service }
        jump new_sample;
.ENDSEG;

.SEGMENT /PM    pm_code;                 { selected by PMS0~ }
```

(listing continues on next page)

8 Programming Tutorial

initial setups:

```
begin:  pmwait = 0x0021;                { RAM = zero wait states }
        dmwait = 0xc401;                { RAM = zero wait states
                                        in_channel = ext. hardware ACK
                                        out_channel = 5 automatic waits }
        pmbank1=0x000800;               { first addr of pm bank 1 (PMS1~) }
        dmbank1=0x00001000;             { first addr of dm bank 1 (DMS1~) }
        dmbank2=0x00002000;             { first addr of dm bank 2 (DMS2~) }
        l0=0; l1=0; l8=0;
        m1=1; m8=1;
        call cascaded_biquad_init (db);
                                        { zero the delay line }
         r0=SECTIONS;
         b0=dline;
        tperiod=199;                    { 100 kHz sampling if CLKIN=20.0 MHz}
        tcount=199;
        bit set imask TMZHI;            { allow timer interrupt }
        bit set mode2 TIMEN;            { turn on timer }
        bit set mode1 IRPTEN;           { allow interrupts }
```

main processing loop:

```
wait:   idle;                           { wait for interrupts indefinitely }
        jump wait;                      { after rti, go wait for more }
```

interrupt service routine which does the filtering:

```
new_sample:
        f8=dm(in_channel);              { simulate this channel with a file }
        call cascaded_biquad (db);      { input=F8, output=F8 }
         b0=dline;
         b8=coefs;
        rti (db);
         dm(out_channel)=f8;            { simulate this channel with file }
         nop;
.ENDSEG;
```

Listing 8.4 iirirq.asm

```
{-------------------------------------------------------------------------
def21020.h - STATUS REGISTER BIT DEFINITIONS FOR ADSP-21020

This include file contains a list of "defines" to enable the programmer to
use symbolic names for all of the system register bits for the ADSP-21020.
-------------------------------------------------------------------------}

{ MODE1 register }
#define BR0      0x00000002 { Bit  1: Bit-reverse for I0 (uses DMS0- only ) }
#define SRCU     0x00000004 { Bit  2: Alt. register select for comp. units  }
#define SRD1H    0x00000008 { Bit  3: DAG1 alt. register select (7-4)       }
#define SRD1L    0x00000010 { Bit  4: DAG1 alt. register select (3-0)       }
#define SRD2H    0x00000020 { Bit  5: DAG2 alt. register select (15-12)     }
#define SRD2L    0x00000040 { Bit  6: DAG2 alt. register select (11-8)      }
#define SRRFH    0x00000080 { Bit  7: Register file alt. select for R(15-8) }
#define SRRFL    0x00000400 { Bit 10: Register file alt. select for R(7-0)  }
#define NESTM    0x00000800 { Bit 11: Interrupt nesting enable              }
#define IRPTEN   0x00001000 { Bit 12: Global interrupt enable               }
#define ALUSAT   0x00002000 { Bit 13: Enable ALU fixed-pt. saturation       }
#define TRUNC    0x00008000 { Bit 15: 1=fltg-pt. truncation 0=Rnd to nearest }
#define RND32    0x00010000 { Bit 16: 1=32-bit fltg-pt.rounding 0=40-bit rnd }

{ MODE2 register }
#define IRQ0E    0x00000001 { Bit  0: IRQ0- 1=edge sens. 0=level sens.      }
#define IRQ1E    0x00000002 { Bit  1: IRQ1- 1=edge sens. 0=level sens.      }
#define IRQ2E    0x00000004 { Bit  2: IRQ2- 1=edge sens. 0=level sens.      }
#define IRQ3E    0x00000008 { Bit  3: IRQ3- 1=edge sens. 0=level sens.      }
#define CADIS    0x00000010 { Bit  4: Cache disable                         }
#define TIMEN    0x00000020 { Bit  5: Timer enable                          }
#define FLG0O    0x00008000 { Bit 15: FLAG0 1=output 0=input                }
#define FLG1O    0x00010000 { Bit 16: FLAG1 1=output 0=input                }
#define FLG2O    0x00020000 { Bit 17: FLAG2 1=output 0=input                }
#define FLG3O    0x00040000 { Bit 18: FLAG3 1=output 0=input                }
#define CAFRZ    0x00080000 { Bit 19: Cache freeze                          }

{ ASTAT register }
#define AZ       0x00000001 { Bit  0: ALU result zero or fltg-pt. underflow }
#define AV       0x00000002 { Bit  1: ALU overflow                          }
#define AN       0x00000004 { Bit  2: ALU result negative                   }
#define AC       0x00000008 { Bit  3: ALU fixed-pt. carry                   }
#define AS       0x00000010 { Bit  4: ALU X input sign (ABS and MANT ops)   }
#define AI       0x00000020 { Bit  5: ALU fltg-pt. invalid operation        }
#define MN       0x00000040 { Bit  6: Multiplier result negative            }
#define MV       0x00000080 { Bit  7: Multiplier overflow                   }
#define MU       0x00000100 { Bit  8: Multiplier fltg-pt. underflow         }
#define MI       0x00000200 { Bit  9: Multiplier fltg-pt. invalid operation }
#define AF       0x00000400 { Bit 10: ALU fltg-pt. operation                }
#define SV       0x00000800 { Bit 11: Shifter overflow                      }
#define SZ       0x00001000 { Bit 12: Shifter result zero                   }
#define SS       0x00002000 { Bit 13: Shifter input sign                    }
#define BTF      0x00040000 { Bit 18: Bit test flag for system registers    }
#define FLG0     0x00080000 { Bit 19: FLAG0 value                           }
#define FLG1     0x00100000 { Bit 20: FLAG1 value                           }
```

(listing continues on next page)

8 Programming Tutorial

```
#define FLG2    0x00200000 { Bit 21: FLAG2 value                              }
#define FLG3    0x00400000 { Bit 22: FLAG3 value                              }
#define CACC0   0x01000000 { Bit 24: Compare Accumulation Bit 0               }
#define CACC1   0x02000000 { Bit 25: Compare Accumulation Bit 1               }
#define CACC2   0x04000000 { Bit 26: Compare Accumulation Bit 2               }
#define CACC3   0x08000000 { Bit 27: Compare Accumulation Bit 3               }
#define CACC4   0x10000000 { Bit 28: Compare Accumulation Bit 4               }
#define CACC5   0x20000000 { Bit 29: Compare Accumulation Bit 5               }
#define CACC6   0x40000000 { Bit 30: Compare Accumulation Bit 6               }
#define CACC7   0x80000000 { Bit 31: Compare Accumulation Bit 7               }

{ STKY register }
#define AUS     0x00000001 { Bit  0: ALU fltg-pt. underflow                   }
#define AVS     0x00000002 { Bit  1: ALU fltg-pt. overflow                    }
#define AOS     0x00000004 { Bit  2: ALU fixed-pt. overflow                   }
#define AIS     0x00000020 { Bit  5: ALU fltg-pt. invalid operation           }
#define MOS     0x00000040 { Bit  6: Multiplier fixed-pt. overflow            }
#define MVS     0x00000080 { Bit  7: Multiplier fltg-pt. overflow             }
#define MUS     0x00000100 { Bit  8: Multiplier fltg-pt. underflow            }
#define MIS     0x00000200 { Bit  9: Multiplier fltg-pt. invalid operation    }
#define CB7S    0x00020000 { Bit 17: DAG1 circular buffer 7 overflow          }
#define CB15S   0x00040000 { Bit 18: DAG2 circular buffer 15 overflow         }
#define PCFL    0x00200000 { Bit 21: PC stack full                            }
#define PCEM    0x00400000 { Bit 22: PC stack empty                           }
#define SSOV    0x00800000 { Bit 23: Status stack overflow (MODE1 and ASTAT)  }
#define SSEM    0x01000000 { Bit 24: Status stack empty                       }
#define LSOV    0x02000000 { Bit 25: Loop stack overflow                      }
#define LSEM    0x04000000 { Bit 26: Loop stack empty                         }

{ IRPTL and IMASK and IMASKP registers }
#define RSTI    0x00000002 { Bit  1: Address: 08: Reset                       }
#define SOVFI   0x00000008 { Bit  3: Address: 18: Stack overflow              }
#define TMZHI   0x00000010 { Bit  4: Address: 20: Timer = 0 (high priority)   }
#define IRQ3I   0x00000020 { Bit  5: Address: 28: IRQ3- asserted              }
#define IRQ2I   0x00000040 { Bit  6: Address: 30: IRQ2- asserted              }
#define IRQ1I   0x00000080 { Bit  7: Address: 38: IRQ1- asserted              }
#define IRQ0I   0x00000100 { Bit  8: Address: 40: IRQ0- asserted              }
#define CB7I    0x00000800 { Bit 11: Address: 58: Circ. buffer 7 overflow     }
#define CB15I   0x00001000 { Bit 12: Address: 60: Circ. buffer 15 overflow    }
#define TMZLI   0x00004000 { Bit 14: Address: 70: Timer = 0 (low priority)    }
#define FIXI    0x00008000 { Bit 15: Address: 78: Fixed-pt. overflow          }
#define FLTOI   0x00010000 { Bit 16: Address: 80: fltg-pt. overflow           }
#define FLTUI   0x00020000 { Bit 17: Address: 88: fltg-pt. underflow          }
#define FLTII   0x00040000 { Bit 18: Address: 90: fltg-pt. invalid            }
#define SFT0I   0x01000000 { Bit 24: Address: C0: user software int 0         }
#define SFT1I   0x02000000 { Bit 25: Address: C8: user software int 1         }
#define SFT2I   0x04000000 { Bit 26: Address: D0: user software int 2         }
#define SFT3I   0x08000000 { Bit 27: Address: D8: user software int 3         }
#define SFT4I   0x10000000 { Bit 28: Address: E0: user software int 4         }
#define SFT5I   0x20000000 { Bit 29: Address: E8: user software int 5         }
#define SFT6I   0x40000000 { Bit 30: Address: F0: user software int 6         }
#define SFT7I   0x80000000 { Bit 31: Address: F8: user software int 7         }
```

Listing 8.5 def21020.h

8.3.3.1 Initialization Following Reset (Initial Setups)

This section describes each of the initial setups.

Disabling interrupts. A processor reset automatically clears the IMASK register, effectively blocking any interrupts from interfering with instruction execution. There is nothing the programmer is required to do for this step.

Initializing the interrupt vector table. As in example #1, the beginning of program memory is used to store the interrupt vector table. Part of the initialized vector table can be found in Listing 8.4.

Altering the values in the memory hardware configuration registers. Memory wait states are configured by the PMWAIT and DMWAIT registers. Upon processor reset, these registers contain these default values:

PMWAIT: 0x0000 03DE
DMWAIT: 0x000F 7BDE

In the example system in Figure 8.5, all RAM can operate with zero wait states in both program memory and data memory spaces. The two I/O channels mapped into data memory require wait states, however. The input channel sends a hardware acknowledge when it is ready to end its bus (read) cycle; thus, the input device controls the number of wait states. The output channel functions properly with five wait states.

To set the wait state registers for this example system, execute the following instructions:

```
pmwait=0x0021; {RAM = no waits}
dmwait=0xC401; {RAM = no waits,
               in_channel = ext. hardware-generated ACK,
               out_channel = automatic 5 cycle waits}
```

The on-chip memory bank select decoding simplifies the hardware memory interface shown in Figure 8.5. Notice that three memory devices are required on the data memory side: a RAM storage area, an input device, and an output device. The two I/O channels could be A/D or D/A converters or buffers to a host computer bus, for example.

When the processor accesses a data memory location, one of four memory select lines is activated. Program memory space is likewise divided into two banks. Figure 8.4 shows the standard memory subdivision created by the initial values of the PMBANK and DMBANK registers after processor reset. In this example, the following lines of code modify this memory configuration to the one shown in Figure 8.6.

```
pmbank1 = 0x000800;      {change PMS1~ start address}
dmbank1 = 0x00001000;    {change DMS1~ start address}
dmbank2 = 0x00002000;    {change DMS2~ start address}
```

Initializing address and data registers. The same considerations presented in example #1 apply here. In this example, circular addressing is not used as reflected in the following code fragment:

```
l0=0; l1=0; l8=0;
m1=1; m8=1;
```

The index (I) registers and the base (B) registers are set in this example during the two instructions which immediately follow delayed call instructions to the subroutines which use the DAG registers.

Initializing memory locations and buffers. In the software development stage, RAM or ROM buffers in memory can be initialized by the assembler using directives such as:

```
.VAR cosine[256]   = "cos.dat";
.VAR list[4]       = 18.37, 1.0, -300.28769, 0.0;
```

In actual hardware, however, RAM sections cannot make use of these assembly-time initializations. The PROM Splitter creates files which PROM programmers use to initialize ROM memory. Emulation tools allow downloading initialized values to RAM memory. In the latter case, the initialization only occurs **once**—before processor reset. If the processor changes the initialized memory, subsequent reset operations would not re-initialize memory, causing the system to restart in a state that is different than the original one.

A good programming practice is to write code which reliably reinitializes your RAM memory buffers during the initial setup phase after processor reset. In this example, a subroutine is called to zero out the biquad filter delay element storage locations.

```
call cascaded_biquad_init (db);      {zero the delay line}
 r0=SECTIONS;
 b0=dline;
```

Here the B0 (and I0 automatically) registers are initialized as well.

Configuring and initializing on-chip peripherals. The timer on the ADSP-21020 is utilized in this example for creating the sampling interrupts which control the filtering operation. In our example, the ADSP-21020 is clocked at a 20.0 MHz rate. The timer is configured for a sampling interval such that the sampling frequency is 100 kHz. The TPERIOD register is set accordingly as well as TCOUNT.

```
tperiod=199;          {100 kHz intervals at 20.0 MHz CLKIN}
tcount=199;
```

The general formula for calculating the proper TPERIOD value is:

Interrupt Rate = CLKIN frequency/(TPERIOD+1)

An interrupt rate of 9.6 kHz, for example, with a 20 MHz CLKIN frequency requires a TPERIOD value of 0x822. TCOUNT is the register which decrements during every processor cycle, and TPERIOD holds the value which is automatically reloaded into TCOUNT when the timer expires (decrements to zero and causes an interrupt).

Configuring and enabling interrupts. Good programming practice dictates that all setup operations should conclude before interrupts are allowed to affect program execution. The last two setup tasks are to configure the interrupting scheme and then enable interrupts to be recognized. In this example, the only interrupt being used is from the timer, which controls the sampling rate of the filter.

Table 8.3 shows all five registers in the ADSP-21020 which affect interrupt configuration. Only some of the functions controlled by these registers are used in this example; the others are left in their default states. See the Interrupts section in Chapter 3 for complete information on these registers.

Name	*Function*
IMASK	which interrupts are to be recognized?
IMASKP	what to do in the case of interrupt nesting? (configured by processor automatically)
IRPTL	which interrupts have occurred?
MODE1	bit 12 turns interrupts on or off bit 11 turns interrupt nesting on or off
MODE2	bit 5 turns the interval timer on or off bits 0-3 set IRQ0-3 edge- or level-sensitive

Table 8.3 Interrupt-Related Registers

These registers are set in this example as follows:

```
bit set imask 0x10;
bit set mode2 0x20;
bit set mode1 0x1000;       {last initial setup}
```

Using standard definitions in the #include file called "def21020.h," shown in Listing 8.5, the bit positions specified by the values in these instructions translate to more readable bit names:

```
#include "def21020.h"       {place at top of file}
...
...
bit set imask TMZHI;
bit set mode2 TIMEN;
bit set mode1 IRPTEN;       {last initial setup}
```

The IMASK register is set in such a way to allow the timer to interrupt the processor. This register is automatically cleared to zero during a procesor reset. The timer has two different mask bits associated with it:

TMZHI (bit position 4 or 0x00000010)
TMZLI (bit position 14 or 0x00004000)

The timer interrupts are described in Chapter 5. You may select to either use the higher priority or the lower priority interrupt. The higher priority one was chosen in this example, but since no other interrupts are being used, either position could have been selected.

The timer is enabled by setting TIMEN to 1 in the MODE2 register. Once the timer is enabled, it automatically decrements the TCOUNT register once during every processor cycle. During this initial setup phase, the TCOUNT and the TPERIOD registers are typically set to the same value. This gives the processor some time to finish the last few setup instructions before going to the main loop and waiting there for interrupts. Keep in mind that as soon as you enable the timer, it begins to decrement on the next cycle.

The IRPTL register is where interrupt requests are latched and cleared. This register is unaffected by a processor reset, and conseqently it is the **programmer's responsibility** to clear this register before enabling interrupts. This can be done with any of the following equivalent instructions:

```
irptl = 0;
bit clr irptl 0xFFFFFFFF;
```

It is good programming practice to execute this instruction just before executing the instruction which enable interrupts (IRPTEN bit in the MODE1 register).

The MODE1 register has two bits (NESTM, IRPTEN) which impact interrupt operation. This register is automatically cleared to zero during a processor reset. The NESTM bit enables interrupt nesting. This example does not use nesting, so this bit is left unaltered after processor reset. The IRPTEN bit is the global interrupt enable bit. In order for the ADSP-21020 to service any interrupts whatsoever, this bit must be set. It is the **programmer's responsibility** to set this bit to a 1. It is good programming practice to only do so once all other initial setup operations are complete. The last instruction before the main processing loop section of code should be this:

```
bit set mode1 0x1000;
```

or

```
#include "def21020.h"
...
...
bit set mode1 IRPTEN;
```

8.3.3.2 Main Processing Loop

Having completed all the necessary initial setup operations, the program is ready to execute the main processing loop. The main processing loop typically consists of either:

- a list of tasks which eventually terminates, possibly with interrupt intervention, or
- an endless loop, waiting to be interrupted, in which most tasks are performed during interrupt service routines.

In this example, the second method is implemented. The endless loop consists of nothing more than:

```
wait: idle;
      jump wait;
```

This keeps the ADSP-21020 in an idle state, waiting for an interrupt to tell it to process the next sample. It is good programming practice to be executing an IDLE instruction while waiting for interrupts (without doing anything else) because this technique lowers the power consumption of the processor in a system. The IDLE instruction is described in Chapter 9.

The total power budget is calculated by summing the power dissipated in idle mode as well as the power dissipated in servicing interrupts. A shorter interrupt service routine means that a greater percentage of time is spent using less power.

8.3.3.3 Terminating The Main Processing Loop

This main processing loop runs indefinitely, without termination. To stop execution during simulation, however, open the input port simulation file and select Autowrap=NO as an option. This causes the simulator to stop when the end-of-file (EOF) is reached in the input file. More details on this follow.

8.3.4 Creating The Executable Program

The executable program for this system example is created using these commands to invoke the ADSP-21000 Family Assembler and Linker:

```
asm21k iirirq          NOTE: def21020.h must be in current directory
asm21k cascade
ld21k iirirq cascade -a iirirq -m
```

8.3.5 Simulation

This example system can be simulated using these commands to invoke the ADSP-21000 Family Simulator:

```
sim21k -e iirirq -a iirirq    NOTE: input.dat must be in current directory
```

In this example, the input samples are read from an I/O port simulation file. The file called "input.dat" is chosen as the input data. The contents of this file (see Listing 8.6) represent a normalized unit impulse function. Notice that the file can contain comments.

8 Programming Tutorial

Once inside the simulation session, open the ports, and use the Autowrap=NO option. After simulation is complete, the output file generated by writing to the simulated output port should contain the filter's impulse response function (see Listing 8.7).

```
1.000    This is the input data
0.000    for the biquads
0.000
0.000
...
...
0.000
0.000    (300 samples total)
```

Listing 8.6 Input Data Read by Input Port (Normalized Unit Impulse)

```
1.000000000
1.787410300
1.332763443
0.832507949
...
...
0.042204879
0.047202070    (300 samples total)
```

Listing 8.7 Output Data Stored by Output Port (Impulse Response)

8.4 CALLED SUBROUTINES (cascade.asm)

The two routines in the file "cascade.asm" perform the cascaded biquad IIR filtering operations:

cascaded_biquad_init	(clears delay line storage elements)
cascaded_biquad	(passes a sample through filter)

Each routine begins with a program memory label (*cascaded_biquad_init* or *cascaded_biquad*) and ends with an RTS instruction. These labels are global (declared with the `.GLOBAL` directive), which makes their names known to other files, for example, the main program which calls these subroutines. The other labels (*clear* and *quads*) remain unknown outside cascade.asm because they are not declared global. It is not possible to refer to them by name from another file.

The routines in cascade.asm demonstrate several important programming concepts, namely:

- Writing looped code
- "Rolling" loops for more efficient code
- Multifunction instructions and associated register restrictions

8.4.1 Writing Looped Code

Looped code is easily written using the nestable DO UNTIL construct. The advantage of using the DO UNTIL construct is that the ADSP-21020 automatically tests termination status and branches appropriately—without any programming or execution overhead. For counter-controlled loops, the ADSP-21020 even allows setting the loop counter register (LCNTR) in the same instruction cycle that the DO UNTIL instruction is executed. Of course, the loop can terminate on conditions other than LCE (loop counter expired), such as an arithmetic status flag. See Chapter 3 for more details and for a list of loop restrictions.

The R0 register is set by the calling program to tell the *cascaded_biquad* routine how many biquad sections to compute. For example, a sixth-order structure (which consists of three cascaded biquads) is computed if R0=3. The DO UNTIL loop is set up by the instruction:

```
lcntr=r0, do quads until lce;
```

The assembly source code within the loop is:

```
            f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
            f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
            f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
quads:      f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
```

Here is a cycle-by-cycle trace of the loop execution with r0=3:

```
1  lcntr=r0, do quads until lce;
2  f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
3  f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
4  f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
5  f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
6  f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
7  f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
8  f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
9  f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
10 f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
11 f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
12 f12=f2*f4,  f8=f8+f12,  f3=dm(i0,m1),  f4=pm(i8,m8);
13 f12=f3*f4,  f8=f8+f12,  dm(i1,m1)=f3,  f4=pm(i8,m8);
14 <next instruction after loop code>
15 <next instruction> , etc.
```

The above code is extremely efficient. Many resources are operating concurrently during every instruction cycle. Guidelines for efficiency in looped code are described in the following section.

8.4.2 Rolling Loops For More Efficient Code

"Rolling" a loop means pipelining operations to minimize instructions within a loop, exploiting the ADSP-21020's parallel architecture to maximize concurrent operations. This involves scheduling operations and adding some extra lines of code before and after the loop to "fill" and "drain" the pipeline. The basic rolled structure is shown in Figure 8.7.

pipe(1)						*loop prologue*
pipe(1)	pipe(2)					*loop prologue*
pipe(1)	pipe(2)	pipe(3)				*loop prologue*
pipe(1)	pipe(2)	pipe(3)	...			*loop prologue*
pipe(1)	pipe(2)	pipe(3)	...	pipe(n-1)		*loop prologue*
pipe(1)	**pipe(2)**	**pipe(3)**	**...**	**pipe(n-1)**	**pipe(n)**	***loop body (iterate here)***
	pipe(2)	pipe(3)	...	pipe(n-1)	pipe(n)	*loop epilogue*
		pipe(3)	...	pipe(n-1)	pipe(n)	*loop epilogue*
			...	pipe(n-1)	pipe(n)	*loop epilogue*
				pipe(n-1)	pipe(n)	*loop epilogue*
					pipe(n)	*loop epilogue*

Loop prologue	Instructions to fill the pipeline
Loop body	Instructions executed during looped steady state
Loop epilogue	Instructions to drain the pipeline

Figure 8.7 Filling and Draining the Pipeline

Figure 8.8 shows the instructions to be executed for the three biquad sections in this example. The operations are listed in chronological order and are vertically arranged according to the computation unit or memory bus used. When these operations are consolidated into multifunction instructions to match the model shown in Figure 8.7, the code in Listing 8.8 results.

```
1                f8=<input data>;
2   f12=0;
    *** begin first section ***
3                                f2=dm(i0,m1),   f4=pm(i8,m8);
4   f12=f2*f4,                   f3=dm(i0,m1),   f4=pm(i8,m8);
5   f12=f3*f4,   f8=f8+f12,      dm(i1,m1)=f3,   f4=pm(i8,m8);
6   f12=f2*f4,   f8=f8+f12,                      f4=pm(i8,m8);
7   f12=f3*f4,   f8=f8+f12,      dm(i1,m1)=f8;
8                f8=f8+f12;
    *** begin second section ***
9                                f2=dm(i0,m1),   f4=pm(i8,m8);
10  f12=f2*f4,                   f3=dm(i0,m1),   f4=pm(i8,m8);
11  f12=f3*f4,   f8=f8+f12,      dm(i1,m1)=f3,   f4=pm(i8,m8);
12  f12=f2*f4,   f8=f8+f12,                      f4=pm(i8,m8);
13  f12=f3*f4,   f8=f8+f12,      dm(i1,m1)=f8;
14               f8=f8+f12;
    *** begin third section ***
15                               f2=dm(i0,m1),   f4=pm(i8,m8);
16  f12=f2*f4,                   f3=dm(i0,m1),   f4=pm(i8,m8);
17  f12=f3*f4,   f8=f8+f12,      dm(i1,m1)=f3,   f4=pm(i8,m8);
18  f12=f2*f4,   f8=f8+f12,                      f4=pm(i8,m8);
19  f12=f3*f4,   f8=f8+f12,      dm(i1,m1)=f8;
20               f8=f8+f12;
21               <output data>=f8;
```

Figure 8.8 Loop Code Before Rolling

8.4.3 Multifunction Instructions And Register Restrictions

The 48-bit wide instruction word provides great single-cycle flexibility and parallelism in the ADSP-21020 architecture. For example, the ALU, multiplier, program sequencer and two separate address generators can all function simultaneously on a wide selection of input and output registers. There are, however, tradeoffs to be made when several blocks are simultaneously active. For example, multifunction instructions require indirect addressing using I and M registers because the instruction word is not wide enough to accommodate multiple instructions and direct, immediate address or modify amounts.

The multiport register file (F0-F15 or R0-R15) can normally be read from and written to without restriction; however, in multifunction instructions, the ALU and multiplier inputs are restricted to particular sets of registers, while the outputs are unrestricted. The architecture dictates that when ALU and multiply operations are concurrent, the multiplier X-input may be either F0, F1, F2 or F3 while the multiplier Y-input is chosen from F4, F5, F6 or F7. The ALU X-input may be F8, F9, F10 or F11 while the Y-input is chosen from F12, F13, F14 or F15. In floating-point multiply/accumulates, the destination of the ALU is typically the same register as one of its input registers (i.e., the previous accumulated total).

In the *quads* loop in Listing 8.8, the register restrictions for multifunction instructions do not deter efficient computation. Note also that for ease of programming and legibility, the pipes of the multifunction instructions are vertically aligned.

```
cascaded_biquad:

        b1=b0;
        r8=r8 xor r8, f2=dm(i0,m1), f4=pm(i8,m8);
        lcntr=r0, do quads until lce;
          f12=f2*f4, f8=f8+f12, f3=dm(i0,m1), f4=pm(i8,m8);
          f12=f3*f4, f8=f8+f12, dm(i1,m1)=f3, f4=pm(i8,m8);
          f12=f2*f4, f8=f8+f12, f2=dm(i0,m1), f4=pm(i8,m8);
quads:    f12=f3*f4, f8=f8+f12, dm(i1,m1)=f8, f4=pm(i8,m8);
        rts (db),    f8=f8+f12;
         nop;
         nop;
```

Listing 8.8 Final Rolled Loop Example in "cascade.asm"

8.5 DEVELOPING THE IIR FILTER AND COEFFICIENTS

The IIR filter can be developed using computer-aided filter design software. In this example, FDAS (Filter Design and Analysis Software), which is a product of Momentum Data Systems, was used. The file created by FDAS is shown in Listing 8.9. This file lists the filter specifications input by the user, as well as the filter coefficients computed by FDAS.

```
FILTER COEFFICIENT FILE
IIR DESIGN
FILTER TYPE                 BAND PASS
ANALOG FILTER TYPE          ELLIPTIC
PASSBAND RIPPLE IN -dB            -.1000
STOPBAND RIPPLE IN -dB           -1.0000
PASSBAND CUTOFF FREQUENCIES  .400000E+03  .500000E+03 HERTZ
STOPBAND CUTOFF FREQUENCIES  .300000E+03  .600000E+03 HERTZ
SAMPLING FREQUENCY           .800000E+04 HERTZ
FILTER DESIGN METHOD: BILINEAR TRANSFORMATION
FILTER ORDER            6 0006h
NUMBER OF SECTIONS      3 0003h
NO. OF QUANTIZED BITS  32 0020h
QUANTIZATION TYPE - FLOATING POINT
COEFFICIENTS SCALED FOR FLOATING POINT IMPLEMENTATION
  .67730926E-02        /* overall gain                    */
  .00000000            /* section  1 coefficient B1       */
 -1.0000000            /* section  1 coefficient B2       */
  1.8039191            /* section  1 coefficient A1       */
 -.92128010            /* section  1 coefficient A2       */
 -1.7640328            /* section  2 coefficient B1       */
  1.0000000            /* section  2 coefficient B2       */
  1.8060702            /* section  2 coefficient A1       */
 -.96266572            /* section  2 coefficient A2       */
 -1.9376569            /* section  3 coefficient B1       */
  1.0000000            /* section  3 coefficient B2       */
  1.8791107            /* section  3 coefficient A1       */
 -.97108089            /* section  3 coefficient A2       */
```

Listing 8.9 FDAS File

8.5.1 Normalized b_0 Coefficient Biquad Filter Design Method

Many digital filter design techniques exist for determining filter coefficients. To minimize coefficient quantization effects, IIR filters of high order are usually implemented as cascaded biquad sections. Each biquad section requires five filter coefficients, three feedforward and two feedback. These sections can be in normal order or transposed order. Refer to texts on digital filters for further information.

The dynamic range offered by floating-point numbers allows us to normalize the coefficients in this example to the b_0 coefficient. FDAS offers b_0 normalization in its menu of filter design techniques. The value of b_0 becomes unity (1.0) and therefore any multiplication by this coefficient does not have to be carried out. The biquad rewritten without b_0 multiplication results in a four-coefficient, four-instruction-cycle per biquad routine.

8.5.2 DSP Code Generation

Filtering routines are general-purpose; only coefficients and the length of the coefficient and delay-line buffers need to change for different filters. Buffer lengths are modified by changing the #define preprocessor directives, and reassembling and relinking the source code. The filter coefficients themselves are changed by simply extracting, using any standard text editor, the floating-point coefficient values from the file which FDAS generates, placing them in another file that the source code references for coefficient buffer initialization. An alternative is to simply delete all lines in the FDAS file which are not coefficient values. In that case, make sure the filename is the one which the ADSP-21020 source code references for coefficient buffer initialization. For example, Listing 8.2 refers to "iircoefs.dat."

8.5.3 Coefficient Formatting

The coefficients in the coefficient buffer are initialized in the ADSP-21020 source code with the .VAR assembler directive. Note that this directive not only defines the buffer, but also initializes its contents with the values in the specified file, "iircoefs.dat."

```
.var coefs[SECTIONS*4] = "iircoefs.dat";
```

The contents of "iircoefs.dat" are shown in Listing 8.10. This file was created by editing the FDAS file shown in Listing 8.9.

```
 -.92128010
  1.8039191
 -1.0000000
  .00000000
 -.96266572
  1.8060702
  1.0000000
 -1.7640328
 -.97108089
  1.8791107
  1.0000000
 -1.9376569
```

Listing 8.10 "iircoefs.dat" File

When saving the filter coefficients in FDAS, select the maximum allowable bits per coefficient (i.e., the least amount of quantization error). The floating-point coefficient values are stored in ASCII representation and read in the ADSP-21020 initialization in ASCII representation. For this reason, it is a good idea to use as many ASCII digits as possible.

8.6 PROGRAMMING HINTS

This section describes good and poor programming practices. Strict adherence to these suggestions is not required but is strongly recommended.

See Chapter 7 for a list of programming reminders and restrictions, as well as an overview of the instruction set.

8.6.1 System Considerations For Scoping

The scope of variables and code labels refers to where they are declared and which portions of software know of their existence. Variables and code labels can be global or local in scope. They must always be declared within a .SEGMENT definition. For example:

Do This:

```
.SEGMENT /DM        dm_data;
.VAR                inbuf [SAMPLES];
.VAR                outbuf [SAMPLES];
.ENDSEG;
```

By default, a .VAR declaration forces this variable to remain local within the file in which it is declared. To make a variable or code label global, it must be declared global using the .GLOBAL directive in the same file in which the variable or code label is originally declared. Any other file must use the .EXTERN directive to make a global variable known to it.

The context in which a subroutine is used can suggest how to scope the variables it references. For example, routines such as the IIR filtering ones shown here may be used to implement many different filters. It may even be included in a library of general-purpose routines. In such a case, the variables the routine uses (*coefs* and *dline*) should be declared in the main routine (iirmem.asm or iirirq.asm) and not in the called routine (cascade.asm). This is because each filter has a unique set of coefficients and delay line storage.

Notice that the example subroutines in "cascade.asm" do not refer to *coefs* or *dline* by name. For that reason, it is not necessary (nor desirable) to use the .GLOBAL or .EXTERN directives. The called subroutines only need to know the start addresses of the *coefs* and *dline* buffers. These addresses are passed to the subroutines by assignments to B or I registers in the calling program. The routine then uses the I registers for data addressing. This example shows those assignments being performed as the two instructions executed following a delayed branch call to the subroutines.

Main Code:

```
.VAR coefs1 [SECTIONS*4] = "iir1.dat";
.VAR coefs2 [SECTIONS*4] = "iir2.dat";
.VAR coefs3 [SECTIONS*4] = "iir3.dat";

.VAR dline1 [SECTIONS*2];
.VAR dline2 [SECTIONS*2];
.VAR dline3 [SECTIONS*2];
...
...
...
call readem (db);
   b0=dline1;                          {loads i0 register}
   b8=coefs1;                          {loads i8 register}
```

Subroutine:

```
...
dm(i1,m1)=f3, f4=pm(i8,m8);            {i0, i8 used}
...                                    {no names referenced}
...
rts;
```

This approach has two advantages. It is simple and quick to edit the main calling code only, allowing the subroutine to be general purpose. Also, less overhead is incurred when branching to the subroutine using a delayed branch. The two extra cycles following the delayed CALL instruction are conveniently employed to pass the coefficient and delay line buffer addresses to the subroutine.

For example:

Do This:

```
call cascaded_biquad (db);
 b0=dline1;
 b8=coefs1;
```

{takes 3 cycles}

Not This:

```
b0=dline1;
b8=coefs1;
call cascaded_biquad;
```

{takes 5 cycles}

The former shows a delayed branch, the latter depicts a normal branch. The delayed branch saves two instruction cycles during execution. To use the faster delayed branch subroutine call, simply take two instructions from before the call and place them immediately after the delayed call. Do not use any of the restricted instructions such as branches or looping constructs as these two. See Chapter 3 for details on delayed branching.

8.6.2 Delayed Branches

Understanding delayed branches requires a non-intuitive leap. Instructions *following* a branch instruction get executed *before* program flow continues at the branch destination. For this reason, a good programming practice is to highlight these two instructions. In the same way that loop nesting is emphasized by indentation, we use indentation to highlight the two instructions following a delayed branch instruction. This convention is not confused with loop indentation in this example, because this example uses only a single space indentation. When reading code, the space reminds you that these instructions execute *before* the branch is taken.

The two instructions after a delayed branch can be used to pass parameters to the destination code branched to, especially when the delayed branch is a subroutine call. If the branch is a delayed return from a subroutine or interrupt, the two instructions associated with the delayed branch may actually finish up the subroutine or interrupt service tasks.

Here is an example code fragment showing indentation used for nested loops as well as for delayed branches. Note that for loops and delayed branches, a different amount of indentation distinguishes from the other.

```
#define DATASETS        4
#define SAMPLES         300

         ...
         lcntr=DATASETS, do sets until lce;
            lcntr=SAMPLES, do filtering until lce;
               f8=dm(i3,1);
               call biquad (db);
                b0=dline;  {executes BEFORE biquad routine begins}
                b8=coefs;  {executes BEFORE biquad routine begins}
filtering:     dm(i4,1)=f8;
sets:       nop;
         ...
```

8.6.3 Multifunction Instruction Coding

Multifunction instructions which activate the ALU, the muliplier and the two DAGs simultaneously can be quite lengthy. To graphically depict operations in progress as well as resource utilization, it is good programming practice to write sequential multifunction instructions in such a way that the parts of the multifunction instructions that use the same resource (e.g., ALU operations) line up vertically. If the instruction does not use a resource, the space for that resource is left blank. The *quads* loop in Listing 8.8 in the *cascaded_biquad* subroutine shows this vertical alignment.

This graphically shows resource utilization and, more importantly for hand code compaction, resource *non*-utilization. The programmer may find a way later to fill those blank spaces (i.e., use the unused resource) and reduce the total instruction count. This is described in an earlier section, "Rolling Loops."

8.7 COMPLETE FFT EXAMPLE

In addition to the two IIR examples presented in this chapter, an FFT program is presented in this section to show a larger example. This program features efficient memory usage in conjunction with fast computational throughput.

```
{_____________________________________________________________________
FFTRAD4.ASM      ADSP-21020 Radix-4 Complex Fast Fourier Transform

This routine performs a complex, radix 4 Fast Fourier Transform (FFT). The FFT
length (N) must be a power of 4 and a minimum of 64 points. The real part of
the input data is placed in DM and the complex part in PM. This data is
destroyed during the course of the computation. The real and complex output of
the FFT is placed in separate locations in DM.

Since this routine takes care of all necessary address digit-reversals, the
input and output data are in normal order. The digit reversal is accomplished
by using a modified radix 4 butterfly throughout which swaps the inner two
nodes resulting with bit reversed data. The digit reversal is completed by
bit reversing the real data in the final stage and then bit reversing the
imaginary so that it ends up in DM.

To implement an inverse FFT, you only have to (1) swap the incoming data, real
and imaginary parts, (2) run the forward FFT, and (3) swap the outgoing data,
real and imaginary parts.

For this routine to work correctly, the program "twidrad4.C" must be used to
generate the special twiddle factor tables for this program.

Author:     Karl Schwarz & Raimund Meyer, Universitaet Erlangen Nuernberg
Revision:   27-MAR-91, Ronnin Yee, Analog Devices, DSP div., (617) 461-3672

Calling Information:
    costwid table at DM : cosine    length 3*N/4
    sintwid table at PM : sine      length 3*N/4
    real  input  at DM : redata     length N, normal order
    imag  input  at PM : imdata     length N, normal order

Results:
    real  output at DM : refft      length N, normal order
    imag  output at DM : imfft      length N, normal order

    (Note:  Because the bit reversed addressing mode is used with the arrays
    refft and imfft, they must start at addresses that are integer
    multiples of the length (N) of the transform, (i.e. 0,N,2N,3N,...).
    This is accomplished by specifying two segments starting at those addresses
    in the architecture file and placing the variables alone in their
    respective segments. These addresses must also be reflected in the
    preprocessor variables ORE and OIM in bit reversed format.)

Altered Registers:
    All I, M, L and R registers.
    Three levels of looping.
```

(listing continues on next page)

8 Programming Tutorial

Benchmarks: radix-4, complex with digit reversal

FFT Length	cycles	ms @ 20 MHz CLK	ms @ 25 MHz CLK
64	920	.046	.037
256	4044	.202	.162
1024	19245	.962	.770
4096	90702	4.535	3.628
16384	419434	20.972	16.777

First Stage - 8 cycles per radix-4 butterfly
Other Stages - 14 cycles per radix-4 butterfly

Memory Usage:
pm code = 192 words, pm data = 1.75*N words, dm data = 3.75*N words

Assembler Preprocessor Variables:
N Number of points in FFT. Must be a power of four, minimum of 64.
STAGES Set to log4(N) or (log(N)/log(4))
OST = bitrev(32 bit N/2)
ORE = bitrev(32 bit addr of output real in dm), addr is 0,N,2N,3N,...
OIM = bitrev(32 bit addr of output imag. in dm), addr is 0,N,2N,3N,...

```
{ include for symbolic definition of system regster bits }
#include "def21020.h"

{_______The constants below must be changed for different length FFTs_______
 N      = number of points in the FFT
 STAGES = log4(N)
 OST    = bitrev(0x00000080=N/2), used as a modifier for bit reversal
 ORE    = bitrev(0x00000000=output real in dm)
 OIM    = bitrev(0x00004000=output imag in dm)
_____________________________________________________________________________}
#define N       256
#define STAGES  4
#define OST     0x01000000
#define ORE     0x00000000
#define OIM     0x00020000

.SEGMENT/DM             dm_data;
.VAR    cosine[3*N/4]="tc.dat"; {Cosine twiddle factors, from TWIDRAD4 program}
.VAR    redata[N]="inreal.dat"; { Input real data }
.GLOBAL redata;
.ENDSEG;

.SEGMENT/DM             dm_rdat; { this segment is an integer multiple of N }
.VAR    refft[N];               { Output real data }
.GLOBAL refft;
.ENDSEG;
```

```
.SEGMENT/DM             dm_idat; { this segment is an integer multiple of N }
.VAR    imfft[N];                { Output imaginary data }
.GLOBAL imfft;
.ENDSEG;

.SEGMENT/PM             pm_data;
.VAR    sine[3*N/4]="ts.dat";   { Sine twiddle factors, from TWIDRAD4 program}
.VAR    imdata[N]="inimag.dat"; { Input imaginary data }
.GLOBAL imdata;
.ENDSEG;

.SEGMENT/PM             rst_svc; { program starts at the reset vector }
        pmwait=0x0021;          {pgsz=0,pmwtstates=0,intrn.wtstates only}
        dmwait=0x008421;        {pgsz=0,dmwtstates=0,intrn.wtstates only}
        call    fft;
stop:   idle;
        nop;
        nop;
.ENDSEG;

.SEGMENT/PM             pm_code;
fft:
{_______first stage radix-4 butterfly without twiddles_____}
                        i0=redata;
                        i1=redata+N/4;
                        i2=redata+N/2;
                        i3=redata+3*N/4;
                        i4=i0;
                        i5=i1;
                        i6=i2;
                        i7=i3;
                        m0=1;
                        m8=1;
                        i8=imdata;
                        i9=imdata+N/4;
                        i10=imdata+N/2;
                        i11=imdata+3*N/4;
                        i12=i8;
                        i13=i9;
                        i14=i10;
                        i15=i11;
                        l0 = 0;
                        l1 = l0;
                        l2 = l0;
                        l3 = l0;
                        l4 = l0;
                        l5 = l0;
                        l6 = l0;
                        l7 = l0;
                        l8 = l0;
```

(listing continues on next page)

```
                        l9 = l0;
                        l10 = l0;
                        l11 = l0;
                        l12 = l0;
                        l13 = l0;
                        l14 = l0;
                        l15 = l0;
                                                    f0=dm(i0,m0),    f1=pm(i8,m8);
                                                    f2=dm(i2,m0),    f3=pm(i10,m8);
                f0=f0+f2,       f2=f0-f2,           f4=dm(i1,m0),    f5=pm(i9,m8);
                f1=f1+f3,       f3=f1-f3,           f6=dm(i3,m0),    f7=pm(i11,m8);
                f4=f6+f4,       f6=f6-f4;
                f5=f5+f7,       f7=f5-f7;
                f8=f0+f4,       f9=f0-f4;
                f10=f1+f5,      f11=f1-f5;

lcntr=N/4,      do fstage until lce;    { do N/4 simple radix-4 butterflies }
                f12=f2+f7,      f13=f2-f7,          f0=dm(i0,m0),    f1=pm(i8,m8);
                f14=f3+f6,      f15=f3-f6,          f2=dm(i2,m0),    f3=pm(i10,m8);
                f0=f0+f2,       f2=f0-f2,           f4=dm(i1,m0),    f5=pm(i9,m8);
                f1=f1+f3,       f3=f1-f3,           f6=dm(i3,m0),    f7=pm(i11,m8);
                f4=f6+f4,       f6=f6-f4,           dm(i4,m0)=f8,    pm(i12,m8)=f10;
                f5=f5+f7,       f7=f5-f7,           dm(i5,m0)=f9,    pm(i13,m8)=f11;
                f8=f0+f4,       f9=f0-f4,           dm(i6,m0)=f12,   pm(i14,m8)=f14;
fstage:
                f10=f1+f5,      f11=f1-f5,          dm(i7,m0)=f13,   pm(i15,m8)=f15;

{_____________Middle stages with radix-4 main butterfly_____________________}

{ m0=1 and m8=1 is still preset }
                        m1=-2;          { reverse step for twiddles }
                        m9=m1;
                        m2=3;           { forward step for twiddles }
                        m10=m2;
                        m5=4;           { first there are 4 groups }
                        r2=N/16;        { with N/16 butterflies in each group }
                        r3=N/16*3;      { step to next group }

lcntr=STAGES-2, do mstage until lce;    { do STAGES-2 stages }

                        i7=cosine;      { first real twiddle }
                        i15=sine;       { first imag twiddle }

                        r8=redata;
                        r9=imdata;

                        i0=r8;          { upper real      path }
        r10=r8+r2;      i8=r9;          { upper imaginary path }
                        i1=r10;         { second real input path }
        r10=r10+r2,     i4=r10;         { second real output path }
```

```
                    i2=r10;          { third real input path }
        r10=r10+r2,     i5=r10;      { third real output path }
                    i3=r10;          { fourth real input path }
        r10=r9+r2,      i6=r10;      { fourth real output path }
                    i9=r10;          { second imag input path }
        r10=r10+r2,     i12=r10;     { second imag output path }
                    i10=r10;         { third imag input path }
        r10=r10+r2,     i13=r10;     { third imag output path }
                    i11=r10;         { fourth imag input path }
                    i14=r10;         { fourth imag output path }
                    m4=r3;
                    m12=r3;
        r4=r3+1,    m6=r2;
                    m3=r4;
        r2=r2-1,    m11=r4;
                    m7=r2;

lcntr=m5,       do mgroup until lce;    { do m5 groups }
                                            f0=dm(i7,m0),   f5=pm(i9,m8);
f8=f0*f5,                                   f4=dm(i1,m0),   f1=pm(i15,m8);
f9=f0*f4;
f12=f1*f5,                                  f0=dm(i7,m0),   f5=pm(i11,m8);
f13=f1*f4,      f12=f9+f12,                 f4=dm(i3,m0),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13;
f13=f1*f5;
f9=f0*f5,       f8=f8+f13,                  f0=dm(i7,m1),   f5=pm(i10,m8);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12, f4=dm(i2,m0),   f1=pm(i15,m9);
f11=f0*f4;
f13=f1*f5,                      f6=f9-f13;
f9=f0*f5,       f13=f11+f13,                f11=dm(i0,0);
f13=f1*f4,      f8=f11+f13,     f10=f11-f13;
          {___________Do m7 radix-4 butterflies___________}
lcntr=m7,       do mr4bfly until lce;
                                f13=f9-f13, f4=dm(i1,m0),   f5=pm(i9,m8);
                f2=f2+f6,       f15=f2-f6,  f0=dm(i7,m0),   f1=pm(i15,m8);
f8=f0*f4,       f3=f8+f12,      f7=f8-f12,                  f9=pm(i8,0);
f12=f1*f5,      f9=f9+f13,      f11=f9-f13, f13=f2;
f8=f0*f5,       f12=f8+f12,                 f0=dm(i7,m0),   f5=pm(i11,m8);
f13=f1*f4,      f9=f9+f13,      f6=f9-f13,  f4=dm(i3,m0),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13,  dm(i0,m0)=f3,   pm(i8,m8)=f9;
f13=f1*f5,      f11=f11+f14,    f7=f11-f14, dm(i4,m0)=f7,   pm(i12,m8)=f6;
f9=f0*f5,       f8=f8+f13,                  f0=dm(i7,m1),   f5=pm(i10,m8);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12, f4=dm(i2,m0),   f1=pm(i15,m9);
f11=f0*f4,      f3=f10+f15,     f8=f10-f15,                 pm(i13,m8)=f11;
f13=f1*f5,                      f6=f9-f13,  dm(i6,m0)=f8,   pm(i14,m8)=f7;
f9=f0*f5,       f13=f11+f13,                f11=dm(i0,0);
mr4bfly:
f13=f1*f4,      f8=f11+f13,     f10=f11-f13,dm(i5,m0)=f3;
          {___________End radix-4 butterfly_____________}
{                   dummy for address update      *              * }
```

(listing continues on next page)

8 Programming Tutorial

```
                                f13=f9-f13,     f0=dm(i7,m2),   f1=pm(i15,m10);
                f2=f2+f6,       f15=f2-f6,      f0=dm(i1,m4),   f1=pm(i9,m12);
                f3=f8+f12,      f7=f8-f12,                      f9=pm(i8,0);
                f9=f9+f13,      f11=f9-f13,     f0=dm(i2,m4);
                f9=f9+f2,       f6=f9-f2,       f0=dm(i3,m4),   f1=pm(i10,m12);
                                                dm(i0,m3)=f3,   pm(i8,m11)=f9;
                f11=f11+f14,    f7=f11-f14,     dm(i4,m3)=f7,   pm(i12,m11)=f6;
                f3=f10+f15,     f8=f10-f15,                     pm(i13,m11)=f11;
                                                dm(i6,m3)=f8,   pm(i14,m11)=f7;
mgroup:                                         dm(i5,m3)=f3,   f1=pm(i11,m12);

                                r3=m4;
                                r1=m5;
                                r2=m6;
        r3=ashift r3 by -2;             { groupstep/4 }
        r1=ashift r1 by 2;              { groups*4 }
                                m5=r1;
mstage: r2=ashift r2 by -2;             { butterflies/4 }

{_____________________Last radix-4 stage_____________________________________}
{ Includes bitreversal of the real data in dm }

        bit set mode1 BR0;              { bitreversal in i0 }
{ with: m0=m8=1 preset }
        i4=redata;                      { input }
        i1=redata+1;
        i2=redata+2;
        i3=redata+3;
        i0=ORE;  { real output array base must be an integer multiple of N }
        m2=OST;
        i7=cosine;
        i8=imdata;                      { input }
        i9=imdata+1;
        i10=imdata+2;
        i11=imdata+3;
        i12=imdata;                     { output }
        i15=sine;
        m1=4;
        m9=m1;

                                                        f0=dm(i7,m0),   f5=pm(i9,m9);
f8=f0*f5,                                               f4=dm(i1,m1),   f1=pm(i15,m8);
f9=f0*f4;
f12=f1*f5,                                              f0=dm(i7,m0),   f5=pm(i11,m9);
f13=f1*f4,      f12=f9+f12,                             f4=dm(i3,m1),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13;
f13=f1*f5;
f9=f0*f5,       f8=f8+f13,                              f0=dm(i7,m0),   f5=pm(i10,m9);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12,             f4=dm(i2,m1),   f1=pm(i15,m8);
f11=f0*f4;
f13=f1*f5,                      f6=f9-f13;
```

```
f9=f0*f5,       f13=f11+f13,                    f11=dm(i4,m1);
f13=f1*f4,      f8=f11+f13,     f10=f11-f13;
          {________Do N/4-1 radix-4 butterflies_______}
lcntr=N/4-1, do lstage until lce;
                                f13=f9-f13,     f4=dm(i1,m1),   f5=pm(i9,m9);
                f2=f2+f6,       f15=f2-f6,      f0=dm(i7,m0),   f1=pm(i15,m8);
f8=f0*f4,       f3=f8+f12,      f7=f8-f12,                      f9=pm(i8,m9);
f12=f1*f5,      f9=f9+f13,      f11=f9-f13,     f13=f2;
f8=f0*f5,       f12=f8+f12,                     f0=dm(i7,m0),   f5=pm(i11,m9);
f13=f1*f4,      f9=f9+f13,      f6=f9-f13,      f4=dm(i3,m1),   f1=pm(i15,m8);
f8=f0*f4,                       f2=f8-f13,      dm(i0,m2)=f3,   pm(i12,m8)=f9;
f13=f1*f5,      f11=f11+f14,    f7=f11-f14,     dm(i0,m2)=f7,   pm(i12,m8)=f6;
f9=f0*f5,       f8=f8+f13,                      f0=dm(i7,m0),   f5=pm(i10,m9);
f13=f1*f4,      f12=f8+f12,     f14=f8-f12,     f4=dm(i2,m1),   f1=pm(i15,m8);
f11=f0*f4,      f3=f10+f15,     f8=f10-f15,                     pm(i12,m8)=f11;
f13=f1*f5,                      f6=f9-f13,      dm(i0,m2)=f3,   pm(i12,m8)=f7;
f9=f0*f5,       f13=f11+f13,                    f11=dm(i4,m1);
lstage:
f13=f1*f4,      f8=f11+f13,     f10=f11-f13,    dm(i0,m2)=f8;

                                f13=f9-f13;
                f2=f2+f6,       f15=f2-f6;
                f3=f8+f12,      f7=f8-f12,                      f9=pm(i8,m9);
                f9=f9+f13,      f11=f9-f13,     dm(i0,m2)=f3;
                f9=f9+f2,       f6=f9-f2,       dm(i0,m2)=f7;
                                                                pm(i12,m8)=f9;
                f11=f11+f14,    f7=f11-f14,                     pm(i12,m8)=f6;
                f3=f10+f15,     f8=f10-f15,                     pm(i12,m8)=f11;
                                                dm(i0,m2)=f3,   pm(i12,m8)=f7;
                                                dm(i0,m2)=f8;

{________Do the bitreversal of the imaginary part from pm to dm__________}
        i8=imdata;
        i0=OIM;  { image output array base must be an integer multiple of N }
        f0=pm(i8,m8);

        lcntr=N-1, do pmbr until lce;    { do N-1 bitreversals }
pmbr:     dm(i0,m2)=f0, f0=pm(i8,m8);

        rts (db);
        dm(i0,m2)=f0;
        bit clr mode1 BR0;               { no bitreversal in i0 any more }
.ENDSEG;
```

Hardware System Design 9

9.1 OVERVIEW

This chapter describes considerations for designing hardware systems based on the ADSP-21020/21010 processor. It also supplies examples of some common configurations.

9.1.1 Basic System Configuration

Figure 9.1, on the following page, shows a basic configuration for a system based on the ADSP-21020. The following two signals coordinate the operation of the ADSP-21020 and other devices in the system:

- The CLKIN signal comes from a clock oscillator that provides clocking for the ADSP-21020 and other devices operating synchronously with it.
- The $\overline{\text{RESET}}$ signal is provided by a circuit that resets all or part of the system.

These signals are described in greater detail in later sections of this chapter.

The basic configuration in Figure 9.1 features program memory, data memory and peripherals that are mapped into data memory space. The connections in each memory interface are:

- Address buses (PMA23-0 and DMA31-0)
- Data buses (PMD47-0 and DMD39-0)
- Bank selects (PMS1-0 and DMS3-0)
- Read signals ($\overline{\text{PMRD}}$ and $\overline{\text{DMRD}}$)
- Write signals ($\overline{\text{PMWR}}$ and $\overline{\text{DMWR}}$)

Example memory configurations for both single and multiple processors are shown later in this chapter.

9 Hardware System Design

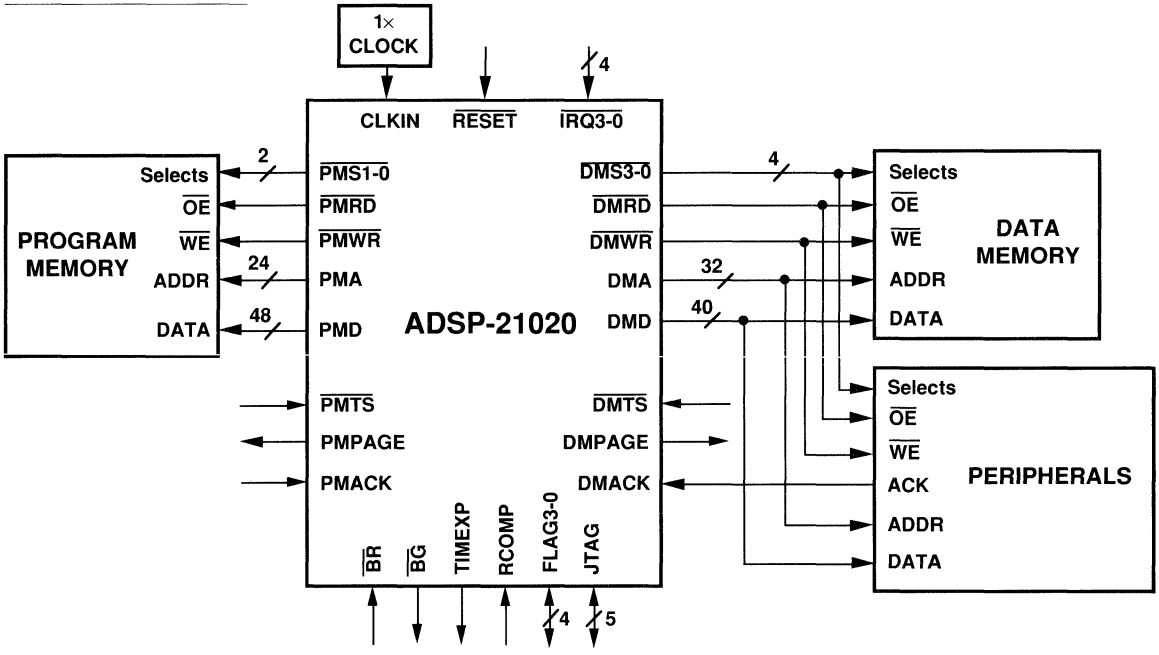


Figure 9.1 Basic ADSP-21020 System Configuration

9.1.2 More Complex Configurations

Other signals shown in Figure 9.1 but not connected in this configuration could be used in a system with more complex memory interface requirements:

- Bus acknowledges (PMACK or DMACK), for hardware-controlled wait states.
- Three-state enables ($\overline{\text{PMTS}}$ or $\overline{\text{DMTS}}$), to hold the processor off the memory bus during an external cache update (for example).
- Page fault indicators (PMPAGE or DMPAGE), for interfacing to page-mode and static-column dynamic RAMs (DRAMs).
- Bus request ($\overline{\text{BR}}$) and bus grant ($\overline{\text{BG}}$), for granting the memory buses and control signals to another master.

There are pins through which the ADSP-21020 sends and receives control signals to and from other devices in the system. These pins may or may not be used, depending on the system:

- Hardware interrupts ($\overline{IRQ3-0}$) can come from devices that require the ADSP-21020 to perform some task on demand. One of the memory-mapped peripherals, for example, can use an interrupt to alert the processor that it has data available. Interrupts are described in detail in Chapter 3.

- The flags (FLAG3-0), each of which can be programmed to be an input or an output, allow signalling between the ADSP-21020 and another device. For example, the ADSP-21020 can raise an output flag to interrupt some other device. Flags are described in detail later in this chapter.

- The TIMEXP output is controlled by the on-chip timer. It indicates to other devices that the programmed time period has expired. The timer is described in detail in Chapter 5.

- The test access port (TCK, TMS, TDI, TDO and $\overline{TRST}$) can be connected to a controller that performs a boundary scan for test purposes or for powerup boot loading of external program memory. This port is also used by the ADSP-21020 EZ-ICE® Emulator to access on-chip emulation features. Use of this emulator requires a connector for access to the test access port. The connector is described in this chapter, in section 9.9. The test access port is described in detail in Appendix C.

9.2 CLOCKS & SYNCHRONIZATION

The ADSP-21020 receives its clock input on the CLKIN pin. The processor uses an on-chip phase-locked loop to generate its internal clock.

Because the phase-locked loop requires some time to achieve phase lock, CLKIN must be valid for a minimum time period during reset before the RESET signal can be deasserted; this time period is specified in the *ADSP-21020 Data Sheet*.

9.2.1 Synchronization Delay

The ADSP-21020 has several asynchronous inputs, namely, $\overline{RESET}$, $\overline{TRST}$, $\overline{BR}$, $\overline{IRQ3-0}$ and FLAG3-0 (when configured as inputs). These inputs can be asserted in arbitrary phase to the processor clock, CLKIN. The ADSP-21020 synchronizes them prior to recognizing them. The delay associated with recognition is called the synchronization delay.

9 Hardware System Design

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 may be recognized in the current cycle or during the next cycle.

Therefore, to ensure recognition of an asynchronous input, it must be asserted for at least one full processor cycle plus setup and hold time (except for RESET, which must be asserted for at least four processor cycles). The minimum time prior to recognition (the setup and hold time) is specified in the *ADSP-21020 Data Sheet*.

9.3 POWERUP & RESET

RESET halts execution and returns all registers to a state defined in Table 9.1. On powerup, RESET must be asserted (low). After the clock is stable for a minimum period (specified in the *ADSP-21020 Data Sheet*), RESET can be deasserted.

Table 9.1 shows the states of the processor registers after reset. If a value is unchanged, it is uninitialized at powerup. Table 9.2 shows the states of outputs during reset (i.e. while RESET is low).

Pin Name	*Type*	*State During Reset*
PMA_{23-0}	Output	Driven, Value Undefined
PMD_{47-0}	Bidirectional	High Impedance
$\overline{PMS_0}$, $\overline{PMS_1}$	Output	One High, the other Low
$\overline{PMRD}$	Output	High
$\overline{PMWR}$	Output	High
PMPAGE	Output	Low
DMA_{31-0}	Output	Driven, Value Undefined
DMD_{39-0}	Bidirectional	High Impedance
$\overline{DMS_0}$	Output	High
$\overline{DMS_1}$	Output	High
$\overline{DMS_2}$	Output	High
$\overline{DMS_3}$	Output	High
$\overline{DMRD}$	Output	High
$\overline{DMWR}$	Output	High
DMPAGE	Output	Low
FLAG0	Bidirectional	High Impedance
FLAG1	Bidirectional	High Impedance
FLAG2	Bidirectional	High Impedance
FLAG3	Bidirectional	High Impedance
$\overline{BG}$	Output	Depends on $\overline{BR}$
TIMEXP	Output	Low
TDO	Output	Depends on $\overline{TRST}$ and TCK

Table 9.2 ADSP-21020 Pin States During Reset (while $\overline{RESET}$=0)

Register	*Value after Reset*
PC	0x0008
PCSTK	unchanged
PCSTKP	0x0000 (cleared)
FADDR	0x0008
DADDR	unchanged
LADDR	unchanged
CURLCNTR	unchanged
LCNTR	0x0000 (cleared)
R15 - R0	unchanged
I15 - I0	unchanged
M15 - M0	unchanged
L15 - L0	unchanged
B15 - B0	unchanged
MODE1	0x0000 (cleared)
MODE2	0x*n*000 0000 (bits 28-31 are the device identification field, identifying the silicon revision #)
IRPTL	0x0000 (cleared)
IMASK	0x0003
IMASKP	0x0000 (cleared)
ASTAT	0x00*nn* 0000 (bits 19-22 are equal to the values of the FLAG0-3 input pins; the flag pins are configured as inputs after reset)
STKY	0x0540 0000
USTAT1	0x0000 (cleared)
USTAT2	0x0000 (cleared)
DMWAIT	0x000F 7BDE
DMBANK1	0x2000 0000
DMBANK2	0x4000 0000
DMBANK3	0x8000 0000
DMADR	unchanged
PMWAIT	0x0003DE
PMBANK1	0x800000
PMADR	unchanged
PX	unchanged
PX1	unchanged
PX2	unchanged
TPERIOD	unchanged
TCOUNT	unchanged

Table 9.1 ADSP-21020 Register Values After Reset

The timing of the program memory interface for the first instruction fetch after a reset is shown in Figure 9.2 below. The first address output is the reset vector, 0x000008. PMPAGE is asserted because this is the first access to this page of memory. PMS0 is asserted and PMS1 deasserted because 0x000008 lies in bank 0 in the default configuration of memory banks. During the first two memory accesses, which have seven wait states each (due to the default value of the PMWAIT register), the first instruction is fetched and decoded. It is executed when the fetch of the third instruction begins.

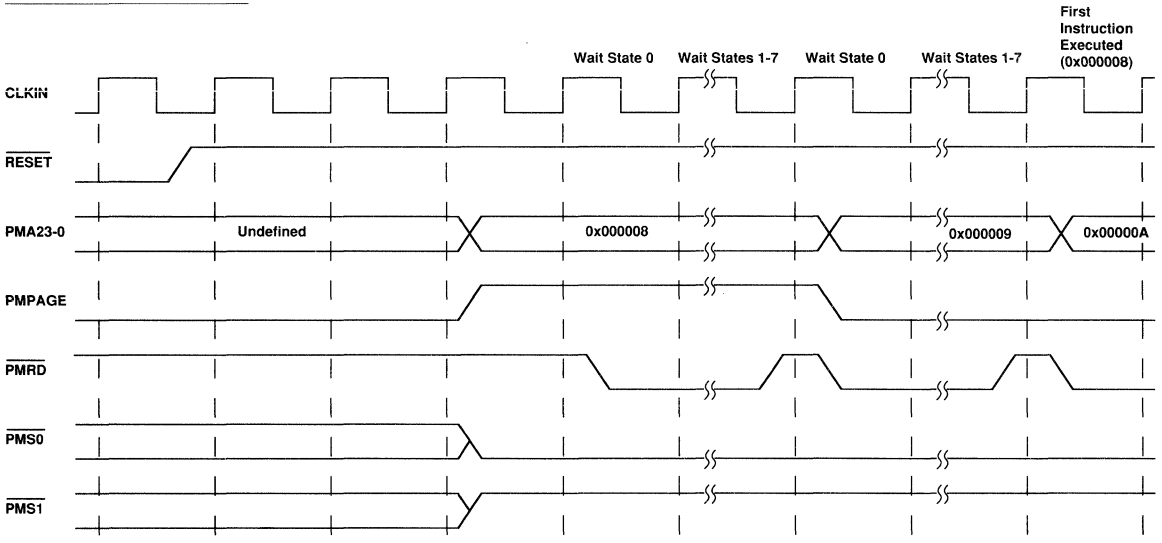


Figure 9.2 Program Memory Interface Timing at Reset

9.4 RCOMP PIN

The ADSP-21020's RCOMP pin is a compensation resistor input that controls the processor's output driver/buffers.

To reduce system noise at low temperatures when transistors switch fastest, the ADSP-21020 employs compensated output drivers. These drivers equalize slew rate over temperature extremes and process variations. A 1.8 kΩ resistor placed between the RCOMP pin and EVDD (+5 V) provides a reference for the compensated drivers. Use of a capacitor, approximately 100 pF, placed in parallel with the 1.8 kΩ resistor, is recommended.

9.5 FLAGS

Four external pins on the ADSP-21020—FLAG0, FLAG1, FLAG2 and FLAG3—allow single-bit signalling between processors. Many instructions can be conditioned on a flag's value, enabling efficient communication and synchronization between multiple processors or in other interfaces. Examples of flag use are included in the multiprocessing memory examples later in this chapter.

9.5.1 Flag Direction

The flags are bidirectional pins, each with the same functionality. Whether or not each flag is an input or an output is controlled by bits in the MODE2 register. The control for each flag is independent. On reset, the MODE2 register is cleared, so all the flags are inputs.

MODE2

Bit	*Name*	*Definition*
15	FLG0O	FLAG0 1=output; 0=input
16	FLG1O	FLAG1 1=output; 0=input
17	FLG2O	FLAG2 1=output; 0=input
18	FLG3O	FLAG3 1=output; 0=input

9.5.2 Flag Input

When a flag pin is programmed as an input, its value is stored in a bit in the ASTAT register. These flag bits are not changed when the ASTAT register is pushed onto or popped off the status stack. The bit is updated each cycle with the input value from the pin. Flag inputs can be asynchronous to the ADSP-21020 clock, so there is a one-cycle delay before a change on the pin appears in the ASTAT bit if the rising edge of the flag input misses the setup requirement for that cycle. The states of ASTAT flag bits are conditions that you can specify in conditional instructions.

ASTAT

Bit	*Name*	*Definition*
19	FLG0	FLAG0 value
20	FLG1	FLAG1 value
21	FLG2	FLAG2 value
22	FLG3	FLAG3 value

An ASTAT flag bit is read-only if the flag is configured as an input. Otherwise, the bit is readable and writeable.

9.5.3 Flag Output

When a flag is an output pin, the value on the flag pin follows the data bit in the ASTAT register. These flag bits are not changed when the ASTAT register is pushed onto or popped off the status stack. A program can set or clear the ASTAT flag bit to provide a signal to another processor or peripheral. The timing of a flag output is shown in Figure 9.3.

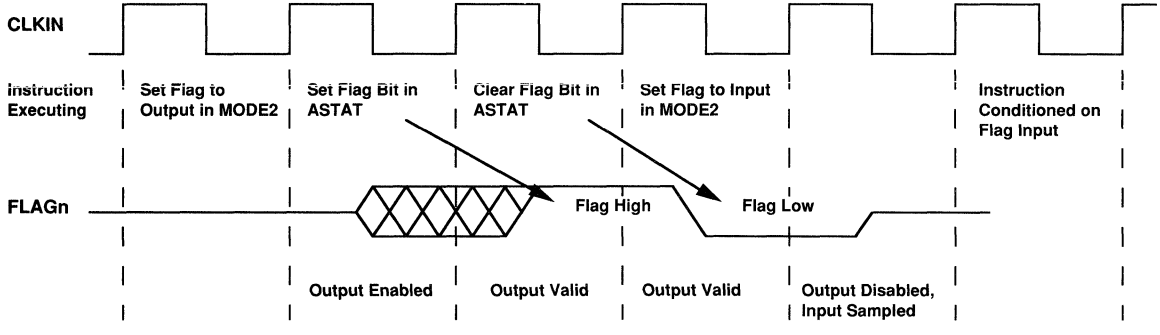


Figure 9.3 Flag Output Timing

9.6 MEMORY CONFIGURATIONS

The ADSP-21020 provides memory management and interface features to support a variety of memory configurations. These features are described in detail in Chapter 6. This section presents examples of several configurations: systems based on a single ADSP-21020 processor and systems based on multiple ADSP-21020s. These examples may serve as a starting point for your hardware design.

9.6.1 Single Processor Configurations

The memory configurations in this section are based on a single ADSP-21020 processor. The examples range from simple to complex. These examples show general concepts only; the details of a specific design would depend on the application.

9.6.1.1 One Memory Bank

Figure 9.4 shows the ADSP-21020 connected to a single bank of data memory. Each memory device is a 15 ns static RAM (SRAM), which allows the ADSP-21020 access with zero wait states. (Refer to the *ADSP-21020 Data Sheet* for specific timing parameters.) The memory devices are each 4 bits wide by 64K locations. The 64K locations require only the lower 16 address bits of the ADSP-21020; the upper bits are unused.

In this example, eight memory devices are used to provide 32-bit width for IEEE standard floating-point data. (If extended 40-bit data were needed, 10 devices would be used.) The lower eight bits of the data bus (D7-0) are not connected. No pullup or pulldown resistors are needed on these unused pins—this is taken care of on-chip. The ADSP-21020 drives and reads these lower eight bits even when it is internally configured for 32-bit data (the computation units ignore the lower eight bits).

The DMWR output of the ADSP-21020 controls each memory device's write enable ($\overline{WE}$), and the DMRD output controls each memory device's output enable ($\overline{OE}$). The $\overline{DMS0}$ signal, which in this case is the only memory select ever activated, is connected to the chip enable ($\overline{CE}$) of each device. This particular memory device has a second chip enable, which is not used (tied low) in this example.

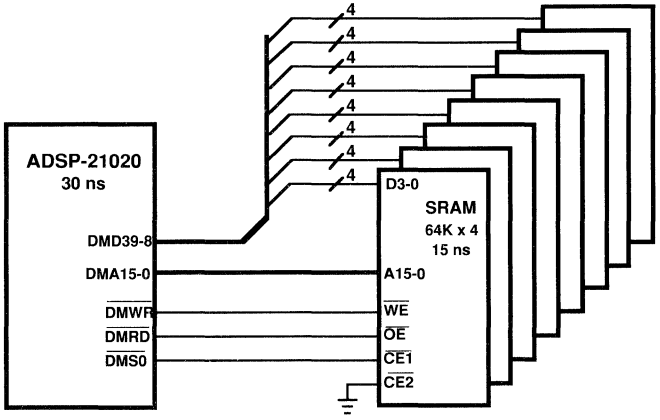


Figure 9.4 Interface to Single Data Memory Bank

9.6.1.2 Several Memory Banks

Three banks of data memory are connected to the ADSP-21020 in the example in Figure 9.5. As in the previous example, each memory device is a 35 ns SRAM, for zero wait states. The memory devices are each 8 bits wide by 32K locations, for a total of 96K locations. Five devices in each bank are needed for 40-bit floating-point data.

Bank 0 extends from 0x0000 to 0x7FFF; bank 1 from 0x8000 to 0xFFFF; and bank 2 from 0x10000 to 0x17FFF. However, only the lower 15 address bits of the ADSP-21020 are needed for addressing the 32K locations in each bank because the data memory selects enable only one bank at a time. $\overline{\text{DMS0}}$, $\overline{\text{DMS1}}$, and $\overline{\text{DMS2}}$ are connected to the chip enables ($\overline{\text{CE}}$) of banks 0, 1 and 2, respectively. The $\overline{\text{DMS3}}$ memory select, not used in this example, could be used to select a fourth bank.

The $\overline{\text{DMWR}}$ output of the ADSP-21020 controls each memory device's write enable ($\overline{\text{WE}}$), and the $\overline{\text{DMRD}}$ output controls each memory device's output enable ($\overline{\text{OE}}$).

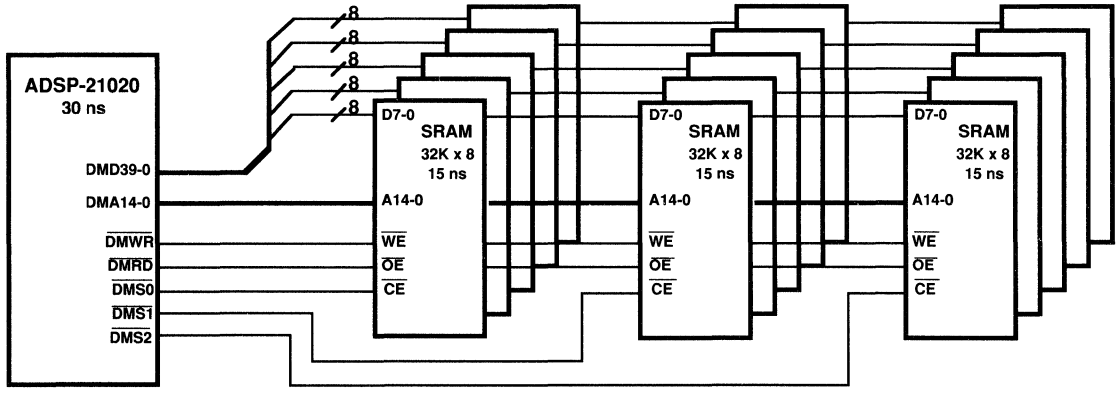


Figure 9.5 Interface to Three Data Memory Banks

9.6.1.3 Memory & I/O Devices

The example in Figure 9.6 is identical to the previous one but shows memory-mapped I/O devices added. These devices, one input-only and one output-only, can be mapped to any location in bank 3 (any location greater than 0x17FFF in this example), since there are no other devices in that bank. The $\overline{\text{DMS3}}$ signal selects the I/O devices for access; the read and write strobes differentiate between the two, enabling the latch of the output port on writes and the buffer of the input port on reads.

I/O devices should be connected to the 32-bit integer field (the *upper* 32 bits) of the DMD or PMD data buses—bits 39-8 of the DMD bus, and bits 47-16 of the PMD bus.

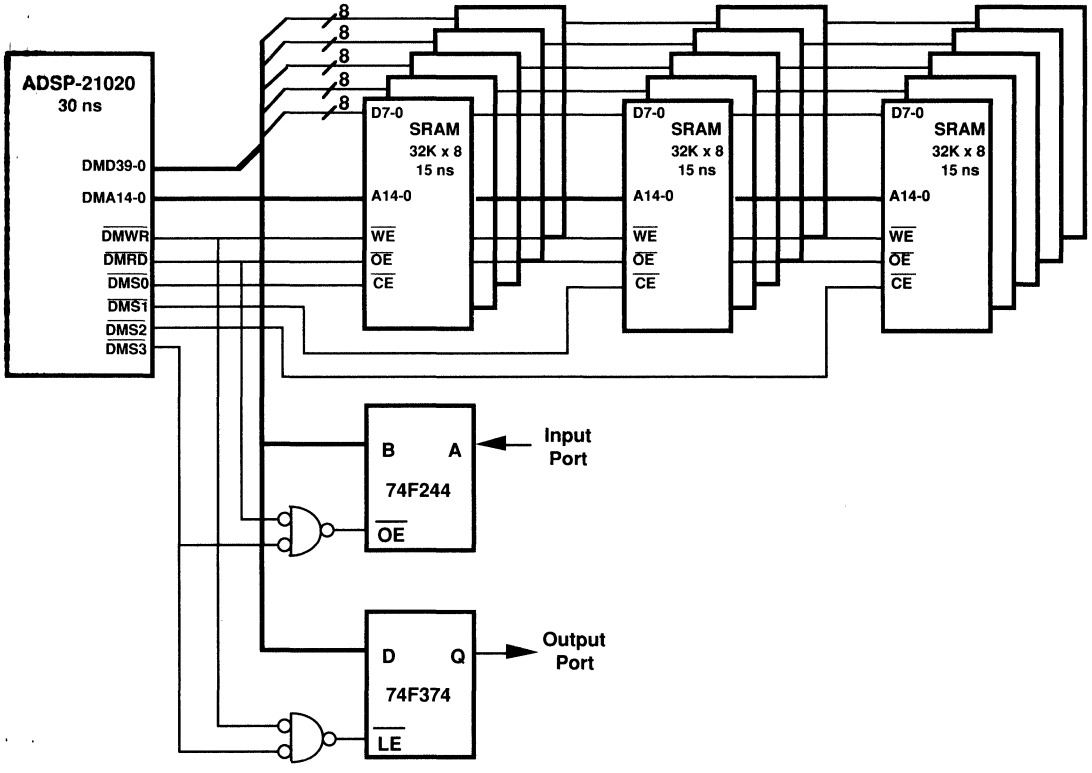


Figure 9.6 Interface to Three Data Memory Banks and Two I/O Devices

9.6.1.4 Hardware Acknowledge

Figure 9.7 shows the interface to a relatively slow I/O device that uses the hardware acknowledge (DMACK, in this case) to insert wait states in the memory cycle. This device is mapped to data memory bank 3. No other devices are in the same bank, so the $\overline{\text{DMS3}}$ signal can be used to gate the strobe that enables the I/O device to read or write data (N bits).

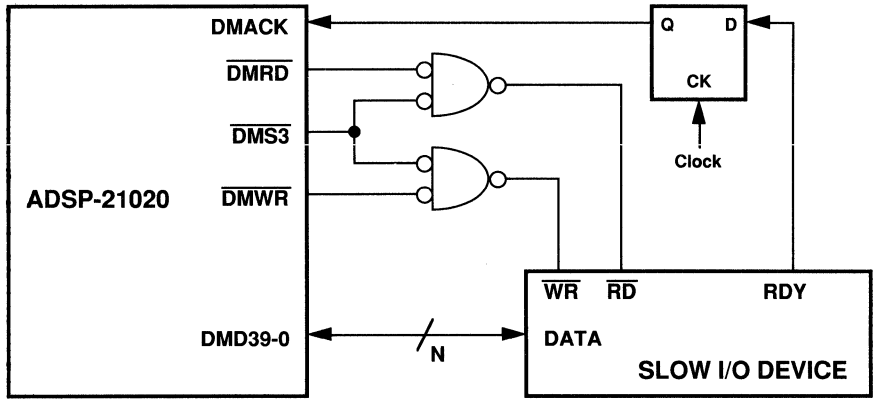


Figure 9.7 I/O Device Interface with Hardware Acknowledge

The latch on the RDY output of the I/O device synchronizes the signal to the ADSP-21020. This latch is not needed if the RDY signal meets the setup requirement for DMACK.

If the I/O device needs extra time to deassert RDY after an access is initiated (so that DMACK will not be erroneously sampled high), the ADSP-21020 can be programmed to require both internal wait states and a hardware acknowledge to terminate the memory cycle. The programmed wait states give the I/O device extra cycles in which to deassert RDY. But the RDY signal will still determine the end of the memory cycle.

9.6.1.5 Cache Memory

Figure 9.8 shows how an external cache controller would use the three-state enable (DMTS, for data memory). The cache controller monitors the ADSP-21020 address to detect a cache miss. When a miss occurs, the controller asserts DMTS in time to prevent the ADSP-21020 from completing the access. The ADSP-21020 places the data memory interface in a high-impedance state and halts. This allows the cache controller to retrieve the needed data from main memory and load it into the cache. It then deasserts DMTS and deasserts DMACK for one cycle to allow the ADSP-21020 to complete the memory access.

The 20 kΩ pullup resistors on DMRD, DMWR and DMS3 are needed to hold these controls inactive (high) during the transition of control between the ADSP-21020 and the cache controller.

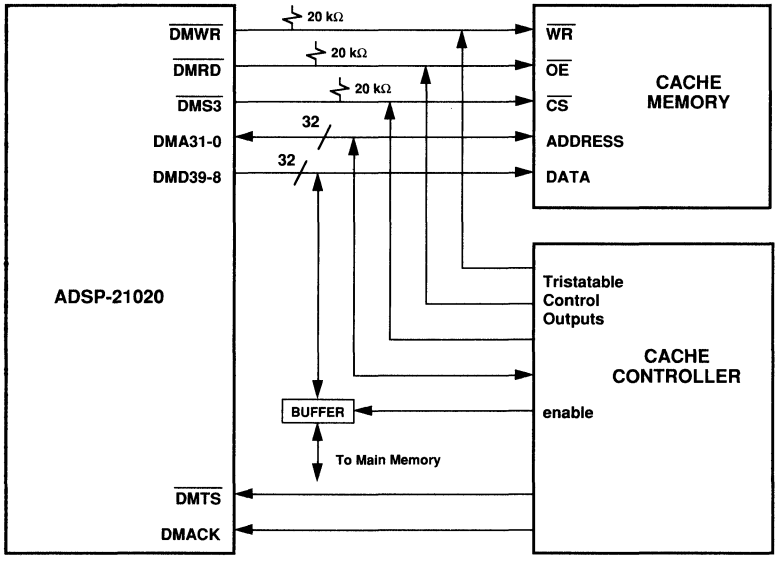


Figure 9.8 Cache Memory Interface

If the cache memory requires the ADSP-21020 to use wait states, the processor can be programmed to recognize the AND of internal and external acknowledges as the terminator of the memory cycle. After the controller updates the cache, it holds DMACK low for the required number of wait states plus one, the extra cycles allowing for the completion of the access that caused the cache miss. The minimum timing for DMACK is shown in Figure 9.9.

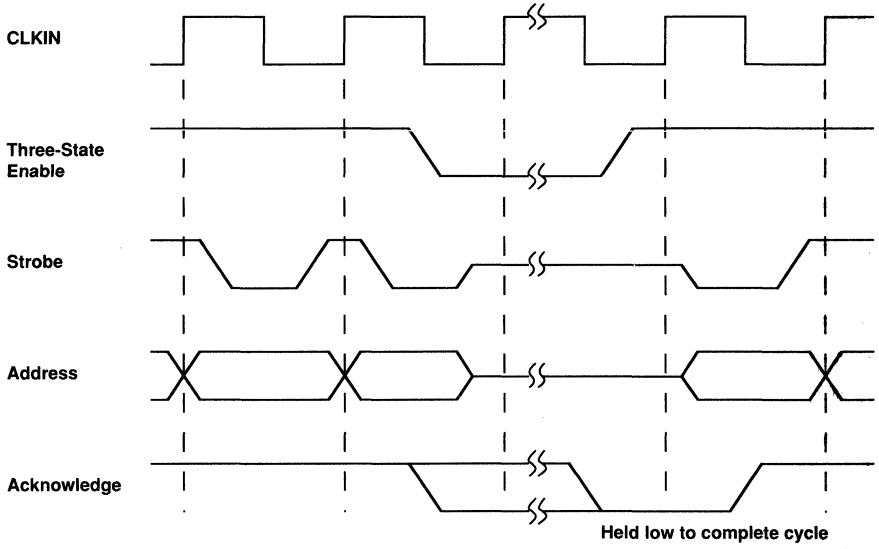


Figure 9.9 Timing on Cache Miss

9.6.1.6 DRAM With Paging

The example in Figure 9.10 shows the ADSP-21020 interface to a page-mode or static-column DRAM using a DRAM controller (which may be implemented with PALs and PGAs). This example is similar to the cache memory example in that it uses $\overline{\text{DMTS}}$ to hold off the memory access while an external memory controller takes over.

The DRAM controller's outputs are normally tristated. The DMPAGE output signals a change of page to the controller. This can be accompanied by an automatic wait state if the controller requires extra time. The DRAM controller responds by latching in the address and by asserting $\overline{\text{DMTS}}$ to prevent the ADSP-21020 from completing the access. It then controls the DRAM to effect the page change, using the latched address, by driving the 10 MSBs of the address onto the 10 LSBs of the DRAM address input. When finished, the DRAM controller tristates its memory controls and deasserts $\overline{\text{DMTS}}$ and deasserts DMACK in the same cycle for the appropriate number of wait state cycles plus one. This timing is the same as for the cache controller, shown in Figure 9.9.

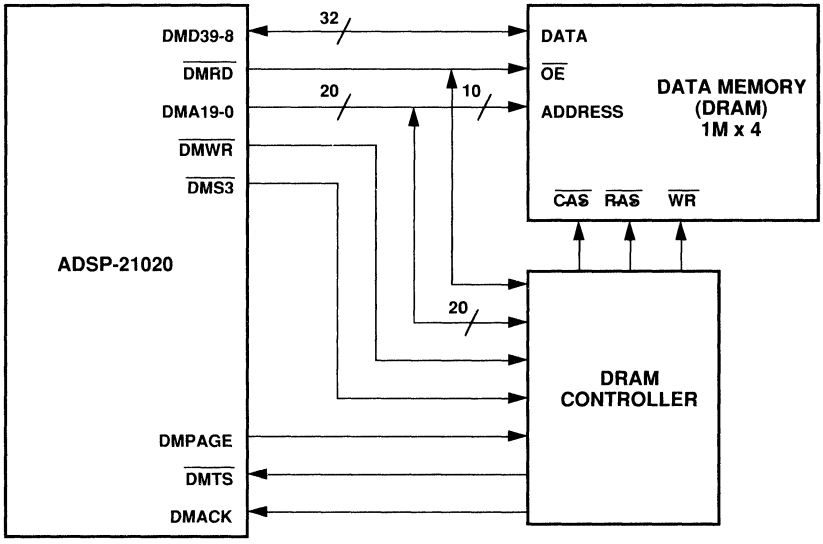


Figure 9.10 Page-Mode DRAM Interface

9.6.1.7 Direct Memory Access (DMA)

The example shown in Figure 9.11 uses the bus request/bus grant protocol of the ADSP-21020 to perform direct memory access (DMA) on the ADSP-21020's data memory. The DMA controller is driven by a host processor (not shown) that occasionally needs to access the data memory to read or write a buffer of data. When the host requests an access, the DMA controller asserts bus request ($\overline{BR}$) on the ADSP-21020. The ADSP-21020 completes its current instruction, places its memory buses in a high impedance state, and asserts bus grant ($\overline{BG}$). The ADSP-21020 idles while $\overline{BR}$ is asserted.

The bus grant allows the DMA controller to access the data memory, using counters to generate the necessary addresses and clocking the data to or from the host through the bidirectional latch. The DMA controller must also provide the appropriate memory strobes.

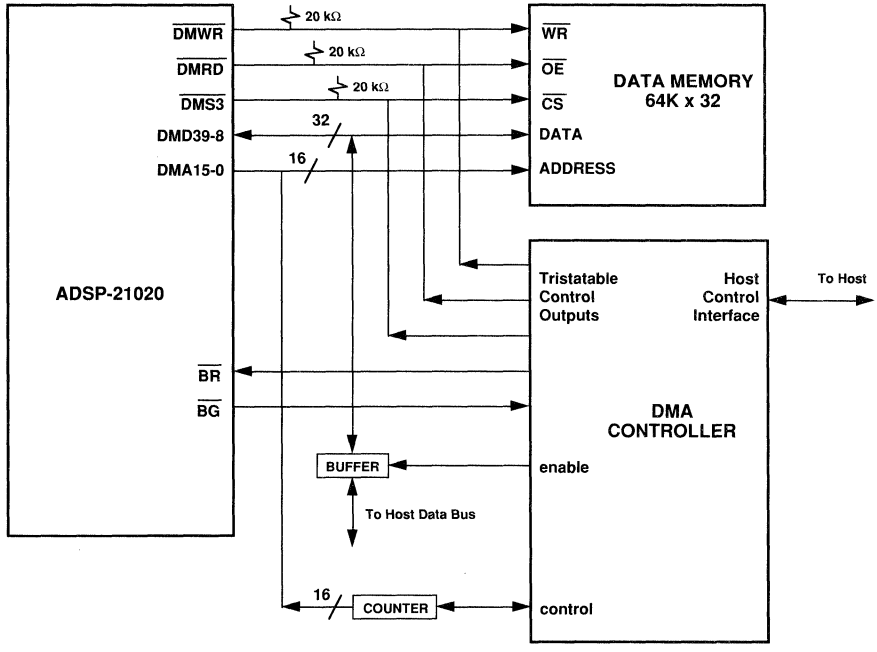


Figure 9.11 DMA Controller Interface Using Bus Request

When the DMA access is complete, the controller deasserts $\overline{BR}$ and the ADSP-21020 continues program execution from where it left off. Timing for the bus request/bus grant cycle is shown in Figure 9.12. Note that there is at least one cycle of delay after BR is asserted and before $\overline{BG}$ goes low (more if the ADSP-21020 is executing an instruction requiring extra cycles). After BG goes low, there may be one cycle of overhead during which no instructions are executed and no data is transferred. There may be another cycle of overhead when exiting bus grant if the DMA controller cannot tristate its outputs before the ADSP-21020 drives the bus. In this case, the controller must tristate its outputs in the previous cycle.

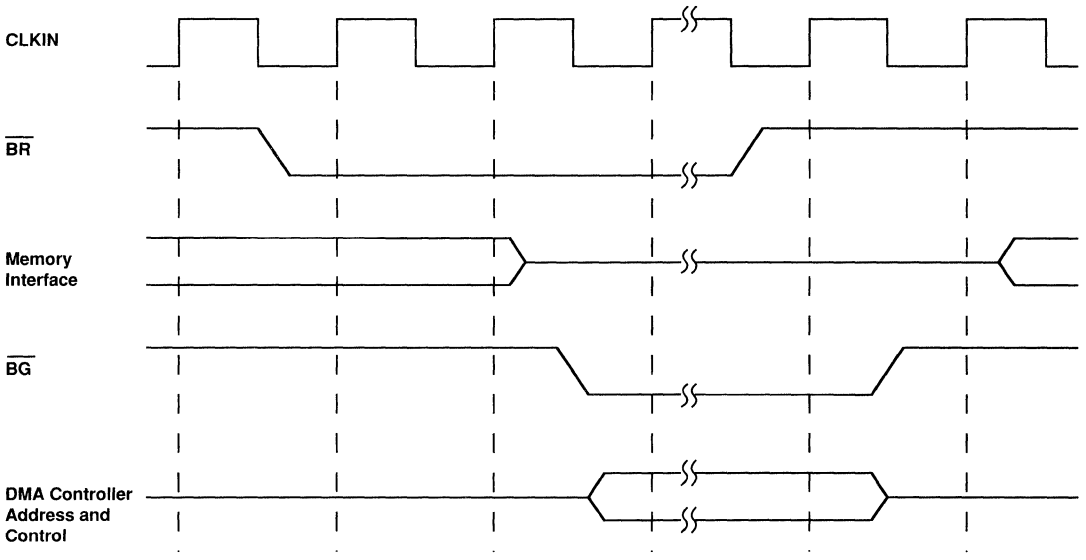


Figure 9.12 Bus Request Timing for DMA

9.6.2 Multiprocessor Configurations

This section describes several memory configurations featuring multiple ADSP-21020s operating in the same system. In these examples, the processors pass data between one another through shared memory. The configuration appropriate for a specific application depends on the data flow and timing required by the system.

9.6.2.1 Multiport Memory

Figure 9.13 shows the minimal hardware for connecting three ADSP-21020s and a host processor to a 4-port RAM. The particular memory device in this example is 8 bits wide by 2K locations. Four devices are needed for 32-bit data in Figure 9.13; the number of devices actually used depends on the data width to be supported.

The memory provides four identical interfaces consisting of address, data, write strobe, output enable, and chip enable. In this example, each processor maps accesses to this memory to bank 1 of data memory (enabled by $\overline{\text{DMS1}}$). Only 11 of each processor's 32 address lines are needed to address the 2K locations. Also shown for each processor is local data memory that may use all of the address lines. The local memory would occupy a different bank of memory and thus be enabled by a different memory select signal.

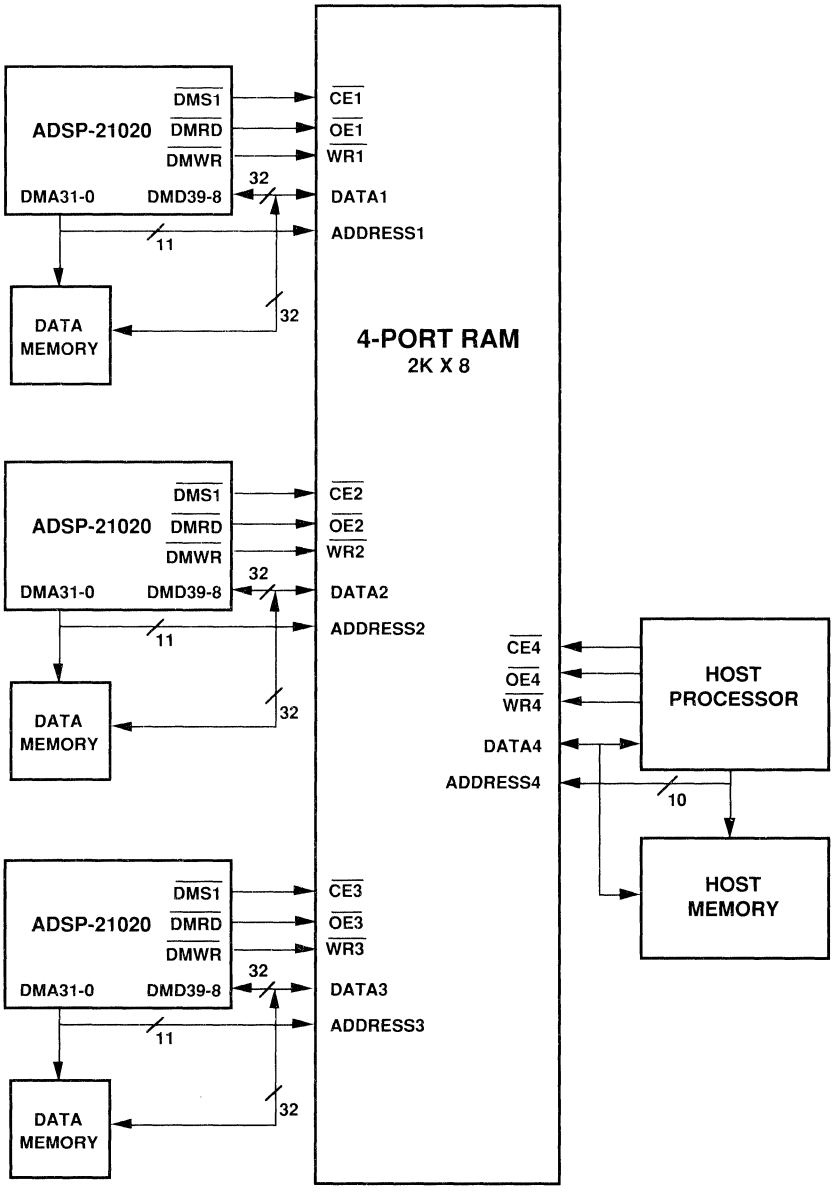


Figure 9.13 Four-Port RAM Configuration

9.6.2.2 Serial Data Flow

If the flow of data in a multiprocessor system is serial, that is, data moves in sequence along a linear path from one processor to the next, then several configurations are suggested.

Buffer Latches

Figure 9.14 shows a simple, low-cost solution for synchronous transfers. The linkage between every two processors in a serial path consists of a set of buffer latches, the number of which is determined by the width of the data to be transferred (8, 16, 24, 32 or 40 bits). Each processor in the serial path outputs data to latches on its DMD bus, and the reads data from latches on its PMD bus. Each processor also has local program memory and data memory. Reads from and writes to the latches are distinguished

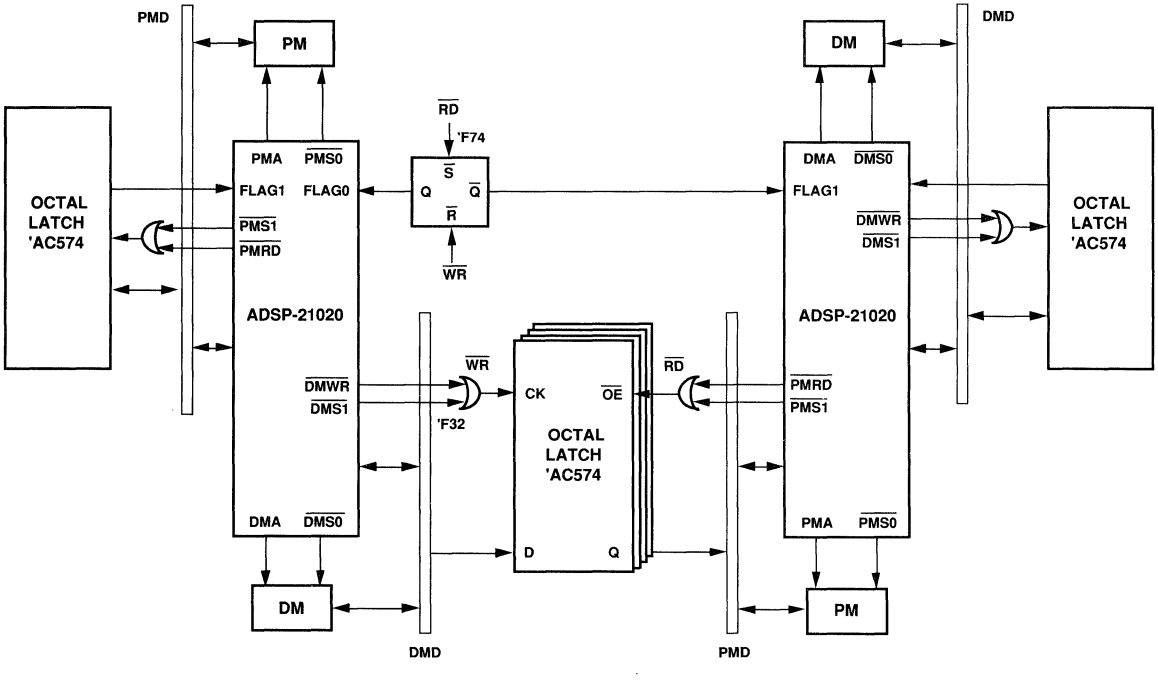


Figure 9.14 Serial Data Flow Using Buffers

by the bank 1 selects ($\overline{DMS1}$ and $\overline{PMS1}$) in this case. Data is clocked into a latch by the rising edge of the $\overline{DMWR}$ signal; the output of a latch is enabled by the $\overline{PMRD}$ signal.

Two of each processor's flags (FLAG0 and FLAG1, programmed as inputs) are used for synchronizing the data transfer between processors. When one processor writes to the latch, it asserts the external semaphore which in turn asserts the FLAG1 input of the next processor to indicate that there is data in the latch. When the read occurs, the second processor resets the signal on its FLAG1 input and also sets the FLAG0 input of the first processor to indicate that the data has been read. The first processor then writes the latch again, resetting the signal on its FLAG0 input.

A processor reads the latch only if FLAG1 indicates that there is data to read. The instruction would be a conditional instruction that reads program memory:

```
IF FLAG1_IN F3=PM(buffer);    {"buffer" is a location }
                              { in bank 1             }
```

Likewise, a processor would write the latch only if FLAG0 indicates that the previous data has been read.

In this configuration, a processor can read the latch in the cycle after it was written. The maximum throughput is therefore one data transfer every two cycles. High transfer rates, however, require the operations of all processors in the system to be closely synchronized, since there is no external storage in which to accumulate data.

The use of both memory interfaces is optional. Alternatively, both input and output latches could be placed on the data memory interface. In this case, the program memory interface would be used only for program memory accesses.

FIFOs

Figure 9.15 shows an example similar to the previous one, except that the buffer latches are replaced with FIFOs. This configuration has the advantage that each processor does not have to wait for the next one to read data before it can write data. The processors can operate at full speed, and bottlenecks are avoided.

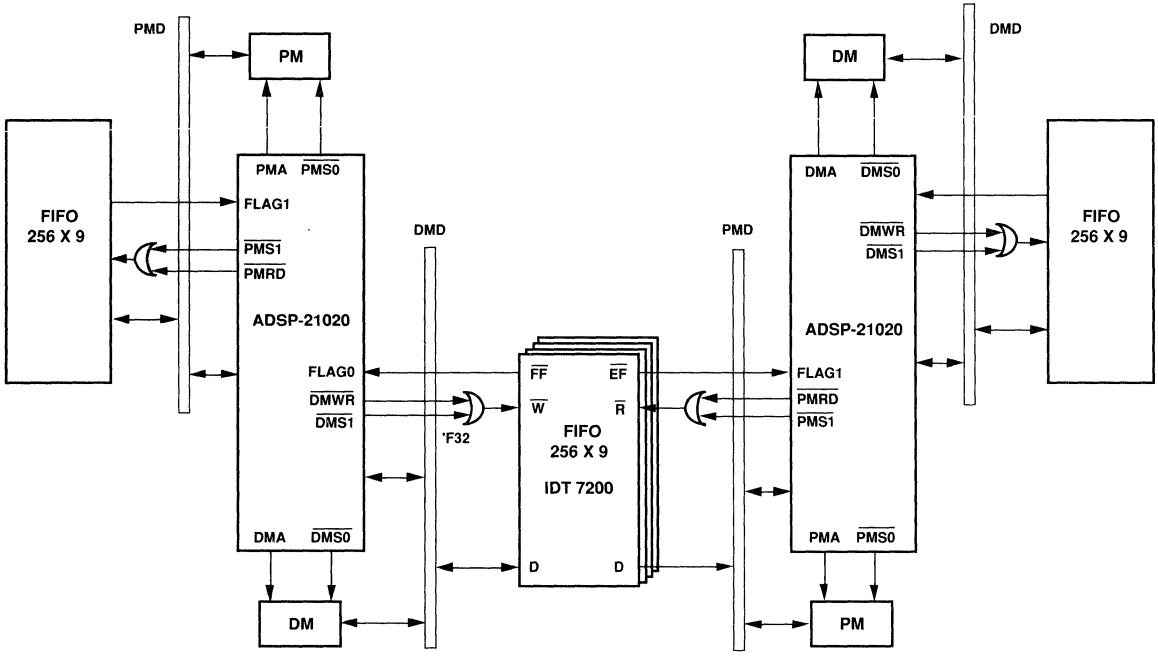


Figure 9.15 Serial Data Flow Using FIFOs

In this configuration, a low FLAG0 input indicates that the FIFO is full ($\overline{FF}$ flag is asserted). Thus, writes to the FIFO should be conditioned on FLAG0:

```
IF FLAG0_IN DM(fifo)=F7;      {"fifo" is a location in bank 1}
```

or

```
DO loop UNTIL NOT FLAG0_IN;   {loop is at least 5 instructions, so flag }
                              { can go low before loop restarts.        }
      compute, DM(fifo)=F7;   {"fifo" is a location in bank 1}
      instruction 2 ;         {FLAG0 is low here on last loop}
      instruction 3 ;
      instruction 4 ;
loop: instruction 5 ;
```

Similarly, when the FIFO is empty it asserts its $\overline{EF}$ flag which deasserts the FLAG1 input of the processor receiving data. FIFO reads should be conditioned on FLAG1.

As in the previous example, the use of the program memory interface is optional; both the input FIFO and output FIFO can be connected to a processor's data memory interface.

9 Hardware System Design

Dual-Port Memory

Figure 9.16 shows an example using dual-port RAM to transfer data between processors. This configuration is similar to the previous one, but has even more data storage and allows bidirectional data flow on the serial path.

The INT pin is a general-purpose output that is set when location 0x3FE or 0x3FF of the dual-port RAM is written. These locations are mailbox registers that can be used to pass messages. In this example, the INT output is connected to a flag input on the ADSP-21020. The processor can read the mailbox register if the flag is set. Alternatively, INT could trigger an interrupt, with the service routine reading the mailbox.

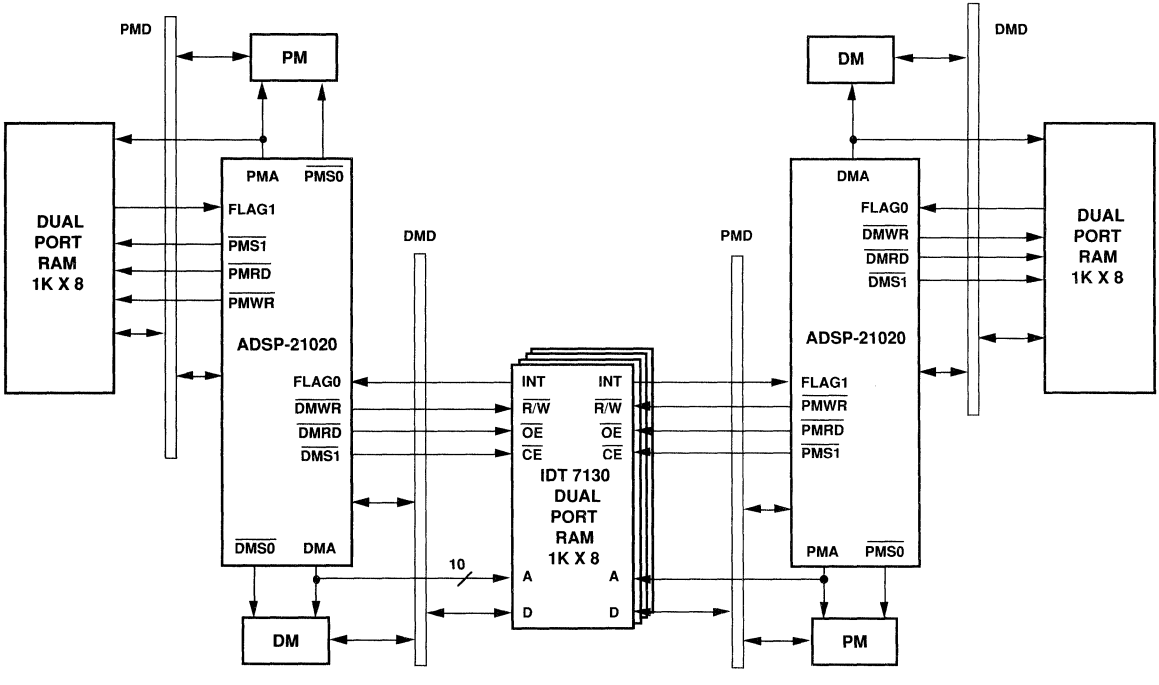


Figure 9.16 Serial Data Flow Using Dual-Port RAM

The dual-port RAM has a BUSY output that it asserts if contention occurs (both processors try to write the same location at the same time). If both processors are operating with zero wait states, however, this function can not be used. Contention can be avoided if each processor is constrained in software to read only those locations that the other processor writes and to write only those locations that the other processor reads. In this way, two writes to the same location will never occur.

9.7 PROGRAM MEMORY BOOT AT RESET

After reset, the ADSP-21020 automatically fetches its first instruction from location 0x08 of program memory. If program memory consists of ROM, the instructions are available in memory at powerup. If program memory is made up of RAM, however, there must be a mechanism for loading the program into memory at powerup. A single RAM and no ROM is frequently an attractive option because the addition of 8-bit ROMs would require six more memory devices, resulting in a higher cost, more board space and higher capacitive loads on the address lines than with RAM alone.

In a boot operation the ADSP-21020 executes a minimal program that loads the rest of program memory. A way to implement the boot operation is to load the instructions of a boot program through the ADSP-21020's test access port. This port, which conforms to the IEEE 1149.1 specification, is described in Appendix C. It allows serial data to be shifted into and out of the ADSP-21020. The internal serial path connects to every input and output pin, so that the value of any pin can be read or written to using the serial shift mechanism.

The boot operation can be controlled by a host processor or a dedicated microcontroller. The operation would proceed generally as follows:

1. The host or controller shifts an instruction and address into the program memory data inputs with PMWR deasserted.

2. The host or controller shifts the same instruction and address into the program memory data inputs with $\overline{\text{PMWR}}$ asserted.

3. The host or controller shifts the same instruction and address into the program memory data inputs with $\overline{\text{PMWR}}$ deasserted.

4. Steps 1-3 are repeated for a series of instructions and sequential addresses that cause the processor to load a loader program into program memory.

5. When the loader program has been loaded, RESET is deasserted, and the processor begins executing the loader program from location 0x08 in memory to bring in the main program.

An example loader program is shown below in Figure 9.17. In this example, data is read in byte-wise on DMD15-8. The 48-bit instructions are reconstructed in the shifter, then transferred to the PMD bus and to program memory using the PX registers. This routine assumes that data is coming from an 8-bit EPROM. If a host is supplying the data, the address lines can be ignored. This routine requires 18 instructions.

```
          I8=START_ADR;          {load program at START_ADR}
          M8=1;                  {M8 increments by 1}
          L8=0;                  {no circular buffer}
          I1=EPROM_ADR;          {pointer to EPROM}
          M1=1;                  {M1 increments by 1}
          L1=0;                  {no circular buffer}

          LCNTR=LOAD_COUNT, DO LOAD_LP UNTIL LCE;
          R0=DM(I1,M1);                                {load byte 1 LSB}
          R1=FDEP R0 BY 0:8,          R0=DM(I1,M1);    {load byte 2}
          R1=R1 OR FDEP R0 BY 8:8,    R0=DM(I1,M1);    {load byte 3}
          PX1=R1;
          R1=FDEP R0 BY 0:8,          R0=DM(I1,M1);    {load byte 4}
          R1=R1 OR FDEP R0 BY 8:8,    R0=DM(I1,M1);    {load byte 5}
          R1=R1 OR FDEP R0 BY 16:8,   R0=DM(I1,M1);    {load byte 6 MSB}
          R1=R1 OR FDEP R0 BY 24:8;
          PX2=R1;
LOAD_LP:  PM(I8,M8)=PX;          {write instr. to PM}
          JUMP START_ADR;        {begin execution}
```

Figure 9.17 Example Loader Program

9.8 MEMORY INTERFACE CAPACITIVE LOAD

The timing parameters for the memory interfaces of the ADSP-21020 are specified at capacitive loads of 100 pF on the address, memory select, page boundary, read strobe and write strobe outputs. For the data and other outputs, the nominal load is 50 pF. If the capacitive load on an output is different than the nominal load, its switching characteristic is affected. Specifically, as the capacitance on a pin increases, so do its rise and fall times. If an output delay is measured at the point that the output reaches a particular voltage level, then the delay increases for larger capacitive loads and decreases for smaller ones. Consequently, pin-to-pin variations in capacitive loading can alter the relative timing of outputs to the point where they no longer meet the input requirements of the memory device.

The drive strength of the ADSP-21020 memory outputs is sufficient for large capacitive loads, so most variations in loading will not change relative timing enough to violate any memory device specifications. This section describes how to determine whether load variations in your system will cause timing problems and how to correct them.

9.8.1 Load Variations

A typical application that can contain large variations in loading between pins is one that requires multiple banks of memory. Figure 9.18 (on page 9-28) shows an example with three banks of external data memory, one of which is 32K words long and the other two which are 8K words each. Address lines DMA12-0, $\overline{\text{DMRD}}$ and $\overline{\text{DMWR}}$ are loaded by all three banks of memory, a total of 15 RAMs. The address lines DMA13 and DMA14 and memory selects $\overline{\text{DMS2-0}}$ are connected to only one of the three memory banks and have approximately a third the load of the other outputs.

In this scenario, there are a number of RAM specifications that can be negatively affected by the load variation. One of these is the address hold from write end (write strobe deasserted), typically specified at 0 ns minimum. The timing of the ADSP-21020 data memory address and $\overline{\text{DMWR}}$ outputs is specified to guarantee a positive hold time at nominal capacitive loads. However, if the load on an address line is much less than that on the write strobe, which is the case for DMA13 and DMA14, the address line switches faster relative to the strobe. If the result is that the address line switches before the strobe does, then the address hold time is negative, and the RAM input requirement is not met. This situation is depicted in Figure 9.19.

There are other capacitive loading variations that can lead to violations of RAM specifications. If data lines are more heavily loaded than the write strobe, for example, the data may not be valid in time to meet the setup requirement before the write end. Heavily loaded address lines can be slowed enough to adversely affect the address-to-read-data-valid and address-to-acknowledge requirements.

To determine whether capacitive loading variations in a particular situation lead to specification violations, refer to graphs in the *ADSP-21020 Data Sheet* that specify for each type of output how the delay changes with capacitive load. Adjust the nominal delay for each output based on its capacitive load, then assess whether the relative timing of signals meets the input specifications of the RAM.

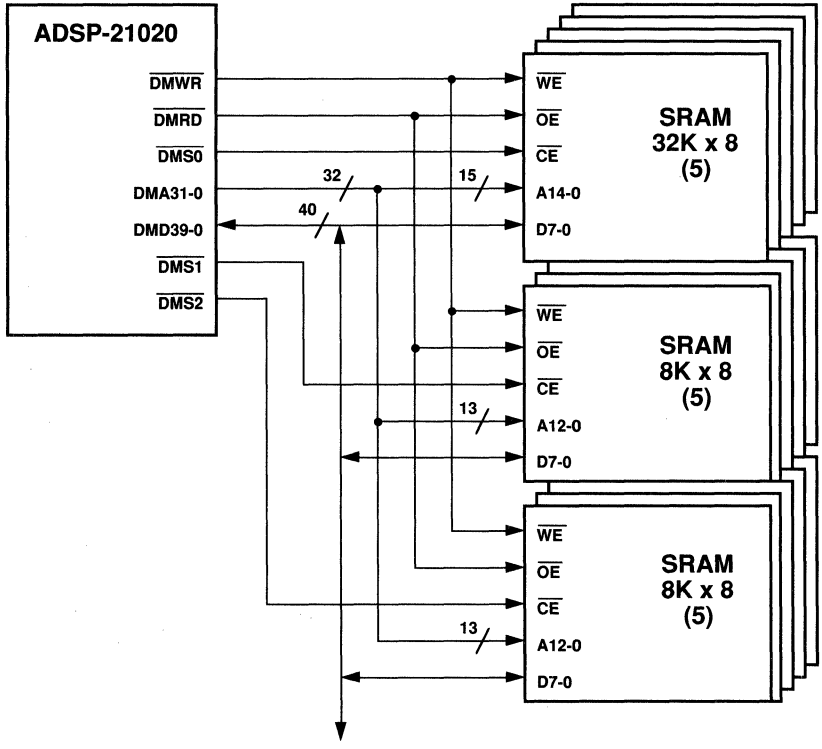


Figure 9.18 Memory Configuration with Unequal Capacitive Loads

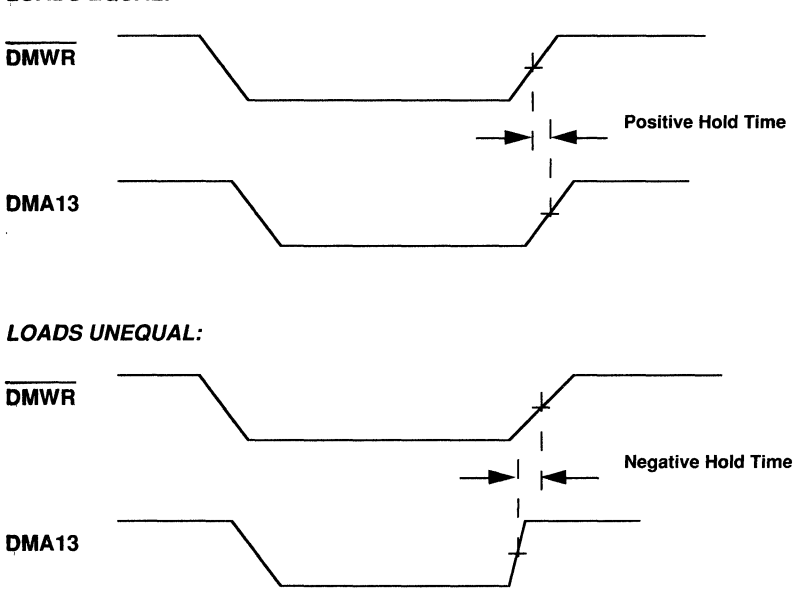


Figure 9.19 Effect of Unequal Capacitive Loads

9.8.2 Correcting The Timing

Timing violations caused by load variations can be corrected by slowing down the faster outputs. Two ways to accomplish this are adding discrete capacitance or adding series resistance to the faster outputs. Each method results in a longer delay, but discrete capacitance may add ringing whereas series resistance may increase susceptibility to noise coupling. If series resistance is chosen, the following formula yields the approximate amount of resistance required:

$$R = \Delta t / C_{load}$$

Thus, to increase the delay by 2 ns on an output that has a 40 pF load, add about 50 Ω of resistance in series between the output and the RAM input, near the output pin. Access times may suffer if too much delay is added, however.

9 Hardware System Design

9.9 EZ-ICE EMULATOR CONSIDERATIONS

The ADSP-21020 EZ-ICE® Emulator is a development tool for debugging programs running in real time on your ADSP-21020 or ADSP-21010 target system hardware.

The EZ-ICE provides a controlled environment for observing, debugging, and testing activities in a target system by connecting directly to the target processor through its JTAG interface. The emulator can monitor system behavior while running at full speed. It lets you examine and alter memory locations as well as processor registers and stacks.

Because EZ-ICE controls the target system's ADSP-21020 (or ADSP-21010) through the processor's IEEE 1149.1 (JTAG) Test Access Port, non-intrusive in-circuit emulation is assured; the emulator does not impact target loading or timing. The emulator's in-circuit probe connects to an IBM PC host computer with an ISA bus plug-in board.

Target system boards must have an 11-pin JTAG connector to accept the EZ-ICE's in-circuit probe (a 12-pin plug).

9.9.1 Target Board Connector For EZ-ICE Probe

The EZ-ICE uses the IEEE 1149.1 JTAG test access port of the ADSP-21020/21010 to monitor and control the target board processor during emulation. The EZ-ICE probe requires that CLKIN, TMS, TCK, TRST, TDI, TDO, and GND be made accessible on the target system via a 12-pin connector (pin strip header) such as that shown in Figure 9.20. The EZ-ICE probe plugs directly onto this connector for chip-on-board emulation; you must add this connector to your target board design if you intend to use the EZ-ICE.

The 12-pin, 2-row pin strip header is keyed at the pin 1 location—you must clip pin 1 off of the header. The pins must be 0.025 inch square and at least 0.318 inch in length. Pin spacing is 0.1 x 0.1 inches. Pin strip headers are available from vendors such as 3M, Samtec, and McKenzie.

The tip of the pins must be at least 0.18 inch higher than the tallest component on your board to allow clearance for the bottom of the emulator's probe.

The length of the traces between the EZ-ICE probe connector and the ADSP-21020/21010's test access port pins should be less than 1 inch.

The BMTS, BTCK, $\overline{\text{BTRST}}$, and BTDI signals are provided so that the test access port can also be used for board-level testing. When the connector is not being used for emulation, place jumpers between the BXXX pins and the XXX pins as shown in Figure 9.20. If you are not going to use the test access port for board test, tie $\overline{\text{BTRST}}$ to GND and tie or pullup BTCK to VDD. The $\overline{\text{TRST}}$ pin must be asserted (pulsed low) after powerup (through $\overline{\text{BTRST}}$ on the connector) or held low for proper operation of the ADSP-21020/21010.

9.9.2 Other Hardware Considerations

- The EZ-ICE probe adds two TTL loads to the CLKIN pin of the ADSP-21020/21010.
- The target system design must use $\overline{\text{PMRD}}$ and $\overline{\text{DMRD}}$ to gate the output enable of any device that can drive the ADSP-21020/21010 buses.

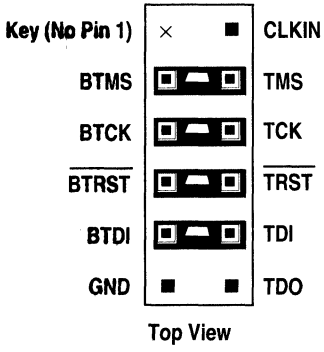


Figure 9.20 Target Board Connector For EZ-ICE Probe (jumpers in place)

9 Hardware System Design

9.10 HOST PROCESSOR INTERFACE

In this section a bidirectional interface between an ADSP-21020 and a host microprocessor is described. The interface consists of three channels: the write channel, the read channel, and the status channel. The write channel transfers data from the host to the ADSP-21020. The read channel transfers data from the ADSP-21020 to the host. The status channel provides the host with information regarding the state of the read and write channels.

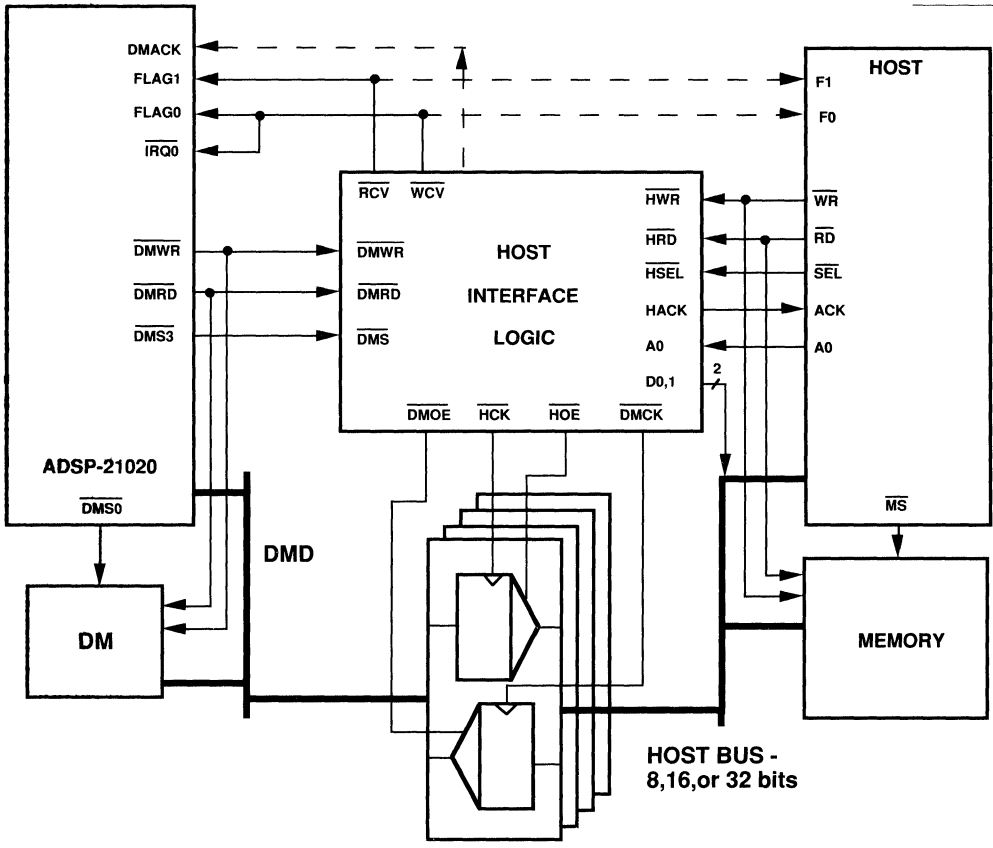
The system configuration is shown in Figure 9.21. Figure 9.22 shows the details of the interface logic.

When the host writes data to the port, the write channel valid flag ($\overline{WCV}$) goes active. This flag informs the ADSP-21020 that valid data is present in the write channel. Similarly, when the ADSP-21020 writes data to the port, the read channel valid flag ($\overline{RCV}$) goes active. Both flags are cleared when their respective channels are read. It is the channel valid flags that the host accesses when it reads the status channel.

The channel valid flags are set and cleared on the rising edges of the strobes. This ensures that the flags reflect the true state of the channels at all times. For example, assume the flags are set on the falling edges of the strobes and that the host is writing data to the port. If the ADSP-21020 is much faster than the host, it may sense the flag and read the channel before the host has had time to put its data into the port. By changing the state of the flags on the rising edge of the strobes, we guarantee that the flags change state only after the channels do.

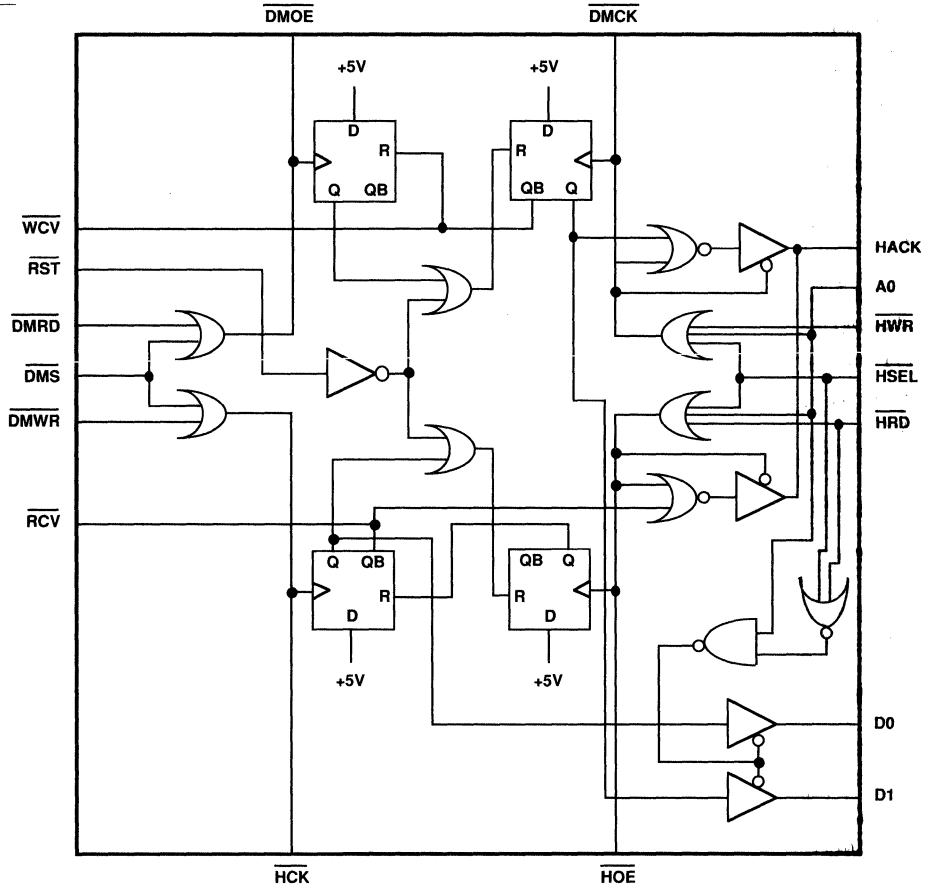
9.10.1 Data Transfer Sequences

Described below are the two basic transfer sequences. See the timing shown in Figures 9.23 and 9.24 for more information. In these figures, the falling edges of $\overline{WCV}$ and $\overline{RCV}$ indicate the presence of valid data in the host port latches. The rising edges indicate that the data has been read and the buffer is empty. The data bus shows the host port latch data.



Dashed lines indicate optional connections.

Figure 9.21 Host Interface Block Diagram



NOTES:

1. The latches shown are positive edge triggered D latches with asynchronous reset to 0.
2. Because the host acknowledge line (ACK) is shared by the memory system, HACK is tristated except when either of the host strobes is low.

Figure 9.22 Host Interface Logic

Host Write (Host Data to ADSP-21020):

1. The host writes data to the port. The $\overline{WCV}$ flag is set on the rising edge of the host write strobe.
2. The ADSP-21020 samples the $\overline{WCV}$ flag, either by polling or interrupt.
3. The ADSP-21020 reads the port, clearing the $\overline{WCV}$ flag.

Because the $\overline{WCV}$ flag is cleared by the ADSP-21020 read before the host begins its write, the $\overline{HACK}$ line is asserted immediately after the host write strobe is asserted, ending the host cycle without wait states. If the host attempts to write the port again before the ADSP-21020 reads it, the $\overline{HACK}$ line is deasserted immediately after the write strobe is asserted and remains deasserted until the ADSP-21020 reads the data. Then the $\overline{HACK}$ line is asserted, ending the host cycle. The previous host data is not corrupted by the second host write because the data is clocked in on the rising edge of the write strobe. The host can avoid hanging up by reading the status bits before each transfer.

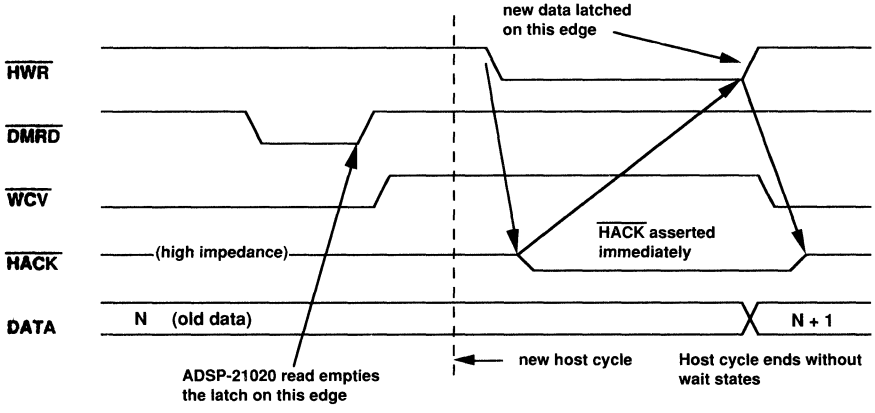


Figure 9.23 Host Write Timing

Host Read (ADSP-21020 Data to Host):

1. The ADSP-21020 writes data to the port, setting the $\overline{\text{RCV}}$ flag on the rising edge of the write strobe.
2. The host samples the $\overline{\text{RCV}}$ flag, either by being interrupted, or by reading the status channel.
3. The host reads the port, clearing the $\overline{\text{RCV}}$ flag on the rising edge of the read strobe.

Because the $\overline{\text{RCV}}$ flag is set by the ADSP-21020 write before the host begins its read, the $\overline{\text{HACK}}$ line is asserted immediately after the host read strobe is asserted, ending the host cycle without wait states. If the host attempts to read the port before new data has arrived, the $\overline{\text{HACK}}$ line is deasserted immediately after the host read strobe is asserted until the ADSP-21020 writes data to the port. Then the $\overline{\text{HACK}}$ line is asserted, ending the host cycle.

Since this is an asynchronous interface, the status of the channels may change even as the host is reading the status. At worst, this could cause the host to wait longer than necessary before initiating the next access.

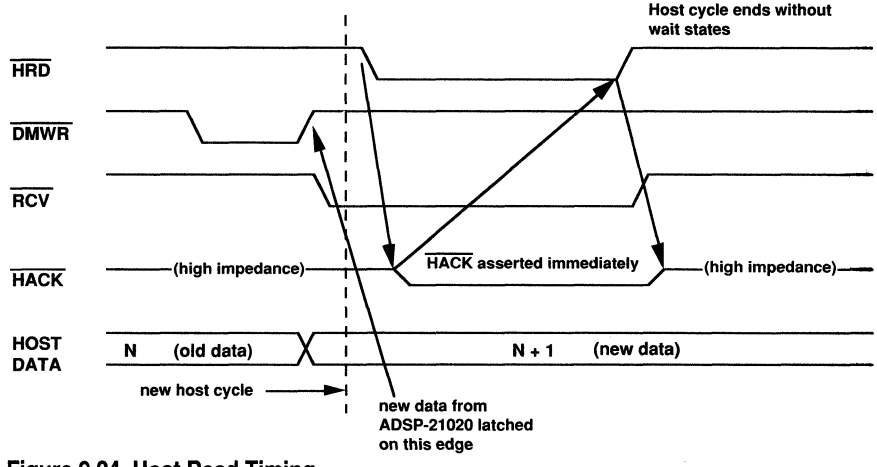


Figure 9.24 Host Read Timing

9.10.2 Host Interface Code Examples

This section describes several ways to program data transfers using the host port.

9.10.2.1 Buffer Transfers

The code segment below shows buffer transfers from the host while the ADSP-21020 is executing a loop. It has minimal overhead for a transfer, and the loop length is the same whether or not there is a transfer.

```
      LCNTR=length, DO END UNTIL LCE;          {Loop}
      IF NOT FLAG0 R3=DM(HOST), R1=LEFTZ R1;   {Read host if not flag0}
      IF SZ DM(I0,M0)=R3, R2=LEFTZ R2;         {Write buffer if SZ=1}
      instruction 1 ;                          {Main part of loop}
      instruction 2 ;
      ...
      ...
END:  instruction N ;
```

The I0 register is the address pointer to the buffer in data memory. FLAG0 is low when data is ready to be read by the ADSP-21020. R1 is initialized to 0xFFFFFFFF, R2 to 0. The use of the SZ flag permits the read from the host and the write to the buffer to be indivisible. If the shifter, which tests R1 and R2 and sets and clears the SZ flag, is used elsewhere in the loop, the SZ flag should not be left set prior to the FLAG0 test.

9.10.2.2 Interrupt-Driven Transfers

Interrupt-driven transfers are useful because they allow the ADSP-21020 to continue processing without overhead until data is ready to be transferred. This is desirable when communicating with a slow host.

The code below is a simple example of an interrupt-driven I/O driver.

```
transfer:  RTI (DB);              {Return (delayed)}
           R15=DM(HOST_ADDR);     {Get data from port}
           DM(I6,1)=R15;          {Transfer data to buffer, incr pointer}
```

The transfer interrupt request is asserted by the host interface logic each time the host writes to the port. An alternative to a host interrupt is a timer interrupt (using the ADSP-21020 timer).

9.10.2.3 High Speed Transfers

For transfer rates comparable to the ADSP-21020 cycle time, the host interface should be connected to the ADSP-21020's program memory port. Then data can be read from the host port and written to the data memory on every cycle. This may even occur in parallel with a computation.

```
init: I5=HOST_ADDR;           {Load addr pointer to host port}
      M5=0;
      I6=DATA_BUFFER;         {Load addr pointer to buffer}
      M6=1;
go:   R15=PM(I5,0);                              {Get first data}
      LCNTR=length, DO end UNTIL LCE;
end:    compute, DM(I6,M6)=R15, R15=PM(I5,M5);  {Write data,  }
                                                  {then get data}
      DM(I6,M6)=R15;                            {Write last data}
```

The dual fetch instruction in the loop transfers R15 to data memory first, then loads R15 from program memory (the host port). In effect, it writes the last value to data memory while reading the next value from the host port. That is why R15 is loaded from the port prior to entering the loop, and why R15 is written to data memory after the loop.

If transfers at the full clock rate are desired, then the host and ADSP-21020 should be synchronized (i.e., use the same clock). DMACK may be used to control throughput in this case.

If a handshake with the host is needed, as shown in Figures 9.23 and 9.24, then the maximum transfer rate will be half the ADSP-21020 clock rate.

Instruction Set Reference A

A.1 OVERVIEW

This appendix and the next one describe the ADSP-21020/21010 instruction set in detail. This appendix explains each instruction type, including the assembly language syntax and the opcode that the instruction assembles to. Many instruction types contain a field for specifying a compute operation (an operation that uses the ALU, multiplier or shifter). Because there are a large number of options available for this field, they are described separately in Appendix B. (Note that data moves between the MR registers and the register file are considered multiplier operations.)

Each instruction is specified in this section. The specification shows the **syntax** of the instruction, describes its **function**, gives one or two assembly-language **examples**, and specifies and describes the various fields of its **opcode**. The instructions are grouped into four categories:

I. Compute and move or modify instructions, which specify a compute operation in parallel with one or two data moves or an index register modify.

II. Program flow control instructions, which specify various types of branches, calls, returns and loops. Some of these instructions may also specify a compute operation and/or a data move.

III. Immediate data move instructions, which use immediate instruction fields as operands, or use immediate instruction fields for addressing.

IV. Miscellaneous instructions, such as bit modify and test, no operation and idle.

Many instructions can be conditional. These instructions are prefaced by an "IF" plus a condition mnemonic. In a conditional instruction, the execution of the entire instruction is based on the specified condition.

A Instruction Set Reference

Several sections that appear before the instruction specifications explain the notation conventions used in this instruction set reference (for both Appendix A and Appendix B).

- Section A.2 describes the notation and abbreviations used in the syntax description for each instruction.
- Section A.3 describes the notation and abbreviations used in the opcode description for each instruction.
- Section A.4 lists all condition and termination codes and their assembly language mnemonics.
- Section A.5 lists the assembly language mnemonics and opcode addresses for all universal registers.

A.2 INSTRUCTION SYNTAX NOTATION

The conventions in this section are used to describe the syntax of each instruction.

Notation	*Meaning*
UPPERCASE	explicit syntax; assembler keyword
;	instruction terminator
,	separates parallel operations in an instruction
italics	optional part of instruction
| between lines |	list of options (choose one)
<data*n*>	*n*-bit immediate data value
<addr*n*>	*n*-bit immediate address value
<reladdr*n*>	*n*-bit immediate PC-relative address value
<bit6>:<len6>	6-bit immediate bit position and length values (for shifter immediate operations)
compute	ALU, multiplier, shifter or multifunction operation (see Appendix B)
shiftimm	shifter immediate operation (see Appendix B)
condition	status condition (see Condition Codes)
termination	termination condition (see Condition Codes)
ureg	universal register (see Universal Registers)
sreg	system register (see Universal Registers)
dreg	R15-R0, F15-F0; register file location
Rn, Rx, Ry, Ra, Rm, Rs	R15-R0; register file location, fixed-point
Fn, Fx, Fy, Fa, Fm, Fs	F15-F0; register file location, floating-point
R3-0	R3, R2, R1, R0
R7-4	R7, R6, R5, R4
R11-8	R11, R10, R9, R8

Notation	*Meaning*
R15-12	R15, R14, R13, R12
F3-0	F3, F2, F1, F0
F7-4	F7, F6, F5, F4
F11-8	F11, F10, F9, F8
F15-12	F15, F14, F13, F12
Ia	I7-I0; DAG1 index register
Mb	M7-M0; DAG1 modify register
Ic	I15-I8; DAG2 index register
Md	M15-M8; DAG2 modify register
(DB)	Delayed branch
(LA)	Loop abort (pop loop, PC stacks on branch)
MR0F	Multiplier result accumulator 0, foreground
MR1F	Multiplier result accumulator 1, foreground
MR2F	Multiplier result accumulator 2, foreground
MR0B	Multiplier result accumulator 0, background
MR1B	Multiplier result accumulator 1, background
MR2B	Multiplier result accumulator 2, background

A.3 OPCODE NOTATION

In ADSP-21020 opcodes, some bits are explicitly defined to be zeros or ones. The values of other bits or fields set various parameters for the instruction. The terms in this section define these opcode bits and fields. Bits which are unspecified are ignored when the processor decodes the instruction, but are reserved for future use.

A Loop abort code

0	Do not pop loop, PC stacks on branch
1	Pop loop, PC stacks on branch

ADDR Immediate address field

AI Computation unit register

0000	MR0F
0001	MR1F
0010	MR2F
0100	MR0B
0101	MR1B
0110	MR2B

A Instruction Set Reference

B Branch type

0	Jump
1	Call

BOP Bit Operation select codes

000	Set
001	Clear
010	Toggle
100	Test
101	XOR

COMPUTE Compute operation field (see Appendix B)

COND Status Condition codes

0 - 31 (see Condition Codes)

CU Computation unit select codes

00	ALU
01	Multiplier
10	Shifter

DATA Immediate data field

DEC Counter decrement code

0	No counter decrement
1	Counter decrement

DMD Memory access direction

0	Read
1	Write

DMI Index (I) register numbers, DAG1

0 - 7

DMM Modify (M) register numbers, DAG1

0 - 7

DREG Register file locations

0 - 15

G DAG/Memory select

0 DAG1 or Data Memory
1 DAG2 or Program Memory

INC Counter increment code

0 No counter increment
1 Counter increment

J Jump Type

0 Non-delayed
1 Delayed

LPO Loop stack pop code

0 No stack pop
1 Stack pop

LPU Status stack push code

0 No stack push
1 Stack push

NUM Interrupt vector

0 - 7

OPCODE Computation unit opcodes (see Appendix B)

PMD Memory access direction

0 Read
1 Write

PMI Index (I) register numbers, DAG2

8 - 15

PMM Modify (M) register numbers, DAG2

8 - 15

RELADDR PC-relative address field

SPO Status stack pop code

0 No stack pop
1 Stack pop

SPU Loop stack push code

0 No stack push
1 Stack push

SREG System Register address

0 - 15 (see Universal Registers)

TERM Termination Condition codes

0 - 31 (see Condition Codes)

U Update, index (I) register

0 Pre-modify, no update
1 Post-modify with update

UREG Universal Register address

0 - 256 (see Universal Registers)

RA, RM, RN, RS, RX, RY Register file locations for compute operands and results

0 - 15

RXA ALU x-operand register file location for multifunction operations

8 - 11

RXM Multiplier x-operand register file location for multifunction operations

0 - 3

RYA ALU y-operand register file location for multifunction operations

12 - 15

RYM Multiplier y-operand register file location for multifunction operations

4 - 7

A.4 CONDITION CODES

No.	*Mnemonic*	*Description*	*True If*
0	EQ	ALU equal zero	AZ = 1
1	LT	ALU less than zero	[$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN and $\overline{\text{AZ}}$)] = 1
2	LE	ALU less than or equal zero	[$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or AZ = 1
3	AC	ALU carry	AC = 1
4	AV	ALU overflow	AV = 1
5	MV	Multiplier overflow	MV = 1
6	MS	Multiplier sign	MN = 1
7	SV	Shifter overflow	SV = 1
8	SZ	Shifter zero	SZ = 1
9	FLAG0_IN	Flag 0 input	FI0 = 1
10	FLAG1_IN	Flag 1 input	FI1 = 1
11	FLAG2_IN	Flag 2 input	FI2 = 1
12	FLAG3_IN	Flag 3 input	FI3 = 1
13	TF	Bit test flag	BTF = 1
14		Reserved	
15	LCE	Loop counter expired (DO UNTIL term)	CURLCNTR = 1
15	NOT LCE	Loop counter not expired (IF cond)	CURLCNTR ≠ 1
Bits 16-30 are the complements of bits 0-14			
16	NE	ALU not equal to zero	AZ = 0
17	GE	ALU greater than or equal zero	[$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN and $\overline{\text{AZ}}$)] = 0
18	GT	ALU greater than zero	[$\overline{\text{AF}}$ and (AN xor (AV and $\overline{\text{ALUSAT}}$)) or (AF and AN)] or AZ = 0
19	NOT AC	Not ALU carry	AC = 0
20	NOT AV	Not ALU overflow	AV = 0
21	NOT MV	Not multiplier overflow	MV = 0
22	NOT MS	Not multiplier sign	MN = 0
23	NOT SV	Not shifter overflow	SV = 0
24	NOT SZ	Not shifter zero	SZ = 0
25	NOT FLAG0_IN	Not Flag 0 input	FI0 = 0
26	NOT FLAG1_IN	Not Flag 1 input	FI1 = 0
27	NOT FLAG2_IN	Not Flag 2 input	FI2 = 0
28	NOT FLAG3_IN	Not Flag 3 input	FI3 = 0
29	NOT TF	Not bit test flag	BTF = 0
30		Reserved	
31	FOREVER	Always False (DO UNTIL)	always
31	TRUE	Always True (IF)	always

A.5 UNIVERSAL REGISTERS

Map 1 registers:

PC*	program counter
PCSTK	top of PC stack
PCSTKP	PC stack pointer
FADDR*	fetch address
DADDR*	decode address
LADDR	loop termination address
CURLCNTR	current loop counter
LCNTR	loop counter
R15 - R0	register file locations
I15 - I0	DAG1 and DAG2 index registers
M15 - M0	DAG1 and DAG2 modify registers
L15 - L0	DAG1 and DAG2 length registers
B15 - B0	DAG1 and DAG2 base registers

System Registers:

MODE1	mode control 1
MODE2	mode control 2
IRPTL	interrupt latch
IMASK	interrupt mask
IMASKP	interrupt mask pointer
ASTAT	arithmetic status
STKY	sticky status
USTAT1	user status reg 1
USTAT2	user status reg 2

* read-only

System Registers

b3 b2 b1 b0 \ (b7=0) b7 b6 b5 b4	0000	0001	0010	0011	0100	0101	0110	0111
0000	R0	I0	M0	L0	B0		FADDR	USTAT1
0001	R1	I1	M1	L1	B1		DADDR	USTAT2
0010	R2	I2	M2	L2	B2			
0011	R3	I3	M3	L3	B3		PC	
0100	R4	I4	M4	L4	B4		PCSTK	
0101	R5	I5	M5	L5	B5		PCSTKP	
0110	R6	I6	M6	L6	B6		LADDR	
0111	R7	I7	M7	L7	B7		CURLCNTR	
1000	R8	I8	M8	L8	B8		LCNTR	
1001	R9	I9	M9	L9	B9			IRPTL
1010	R10	I10	M10	L10	B10			MODE2
1011	R11	I11	M11	L11	B11			MODE1
1100	R12	I12	M12	L12	B12			ASTAT
1101	R13	I13	M13	L13	B13			IMASK
1110	R14	I14	M14	L14	B14			STKY
1111	R15	I15	M15	L15	B15			IMASKP

Figure A.1 Map 1 Universal Register Addresses

A Instruction Set Reference

Map 2 registers:

DMWAIT	wait state and page size control for data memory
DMBANK1	data memory bank 1 upper boundary
DMBANK2	data memory bank 2 upper boundary
DMBANK3	data memory bank 3 upper boundary
DMADR	copy of last data memory address
PMWAIT	wait state and page size control for program memory
PMBANK1	program memory bank 1 upper boundary
PMADR	copy of last program memory address
PX	48-bit PX1 and PX2 combination
PX1	bus exchange 1 (16 bits)
PX2	bus exchange 2 (32 bits)
TPERIOD	timer period
TCOUNT	timer counter

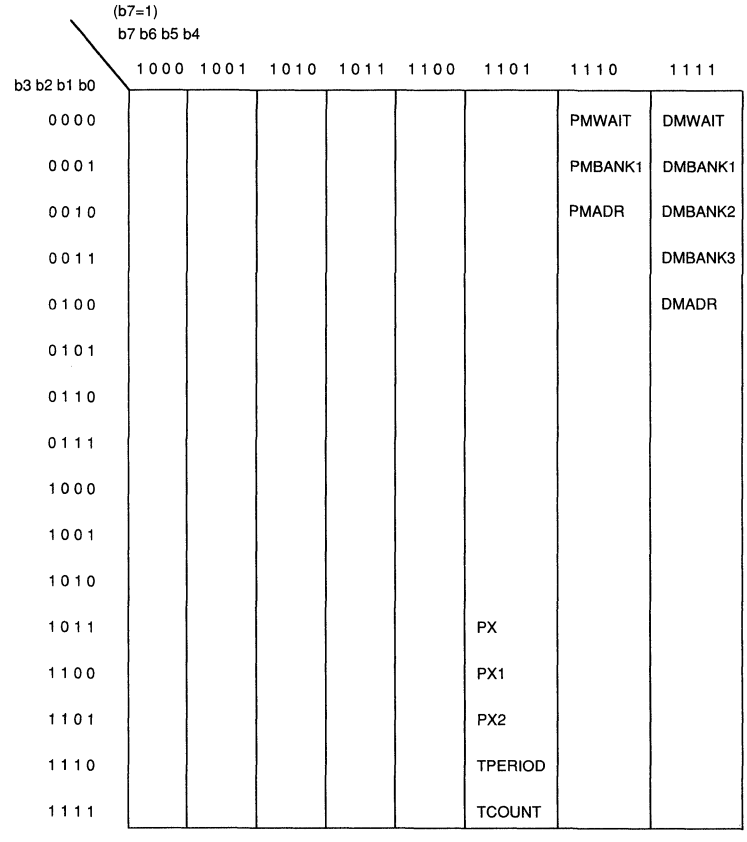


Figure A.2 Map 2 Universal Register Addresses

Instruction Set Reference A

Group I.
Compute and Move Instructions

1. Parallel data memory and program memory transfers with register file, optional compute operation A-12

2. Compute operation, optional condition A-13

3. Transfer between data or program memory and universal register, optional condition, optional compute operation A-14

4. PC-relative transfer between data or program memory and register file, optional condition, optional compute operation A-16

5. Transfer between two universal registers, optional condition, optional compute operation A-18

6. Immediate shift operation, optional condition, optional transfer between data or program memory and register file A-20

7. Index register modify, optional condition, optional compute operation A-22

compute / dreg↔DM / dreg↔PM

Syntax:

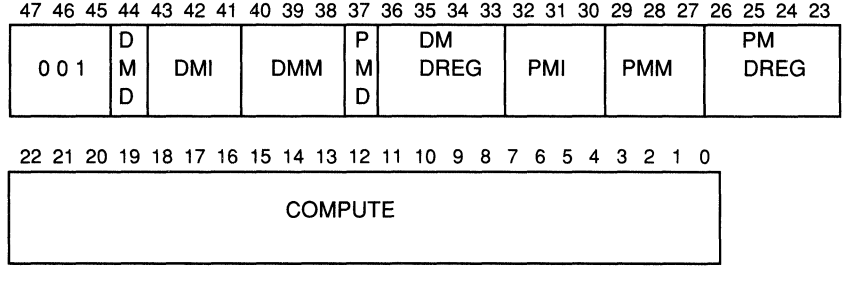
compute, | DM(Ia, Mb) = dreg1 / dreg1 = DM(Ia, Mb) | , | PM(Ic, Md) = dreg2 / dreg2 = PM(Ic, Md) | ;

Function:
Parallel accesses to data memory and program memory from the register file. The specified I registers address data memory and program memory. The I values are post-modified and updated by the specified M registers. Pre-modify offset addressing is not supported.

Examples:

```
R7=BSET R6 BY R0, DM(I0,M3)=R5, PM(I11,M15)=R4;

R8=DM(I4,M1), PM(I12 M12)=R0;
```

Opcode:



DMD and PMD select the access types (read or write). DMDREG and PMDREG specify register file locations. DMI and PMI specify I registers for data and program memory. DMM and PMM specify M registers used to update the I registers. The COMPUTE field defines a compute operation to be performed in parallel with the data accesses; this is a NOP if no compute operation is specified in the instruction.

compute

Syntax:

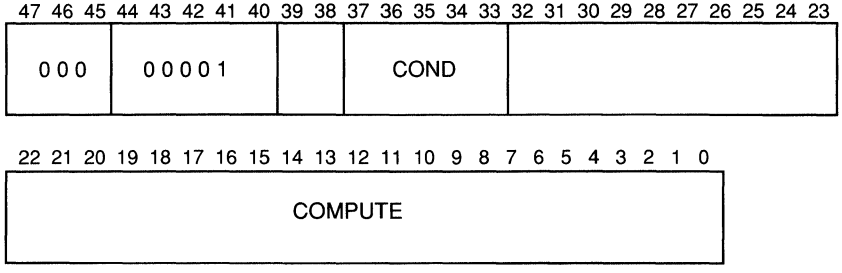
IF condition compute ;

Function:
Conditional compute instruction. The instruction is executed if the specified condition tests true.

Examples:

```
IF MS MRF=0;

F6=(F2+F3)/2;
```

Opcode:



The operation specified in the COMPUTE field is executed if the condition specified by COND is true. If no condition is specified in the instruction, COND is the TRUE condition, and the compute operation is always executed.

compute / ureg↔DM|PM , register modify

Syntax:

a. *IF condition* *compute,* | DM(Ia, Mb) | = ureg ;
| PM(Ic, Md) |

b. *IF condition* *compute,* | DM(Mb, Ia) | = ureg ;
| PM(Md, Ic) |

c. *IF condition* *compute,* ureg = | DM(Ia, Mb) | ;
| PM(Ic, Md) |

d. *IF condition* *compute,* ureg = | DM(Mb, Ia) | ;
| PM(Md, Ic) |

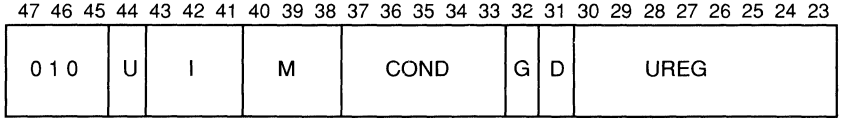
Function:
Access between data memory or program memory and a universal register. The specified I register addresses data memory or program memory. The I value is either pre-modified (M, I order) or post-modified (I, M order) by the specified M register. If it is post-modified, the I register is updated with the modified value. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

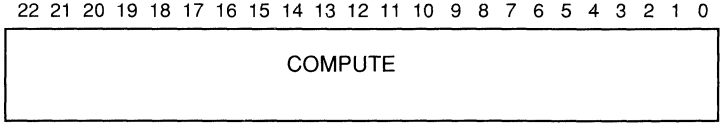
Examples:

```
R6=R3-R11, DM(I0,M1)=ASTAT;

IF NOT SV F8=CLIP F2 BY F14, PX=PM(I12,M12);
```

compute / ureg↔DM|PM , register modify

Opcode:





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

D selects the access type (read or write). G selects data memory or program memory. UREG specifies the universal register. I specifies the I register, and M specifies the M register. U selects either pre-modify without update or post-modify with update. The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

compute / dreg↔DM|PM , immediate modify

Syntax:

a. *IF condition* *compute,* | DM(Ia, <data6>) | = dreg ;
| PM(Ic, <data6>) |

b. *IF condition* *compute,* | DM(<data6>, Ia) | = dreg ;
| PM(<data6>, Ic) |

c. *IF condition* *compute,* dreg = | DM(Ia, <data6>) | ;
| PM(Ic, <data6>) |

d. *IF condition* *compute,* dreg = | DM(<data6>, Ia) | ;
| PM(<data6>, Ic) |

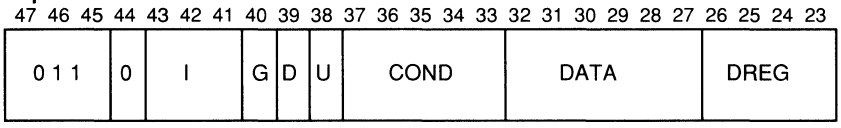
Function:
Access between data memory or program memory and the register file. The specified I register addresses data memory or program memory. The I value is either pre-modified (data order, I) or post-modified (I, data order) by the specified immediate data. If it is post-modified, the I register is updated with the modified value. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

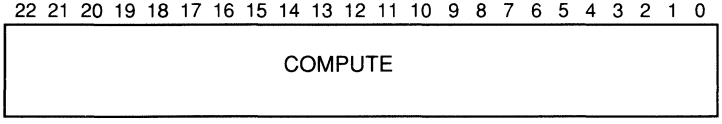
Examples:

```
IF FLAG0_IN F1=F5*F12, F11=PM(I10,40);

R12=R3 AND R1, DM(6,I1)=R6;
```

compute / dreg↔DM|PM , immediate modify

Opcode:





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

D selects the access type (read or write). G selects data memory or program memory. DREG specifies the register file location. I specifies the I register. DATA specifies a 6-bit, twos-complement modify value. U selects either pre-modify without update or post-modify with update. The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

compute / ureg↔ureg

Syntax:

IF condition *compute,* ureg1 = ureg2 ;

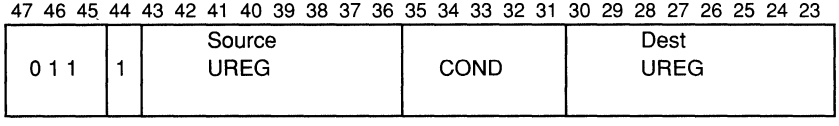
Function:
Transfer from one universal register to another. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

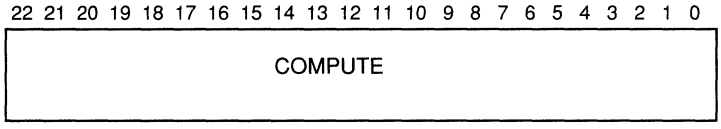
Examples:

```
IF TF MRF=R2*R6(SSFR), M4=R0;

LCNTR=L7;
```

compute / ureg↔ureg

Opcode:





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

Source UREG identifies the universal register source. Dest UREG identifies the universal register destination. The COMPUTE field defines a compute operation to be performed in parallel with the data transfer; this is a no-operation if no compute operation is specified in the instruction.

immediate shift / dreg↔DM|PM

Syntax:

a. *IF condition* shiftimm , | *DM(Ia, Mb)* / *PM(Ic, Md)* | = *dreg* ;

b. *IF condition* shiftimm , *dreg* = | *DM(Ia, Mb)* / *PM(Ic, Md)* | ;

Function:
An immediate shift operation is a shifter operation that takes immediate data as its Y-operand. The immediate data is one 8-bit value or two 6-bit values, depending on the operation. The x-operand and the result are register file locations.

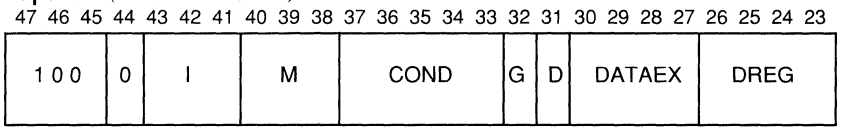
If an access to data or program memory from the register file is specified, it is performed in parallel with the shifter operation. The I register addresses data or program memory. The I value is post-modified by the specified M register and updated with the modified value. If a condition is specified, it affects entire instruction.

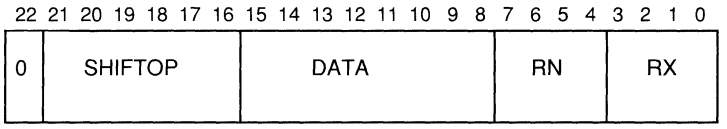
Examples:

```
IF GT R2=R6 LSHIFT BY 30, DM(I4,M4)=R0;

IF NOT SZ R3=FEXT R1 BY 8:4;
```

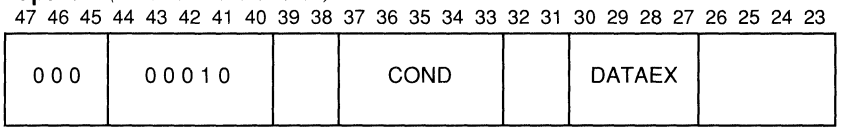
immediate shift / dreg↔DM|PM

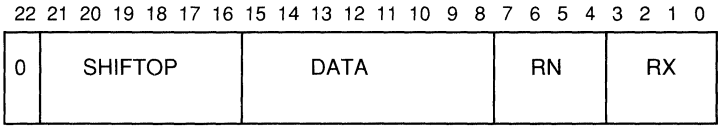
Opcode: *(with data access)*





Opcode: *(without data access)*





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

SHIFTOP specifies the shifter operation. The DATA field specifies an 8-bit immediate shift value. For shifter operations requiring two 6-bit values (a shift value and a length value), the DATAEX field adds 4 MSBs to the DATA field, creating a 12-bit immediate value. The six LSBs are the shift value, and the six MSBs are the length value.

If a memory access is specified, D selects the access type (read or write). G selects data memory or program memory. DREG specifies the register file location. I specifies the I register, which is post-modified and updated by the M register identified by M.

The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

compute / modify

Syntax:

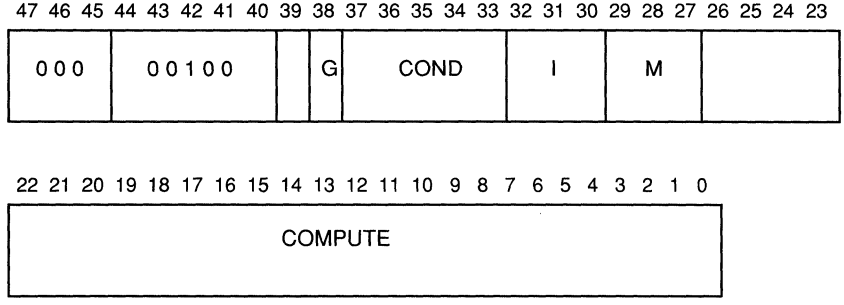
IF condition *compute,* MODIFY | (Ia, Mb) | ;
| (Ic, Md) |

Function:
Update of the specified I register by the specified M register. If a compute operation is specified, it is performed in parallel with the data access. If a condition is specified, it affects entire instruction.

Examples:

```
IF NOT FLAG2_IN R4=R6*R12(SUF), MODIFY(I10,M8);

IF NOT LCE MODIFY(I3,M1);
```

Opcode:



COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

G selects DAG1 or DAG2. I specifies the I register, and M specifies the M register. The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

Instruction Set Reference A

Group II.
Program Flow Control

8. Direct or PC-relative branch, optional condition A-24

9. Indirect or PC-relative branch, optional condition, optional compute operation A-26

11. Return from subroutine or interrupt, optional condition, optional compute operation A-28

12. Load loop counter, do loop until loop counter expired A-30

13. Do until termination A-32

direct jump|call

Syntax:

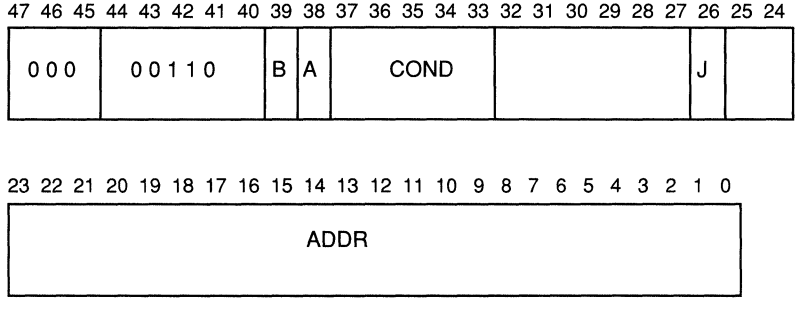
IF condition | JUMP / CALL | <addr24> / (PC, <reladdr24>) | (| *DB* / *LA* / *DB, LA* |) ;

Function:
A jump or call to the specified address or PC-relative address. The PC-relative address is a 24-bit, twos-complement value. If the delayed branch (DB) modifier is specified, the branch is delayed; otherwise, it is non-delayed. If the loop abort (LA) modifier is specified for a jump, the loop stacks and PC stack are popped when the jump is executed. You should use the (LA) modifier if the jump will transfer program execution outside of the loop. If there is no loop, or if the jump address is within the loop, you should not use the (LA) modifier. The (LA) modifier does not affect a call. If a condition is specified, it affects entire instruction.

Examples:

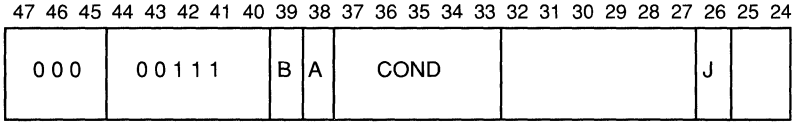
```
IF AV JUMP(PC,0x00A4)(LA);

CALL init (DB);                    {init is user-defined label}
```

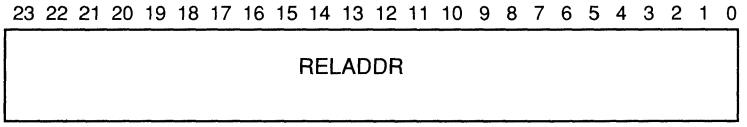
Opcode: *(with direct branch)*



direct jump/call

Opcode: *(with PC-relative branch)*





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

B selects the branch type, jump or call. J determines whether the branch is delayed or non-delayed. The ADDR field specifies a 24-bit program memory address. RELADDR is a 24-bit, twos-complement value that is added to the current PC value to generate the branch address. The A bit activates loop abort; a jump with loop abort pops the loop and PC stacks. (For calls, A is ignored.)

indirect jump|call / compute

Syntax:

IF condition | JUMP / CALL | | (Md, Ic) / (PC, <reladdr6>) | (| *DB* / *LA* / *DB, LA* |) , *compute* ;

Function:
A jump or call to the specified PC-relative address or pre-modified I register value. The PC-relative address is a 6-bit, twos-complement value. If an I register is specified, it is modified by the specified M register to generate the branch address. The I register is not affected by the modify operation.

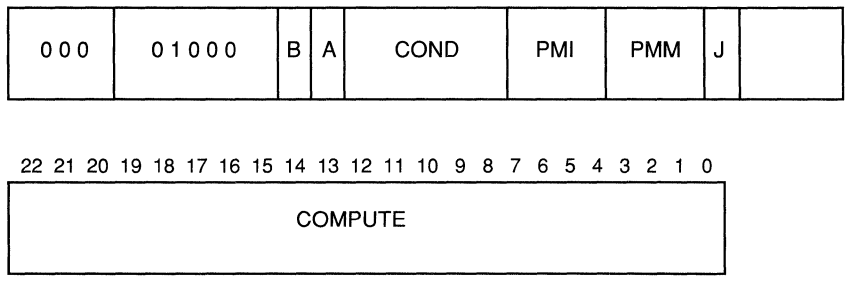
If the delayed branch (DB) modifier is specified, the branch is delayed; otherwise, it is non-delayed. If the loop abort (LA) modifier is specified for a jump, the loop stacks and PC stack are popped when the jump is executed. You should use the (LA) modifier if the jump will transfer program execution outside of the loop. If there is no loop, or if the jump address is within the loop, you should not use the (LA) modifier. The (LA) modifier does not affect a call.

If a compute operation is specified, it is performed in parallel with the branch. If a condition is specified, it affects entire instruction.

Examples:

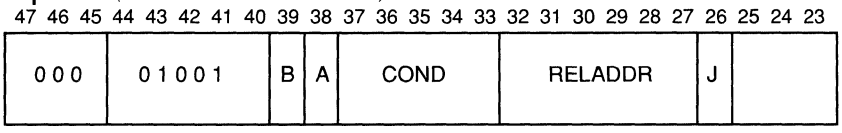
```
IF EQ JUMP(M8,I12), R6=R6-1;

CALL (PC,17)(DB), R12=MR2F;
```

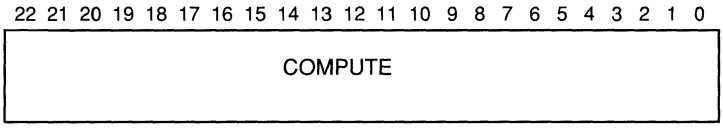
Opcode: *(with indirect branch)*



indirect jump/call / compute

Opcode: *(with PC-relative branch)*





COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

B selects the branch type, jump or call. J determines whether the branch is delayed or non-delayed. The A bit activates loop abort; a jump with loop abort pops the loop and PC stacks. (For calls, A is ignored.)

RELADDR is a 6-bit, twos-complement value that is added to the current PC value to generate the branch address. PMI specifies the I register for indirect branches. The I register is pre-modified but not updated by the M register specified by PMM.

The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

return from subroutinelinterrupt / compute

Syntax:

IF condition | RTS / RTI | (*DB*) , *compute* ;

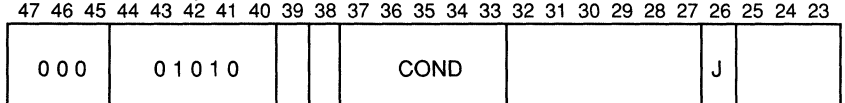
Function:
A return from a subroutine (RTS) or from an interrupt service routine (RTI). If the delayed branch (DB) modifier is specified, the return is delayed; otherwise, it is non-delayed.

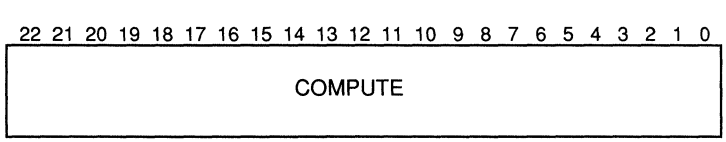
If a compute operation is specified, it is performed in parallel with the branch. If a condition is specified, it affects entire instruction.

Examples:

```
RTI, R6=R5 XOR R1;

IF NOT GT RTS(DB);
```

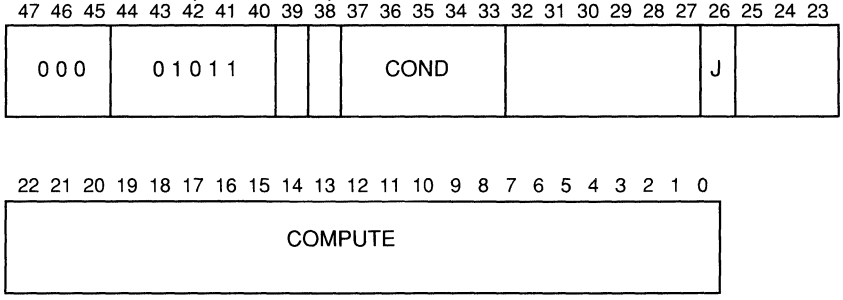
Opcode: *(return from subroutine)*





return from subroutinelinterrupt / compute

Opcode: *(return from interrupt)*



COND specifies the condition to test. If no condition is specified in the instruction, COND is the TRUE condition, and the instruction is always executed.

J determines whether the return is delayed or non-delayed. The COMPUTE field defines a compute operation to be performed in parallel with the data access; this is a no-operation if no compute operation is specified in the instruction.

do until counter expired

Syntax:

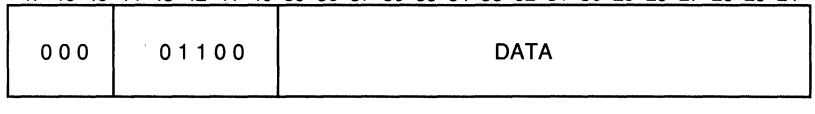
LCNTR = | <data16> / ureg | , DO | <addr24> / (<PC, reladdr24>) | UNTIL LCE ;

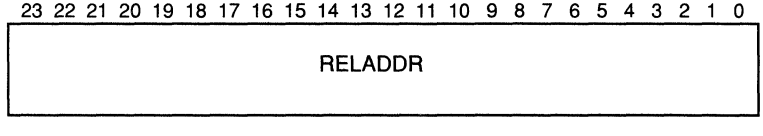
Function:
Sets up a counter-based program loop. The loop counter LCNTR is loaded with 16-bit immediate data or from a universal register. The loop start address is pushed on the PC stack. The loop end address and the LCE termination condition are pushed on the loop address stack. The end address can be either a label for an absolute 24-bit program memory address, or a PC-relative 24-bit twos-complement address. The LCNTR is pushed on the loop counter stack and becomes the CURLCNTR value. The loop executes until the CURLCNTR reaches zero.

Examples:

```
LCNTR=100, DO fmax UNTIL LCE;     {fmax is a program label}

LCNTR=R12, DO (PC,16) UNTIL LCE;
```

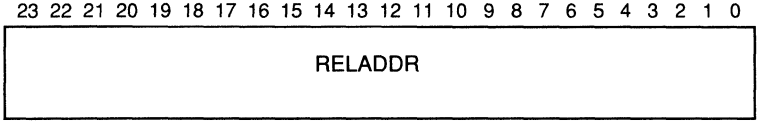
Opcode: *(with immediate loop counter load)*





do until counter expired

Opcode: *(with loop counter load from a universal register)*



RELADDR specifies the end-of-loop address relative to the DO LOOP instruction address. (The Assembler accepts an absolute address as well; it converts the absolute address to the equivalent relative address for coding.) The loop counter (LCNTR) is loaded with the 16-bit DATA value or with the contents of the register specified by UREG.

do until

Syntax:

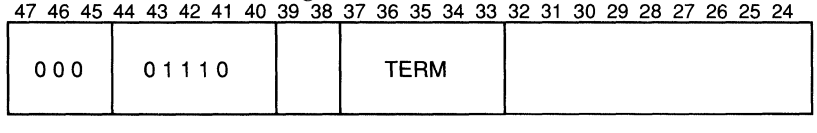
DO | <addr24> / (PC, <reladdr24>) | UNTIL termination ;

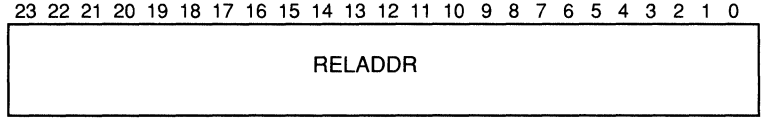
Function:
Sets up a condition-based program loop. The loop start address is pushed on the PC stack. The loop end address and the termination condition are pushed on the loop stack. The end address can be either a label for an absolute 24-bit program memory address or a PC-relative, 24-bit twos-complement address. The loop executes until the termination condition tests true.

Examples:

```
DO end UNTIL FLAG1_IN;              {end is a program label}

DO (PC,7) UNTIL AC;
```

Opcode: *(relative addressing)*





RELADDR specifies the end-of-loop address relative to the DO LOOP instruction address. (The Assembler accepts an absolute address as well; it converts the absolute address to the equivalent relative address for coding.) TERM specifies the termination condition.

Instruction Set Reference A

Group III.
Immediate Move

14. Transfer between data or program memory and universal register, direct addressing, immediate address A-34

15. Transfer between data or program memory and universal register, indirect addressing, immediate modifier A-35

16. Immediate data write to data or program memory A-36

17. Immediate data write to universal register A-37

A Immediate Move

ureg↔DM|PM (direct addressing)

Syntax:

a. | DM(<addr32>) | = ureg ;
   | PM(<addr24>) |

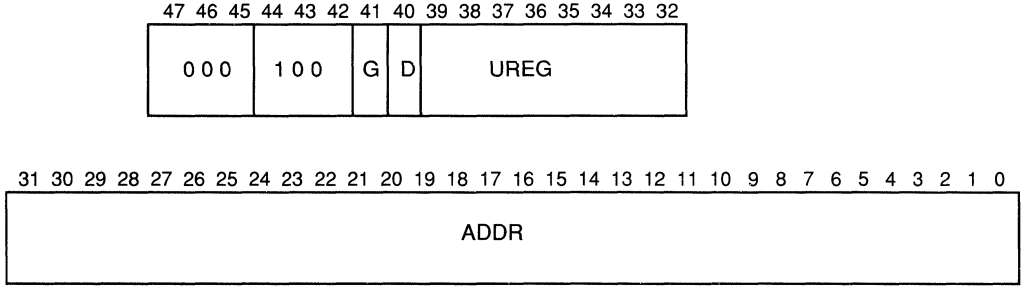
b. ureg = | DM(<addr32>) | ;
          | PM(<addr24>) |

Function:
Access between data memory or program memory and a universal register, with direct addressing. The entire data memory or program memory address is specified in the instruction. Data memory addresses are 32 bits wide (0 to $2^{32}-1$). Program memory addresses are 24 bits wide (0 to $2^{24}-1$).

Examples:

```
DM(temp)=MODE1;                    {temp is a program label}

DMWAIT=PM(0x489060);
```

Opcode:



D selects the access type (read or write). G selects the memory type (data or program). UREG specifies the number of a universal register. ADDR contains the immediate address value.

ureg↔DM|PM (indirect addressing)

Syntax:

a. | DM(<data32>, Ia) | = ureg ;
   | PM(<data24>, Ic) |

b. ureg = | DM(<data32>, Ia) | ;
          | PM(<data24>, Ic) |

Function:
Access between data memory or program memory and a universal register, with indirect addressing using I registers. The I register is pre-modified with an immediate value specified in the instruction. The I register is not updated. Data memory address modifiers are 32 bits wide (0 to 2^{32}–1). Program memory address modifiers are 24 bits wide (0 to 2^{24}–1).

Examples:

```
DM(24,I5)=TCOUNT;

USTAT1=PM(offs,I13);        {offs is a defined constant}
```

Opcode:

47 46 45	44	43 42 41	40	39 38 37 36 35 34 33 32
1 0 1	G	I	D	UREG

31 30 29 28 27 26 25 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
DATA

D selects the access type (read or write). G selects the memory type (data or program). UREG specifies the number of a universal register. ADDR contains the immediate address value. The I field specifies the I register. The DATA field specifies the immediate modify value for the I register.

A Immediate Move

immediate data → DM|PM

Syntax:

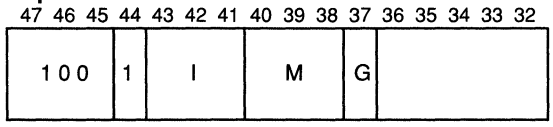
| DM(Ia, Mb) | = <data32> ;
| PM(Ic, Md) |

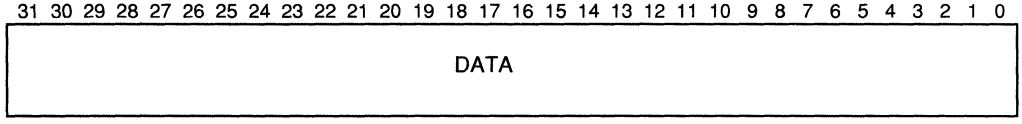
Function:
A write of 32-bit immediate data to data or program memory, with indirect addressing. The data is placed in the most significant 32 bits of the 40-bit memory word. The least significant 8 bits are loaded with 0s. The I register is post-modified and updated by the specified M register.

Examples:

```
DM(I4,M0)=19304;

PM(I14,M11)=count;          {count is user-defined constant}
```

Opcode:





I selects the I register, and M selects the M register. G selects the memory (data or program). DATA specifies the 32-bit immediate data.

immediate data → ureg

Syntax:

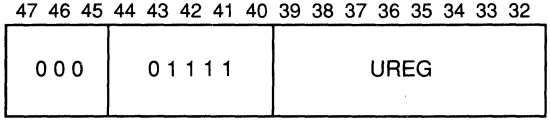
ureg = <data32> ;

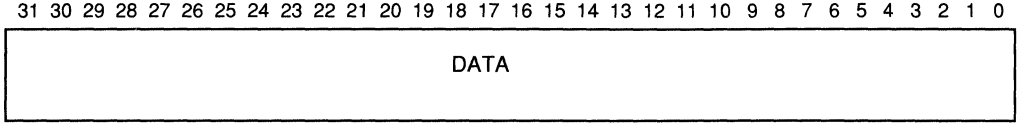
Function:
A write of 32-bit immediate data to a universal register. If the register is 40 bits wide, the data is placed in the most significant 32 bits, and the least significant 8 bits are loaded with 0s.

Examples:

```
IMASK=0xFFFC0060;

M15=mod1;                    {mod1 is user-defined constant}
```

Opcode:





UREG specifies the number of a universal register. The DATA field specifies the immediate data value.

A Instruction Set Reference

Instruction Set Reference A

Group IV. Miscellaneous

18. System register bit manipulation A-40

19. Immediate I register modify, with or without bit-reverse A-42

20. Push or Pop of loop and/or status stacks A-44

21. No operation (NOP) A-45

22. Idle A-46

system register bit manipulation

Syntax:

```
BIT | SET | sreg <data32> ;
    | CLR |
    | TGL |
    | TST |
    | XOR |
```

Function:
A bit manipulation operation on a system register. This instruction can set, clear, toggle or test specified bits, or compare (XOR) the system register with a specified data value. In the first four operations, the immediate data value is a mask. The set operation sets all the bits in the specified system register that are also set in the specified data value. The clear operation clears all the bits that are set in the data value. The toggle operation toggles all the bits that are set in the data value. The test operation sets the bit test flag (BTF in ASTAT) if all the bits that are set in the data value are also set in the system register. The XOR operation sets the bit test flag (BTF in ASTAT) if the system register value is the same as the data value.

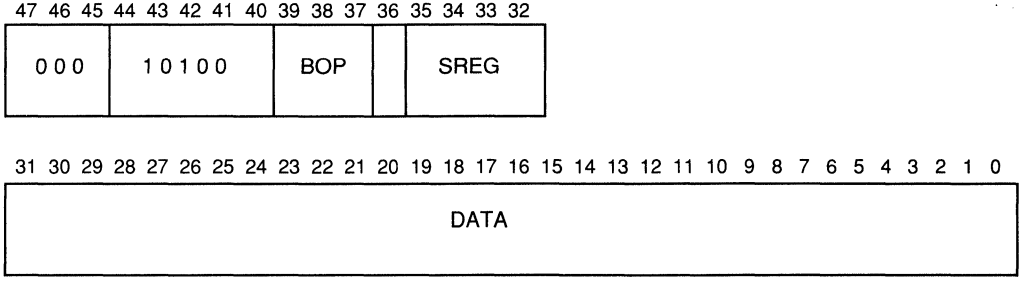
See shifter instructions for bit manipulation of data in the register file. See Appendix E for more information on system registers.

Examples:

```
BIT SET MODE2 0x00000070;

BIT TST ASTAT 0x00002000;
```

system register bit manipulation

Opcode:



BOP selects one of the five bit operations. SREG specifies the system register. DATA specifies the data value.

I register modify / bit-reverse

Syntax:

a. MODIFY | (Ia, <data32>) | ;
| (Ic, <data24>) |

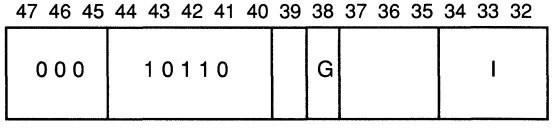
b. BITREV (Ia, <data32>) ;

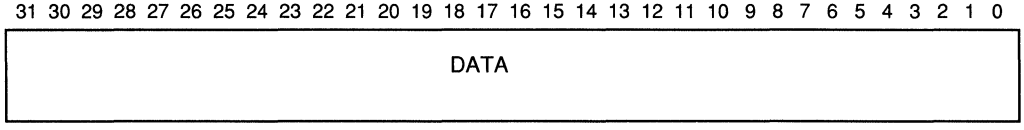
Function:
Modifies and updates the specified I register I by an immediate 32-bit (DAG1) or 24-bit (DAG2) data value. If the address is to be bit-reversed, you must specify a DAG1 register (I0-I7), and the modified value is bit-reversed before being written back to the I register. No address is output in either case.

Examples:

```
BITREV (I7,space);          {space is a defined constant}

MODIFY (I4,304);
```

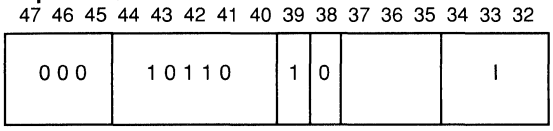
Opcode: *(without bit-reverse)*

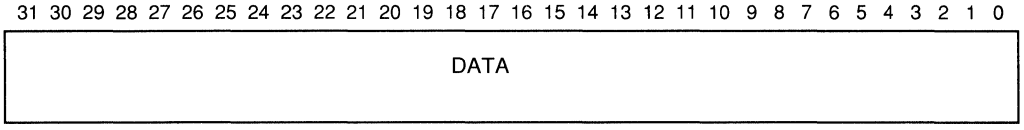




I register modify / bit-reverse

Opcode: *(with bit-reverse)*





G selects the address generator (DAG1 or DAG2). I selects the I register.
DATA specifies the immediate modifier.

push|pop stacks

Syntax:

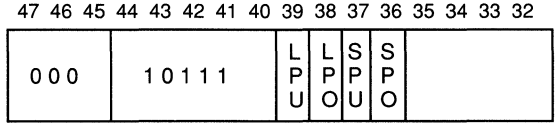
$$\left|\begin{matrix}PUSH\\POP\end{matrix}\right| \quad LOOP \quad , \quad \left|\begin{matrix}PUSH\\POP\end{matrix}\right| \quad STS \quad ;$$

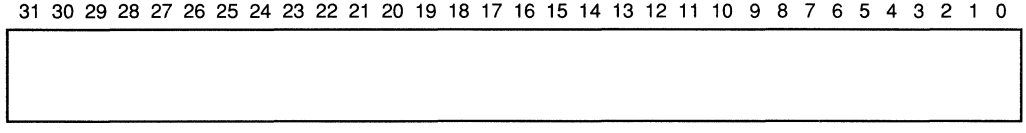
Function:
Pushes or pops the loop address and loop counter stacks, and/or pushes or pops the status stack.

Examples:

```
PUSH LOOP, PUSH STS;

POP STS;
```

Opcode:





LPU pushes the loop stacks. LPO pops the loop stacks. SPU pushes the status stack, and SPO pops the status stack.

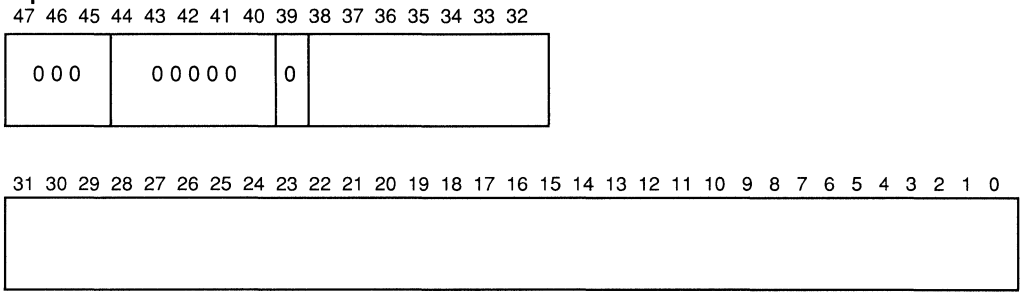
nop

Syntax:

NOP ;

Function:
A null operation; only increments the fetch address.

Opcode:



idle

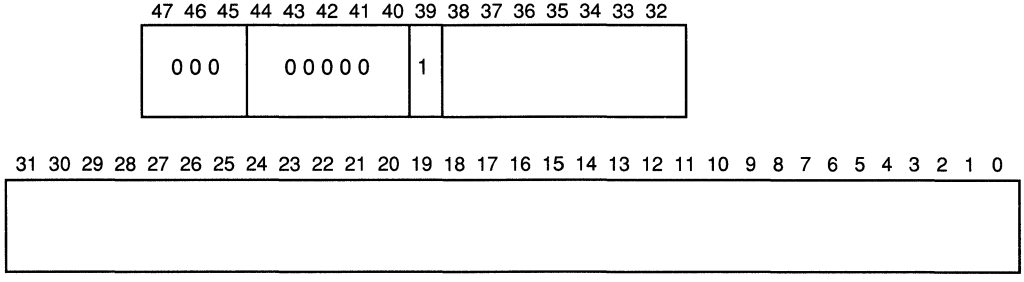
Syntax:

IDLE ;

Function:
Executes a NOP and puts the processor in a low power state. The processor remains in the low power state until an external interrupt occurs.

On return from the interrupt, execution continues at the instruction following the IDLE instruction.

Opcode:



Compute Operation Reference

B

B.1 OVERVIEW

Compute operations execute in the multiplier, the ALU and the shifter. The 23-bit compute field is like a mini-instruction within the ADSP-21000 instruction and can be specified for a variety of compute operations. This appendix describes each compute operation in detail, including its assembly language syntax and opcode field.

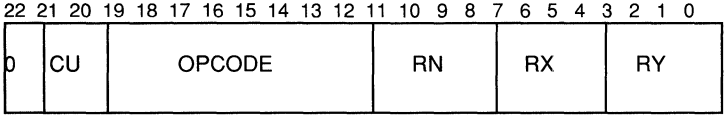
A compute operation is one of the following:

- Single-function operations involve a single computation unit.
- Multifunction operations specify parallel operation of the multiplier and the ALU or two operations in the ALU.
- The MR register transfer is a special type of compute operation dedicated to accessing the fixed-point accumulator in the multiplier. (See p. B-52).

The operations in each category are described in the following sections. For each operation, the assembly language syntax, the function, and the opcode format and contents are specified. Refer to the beginning of Appendix A for an explanation of the notation and abbreviations used.

B.2 SINGLE-FUNCTION OPERATIONS

The compute field of a single-function operation looks like:



An operation determined by OPCODE is executed in the computation unit specified by CU. The x- and the y-operands are received from data registers RX and RY. The result operand is returned to data register RN.

B Compute Operations

The CU (computation unit) field is defined as follows:

CU=00	ALU operations
CU=01	Multiplier operations
CU=10	Shifter operations

In some shifter operations, data register RN is used both as a destination for a result operand and as source for a third input operand.

The available operations and their 8-bit OPCODE values are listed in the following sections, organized by computation unit: ALU, multiplier and shifter. In each section, the syntax and opcodes for the operations are first summarized and then the operations are described in detail.

B.2.1 ALU Operations

The ALU operations are described in this section. Tables B.1 and B.2 summarize the syntax and opcodes for the fixed-point and floating-point ALU operations, respectively. The rest of this section contains detailed descriptions of each operation.

Syntax	*Opcode*
Rn = Rx + Ry	0000 0001
Rn = Rx – Ry	0000 0010
Rn = Rx + Ry + CI	0000 0101
Rn = Rx – Ry + CI – 1	0000 0110
Rn = (Rx + Ry)/2	0000 1001
COMP(Rx, Ry)	0000 1010
Rn = Rx + CI	0010 0101
Rn = Rx + CI – 1	0010 0110
Rn = Rx + 1	0010 1001
Rn = Rx – 1	0010 1010
Rn = –Rx	0010 0010
Rn = ABS Rx	0011 0000
Rn = PASS Rx	0010 0001
Rn = Rx AND Ry	0100 0000
Rn = Rx OR Ry	0100 0001
Rn = Rx XOR Ry	0100 0010
Rn = NOT Rx	0100 0011
Rn = MIN(Rx, Ry)	0110 0001
Rn = MAX(Rx, Ry)	0110 0010
Rn = CLIP Rx BY Ry	0110 0011

Table B.1 Fixed-Point ALU Operations

Compute Operations B

Syntax	*Opcode*
Fn = Fx + Fy	1000 0001
Fn = Fx – Fy	1000 0010
Fn = ABS (Fx + Fy)	1001 0001
Fn = ABS (Fx – Fy)	1001 0010
Fn = (Fx + Fy)/2	1000 1001
COMP(Fx, Fy)	1000 1010
Fn = –Fx	1010 0010
Fn = ABS Fx	1011 0000
Fn = PASS Fx	1010 0001
Fn = RND Fx	1010 0101
Fn = SCALB Fx BY Ry	1011 1101
Rn = MANT Fx	1010 1101
Rn = LOGB Fx	1100 0001
Rn = FIX Fx BY Ry	1101 1001
Rn = FIX Fx	1100 1001
Fn = FLOAT Rx BY Ry	1101 1010
Fn = FLOAT Rx	1100 1010
Fn = RECIPS Fx	1100 0100
Fn = RSQRTS Fx	1100 0101
Fn = Fx COPYSIGN Fy	1110 0000
Fn = MIN(Fx, Fy)	1110 0001
Fn = MAX(Fx, Fy)	1110 0010
Fn = CLIP Fx BY Fy	1110 0011

Table B.2 Floating-Point ALU Operations

Rn = Rx + Ry

Syntax:

Rn = Rx + Ry

Function:
Adds the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx – Ry

Syntax:

Rn = Rx – Ry

Function:
Subtracts the fixed-point field in register Ry from the fixed-point field in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx + Ry + CI

Syntax:

Rn = Rx + Ry + CI

Function:
Adds with carry (AC from ASTAT) the fixed-point fields in registers Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx – Ry + CI – 1

Syntax:

Rn = Rx – Ry + CI – 1

Function:
Subtracts with borrow (AC – 1 from ASTAT) the fixed-point field in register Ry from the fixed-point field in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = (Rx + Ry)/2

Syntax:

Rn = (Rx + Ry)/2

Function:
Adds the fixed-point fields in registers Rx and Ry and divides the result by 2. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in the MODE1 register.

Status flags:

AZ	Is set if the fixed-point output is all 0s, otherwise cleared
AU	Is cleared
AN	Is set if the most significant output bit is 1, otherwise cleared
AV	Is cleared
AC	Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS	Is cleared
AI	Is cleared

COMP(Rx, Ry)

Syntax:

COMP(Rx, Ry)

Function:
Compares the fixed-point field in register Rx with the fixed-point field in register Ry. Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Rx is smaller than the operand in register Ry.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24-31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of ASTAT is set if the X operand is greater than the Y operand (its value is the AND of ~AZ and ~AN); it is otherwise cleared.

Status flags:
AZ Is set if the operands in registers Rx and Ry are equal, otherwise cleared
AU Is cleared
AN Is set if the operand in the Rx register is smaller than the operand in the Ry register, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

ALU Fixed-Point

Rn = Rx + CI

Syntax:

Rn = Rx + CI

Function:
Adds the fixed-point field in register Rx with the carry flag from the ASTAT register (AC). The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF).

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx + CI – 1

Syntax:

Rn = Rx + CI – 1

Function:
Adds the fixed-point field in register Rx with the borrow from the ASTAT register (AC – 1). The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF).

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx + 1

Syntax:

Rn = Rx + 1

Function:
Increments the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status flags:

AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder, stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = Rx – 1

Syntax:

Rn = Rx – 1

Function:
Decrements the fixed-point operand in register Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set), underflow causes the minimum negative number (0x8000 0000) to be returned.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = –Rx

Syntax:

Rn = –Rx

Function:
Negates the fixed-point operand in Rx by twos complement. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. Negation of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status flags:
AZ Is set if the fixed-point output is all 0s
AU Is cleared
AN Is set if the most significant output bit is 1
AV Is set if the XOR of the carries of the two most significant adder stages is 1
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is cleared
AI Is cleared

Rn = ABS Rx

Syntax:

Rn = ABS Rx

Function:
Determines the absolute value of the fixed-point operand in Rx. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. ABS of the minimum negative number (0x8000 0000) causes an overflow. In saturation mode (the ALU saturation mode bit in MODE1 set), overflow causes the maximum positive number (0x7FFF FFFF) to be returned.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage is 1, otherwise cleared
AS Is set if the fixed-point operand in Rx is negative, otherwise cleared
AI Is cleared

Rn = PASS Rx

Syntax:

Rn = PASS Rx

Function:
Passes the fixed-point operand in Rx through the ALU to the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = Rx AND Ry

Syntax:

Rn = Rx AND Ry

Function:
Logically ANDs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = Rx OR Ry

Syntax:

Rn = Rx OR Ry

Function:
Logically ORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = Rx XOR Ry

Syntax:

Rn = Rx XOR Ry

Function:
Logically XORs the fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = NOT Rx

Syntax:

Rn = NOT Rx

Function:
Logically complements the fixed-point operand in Rx. The result is placed in the fixed-point field in Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = MIN(Rx, Ry)

Syntax:

Rn = MIN(Rx, Ry)

Function:
Returns the smaller of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = MAX(Rx, Ry)

Syntax:

Rn = MAX(Rx, Ry)

Function:
Returns the larger of the two fixed-point operands in Rx and Ry. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Rn = CLIP Rx BY Ry

Syntax:

Rn = CLIP Rx BY Ry

Function:
Returns the fixed-point operand in Rx if the absolute value of the operand in Rx is less than the absolute value of the fixed-point operand in Ry. Otherwise, returns |Ry| if Rx is positive, and –|Ry| if Rx is negative. The result is placed in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Status flags:
AZ Is set if the fixed-point output is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is cleared

Fn = Fx + Fy

Syntax:

Fn = Fx + Fy

Function:
Adds the floating-point operands in registers Fx and Fy. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared

Fn = Fx – Fy

Syntax:

Fn = Fx – Fy

Function:
Subtracts the floating-point operand in register Fy from the floating-point operand in register Fx. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, or if they are like-signed Infinities, otherwise cleared

Fn = ABS (Fx + Fy)

Syntax:

Fn = ABS (Fx + Fy)

Function:
Adds the floating-point operands in registers Fx and Fy, and places the absolute value of the normalized result in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns +Infinity (round-to-nearest) or +NORM.MAX (round-to-zero). Post-rounded denormal returns +Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is cleared
AV Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared

Fn = ABS (Fx – Fy)

Syntax:

Fn = ABS (Fx – Fy)

Function:
Subtracts the floating-point operand in Fy from the floating-point operand in Fx and places the absolute value of the normalized result in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns +Infinity (round-to-nearest) or +NORM.MAX (round-to-zero). Post-rounded denormal returns +Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is cleared
AV Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, or if they are like-signed Infinities, otherwise cleared

Fn = (Fx + Fy)/2

Syntax:

Fn = (Fx + Fy)/2

Function:
Adds the floating-point operands in registers Fx and Fy and divides the result by 2, by decrementing the exponent of the sum before rounding. The normalized result is placed in register Fn. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal results return ±Zero. A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:

AZ Is set if the post-rounded result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, or if they are opposite-signed Infinities, otherwise cleared

COMP(Fx, Fy)

Syntax:

COMP(Fx, Fy)

Function:
Compares the floating-point operand in register Fx with the floating-point operand in register Fy. Sets the AZ flag if the two operands are equal, and the AN flag if the operand in register Fx is smaller than the operand in register Fy.

The ASTAT register stores the results of the previous eight ALU compare operations in bits 24-31. These bits are shifted right (bit 24 is overwritten) whenever a fixed-point or floating-point compare instruction is executed. The MSB of ASTAT is set if the X operand is greater than the Y operand (its value is the AND of ~AZ and ~AN); it is otherwise cleared.

Status flags:
AZ Is set if the operands in registers Fx and Fy are equal, otherwise cleared
AU Is cleared
AN Is set if the operand in the Fx register is smaller than the operand in the Fy register, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = –Fx

Syntax:

Fn = –Fx

Function:
Complements the sign bit of the floating-point operand in Fx. The complemented result is placed in register Fn. A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result operand is a ±Zero, otherwise cleared
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN, otherwise cleared

Fn = ABS Fx

Syntax:

Fn = ABS Fx

Function:
Returns the absolute value of the floating-point operand in register Fx by setting the sign bit of the operand to 0. Denormal inputs are flushed to +Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result operand is +Zero, otherwise cleared.
AU Is cleared
AN Is cleared
AV Is cleared
AC Is cleared
AS Is set if the input operand is negative, otherwise cleared
AI Is set if the input operand is a NAN, otherwise cleared

Fn = PASS Fx

Syntax:

Fn = PASS Fx

Function:
Passes the floating-point operand in Fx through the ALU to the floating-point field in register Fn. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result operand is a ±Zero, otherwise cleared
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN, otherwise cleared

Fn = RND Fx

Syntax:

Fn = RND Fx

Function:
Rounds the floating-point operand in register Fx to a 32 bit boundary. Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result operand is a ±Zero, otherwise cleared
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the post-rounded result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN, otherwise cleared

Fn = SCALB Fx BY Ry

Syntax:

Fn = SCALB Fx BY Ry

Function:
Scales the exponent of the floating-point operand in Fx by adding to it the fixed-point twos-complement integer in Ry. The scaled floating-point result is placed in register Fn. Overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the result overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input is a NAN, an otherwise cleared

Rn = MANT Fx

Syntax:

Rn = MANT Fx

Function:
Extracts the mantissa (fraction bits with explicit hidden bit, excluding the sign bit) from the floating-point operand in Fx. The unsigned-magnitude result is left-justified (1.31 format) in the fixed-point field in Rn. Rounding modes are ignored and no rounding is performed because all results are inherently exact. Denormal inputs are flushed to ±Zero. A NAN or an Infinity input returns an all 1s result (–1 in signed fixed-point format).

Status flags:
AZ Is set if the result is zero, otherwise cleared
AU Is cleared
AN Is cleared
AV Is cleared
AC Is cleared
AS Is set if the input is negative, otherwise cleared
AI Is set if the input operands is a NAN or an Infinity, otherwise cleared

Rn = LOGB Fx

Syntax:

Rn = LOGB Fx

Function:
Converts the exponent of the floating-point operand in register Fx to an unbiased twos-complement fixed-point integer. The result is placed in the fixed-point field in register Rn. Unbiasing is done by subtracting 127 from the floating-point exponent in Fx. If saturation mode is not set, a ±Infinity input returns a floating-point +Infinity and a ±Zero input returns a floating-point –Infinity. If saturation mode is set, a ±Infinity input returns the maximum positive value (0x7FFF FFFF) and a ±Zero input returns the maximum negative value (0x8000 0000). Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the fixed-point result is zero, otherwise cleared
AU Is cleared
AN Is set if the result is negative, otherwise cleared
AV Is set if the input operand is an Infinity or a Zero, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input is a NAN, otherwise cleared

Rn = FIX Fx BY Ry / Rn = FIX Fx

Syntax:

Rn = FIX Fx BY Ry
Rn = FIX Fx

Function:
Converts the floating-point operand in Fx to a twos-complement 32-bit fixed-point integer result. If a scaling factor (Ry) is specified, the fixed-point twos-complement integer in Ry is added to the exponent of the floating-point operand in Fx before the conversion. The result of the conversion is right-justified (32.0 format) in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows and +Infinity return the maximum positive number (0x7FFF FFFF), and negative overflows and –Infinity return the minimum negative number (0x8000 0000).

Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode bit in MODE1. A NAN input returns a floating-point all 1s result. If saturation mode is not set, an Infinity input or a result that overflows returns a floating-point all 1s result. All positive underflows return zero. Negative underflows that are rounded-to-nearest return zero, and negative underflows that are rounded by truncation return –1 (0xFF FFFF FF00).

Status flags:
AZ Is set if the fixed-point result is Zero, otherwise cleared
AU Is set if the pre-rounded result is a denormal, otherwise cleared
AN Is set if the fixed-point result is negative, otherwise cleared
AV Is set if the conversion causes the floating-point mantissa to be shifted left, i.e if the floating-point exponent + scale bias is > 157 (127 + 31 – 1) or if the input is ±Infinity, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN or, when saturation mode is not set, either input is an Infinity or the result overflows, otherwise cleared

Fn = FLOAT Rx BY Ry / Fn = FLOAT Rx

Syntax:

Fn = FLOAT Rx BY Ry
Fn = FLOAT Rx

Function:
Converts the fixed-point operand in Rx to a floating-point result. If a scaling factor (Ry) is specified, the fixed-point twos-complement integer in Ry is added to the exponent of the floating-point result. The final result is placed in register Fn.

Rounding is to nearest (IEEE) or by truncation, as defined by the rounding mode, to a 40-bit boundary, regardless of the values of the rounding boundary bits in MODE1. The exponent scale bias may cause a floating-point overflow or a floating-point underflow. Overflow causes a ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero) to be returned; underflow causes a ±Zero to be returned.

Status flags:
AZ Is set if the result is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if the post-rounded result is a denormal, otherwise cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is set if the result overflows (unbiased exponent >127)
AC Is cleared
AS Is cleared
AI Is cleared

Fn = RECIPS Fx

Syntax:

Fn = RECIPS Fx

Function:
Creates an 8-bit accurate seed for 1/Fx, the reciprocal of Fx. The mantissa of the seed is determined from a ROM table using the 7 MSBs (excluding the hidden bit) of the Fx mantissa as an index. The unbiased exponent of the seed is calculated as the twos complement of the unbiased Fx exponent, decremented by one; i.e., if *e* is the unbiased exponent of Fx, then the unbiased exponent of Fn = $-e - 1$. The sign of the seed is the sign of the input. ±Zero returns ±Infinity and sets the overflow flag. If the unbiased exponent of Fx is greater than +125, the result is ±Zero. A NAN input returns an all 1s result.

The following code performs floating-point division using an iterative convergence algorithm.* The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). The following inputs are required: F0=numerator, F12=denominator, F11=2.0. The quotient is returned in F0. (The two highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F0=RECIPS F12, F7=F0;              {Get 8 bit seed R0=1/D}
F12=F0*F12;                        {D' = D*R0}
F7=F0*F7, F0=F11-F12;              {F0=R1=2-D', F7=N*R0}
F12=F0*F12;                        {F12=D'-D'*R1}
F7=F0*F7, F0=F11-F12;              {F7=N*R0*R1, F0=R2=2-D'}
 F12=F0*F12;                        {F12=D'=D'*R2}
 F7=F0*F7, F0=F11-F12;              {F7=N*R0*R1*R2, F0=R3=2-D'}
F0=F0*F7;                          {F7=N*R0*R1*R2*R3}
```

Note that this code segment can be made into a subroutine by adding an `RTS(DB)` clause to the third-to-last instruction.

Status flags:
AZ Is set if the floating-point result is ±Zero (unbiased exponent of Fx is greater than +125), otherwise cleared
AU Is cleared
AN Is set if the input operand is negative, otherwise cleared
AV Is set if the input operand is ±Zero, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is a NAN, otherwise cleared

* Cavanagh, J. 1984. *Digital Computer Arithmetic*. McGraw-Hill. Page 284.

Fn = RSQRTS Fx

Syntax:

Fn = RSQRTS Fx

Function: Creates a 4-bit accurate seed for $1/\sqrt{Fx}$, the reciprocal square root of Fx. The mantissa of the seed is determined from a ROM table using the LSB of the biased exponent of Fx concatenated with the 6 MSBs (excluding the hidden bit) of the mantissa of Fx as an index. The unbiased exponent of the seed is calculated as the twos complement of the unbiased Fx exponent, shifted right by one bit and decremented by one; i.e., if *e* is the unbiased exponent of Fx, then the unbiased exponent of Fn = –INT[*e*/2] – 1. The sign of the seed is the sign of the input. ±Zero returns ±Infinity and sets the overflow flag. +Infinity returns +Zero. A NAN input or a negative nonzero input returns an all 1s result.

The following code calculates a floating-point reciprocal square root ($1/\sqrt{x}$) using a Newton-Raphson iteration algorithm.* The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). To calculate the square root, simply multiply the result by the original input. The following inputs are required: F0=input, F8=3.0, F1=0.5. The result is returned in F4. (The four highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F4=RSQRTS F0;                    {Fetch 4-bit seed}
F12=F4*F4;                       {F12=X0^2}
F12=F12*F0;                      {F12=C*X0^2}
F4=F1*F4, F12=F8-F12;            {F4=.5*X0, F12=3-C*X0^2}
F4=F4*F12;                       {F4=X1=.5*X0(3-C*X0^2)}
F12=F4*F4;                       {F12=X1^2}
F12=F12*F0;                      {F12=C*X1^2}
F4=F1*F4, F12=F8-F12;            {F4=.5*X1, F12=3-C*X1^2}
 F4=F4*F12;                       {F4=X2=.5*X1(3-C*X1^2)}
 F12=F4*F4;                       {F12=X2^2}
 F12=F12*F0;                      {F12=C*X2^2}
 F4=F1*F4, F12=F8-F12;            {F4=.5*X2, F12=3-C*X2^2}
F4=F4*F12;                       {F4=X3=.5*X2(3-C*X2^2)}
```

Note that this code segment can be made into a subroutine by adding an `RTS(DB)` clause to the third-to-last instruction.

Status flags:

AZ Is set if the floating-point result is +Zero (Fx = +Infinity), otherwise cleared
AU Is cleared
AN Is set if the input operand is –Zero, otherwise cleared
AV Is set if the input operand is ±Zero, otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if the input operand is negative and nonzero, or a NAN, otherwise cleared

* Cavanagh, J. 1984. *Digital Computer Arithmetic.* McGraw-Hill. Page 278.

Fn = Fx COPYSIGN Fy

Syntax:

Fn = Fx COPYSIGN Fy

Function:
Copies the sign of the floating-point operand in register Fy to the floating-point operand from register Fx without changing the exponent or the mantissa. The result is placed in register Fn. A denormal input is flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = MIN(Fx, Fy)

Syntax:

Fn = MIN(Fx, Fy)

Function:
Returns the smaller of the floating-point operands in register Fx and Fy. A NAN input returns an all 1s result. MIN of +Zero and –Zero returns –Zero. Denormal inputs are flushed to ±Zero.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared.
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = MAX(Fx, Fy)

Syntax:

Fn = MAX(Fx, Fy)

Function:
Returns the larger of the floating-point operands in registers Fx and Fy. A NAN input returns an all 1s result. MAX of +Zero and –Zero returns +Zero. Denormal inputs are flushed to ±Zero.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared.
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

Fn = CLIP Fx BY Fy

Syntax:

Fn = CLIP Fx BY Fy

Function:
Returns the floating-point operand in Fx if the absolute value of the operand in Fx is less than the absolute value of the floating-point operand in Fy. Else, returns | Fy | if Fx is positive, and –| Fy | if Fx is negative. A NAN input returns an all 1s result. Denormal inputs are flushed to ±Zero.

Status flags:
AZ Is set if the floating-point result is ±Zero, otherwise cleared.
AU Is cleared
AN Is set if the floating-point result is negative, otherwise cleared
AV Is cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, otherwise cleared

B.2.2 Multiplier Operations

The multiplier operations are described in this section. Table B.3 summarizes the syntax and opcodes for the fixed-point and floating-point multiplier operations. The rest of this section contains detailed descriptions of each operation.

Fixed-point:

Syntax					*Opcode*
Rn	= Rx * Ry	*mod2**			01yx f00r
MRF	= Rx * Ry	*mod2**			01yx f10r
MRB	= Rx * Ry	*mod2**			01yx f11r
Rn	= MRF	+ Rx * Ry	*mod2**		10yx f00r
Rn	= MRB	+ Rx * Ry	*mod2**		10yx f01r
MRF	= MRF	+ Rx * Ry	*mod2**		10yx f10r
MRB	= MRB	+ Rx * Ry	*mod2**		10yx f11r
Rn	= MRF	– Rx * Ry	*mod2**		11yx f00r
Rn	= MRB	– Rx * Ry	*mod2**		11yx f01r
MRF	= MRF	– Rx * Ry	*mod2**		11yx f10r
MRB	= MRB	– Rx * Ry	*mod2**		11yx f11r
Rn	= SAT MRF		*mod1***		0000 f00x
Rn	= SAT MRB		*mod1***		0000 f01x
MRF	= SAT MRF		*mod1***		0000 f10x
MRB	= SAT MRB		*mod1***		0000 f11x
Rn	= RND MRF		*mod1***		0001 100x
Rn	= RND MRB		*mod1***		0001 101x
MRF	= RND MRF		*mod1***		0001 110x
MRB	= RND MRB		*mod1***		0001 111x
MRF	= 0				0001 0100
MRB	= 0				0001 0110
MR	= Rn				
Rn	= MR				

Floating-point:

Syntax	*Opcode*
Fn = Fx * Fy	0011 0000

* See Table B.4
** See Table B.5

y y-input; 1=signed, 0=unsigned
x x-input; 1=signed, 0=unsigned
f format; 1=fractional, 0=integer
r rounding; 1=yes, 0=no

Table B.3 Multiplier Operations

Mod2 in Table B.3 is an optional modifier, enclosed in parentheses, consisting of three or four letters that indicate whether the x-input is signed (S) or unsigned (U), whether the y-input is signed or unsigned, whether the inputs are in integer (I) or fractional (F) format and whether the result when written to the register file is to be rounded-to-nearest (R). The options for *mod2* and the corresponding opcode values are listed in Table B.4.

Mod2	*Opcode*
(SSI)	--11 0--0
(SUI)	--01 0--0
(USI)	--10 0--0
(UUI)	--00 0--0
(SSF)	--11 1--0
(SUF)	--01 1--0
(USF)	--10 1--0
(UUF)	--00 1--0
(SSFR)	--11 1--1
(SUFR)	--01 1--1
(USFR)	--10 1--1
(UUFR)	--00 1--1

Table B.4 Multiplier Mod2 Options

Similarly, *mod1* in Table B.3 is an optional modifier, enclosed in parentheses, consisting of two letters that indicate whether the input is signed (S) or unsigned (U) and whether the input is in integer (I) or fractional (F) format. The options for *mod1* and the corresponding opcode values are listed in Table B.5.

Mod1	*Opcode*
(SI) (for SAT only)	---- 0--1
(UI) (for SAT only)	---- 0--0
(SF)	---- 1--1
(UF)	---- 1--0

Table B.5 Multiplier Mod1 Options

Rn|MR = Rx * Ry

Syntax:

Rn	=	Rx * Ry	*mod2*
MRF	=	Rx * Ry	*mod2*
MRB	=	Rx * Ry	*mod2*

Function:
Multiplies the fixed-point fields in registers Rx and Ry. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

Multiplier Fixed-Point
Rn|MR = MR + Rx * Ry

Syntax:

Rn	=	MRF	+ Rx * Ry *mod2*
Rn	=	MRB	+ Rx * Ry *mod2*
MRF	=	MRF	+ Rx * Ry *mod2*
MRB	=	MRB	+ Rx * Ry *mod2*

Function:
Multiplies the fixed-point fields in registers Rx and Ry, and adds the product to the specified MR register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

Rn|MR = MR – Rx * Ry

Syntax:

Rn	= MRF	– Rx * Ry	*mod2*
Rn	= MRB	– Rx * Ry	*mod2*
MRF	= MRF	– Rx * Ry	*mod2*
MRB	= MRB	– Rx * Ry	*mod2*

Function:
Multiplies the fixed-point fields in registers Rx and Ry, and subtracts the product from the specified MR register value. If rounding is specified (fractional data only), the result is rounded. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

Rn|MR = SAT MR

Syntax:

Rn	= SAT MRF	*mod1*
Rn	= SAT MRB	*mod1*
MRF	= SAT MRF	*mod1*
MRB	= SAT MRB	*mod1*

Function:
If the value of the specified MR register is greater than the maximum value for the specified data format, the multiplier sets the result to the maximum value. Otherwise, the MR value is unaffected. The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is cleared
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

Rn|MR = RND MR

Syntax:

Rn	= RND MRF	*mod1*
Rn	= RND MRB	*mod1*
MRF	= RND MRF	*mod1*
MRB	= RND MRB	*mod1*

Function:
Rounds the specified MR value to nearest at bit 32 (the MR1-MR0 boundary). The result is placed either in the fixed-point field in register Rn or one of the MR accumulation registers, which must be the same MR register that provided the input. If Rn is specified, only the portion of the result that has the same format as the inputs is transferred (bits 31-0 for integers, bits 63-32 for fractional). The floating-point extension field in Rn is set to all 0s. If MRF or MRB is specified, the entire 80-bit result is placed in MRF or MRB.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the upper bits are not all zeros (signed or unsigned result) or ones (signed result). Number of upper bits depends on format. For a signed result, fractional=33, integer=49. For an unsigned result, fractional=32, integer=48.
MU Is set if the upper 48 bits of a fractional result are all zeros (signed or unsigned result) or ones (signed result) and the lower 32 bits are not all zeros. Integer results do not underflow.
MI Is cleared

MR=0

Syntax:
MRF = 0
MRB = 0

Function: Sets the value of the specified MR register to zero. All 80 bits (MR2, MR1, MR0) are cleared.

Status flags:
MN Is cleared
MV Is cleared
MU Is cleared
MI Is cleared

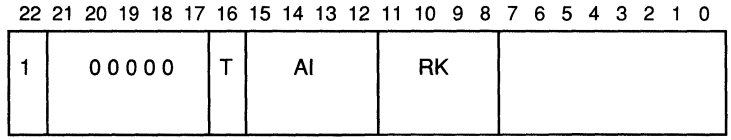
MR=Rn / Rn=MR

Function: A transfer to an MR register places the fixed-point field of register Rn in the specified MR register. The floating-point extension field in Rn is ignored. A transfer from an MR register places the specified MR register in the fixed-point field in register Rn. The floating-point extension field in Rn is set to all 0s.

Syntax:

MR0F = Rn	Rn = MR0F
MR1F = Rn	Rn = MR1F
MR2F = Rn	Rn = MR2F
MR0B = Rn	Rn = MR0B
MR1B = Rn	Rn = MR1B
MR2B = Rn	Rn = MR2B

Compute Field:



The MR register is specified by Ai and the data register by Rk. The direction of the transfer is determined by T (0=to register file, 1=to MR register).

Ai	MR Register
0000	MR0F
0001	MR1F
0010	MR2F
0100	MR0B
0101	MR1B
0110	MR2B

Status flags:
MN Is cleared
MV Is cleared
MU Is cleared
MI Is cleared

Fn = Fx * Fy

Syntax:

Fn = Fx * Fy

Function:
Multiplies the floating-point operands in registers Fx and Fy. The result is placed in the register Fn.

Status flags:
MN Is set if the result is negative, otherwise cleared
MV Is set if the unbiased exponent of the result is greater than 127, otherwise cleared
MU Is set if the unbiased exponent of the result is less than –126, otherwise cleared
MI Is set if either input is a NAN or if the inputs are ±Infinity and ±Zero, otherwise cleared

B.2.3 Shifter Operations

Shifter operations are described in this section. Table B.6 summarizes the syntax and opcodes for the shifter operations. The rest of this section contains detailed descriptions of each operation.

The shifter operates on the register file's 32-bit fixed-point fields (bits 39-8). Two-input shifter operations can take their y-input from the register file or from immediate data provided in the instruction. Either form uses the same opcode. However, the latter case, called an **immediate shift** or **shifter immediate** operation, is allowed only with instruction type 6, which has an immediate data field in its opcode for this purpose. All other instruction types must obtain the y-input from the register file when the compute operation is a two-input shifter operation.

Syntax	*Opcode*
Rn = LSHIFT Rx BY Ry | <data8>	0000 0000
Rn = Rn OR LSHIFT Rx BY Ry | <data8>	0010 0000
Rn = ASHIFT Rx BY Ry | <data8>	0000 0100
Rn = Rn OR ASHIFT Rx BY Ry | <data8>	0010 0100
Rn = ROT Rx BY RY | <data8>	0000 1000
Rn = BCLR Rx BY Ry | <data8>	1100 0100
Rn = BSET Rx BY Ry | <data8>	1100 0000
Rn = BTGL Rx BY Ry | <data8>	1100 1000
BTST Rx BY Ry | <data8>	1100 1100
Rn = FDEP Rx BY Ry | <bit6>:<len6>	0100 0100
Rn = Rn OR FDEP Rx BY Ry | <bit6>:<len6>	0110 0100
Rn = FDEP Rx BY Ry | <bit6>:<len6> (SE)	0100 1100
Rn = Rn OR FDEP Rx BY Ry | <bit6>:<len6> (SE)	0110 1100
Rn = FEXT Rx BY Ry | <bit6>:<len6>	0100 0000
Rn = FEXT Rx BY Ry | <bit6>:<len6> (SE)	0100 1000
Rn = EXP Rx	1000 0000
Rn = EXP Rx (EX)	1000 0100
Rn = LEFTZ Rx	1000 1000
Rn = LEFTO Rx	1000 1100

Instruction modifiers:
(SE) Sign extension of deposited or extracted field
(EX) Extended exponent extract

Table B.6 Shifter Operations

Rn = LSHIFT Rx BY Ry|<data8>

Syntax:

Rn = LSHIFT Rx BY Ry
Rn = LSHIFT Rx BY <data8>

Function:
Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero, otherwise cleared
SV Is set if the input is shifted to the left by more than 0, otherwise cleared
SS Is cleared

Rn = Rn OR LSHIFT Rx BY Ry|<data8>

Syntax:

Rn = Rn OR LSHIFT Rx BY Ry
Rn = Rn OR LSHIFT Rx BY <data8>

Function:
Logically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero, otherwise cleared
SV Is set if the input is shifted left by more than 0, otherwise cleared
SS Is cleared

Rn = ASHIFT Rx BY Ry|<data8>

Syntax:

Rn = ASHIFT Rx BY Ry
Rn = ASHIFT Rx BY <data8>

Function:
Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero , otherwise cleared
SV Is set if the input is shifted left by more than 0, otherwise cleared
SS Is cleared

Rn = Rn OR ASHIFT Rx BY Ry|<data8>

Syntax:

Rn = Rn OR ASHIFT Rx BY Ry
Rn = Rn OR ASHIFT Rx BY <data8>

Function:
Arithmetically shifts the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The shifted result is logically ORed with the fixed-point field of register Rn and then written back to register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a left shift, negative values select a right shift. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a shift of a 32-bit field from off-scale right to off-scale left.

Status flags:
SZ Is set if the shifted result is zero, otherwise cleared
SV Is set if the input is shifted left by more than 0, otherwise cleared
SS Is cleared

Rn = ROT Rx BY Ry|<data8>

Syntax:

Rn = ROT Rx BY Ry
Rn = ROT Rx BY <data8>

Function:
Rotates the fixed-point operand in register Rx by the 32-bit value in register Ry or by the 8-bit immediate value in the instruction. The rotated result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The shift values are twos-complement numbers. Positive values select a rotate left; negative values select a rotate right. The 8-bit immediate data can take values between –128 and 127 inclusive, allowing for a rotate of a 32-bit field from full right wrap around to full left wrap around.

Status flags:
SZ Is set if the rotated result is zero, otherwise cleared
SV Is cleared
SS Is cleared

Rn = BCLR Rx BY Ry|<data8>

Syntax:

Rn = BCLR Rx BY Ry
Rn = BCLR Rx BY <data8>

Function:
Clears a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be cleared. If the bit position value is greater than 31 or less than 0, no bits are cleared.

Status flags:
SZ Is set if the output operand is 0, otherwise cleared
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation affects a bit in a register file location. There is also a bit manipulation *instruction* that affects one or more bits in a system register. This BIT CLR instruction should not be confused with the BCLR shifter operation. See Appendix E for more information on BIT CLR.

Rn = BSET Rx BY Ry|<data8>

Syntax:

Rn = BSET Rx BY Ry
Rn = BSET Rx BY <data8>

Function:
Sets a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be set. If the bit position value is greater than 31 or less than 0, no bits are set.

Status flags:
SZ Is set if the output operand is 0, otherwise cleared
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation affects a bit in a register file location. There is also a bit manipulation *instruction* that affects one or more bits in a system register. This BIT SET instruction should not be confused with the BSET shifter operation. See Appendix E for more information on BIT SET.

Rn = BTGL Rx BY Ry|<data8>

Syntax:

Rn = BTGL Rx BY Ry
Rn = BTGL Rx BY <data8>

Function:
Toggles a bit in the fixed-point operand in register Rx. The result is placed in the fixed-point field of register Rn. The floating-point extension field of Rn is set to all 0s. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be toggled. If the bit position value is greater than 31 or less than 0, no bits are toggled.

Status flags:
SZ Is set if the output operand is 0, otherwise cleared
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation affects a bit in a register file location. There is also a bit manipulation *instruction* that affects one or more bits in a system register. This BIT TGL instruction should not be confused with the BTGL shifter operation. See Appendix E for more information on BIT TGL.

BTST Rx BY Ry|<data8>

Syntax:

BTST Rx BY Ry
BTST Rx BY <data8>

Function:
Tests a bit in the fixed-point operand in register Rx. The SZ flag is set if the bit is a 0 and cleared if the bit is a 1. The position of the bit is the 32-bit value in register Ry or the 8-bit immediate value in the instruction. The 8-bit immediate data can take values between 31 and 0 inclusive, allowing for any bit within a 32-bit field to be tested. If the bit position value is greater than 31 or less than 0, no bits are tested.

Status flags:
SZ Is cleared if the tested bit is a 1, is set if the tested bit is a 0 or if the bit position is greater than 31
SV Is set if the bit position is greater than 31, otherwise cleared
SS Is cleared

Note: This compute operation tests a bit in a register file location. There is also a bit manipulation *instruction* that tests one or more bits in a system register. This BIT TST instruction should not be confused with the BTST shifter operation. See Appendix E for more information on BIT TST.

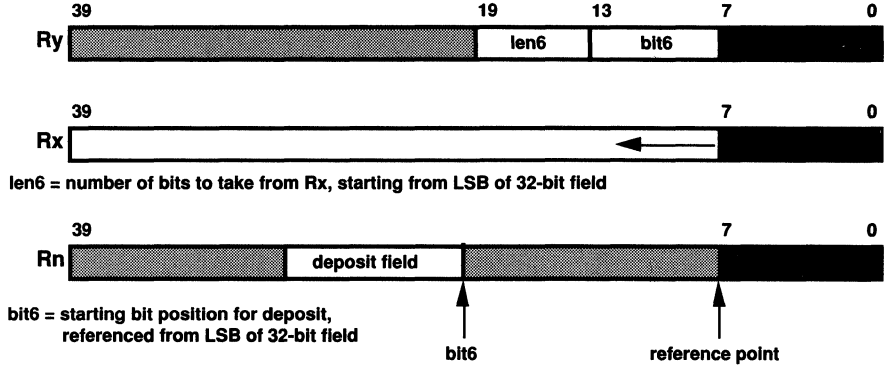
Rn = FDEP Rx BY Ry|<bit6>:<len6>

Syntax:

Rn = FDEP Rx BY Ry
Rn = FDEP Rx BY <bit6>:<len6>

Function:

Deposits a field from register Rx to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left and to the right of the deposited field are set to 0. The floating-pt. extension field of Rn (bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.



Example: If len6=14 and bit6=13, then the 14 bits of Rx are deposited in Rn bits 34-21 (of the 40-bit word).

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|        Rx
                     \-------------/
                          14 bits

39       31       23       15       7        0
|00000abc|defghijk|lmn00000|00000000|00000000|        Rn
      \--------------/
                |
              bit position 13 (from reference point)
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6>

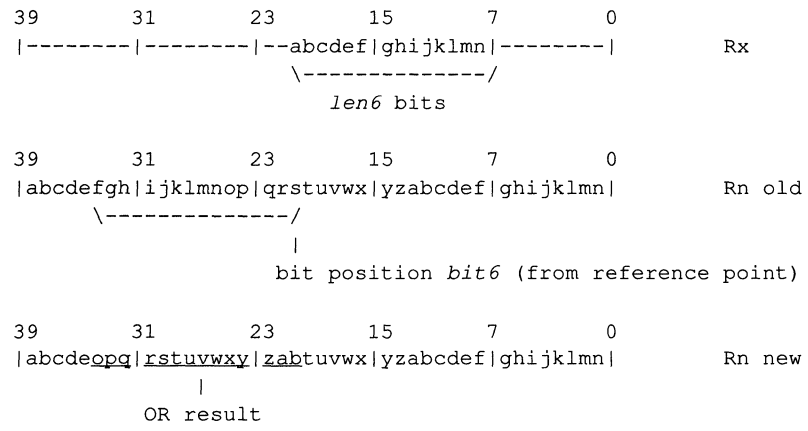
Syntax:

Rn = Rn OR FDEP Rx BY Ry
Rn = Rn OR FDEP Rx BY <bit6>:<len6>

Function:

Deposits a field from register Rx to register Rn. The field value is logically ORed bitwise with the specified field of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits, and to bit positions ranging from 0 to off-scale left.

Example:



Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

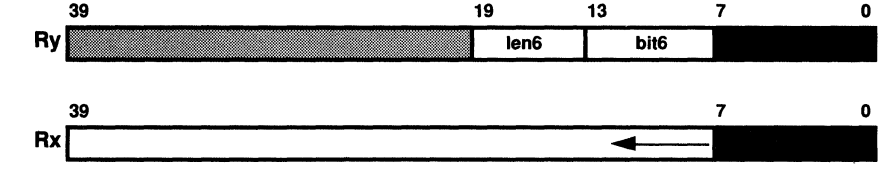
Rn = FDEP Rx BY Ry|<bit6>:<len6> (SE)

Syntax:

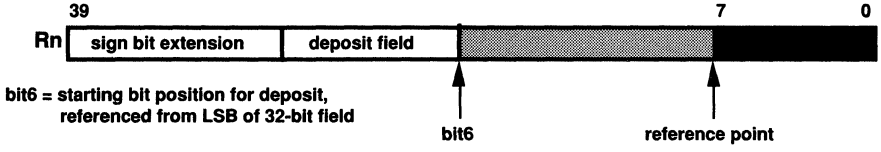
Rn = FDEP Rx BY Ry (SE)
Rn = FDEP Rx BY <bit6>:<len6> (SE)

Function:

Deposits and sign-extends a field from register Rx to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the deposited field, unless the MSB of the deposited field is off-scale left. Bits to the right of the deposited field are set to 0. The floating-point extension field of Rn (bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.



len6 = number of bits to take from Rx, starting from LSB of 32-bit field



bit6 = starting bit position for deposit, referenced from LSB of 32-bit field

bit6 reference point

Example:

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|        Rx
                     \---------------/
                        len6 bits

39       31       23       15       7        0
|aaaaaabc|defghijk|lmn00000|00000000|00000000|        Rn
\----/\--------------/
sign                |
extension          bit position bit6 (from reference point)
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = Rn OR FDEP Rx BY Ry|<bit6>:<len6> (SE)

Syntax:

Rn = Rn OR FDEP Rx BY Ry (SE)
Rn = Rn OR FDEP Rx BY <bit6>:<len6> (SE)

Function:

Deposits and sign-extends a field from register Rx to register Rn. The sign-extended field value is logically ORed bitwise with the value of register Rn and the new value is written back to register Rn. The input field is right-aligned within the fixed-point field of Rx. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is deposited in the fixed-point field of Rn, starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for deposit of fields ranging in length from 0 to 32 bits into bit positions ranging from 0 to off-scale left.

Example:

```
39       31       23       15       7        0
|--------|--------|--abcdef|ghijklmn|--------|        Rx
                     \-------------/
                        len6 bits

39       31       23       15       7        0
|aaaaaabc|defghijk|lmn00000|00000000|00000000|
\----/\--------------/
sign                |
extension          bit position bit6 (from reference point)

39       31       23       15       7        0
|abcdefgh|ijklmnop|qrstuvwx|yzabcdef|ghijklmn|        Rn old

39       31       23       15       7        0
|vwxyzabc|defghijk|lmntuvwx|yzabcdef|ghijklmn|        Rn new
          |
      OR result
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are deposited to the left of the 32-bit fixed-point output field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

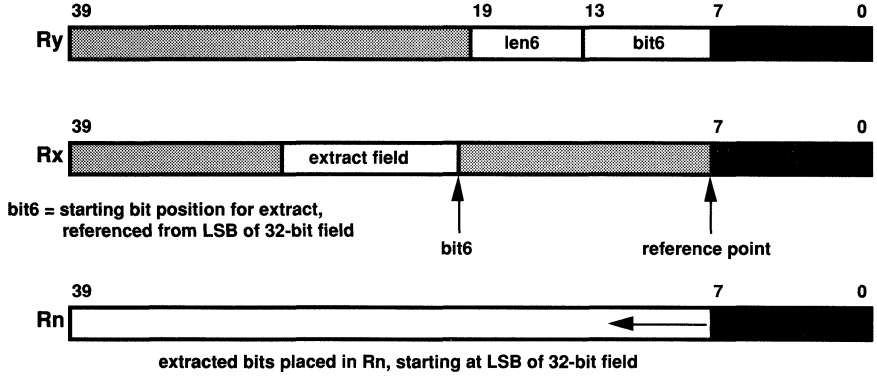
Rn = FEXT Rx BY Ry|<bit6>:<len6>

Syntax:

Rn = FEXT Rx BY Ry
Rn = FEXT Rx BY <bit6>:<len6>

Function:

Extracts a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. Bits to the left of the extracted field are set to 0 in register Rn. The floating-point extension field of Rn (bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits, and from bit positions ranging from 0 to off-scale left.



Example:

```
39       31       23       15       7        0
|-----abc|defghijk|lmn-----|--------|--------|        Rx
      \--------------/
         len6 bits    |
                      bit position bit6 (from reference point)

39       31       23       15       7        0
|00000000|00000000|00abcdef|ghijklmn|00000000|        Rn
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are extracted from the left of the 32-bit fixed-point, input field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = FEXT Rx BY Ry|<bit6>:<len6> (SE)

Syntax:

Rn = FEXT Rx BY Ry (SE)
Rn = FEXT Rx BY <bit6>:<len6> (SE)

Function:

Extracts and sign-extends a field from register Rx to register Rn. The output field is placed right-aligned in the fixed-point field of Rn. Its length is determined by the len6 field in register Ry or by the immediate len6 field in the instruction. The field is extracted from the fixed-point field of Rx starting from a bit position determined by the bit6 field in register Ry or by the immediate bit6 field in the instruction. The MSBs of Rn are sign-extended by the MSB of the extracted field, unless the MSB is extracted from off-scale left. The floating-point extension field of Rn(bits 7-0 of the 40-bit word) is set to all 0s. Bit6 and len6 can take values between 0 and 63 inclusive, allowing for extraction of fields ranging in length from 0 to 32 bits and from bit positions ranging from 0 to off-scale left.

Example:

```
39       31       23       15       7        0
|-----abc|defghijk|lmn-----|--------|--------|        Rx
      \--------------/
         len6 bits   |
                   bit position bit6 (from reference point)

39       31       23       15       7        0
|aaaaaaaa|aaaaaaaa|aaabcdef|ghijklmn|00000000|        Rn
\-------------------/
    sign extension
```

Status flags:

SZ Is set if the output operand is 0, otherwise cleared
SV Is set if any bits are extracted from the left of the 32-bit fixed-point input field (i.e., if len6 + bit6 > 32), otherwise cleared
SS Is cleared

Rn = EXP Rx

Syntax:

Rn = EXP Rx

Function:
Extracts the exponent of the fixed-point operand in Rx. The exponent is placed in the shf8 field in register Rn. The exponent is calculated as the twos complement of:

leading sign bits in Rx – 1

Status flags:
SZ Is set if the extracted exponent is 0, otherwise cleared
SV Is cleared
SS Is set if the fixed-point operand in Rx is negative (bit 31 is a 1), otherwise cleared

Rn = EXP Rx (EX)

Syntax:

Rn = EXP Rx (EX)

Function:
Extracts the exponent of the fixed-point operand in Rx, assuming that the operand is the result of an ALU operation. The exponent is placed in the shf8 field in register Rn. If the AV status bit is set, a value of +1 is placed in the shf8 field to indicate an extra bit (the ALU overflow bit). If the AV status bit is not set, the exponent is calculated as the twos complement of:

leading sign bits in Rx – 1

Status flags:
SZ Is set if the extracted exponent is 0, otherwise cleared
SV Is cleared
SS Is set if the exclusive OR of the AV status bit and the sign bit (bit 31) of the fixed-point operand in Rx is equal to 1, otherwise cleared

Rn = LEFTZ Rx

Syntax:

Rn = LEFTZ Rx

Function:
Extracts the number of leading 0s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status flags:
SZ Is set if the MSB of Rx is 1, otherwise cleared
SV Is set if the result is 32, otherwise cleared
SS Is cleared

Rn = LEFTO Rx

Syntax:

Rn = LEFTO Rx

Function:
Extracts the number of leading 1s from the fixed-point operand in Rx. The extracted number is placed in the bit6 field in Rn.

Status flags:
SZ Is set if the MSB of Rx is 0, otherwise cleared
SV Is set if the result is 32, otherwise cleared
SS Is cleared

B.2.4 Multifunction Computations

Multifunction computations are of three types, each of which has a different format for the 23-bit compute field:

- Dual add/subtract
- Parallel multiplier/ALU
- Parallel multiplier and add/subtract

Dual Add/Subtract (Fixed-Pt.)

The dual add/subtract operation computes the sum and the difference of two inputs and returns the two results to different registers. There are fixed-point and floating-point versions of this operation.

Fixed-Point:

Syntax:

Ra = Rx + Ry, Rs = Rx – Ry

Compute Field:

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	RS	RA	RX	RY

Function:
Does a dual add/subtract of the fixed-point fields in registers Rx and Ry. The sum is placed in the fixed-point field of register Ra and the difference in the fixed-point field of Rs. The floating-point extension fields of Ra and Rs are set to all 0s. In saturation mode (the ALU saturation mode bit in MODE1 set) positive overflows return the maximum positive number (0x7FFF FFFF), and negative overflows return the minimum negative number (0x8000 0000).

Status flags:
AZ Is set if either of the fixed-point outputs is all 0s, otherwise cleared
AU Is cleared
AN Is set if the most significant output bit is 1 of either of the outputs, otherwise cleared
AV Is set if the XOR of the carries of the two most significant adder stages of either of the outputs is 1, otherwise cleared
AC Is set if the carry from the most significant adder stage of either of the outputs is 1, otherwise cleared
AS Is cleared
AI Is cleared

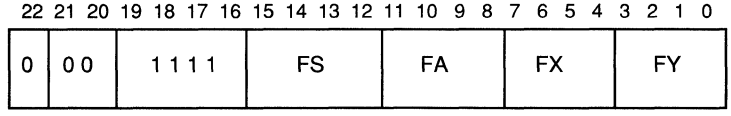
Dual Add/Subtract (Floating-Pt.)

Floating-Point:

Syntax:

Fa = Fx + Fy, Fs = Fx – Fy

Compute Field:



Function:
Does a dual add/subtract of the floating-point operands in registers Fx and Fy. The normalized results are placed in registers Fa and Fs: the sum in Fa and the difference in Fs. Rounding is to nearest (IEEE) or by truncation, to a 32-bit or to a 40-bit boundary, as defined by the rounding mode and rounding boundary bits in MODE1. Post-rounded overflow returns ±Infinity (round-to-nearest) or ±NORM.MAX (round-to-zero). Post-rounded denormal returns ±Zero. Denormal inputs are flushed to ±Zero. A NAN input returns an all 1s result.

Status flags:
AZ Is set if either of the post-rounded results is a denormal (unbiased exponent < –126) or zero, otherwise cleared
AU Is set if either post-rounded result is a denormal, otherwise cleared
AN Is set if either of the floating-point results is negative, otherwise cleared
AV Is set if either of the post-rounded results overflows (unbiased exponent > +127), otherwise cleared
AC Is cleared
AS Is cleared
AI Is set if either of the input operands is a NAN, or if both of the input operands are Infinities, otherwise cleared

Parallel Multiplier & ALU (Fixed-Pt.)

The parallel multiplier/ALU operation performs a multiply or multiply/accumulate and one of the following ALU operations: add, subtract, average, fixed-point to floating-point or floating-point to fixed-point conversion, or floating-point ABS, MIN or MAX.

For detailed information about a particular operation, see the individual descriptions under Single-Function Operations.

Fixed-Point:

Syntax: See Table B.7

Compute Field:

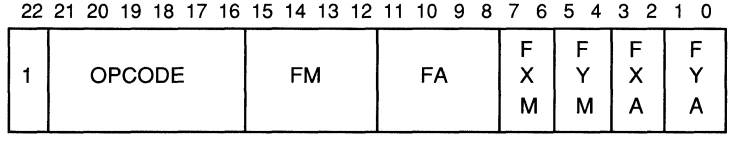
22	21 20 19 18 17 16	15 14 13 12	11 10 9 8	7 6	5 4	3 2	1 0
1	OPCODE	RM	RA	RXM	RYM	RXA	RYA

Parallel Multiplier & ALU (Floating-Pt.)

Floating-Point:

Syntax: See Table B.7

Compute Field:



The multiplier and ALU operations are determined by OPCODE. The selections for the 6-bit OPCODE field are listed in Table B.7. The multiplier x- and y-operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x- and y-operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operand is returned to data register RA (FA).

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a particular set of four data registers.

Input	*Allowed Sources*
Multiplier X:	R3-R0 (F3-F0)
Multiplier Y:	R7-R4 (F7-F4)
ALU X:	R11-R8 (F11-F8)
ALU Y:	R15-R12 (F15-F12)

Syntax	*Opcode*
Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12	000100
Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 – R15-12	000101
Rm=R3-0 * R7-4 (SSFR), Ra=(R11-8 + R15-12)/2	000110
MRF=MRF + R3-0 * R7-4 (SSF), Ra=R11-8 + R15-12	001000
MRF=MRF + R3-0 * R7-4 (SSF), Ra=R11-8 – R15-12	001001
MRF=MRF + R3-0 * R7-4 (SSF), Ra=(R11-8 + R15-12)/2	001010
Rm=MRF + R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12	001100
Rm=MRF + R3-0 * R7-4 (SSFR), Ra=R11-8 – R15-12	001101
Rm=MRF + R3-0 * R7-4 (SSFR), Ra=(R11-8 + R15-12)/2	001110
MRF=MRF – R3-0 * R7-4 (SSF), Ra=R11-8 + R15-12	010000
MRF=MRF – R3-0 * R7-4 (SSF), Ra=R11-8 – R15-12	010001
MRF=MRF – R3-0 * R7-4 (SSF), Ra=(R11-8 + R15-12)/2	010010
Rm=MRF – R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12	010100
Rm=MRF – R3-0 * R7-4 (SSFR), Ra=R11-8 – R15-12	010101
Rm=MRF – R3-0 * R7-4 (SSFR), Ra=(R11-8 + R15-12)/2	010110
Fm=F3-0 * F7-4, Fa=F11-8 + F15-12	011000
Fm=F3-0 * F7-4, Fa=F11-8 – F15-12	011001
Fm=F3-0 * F7-4, Fa=FLOAT R11-8 by R15-12	011010
Fm=F3-0 * F7-4, Fa=FIX R11-8 by R15-12	011011
Fm=F3-0 * F7-4, Fa=(F11-8 + F15-12)/2	011100
Fm=F3-0 * F7-4, Fa=ABS F11-8	011101
Fm=F3-0 * F7-4, Fa=MAX (F11-8, F15-12)	011110
Fm=F3-0 * F7-4, Fa=MIN (F11-8, F15-12)	011111

Table B.7 Parallel Multiplier/ALU Computations

Parallel Multiplier & Dual Add/Subtract

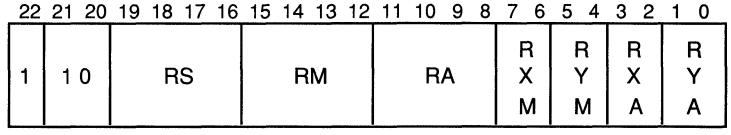
The parallel multiplier and dual add/subtract operation performs a multiply or multiply/accumulate and computes the sum and the difference of the ALU inputs. For detailed information on the multiplier operations, see the individual descriptions under Single-Function Operations. For information on the dual add/subtract operation, see the Dual Add/Subtract section.

Fixed-Point:

Syntax:

Rm=R3-0 * R7-4 (SSFR), Ra=R11-8 + R15-12, Rs=R11-8 – R15-12

Compute Field:

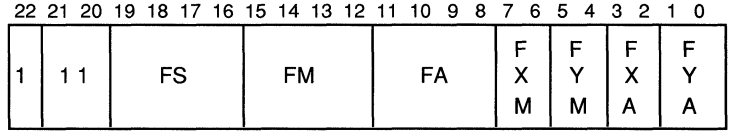


Floating-Point:

Syntax:

Fm=F3-0 * F7-4, Fa=F11-8 + F15-12, Fs=F11-8 – F15-12

Compute Field:



The multiplier x- and y-operands are received from data registers RXM (FXM) and RYM (FYM). The multiplier result operand is returned to data register RM (FM). The ALU x- and y-operands are received from data registers RXA (FXA) and RYA (FYA). The ALU result operands are returned to data register RA (FA) and RS (FS).

Parallel Multiplier & Dual Add/Subtract

The result operands can be returned to any registers within the register file. Each of the four input operands is restricted to a different set of four data registers.

Input	*Allowed Sources*
Multiplier X:	R3-R0 (F3-F0)
Multiplier Y:	R7-R4 (F7-F4)
ALU X:	R11-R8 (F11-F8)
ALU Y:	R15-R12 (F15-F12)

IEEE 1149.1 JTAG Test Access Port

C

C.1 OVERVIEW

A boundary scan allows a system designer to test interconnections on a printed circuit board with minimal test-specific hardware. The scan is made possible by the ability to control and monitor each input and output pin on each chip through a set of serially scannable latches. Each input and output is connected to a latch, and the latches are connected as a long shift register so that data can be read from or written to them through a serial test access port (TAP). The ADSP-21020/21010 contains a test access port compatible with the industry standard IEEE 1149.1 (JTAG) specification.

Only the IEEE 1149.1 features specific to the ADSP-21020/21010 are described here. For more information, see the IEEE 1149.1 specification.

The boundary scan allows a variety of functions to be performed on each input and output signal of the ADSP-21020/21010. Each input has a latch that monitors the value of the incoming signal and can also drive data into the chip in place of the incoming value. Similarly, each output has a latch that monitors the outgoing signal and can also drive the output in place of the outgoing value. For bidirectional pins, the combination of input and output functions is available.

Every latch associated with a pin is part of a single serial shift register path. Each latch is a master/slave type latch with the controlling clock provided externally. This clock (TCK) is asynchronous to the ADSP-21020/21010 system clock (CLKIN).

C IEEE 1149.1 JTAG TAP

C.2 TEST ACCESS PORT

The test access port of the ADSP-21020/21010 controls the operation of the boundary scan. The TAP consists of five pins that control a state machine, including the boundary scan. The state machine and pins conform to the IEEE 1149.1 specification.

TCK (Input)	Test Clock. Used to clock serial data into scan latches and control sequencing of the test state machine. TCK can be asynchronous with CLKIN.
TMS (Input)	Test Mode Select. Primary control signal for the state machine. Synchronous with TCK. A sequence of values on TMS adjusts the current state of the TAP.
TDI (Input)	Test Data Input. Serial input data to the scan latches. Synchronous with TCK.
TDO (Output)	Test Data Output. Serial output data from the scan latches. Synchronous with TCK.
TRST (Input)	Test Reset. Resets the test state machine. Can be asynchronous with TCK.

C.3 INSTRUCTION REGISTER

The instruction register allows an instruction to be shifted into the processor. This instruction selects the test to be performed and/or the test data register to be accessed. The instruction register is 4 bits long with no parity bit. A value of 0001 binary is loaded (LSB nearest TDO) into the instruction register whenever the TAP reset state is entered.

Table C.1 lists the binary code for each instruction. Bit 1 is nearest TDI and bit 4 is nearest TDO. An x specifies a don't-care state. No data registers are placed into test modes by any of the public instructions. The instructions affect the ADSP-21020/21010 as defined in the 1149.1 specification. The optional instructions RUNBIST, IDCODE and USERCODE are not supported by the ADSP-21020/21010.

Instruction Bits 1 2 3 4	Name	Register (Serial Path)	Type
x x x 1	BYPASS	Bypass	Public
0 0 0 0	EXTEST	Boundary	Public
1 0 0 0	SAMPLE/PRELOAD	Boundary	Public
1 1 0 0	INTEST	Boundary	Public
0 1 0 0	Reserved for emulation		Private
x x 1 0	Reserved for emulation		Private

Table C.1 Test Instructions

The entry under "Register" is the serial scan path, either Boundary or Bypass in this case, enabled by the instruction. Figure C.1 shows these register paths. The 1-bit Bypass register is fully defined in the 1149.1 specification. The Boundary register is described in the next section.

No special values need be written into any register prior to selection of any instruction. As Table C.1 shows, certain instructions are reserved for emulator use. See section C.7 for more information.

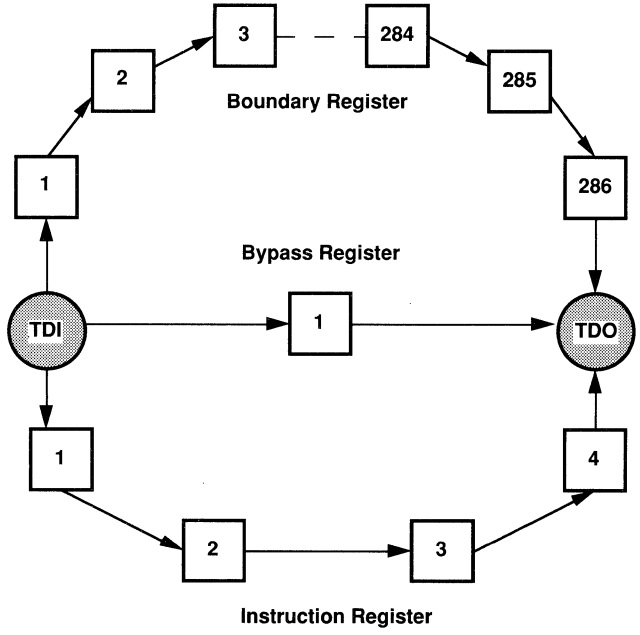


Figure C.1 Serial Scan Paths

C

C.4 BOUNDARY REGISTER

The Boundary register is 286 bits long. This section defines the latch type and function of each position in the scan path. The positions are numbered with 286 being the first bit output (closest to TDO) and 1 being the last (closest to TDI).

Scan Position*	Latch Type	Signal Name
1	Input	$\overline{\text{DMTS}}$
2	Output	$\overline{\text{DMWR}}$
3	Input	DMACK
4	Output	$\overline{\text{DMRD}}$
5	Clock**	CLKIN
6	Output Control††	$\overline{\text{DMRD}}$/$\overline{\text{DMWR}}$ Output Enable
7	Input	$\overline{\text{RESET}}$
8	Output Control††	$\overline{\text{PMRD}}$/$\overline{\text{PMWR}}$ Output Enable
9	Output	$\overline{\text{PMRD}}$
10	Input	PMACK
11	Output	$\overline{\text{PMWR}}$
12	Input	$\overline{\text{PMTS}}$
13	Output Control††	PMD Output Enable
14	Input	PMD47 Input Latch
15	Output	PMD47 Output Latch
16	Input	PMD46 Input Latch
17	Output	PMD46 Output Latch
18	Input	PMD45 Input Latch
19	Output	PMD45 Output Latch
20	Input	PMD44 Input Latch
21	Output	PMD44 Output Latch
22	Input	PMD43 Input Latch
23	Output	PMD43 Output Latch
24	Input	PMD42 Input Latch
25	Output	PMD42 Output Latch
26	Input	PMD41 Input Latch
27	Output	PMD41 Output Latch
28	Input	PMD40 Input Latch
29	Output	PMD40 Output Latch
30	Input	PMD39 Input Latch
31	Output	PMD39 Output Latch
32	Input	PMD38 Input Latch
33	Output	PMD38 Output Latch
34	Input	PMD37 Input Latch
35	Output	PMD37 Output Latch

Scan Position*	Latch Type	Signal Name
36	Input	PMD36 Input Latch
37	Output	PMD36 Output Latch
38	Input	PMD35 Input Latch
39	Output	PMD35 Output Latch
40	Input	PMD34 Input Latch
41	Output	PMD34 Output Latch
42	Input	PMD33 Input Latch
43	Output	PMD33 Output Latch
44	Input	PMD32 Input Latch
45	Output	PMD32 Output Latch
46	Input	PMD31 Input Latch
47	Output	PMD31 Output Latch
48	Input	PMD30 Input Latch
49	Output	PMD30 Output Latch
50	Input	PMD29 Input Latch
51	Output	PMD29 Output Latch
52	Input	PMD28 Input Latch
53	Output	PMD28 Output Latch
54	Input	PMD27 Input Latch
55	Output	PMD27 Output Latch
56	Input	PMD26 Input Latch
57	Output	PMD26 Output Latch
58	Input	PMD25 Input Latch
59	Output	PMD25 Output Latch
60	Input	PMD24 Input Latch
61	Output	PMD24 Output Latch
62	Input	PMD23 Input Latch
63	Output	PMD23 Output Latch
64	Input	PMD22 Input Latch
65	Output	PMD22 Output Latch
66	Input	PMD21 Input Latch
67	Output	PMD21 Output Latch
68	Input	PMD20 Input Latch
69	Output	PMD20 Output Latch
70	Input	PMD19 Input Latch
71	Output	PMD19 Output Latch
72	Input	PMD18 Input Latch
73	Output	PMD18 Output Latch
74	Input	PMD17 Input Latch
75	Output	PMD17 Output Latch
76	Input	PMD16 Input Latch

Scan Position*	Latch Type	Signal Name
77	Output	PMD16 Output Latch
78	Input	PMD15 Input Latch
79	Output	PMD15 Output Latch
80	Input	PMD14 Input Latch
81	Output	PMD14 Output Latch
82	Input	PMD13 Input Latch
83	Output	PMD13 Output Latch
84	Input	PMD12 Input Latch
85	Output	PMD12 Output Latch
86	Input	PMD11 Input Latch
87	Output	PMD11 Output Latch
88	Input	PMD10 Input Latch
89	Output	PMD10 Output Latch
90	Input	PMD9 Input Latch
91	Output	PMD9 Output Latch
92	Input	PMD8 Input Latch
93	Output	PMD8 Output Latch
94	Input	PMD7 Input Latch
95	Output	PMD7 Output Latch
96	Input	PMD6 Input Latch
97	Output	PMD6 Output Latch
98	Input	PMD5 Input Latch
99	Output	PMD5 Output Latch
100	Input	PMD4 Input Latch
101	Output	PMD4 Output Latch
102	Input	PMD3 Input Latch
103	Output	PMD3 Output Latch
104	Input	PMD2 Input Latch
105	Output	PMD2 Output Latch
106	Input	PMD1 Input Latch
107	Output	PMD1 Output Latch
108	Input	PMD0 Input Latch
109	Output	PMD0 Output Latch
110	Output Control††	DMD Output Enable
111	Input	DMD0 Input Latch
112	Output	DMD0 Output Latch
113	Input	DMD1 Input Latch
114	Output	DMD1 Output Latch
115	Input	DMD2 Input Latch
116	Output	DMD2 Output Latch
117	Input	DMD3 Input Latch

Scan Position*	Latch Type	Signal Name
118	Output	DMD3 Output Latch
119	Input	DMD4 Input Latch
120	Output	DMD4 Output Latch
121	Input	DMD5 Input Latch
122	Output	DMD5 Output Latch
123	Input	DMD6 Input Latch
124	Output	DMD6 Output Latch
125	Input	DMD7 Input Latch
126	Output	DMD7 Output Latch
127	Input	DMD8 Input Latch
128	Output	DMD8 Output Latch
129	Input	DMD9 Input Latch
130	Output	DMD9 Output Latch
131	Input	DMD10 Input Latch
132	Output	DMD10 Output Latch
133	Input	DMD11 Input Latch
134	Output	DMD11 Output Latch
135	Input	DMD12 Input Latch
136	Output	DMD12 Output Latch
137	Input	DMD13 Input Latch
138	Output	DMD13 Output Latch
139	Input	DMD14 Input Latch
140	Output	DMD14 Output Latch
141	Input	DMD15 Input Latch
142	Output	DMD15 Output Latch
143	Input	DMD16 Input Latch
144	Output	DMD16 Output Latch
145	Input	DMD17 Input Latch
146	Output	DMD17 Output Latch
147	Input	DMD18 Input Latch
148	Output	DMD18 Output Latch
149	Input	DMD19 Input Latch
150	Output	DMD19 Output Latch
151	Input	DMD20 Input Latch
152	Output	DMD20 Output Latch
153	Input	DMD21 Input Latch
154	Output	DMD21 Output Latch
155	Input	DMD22 Input Latch
156	Output	DMD22 Output Latch
157	Input	DMD23 Input Latch
158	Output	DMD23 Output Latch

Scan Position*	Latch Type	Signal Name
159	Input	DMD24 Input Latch
160	Output	DMD24 Output Latch
161	Input	DMD25 Input Latch
162	Output	DMD25 Output Latch
163	Input	DMD26 Input Latch
164	Output	DMD26 Output Latch
165	Input	DMD27 Input Latch
166	Output	DMD27 Output Latch
167	Input	DMD28 Input Latch
168	Output	DMD28 Output Latch
169	Input	DMD29 Input Latch
170	Output	DMD29 Output Latch
171	Input	DMD30 Input Latch
172	Output	DMD30 Output Latch
173	Input	DMD31 Input Latch
174	Output	DMD31 Output Latch
175	Input	DMD32 Input Latch
176	Output	DMD32 Output Latch
177	Input	DMD33 Input Latch
178	Output	DMD33 Output Latch
179	Input	DMD34 Input Latch
180	Output	DMD34 Output Latch
181	Input	DMD35 Input Latch
182	Output	DMD35 Output Latch
183	Input	DMD36 Input Latch
184	Output	DMD36 Output Latch
185	Input	DMD37 Input Latch
186	Output	DMD37 Output Latch
187	Input	DMD38 Input Latch
188	Output	DMD38 Output Latch
189	Input	DMD39 Input Latch
190	Output	DMD39 Output Latch
191	Output	$\overline{\text{DOMS}}$†
192	Output	$\overline{\text{DMS3}}$ Output Latch
193	Output	$\overline{\text{DMS2}}$ Output Latch
194	Output	$\overline{\text{DMS1}}$ Output Latch
195	Output	$\overline{\text{DMS0}}$ Output Latch
196	Output	$\overline{\text{BG}}$
197	Input	$\overline{\text{BR}}$
198	Output	DMPAGE
199	Output Control††	DMA Output Enable

Scan Position*	Latch Type	Signal Name
200	Output	DMA31
201	Output	DMA30
202	Output	DMA29
203	Output	DMA28
204	Output	DMA27
205	Output	DMA26
206	Output	DMA25
207	Output	DMA24
208	Output	DMA23
209	Output	DMA22
210	Output	DMA21
211	Output	DMA20
212	Output	DMA19
213	Output	DMA18
214	Output	DMA17
215	Output	DMA16
216	Output	DMA15
217	Output	DMA14
218	Output	DMA13
219	Output	DMA12
220	Output	DMA11
221	Output	DMA10
222	Output	DMA9
223	Output	DMA8
224	Output	DMA7
225	Output	DMA6
226	Output	DMA5
227	Output	DMA4
228	Output Control††	FLAG3 Output Enable
229	Output	DMA3
230	Output	DMA2
231	Output	DMA1
232	Output	DMA0
233	Output Control††	FLAG2 Output Enable
234	Input	FLAG3 Input Latch
235	Output	FLAG3 Output Latch
236	Input	FLAG2 Input Latch
237	Output	FLAG2 Output Latch
238	Input	FLAG1 Input Latch
239	Output	FLAG1 Output Latch
240	Input	FLAG0 Input Latch

Scan Position*	Latch Type	Signal Name
241	Output	FLAG0 Output Latch
242	Output Control††	FLAG1 Output Enable
243	Input	$\overline{\text{IRQ0}}$
244	Input	$\overline{\text{IRQ1}}$
245	Input	$\overline{\text{IRQ2}}$
246	Input	$\overline{\text{IRQ3}}$
247	Output	IAS†
248	Output Control††	FLAG0 Output Enable
249	Output	CAV†
250	Output	CA4†
251	Output	CA3†
252	Output	CA2†
253	Output	CA1†
254	Output	CA0†
255	Output	EXAB†
256	Output	TIMEXP
257	Output	PMA0
258	Output	PMA1
259	Output	PMA2
260	Output	PMA3
261	Output	PMA4
262	Output	PMA5
263	Output	PMA6
264	Output	PMA7
265	Output	PMA8
266	Output	PMA9
267	Output	PMA10
268	Output	PMA11
269	Output	PMA12
270	Output	PMA13
271	Output	PMA14
272	Output	PMA15
273	Output	PMA16
274	Output	PMA17
275	Output	PMA18
276	Output	PMA19
277	Output	PMA20
278	Output	PMA21
279	Output	PMA22
280	Output	PMA23
281	Output Control††	PMA Output Enable
282	Output	$\overline{\text{PMS1}}$
283	Output	$\overline{\text{PMS0}}$

Scan Position*	Latch Type	Signal Name
284	Output	PMPAGE
285	Output	$\overline{\text{POMS0}}$†
286	Output	$\overline{\text{POMS1}}$†

* Position 1 = closest to TDI (scan in last); position 286 = closest to TDO (scan in first)
** CLKIN can be sampled but not controlled (read-only). CLKIN continues to clock the ADSP-21020/21010 no matter which instruction is enabled.
† Signals reserved for emulator use. Can be set to any state during scan.
†† 1 = Drive the associated signals during the EXTEST and INTEST instructions
0 = Tristate the associated signals during the EXTEST and INTEST instructions

C.5 DEVICE IDENTIFICATION REGISTER

No device identification register is included in the ADSP-21020/21010.

C.6 BUILT-IN SELF-TEST OPERATION (BIST)

No self-test functions are supported by the ADSP-21020/21010.

C.7 PRIVATE INSTRUCTIONS

Loading a value of 01xx into the instruction register enables the private instructions reserved for emulation. The ADSP-21020/21010 EZ-ICE emulator uses the TAP and boundary scan as a way to access the processor in the target system. Use of the EZ-ICE emulator requires a target board connector for access to the TAP. See "EZ-ICE Emulator Considerations" in Chapter 9 for information on this connector.

C.8 REFERENCES

IEEE Standard 1149.1-1990. *Standard Test Access Port and Boundary-Scan Architecture.* To get a copy, contact IEEE at: 1-800-678-IEEE.

Evanczuk, Stephen. "IEEE 1149.1—A Designer's Reference." *High Performance Systems,* Aug. 1989, pp. 52-60.

Maunder, C.M. and R. Tulloss. 1991. *Test Access Ports and Boundary Scan Architectures.* IEEE Computer Society Press.

Quinnell, Richard A. "Adding Testability Also Aids Debugging." *EDN,* Aug. 2, 1990, pp.67-74.

Numeric Formats D

D.1 OVERVIEW

The ADSP-21020 and ADSP-21010 support the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the ADSP-21020 supports an extended-precision version of the same format with eight additional bits in the mantissa (40 bits total). Both the ADSP-21020 and ADSP-21010 also support 32-bit fixed-point formats—fractional and integer—which can be signed (twos-complement) or unsigned.

D.2 IEEE SINGLE-PRECISION FLOATING-POINT DATA FORMAT

IEEE Standard 754/854 specifies a 32-bit single-precision floating-point format, shown in Figure D.1. A number in this format consists of a sign bit *s*, a 24-bit significand, and an 8-bit unsigned-magnitude exponent *e*. For normalized numbers, the significand consists of a 23-bit fraction *f* and a "hidden" bit of 1 that is implicitly presumed to precede f_{22} in the significand. The binary point is presumed to lie between this hidden bit and f_{22}. The least significant bit (LSB) of the fraction is f_0; the LSB of the exponent is e_0. The hidden bit effectively increases the precision of the floating-point significand to 24 bits from the 23 bits actually stored in the data format. It also insures that the significand of any number in the IEEE normalized-number format is always greater than or equal to 1 and less than 2.

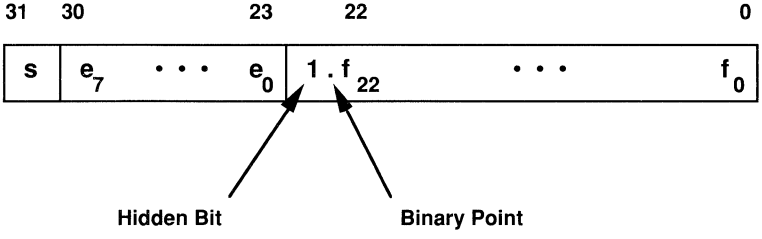


Figure D.1 IEEE 32-Bit Single-Precision Floating-Point Format

The unsigned exponent *e* can range between $1 \leq e \leq 254$ for normal numbers in the single-precision format. This exponent is *biased* by +127 (254 ÷ 2). To calculate the true *unbiased* exponent, 127 must be subtracted from *e*.

D Numeric Formats

The IEEE Standard also provides for several special data types in the single-precision floating-point format:

- An exponent value of 255 (all ones) with a nonzero fraction is a Not-A-Number (NAN). NANs are usually used as flags for data flow control, for the values of uninitialized variables, and for the results of invalid operations such as $0 \bullet \infty$.
- Infinity is represented as an exponent of 255 and a zero fraction. Note that because the fraction is signed, both positive and negative Infinity can be represented.
- Zero is represented by a zero exponent and a zero fraction. As with Infinity, both positive Zero and negative Zero can be represented.

The IEEE single-precision floating-point data types supported by the ADSP-21020 and ADSP-21010 and their interpretations are summarized in Table D.1.

Type	*Exponent*	*Fraction*	*Value*
NAN	255	Nonzero	Undefined
Infinity	255	0	$(-1)^s$ Infinity
Normal	$1 \le e \le 254$	Any	$(-1)^s (1.f_{22-0}) 2^{e-127}$
Zero	0	0	$(-1)^s$ Zero

Table D.1 IEEE Single-Precision Floating-Point Data Types

D.3 EXTENDED FLOATING-POINT FORMAT

The extended precision floating-point format is 40 bits wide, with the same 8-bit exponent as in the standard format but a 32-bit significand. This format is shown in Figure D.2. In all other respects, the extended floating-point format is the same as the IEEE standard format.

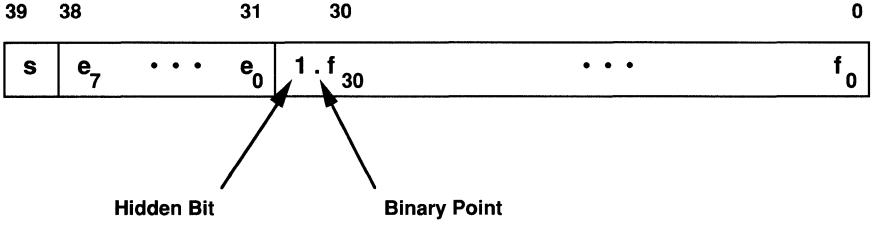


Figure D.2 40-Bit Extended-Precision Floating-Point Format

D.4 FIXED-POINT FORMATS

The ADSP-21020 andADSP-21010 support two 32-bit fixed-point formats: fractional and integer. In both formats, numbers can be signed (twos-complement) or unsigned. The four possible combinations are shown in Figure D.3. In the fractional format, there is an implied binary point to the right of the most significant magnitude bit. In integer format, the binary point is understood to be to the left of the LSB. Note that the sign bit is negatively weighted in a twos-complement format.

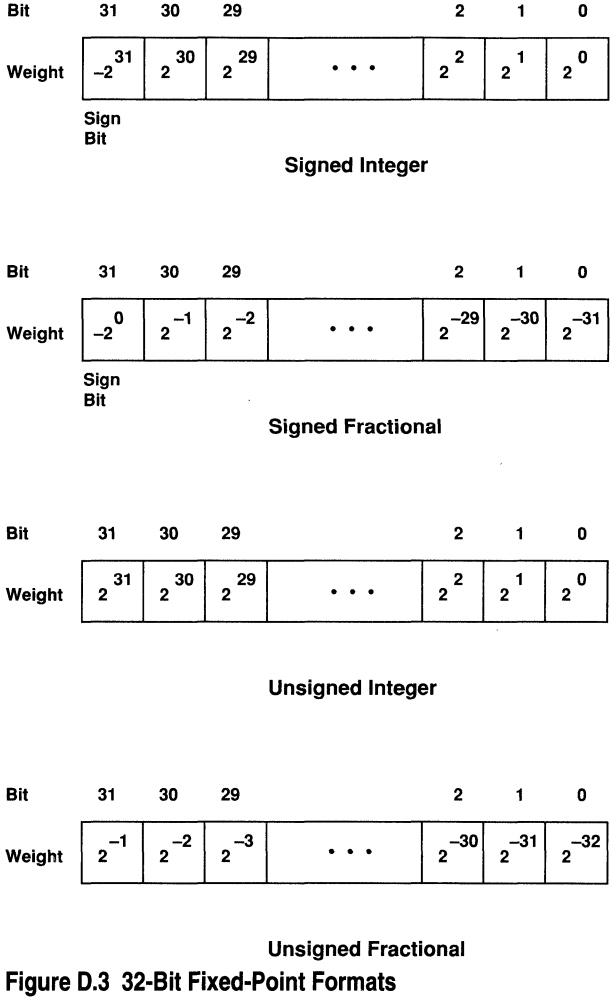


Figure D.3 32-Bit Fixed-Point Formats

D Numeric Formats

ALU outputs always have the same width and data format as the inputs. The multiplier, however, produces a 64-bit product from two 32-bit inputs. If both operands are unsigned integers, the result is a 64-bit unsigned integer. If both operands are unsigned fractions, the result is a 64-bit unsigned fraction. These formats are shown in Figure D.4.

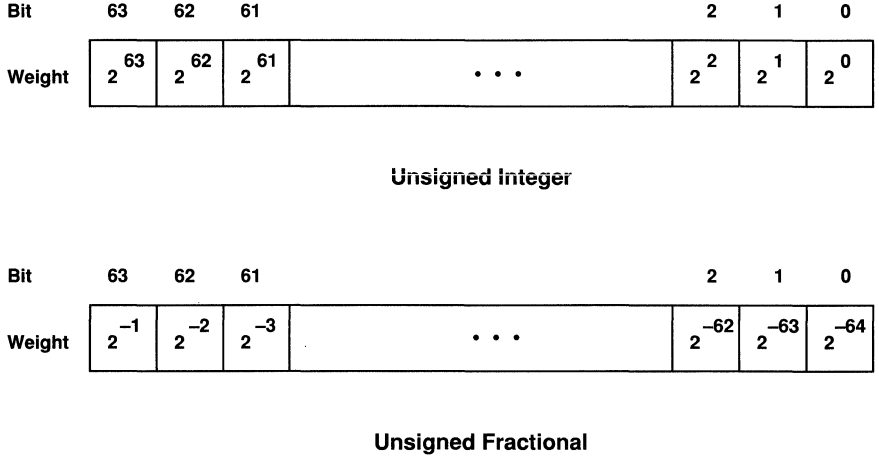
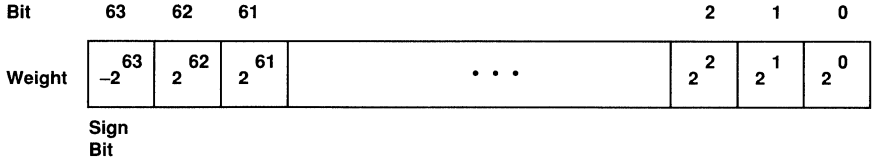


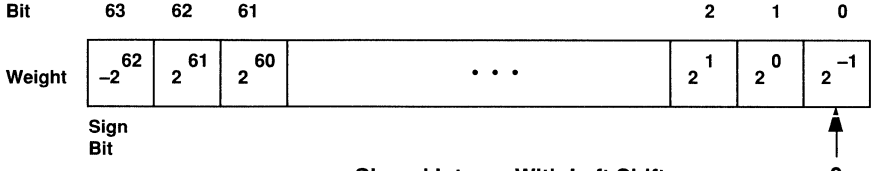
Figure D.4 64-Bit Unsigned Fixed-Point Product

If one operand is signed and the other unsigned, the result is signed. If both inputs are signed, the result is signed and automatically shifted left one bit. The LSB becomes zero and bit 62 moves into the sign bit position. Normally bit 63 and bit 62 are identical when both operands are signed. (The only exception is full-scale negative multiplied by itself.) Thus, the left shift normally removes a redundant sign bit, increasing the precision of the most significant product. Also, if the data format is fractional, a single-bit left shift renormalizes the MSP to a fractional format. The signed formats with and without left shifting are shown in Figure D.5.

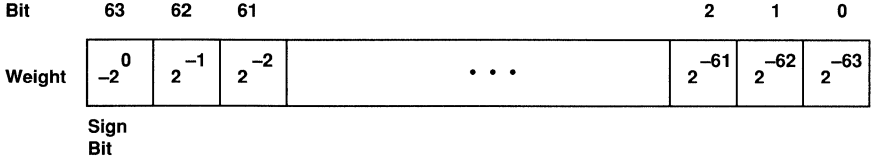
The multiplier has an 80-bit accumulator to allow the accumulation of 64-bit products. The multiplier and accumulator are described in detail in Chapter 2.



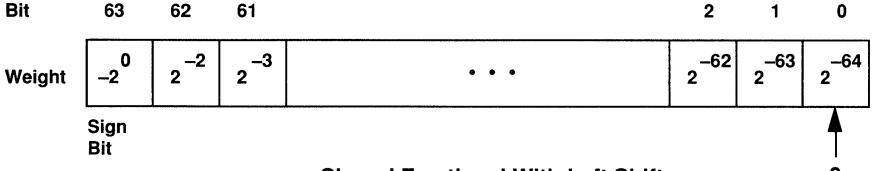
Signed Integer, No Left Shift



Signed Integer With Left Shift



Signed Fractional, No Left Shift



Signed Fractional With Left Shift

Figure D.5 64-Bit Signed Fixed-Point Product

Control/Status Registers E

E.1 OVERVIEW

This appendix describes the subset of universal registers known as **system registers** and the operations that can be performed on them. It also summarizes the bit definitions of the system registers that contain control or status information. For convenience, two other control registers, PMWAIT and DMWAIT, are also summarized here, but these registers are not system registers.

The bit names that appear with each definition are used by convention only; they are not part of the instruction set.

E.2 SYSTEM REGISTERS

System registers are the universal registers listed in Table E.1. The system registers are a subset of the universal register set. They can be written from an immediate field in an instruction or they can be loaded from or stored to data memory. They can also be transferred to or from any other universal register in one cycle.

Register	*Function*	*Value After Reset*
MODE1	mode control 1 (see E.3)	0x0000 (cleared)
MODE2	mode control 2 (see E.4)	0x*n*000 0000 (bits 28-31 are the device identification field, identifying the silicon revision #)
IRPTL	interrupt latch (see E.7)	0x0000 (cleared)
IMASK	interrupt mask (see E.7)	0x0003
IMASKP	interrupt mask pointer	0x0000 (cleared)
ASTAT	arithmetic status (see E.5)	0x00*nn* 0000 (bits 19-22 are equal to the values of the FLAG0-3 input pins; the flag pins are configured as inputs after reset)
STKY	sticky status (see E.6)	0x0540 0000
USTAT1	user status 1	0x0000 (cleared)
USTAT2	user status 2	0x0000 (cleared)

Table E.1 System Registers

E Control/Status Registers

A write to any system register except USTAT1 or USTAT2 has one cycle of latency before any changes are effective. No wait states are inserted. If a write to a system register is immediately followed by a read, the value read is the new one, except for IMASKP which requires a extra cycle before the value is updated.

E.2.1 System Register Bit Operations

The system registers differ from other universal registers in that individual groups of bits can be set, cleared, XORed, toggled or tested using an immediate field in the bit manipulation instruction to specify the affected bits. See the instruction description in Appendix A for specifics.

Although the shifter and ALU have bit manipulation capabilities, these computations operate on register file locations only. System register bit manipulation instructions eliminate the overhead of transferring system registers to and from the register file.

Bit Instruction (System Registers)	*Shifter Operation (Register File Locations)*
BIT SET register data	Rn = BSET Rx BY Ry | data
BIT CLR register data	Rn = BCLR Rx BY Ry | data
BIT TGL register data	Rn = BTGL Rx BY Ry | data
BIT TST register data (result in BTF flag)	BTST Rx BY Ry | data (result in SZ status flag)

E.2.1.1 Bit Test Flag

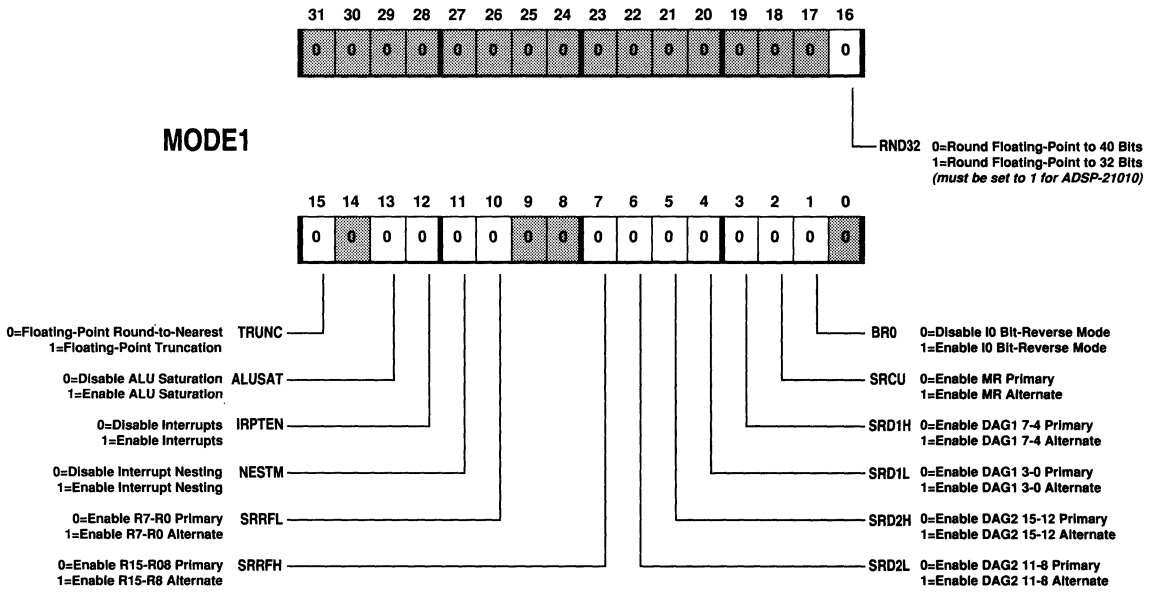
The test and XOR operations of the bit manipulation instruction store the result in the bit test flag (BTF, bit 18 in the ASTAT register). The state of BTF is a condition that you can specify in conditional instructions. The test operation sets BTF if all specified bits in the system register are set. The XOR operation sets BTF if all bits in the system register match the specified bit pattern.

E.2.2 User Registers

Two undedicated 32-bit status registers, USTAT1 and USTAT2, are user-defined. Bits in these registers can be set and tested using system register instructions. You can use these registers for low-overhead, general-purpose software flags or for temporary storage of data.

E.3 MODE1 REGISTER

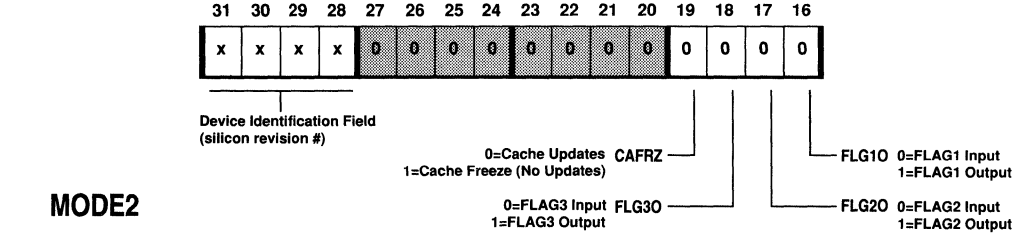
Bit	*Name*	*Definition*
0		**Reserved**
1	BR0	Bit-reverse for I0 (uses DMS0 only)
2	SRCU	Alternate register select for computation units
3	SRD1H	DAG1 alternate register select (7-4)
4	SRD1L	DAG1 alternate register select (3-0)
5	SRD2H	DAG2 alternate register select (15-12)
6	SRD2L	DAG2 alternate register select (11-8)
7	SRRFH	Register file alternate select for R(15-8)
8-9		**Reserved**
10	SRRFL	Register file alternate select for R(7-0)
11	NESTM	Interrupt nesting enable
12	IRPTEN	Global interrupt enable
13	ALUSAT	Enable ALU saturation (full scale in fixed-point)
14		**Reserved**
15	TRUNC	1=Floating-point truncation; 0=Round to nearest
16	RND32	1=Round floating-point data to 32 bits; 0=Round to 40 bits (*must be set to 1 for ADSP-21010*)
17-31		**Reserved**

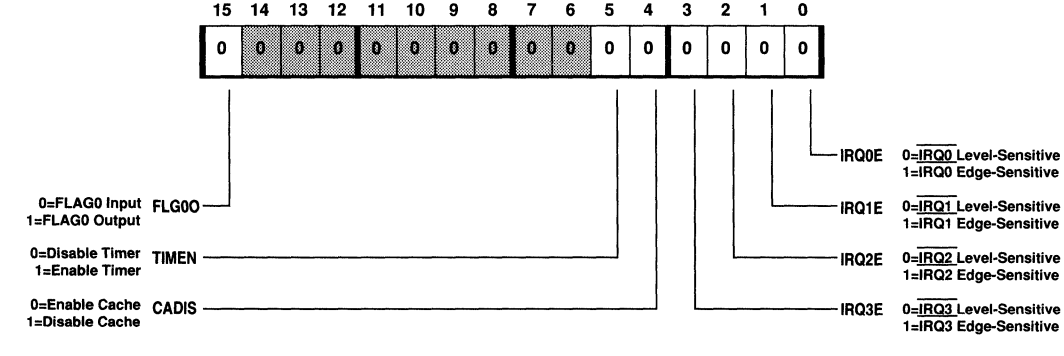


E.4 MODE2 REGISTER

Bit	*Name*	*Definition*
0	IRQ0E	$\overline{IRQ0}$ 1=edge sensitive; 0=level-sensitive
1	IRQ1E	$\overline{IRQ1}$ 1=edge sensitive; 0=level-sensitive
2	IRQ2E	$\overline{IRQ2}$ 1=edge sensitive; 0=level-sensitive
3	IRQ3E	$\overline{IRQ3}$ 1=edge sensitive; 0=level-sensitive
4	CADIS	Cache disable
5	TIMEN	Timer enable
6-14		**Reserved**
15	FLG0O	FLAG0 1=output; 0=input
16	FLG1O	FLAG1 1=output; 0=input
17	FLG2O	FLAG2 1=output; 0=input
18	FLG3O	FLAG3 1=output; 0=input
19	CAFRZ	Cache freeze
20-27		**Reserved**
28-31		Device Identification Field (silicon revision #)

MODE2

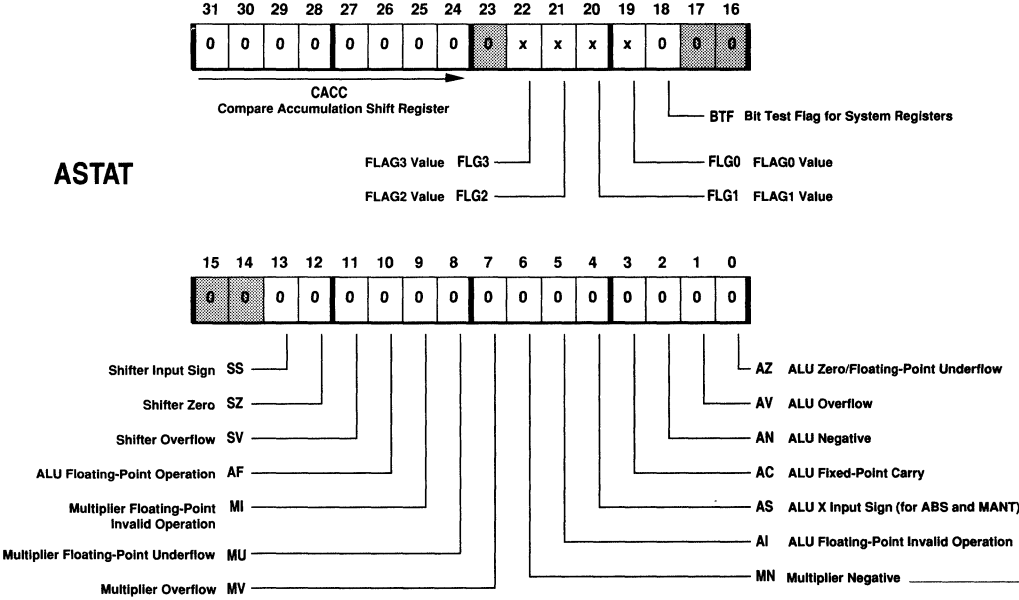




E.5 ARITHMETIC STATUS REGISTER (ASTAT)

Bit	*Name*	*Definition*
0	AZ	ALU result zero or floating-point underflow
1	AV	ALU overflow
2	AN	ALU result negative
3	AC	ALU fixed-point carry
4	AS	ALU X input sign (ABS and MANT operations)
5	AI	ALU floating-point invalid operation
6	MN	Multiplier result negative
7	MV	Multiplier overflow
8	MU	Multiplier floating-point underflow
9	MI	Multiplier floating-point invalid operation
10	AF	ALU floating-point operation
11	SV	Shifter overflow
12	SZ	Shifter result zero
13	SS	Shifter input sign
14-17		**Reserved**
18	BTF	Bit test flag for system registers
19	FLG0	FLAG0 value
20	FLG1	FLAG1 value
21	FLG2	FLAG2 value
22	FLG3	FLAG3 value
23		**Reserved**
24-31		CACC (Compare Accumulation) bits

ASTAT



E Control/Status Registers

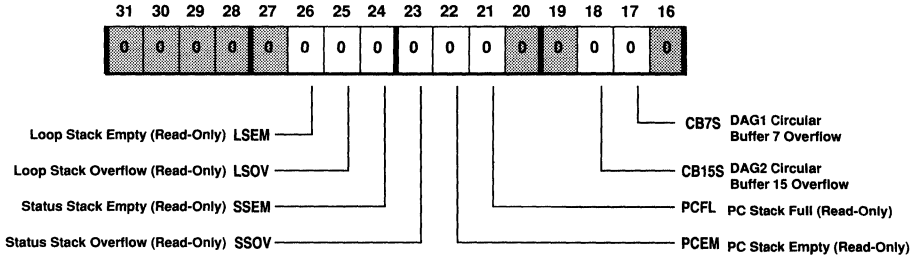
E.6 STICKY ARITHMETIC STATUS REGISTER (STKY)

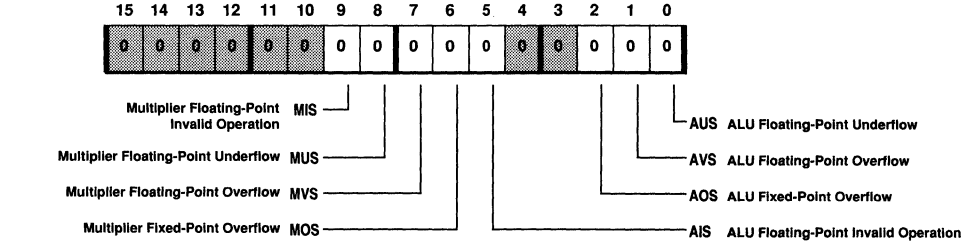
Bit	*Name*	*Definition*
0	AUS	ALU floating-point underflow
1	AVS	ALU floating-point overflow
2	AOS	ALU fixed-point overflow
3-4		**Reserved**
5	AIS	ALU floating-point invalid operation
6	MOS	Multiplier fixed-point overflow
7	MVS	Multiplier floating-point overflow
8	MUS	Multiplier floating-point underflow
9	MIS	Multiplier floating-point invalid operation
10-16		**Reserved**
17	CB7S	DAG1 circular buffer 7 overflow*
18	CB15S	DAG2 circular buffer 15 overflow*
19-20		**Reserved**
21	PCFL	PC stack full (not sticky)
22	PCEM	PC stack empty (not sticky)
23	SSOV	Status stack overflow (MODE1 and ASTAT)
24	SSEM	Status stack empty (not sticky)
25	LSOV	Loop stack overflow (Loop Address and Loop Counter)
26	LSEM	Loop stack empty (not sticky)
27-31		**Reserved**

(*Bits 21-26 are read-only. Writes to the STKY register have no effect on these bits.*)

* Bit 17 (DAG1 circular buffer 7 overflow) and Bit 18 (DAG2 circular buffer 15 overflow) indicate the occurrence of a circular buffer overflow. Rather then remaining set until explicitly cleared, however, these bits are cleared by the next subsequent memory access that uses the corresponding I register (I7, I15). Circular buffer interrupts, therefore, should be used instead of these STKY register bits. See Section 4.3.2.3, "Circular Buffer Overflow Interrupts," in Chapter 4.

STKY


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Loop Stack Empty (Read-Only) LSEM
Loop Stack Overflow (Read-Only) LSOV
Status Stack Empty (Read-Only) SSEM
Status Stack Overflow (Read-Only) SSOV
CB7S DAG1 Circular Buffer 7 Overflow
CB15S DAG2 Circular Buffer 15 Overflow
PCFL PC Stack Full (Read-Only)
PCEM PC Stack Empty (Read-Only)


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
Multiplier Floating-Point Invalid Operation MIS
Multiplier Floating-Point Underflow MUS
Multiplier Floating-Point Overflow MVS
Multiplier Fixed-Point Overflow MOS
AUS ALU Floating-Point Underflow
AVS ALU Floating-Point Overflow
AOS ALU Fixed-Point Overflow
AIS ALU Floating-Point Invalid Operation

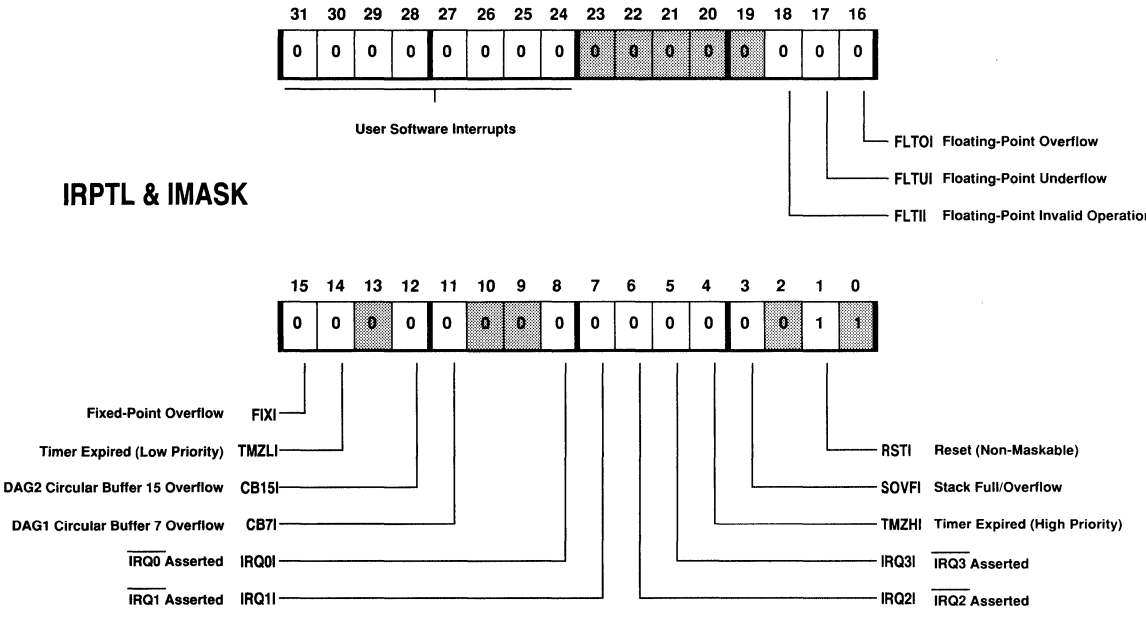
E Control/Status Registers

E.7 INTERRUPT LATCH (IRPTL) & INTERRUPT MASK (IMASK)

Bit (Int#)	*Address*	*Name*	*Function*
0	0x00		**Reserved for emulation**
1	0x08	RSTI	RESET
2	0x10		**Reserved**
3	0x18	SOVFI	Status stack or loop stack overflow or PC stack full
4	0x20	TMZHI	Timer =0 (high priority option)
5	0x28	IRQ3I	$\overline{IRQ_3}$ asserted
6	0x30	IRQ2I	$\overline{IRQ_2}$ asserted
7	0x38	IRQ1I	$\overline{IRQ_1}$ asserted
8	0x40	IRQ0I	$\overline{IRQ_0}$ asserted
9	0x48		**Reserved**
10	0x50		**Reserved**
11	0x58	CB7I	Circular buffer 7 overflow interrupt
12	0x60	CB15I	Circular buffer 15 overflow interrupt
13	0x68		**Reserved**
14	0x70	TMZLI	Timer=0 (low priority option)
15	0x78	FIXI	Fixed-point overflow
16	0x80	FLTOI	Floating-point overflow exception
17	0x88	FLTUI	Floating-point underflow exception
18	0x90	FLTII	Floating-point invalid exception
19-23	0x98-0xB8		**Reserved**
24	0xC0	SFT0I	User software interrupt 0
25	0xC8	SFT1I	User software interrupt 1
26	0xD0	SFT2I	User software interrupt 2
27	0xD8	SFT3I	User software interrupt 3
28	0xE0	SFT4I	User software interrupt 4
29	0xE8	SFT5I	User software interrupt 5
30	0xF0	SFT6I	User software interrupt 6
31	0xF8	SFT7I	User software interrupt 7

For IMASK: 1=unmasked (enabled), 0=masked (disabled)
(interrupts 0 and 1 are not maskable)

IRPTL & IMASK



Default values for IMASK only; IRPTL is cleared after reset.

E.8 PROGRAM MEMORY INTERFACE CONTROL (PMWAIT)

Bit *Definition*

13 1=Automatic wait state for access across page boundary
0=No automatic wait state

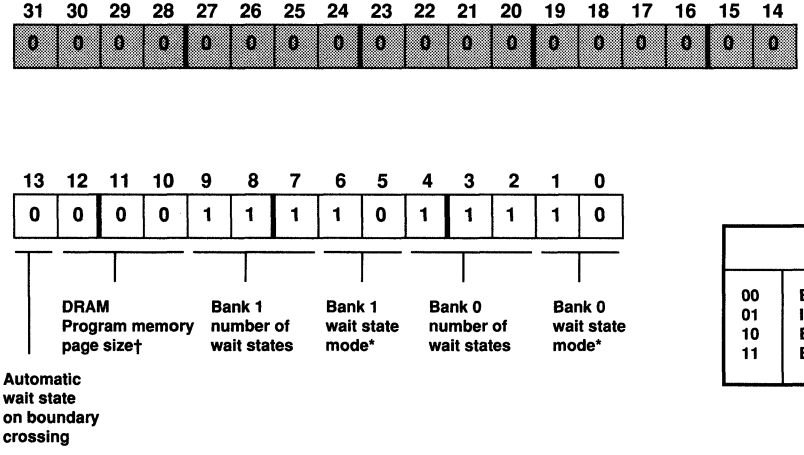
12-10 Memory page size:

000	256 words
001	512 words
010	1024 words
011	2048 words
100	4096 words
101	8192 words
110	16384 words
111	32768 words

9-7 Number of program memory bank 1 wait states (0-7)
6-5 Wait state mode* for program memory bank 1
4-2 Number of program memory bank 0 wait states (0-7)
1-0 Wait state mode* for program memory bank 0

* Wait state mode bits:

0 0	External acknowledge only
0 1	Internal software wait states only
1 0	Both Internal and External acknowledge
1 1	Either Internal or External acknowledge



† DRAM Memory Page Size Codes	
000	256 Words
001	512 Words
010	1024 Words
011	2048 Words
100	4096 Words
101	8192 Words
110	16384 Words
111	32768 Words

* Wait State Mode Codes	
00	External acknowledge only
01	Internal wait states only
10	Both external and internal required
11	Either external or internal sufficient

E.9 DATA MEMORY INTERFACE CONTROL (DMWAIT)

Bit *Definition*

23 1=Automatic wait state for access across page boundary
0=No automatic wait state

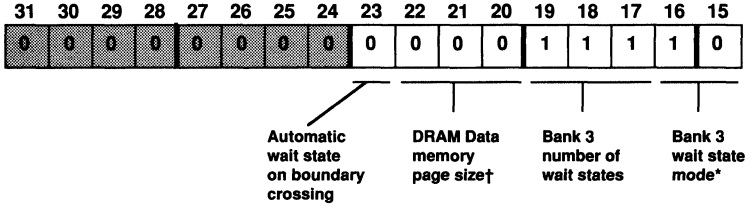
22-20 Memory page size:

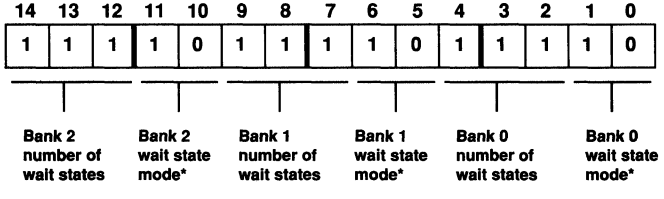
000	256 words
001	512 words
010	1024 words
011	2048 words
100	4096 words
101	8192 words
110	16384 words
111	32768 words

19-17 Number of program memory bank 3 wait states (0-7)
16-15 Wait state mode* for program memory bank 3
14-12 Number of program memory bank 2 wait states (0-7)
11-10 Wait state mode* for program memory bank 2
9-7 Number of program memory bank 1 wait states (0-7)
6-5 Wait state mode* for program memory bank 1
4-2 Number of program memory bank 0 wait states (0-7)
1-0 Wait state mode* for program memory bank 0

* Wait state mode bits:

0 0	External acknowledge only
0 1	Internal software wait states only
1 0	Both Internal and External acknowledge
1 1	Either Internal or External acknowledge





† DRAM Memory Page Size Codes	
000	256 Words
001	512 Words
010	1024 Words
011	2048 Words
100	4096 Words
101	8192 Words
110	16384 Words
111	32768 Words

* Wait State Mode Codes	
00	External acknowledge only
01	Internal wait states only
10	Both external and internal required
11	Either external or internal sufficient

ADSP-21020/21010 User's Manual, 2nd Edition (1993)
Revisions from 1st Edition (1991)

Key

Text deleted from 1st Edition is crossed-out: "~~only by EXP operations~~"
New text added in 2nd Edition is underlined: "SS is cleared"

page (2nd Ed.)	*revision*
p. 1-7, para. 5	"Fixed-point and single-precision floating-point data is aligned to the upper 32 bits of the PMD bus."
p. 1-7, para. 6	"Fixed-point and single-precision floating-point data is aligned to the upper 32 bits of the DMD bus."
p. 2-1, last para.	"The individual registers of the register file are prefixed with an "f" when used in floating-point computations (in assembly language source code). The registers are prefixed with an "r" when used in fixed-point computations. The following instructions, for example, use the same three registers: `F0=F1*F2;` *floating-point multiply* `R0=R1*R2;` *fixed-point multiply* The "f" and "r" prefixes do not affect the 40-bit data transfer; they only determine how the ALU, multiplier, or shifter treat the data."
p. 2-10	ALU Instruction Summary table now shows effects of ALU instructions on all status bits of ASTAT and STKY status registers.
p. 2-17, para. 6	Twos-complement–Fractional: "upper ~~33~~ 17 bits of MR not all zeros or all ones"
p. 2-17, para. 7	Unsigned–Fractional: "upper ~~32~~ 16 bits of MR not all zeros"
p. 2-18	Multiplier Instruction Summary table now shows effects of multiplier instructions on status bits of ASTAT and STKY status registers.

p. 2-20, para. 1	"The X-input and the Z-input are always 32-bit fixed-point values. The Y-input is a 32-bit fixed-point value or ~~two 6-bit fields, bit6 and len6~~ an 8-bit field (shf8), positioned in the register file as shown in Figure 2.4. ~~Bit6 and len6 are interpreted as positive integers. Bit6 holds a bit position value. Len6 holds a field length value~~. Some shifter operations produce 6-bit or 8-bit results. These results are placed in either the bit6 or shf8 field, ~~shown in Figure 2.4~~, (see Figure 2.5) and are sign-extended to 32 bits. Thus the shifter always returns a 32-bit result."
p. 2-20, Figure 2.4	12-bit Y-Input (len6, bit6) representation is deleted from Figure 2.4 and is shown in (new) Figure 2.5 instead
p. 2-20	New section added: "2.7.2 Bit Field Deposit & Extract Instructions"
p. 2-24, para. 1	"The SZ flag indicates if the output is zero, the SV flag indicates an overflow, and the SS flag indicates the sign bit in exponent extract operations."
p. 2-24, last para.	"SS is affected ~~only by the two EXP operations~~ by all shifter operations. For the two EXP (exponent extract) operations, it is set if the fixed-point input operand is negative and cleared if it is positive. For all other shifter operations, SS is cleared."
p. 2-25	In Shifter Instruction Summary table, SS flag is now shown as being cleared by all instructions except EXP Rx and EXP Rx (EX).

p. 2-27, last para. "The individual registers of the register file are prefixed with an "f" when used in floating-point computations (in assembly language source code). The registers are prefixed with an "r" when used in fixed-point computations. The following instructions, for example, use the same three registers:

```
F0=F1*F2; floating-point multiply
R0=R1*R2; fixed-point multiply
```

The "f" and "r" prefixes do not affect the 40-bit (or 32-bit) data transfer; they only determine how the ALU, multiplier, or shifter treat the data."

p. 3-5, para. 2 "The *system register bit manipulation* instruction can be used to set, clear, toggle or test specific bits in these registers. This instruction is described in Appendix A, Group IV–Miscellaneous instructions."

p. 3-5 Table 3.1 now shows which registers are defined as System Registers.

p. 3-7, para. 3 "The bit test flag (BTF) is bit 18 of the ASTAT register. ~~The state of BTF is one of the conditions the ADSP-21020 evaluates. This read-only flag is affected by the system register bit test and XOR instructions. These operations are described at the end of this chapter~~. This flag is set (or cleared) by the results of the BIT TST and BIT XOR forms of the *system register bit manipulation* instruction, which can be used to test the contents of the ADSP-21020's system registers. This instruction is described in Appendix A, Group IV-Miscellaneous instructions. After BTF is set by this instruction, it can be used as the condition in a conditional instruction (with the mnemonic TF; see Table 3.2)."

p. 3-8, Table 3.2

LCE	Loop Cntr Expired (loop term)	CURLCNTR = ~~0~~ 1
NOT LCE	Loop Cntr Expired (condition)	CURLCNTR ≠ ~~0~~ 1

p. 3-9, last para. "This is similar to the *break* instruction of the C programming language used to prematurely terminate execution of a loop."

p. 3-13, para. 3 "Here is a simple example of an ADSP-21020 loop:

```
LCNTR=30, DO label UNTIL LCE;
        R0=DM(I0,M0), F2=PM(I8,M8);
        R1=R0-R15;
label: F4=F2+F3;
```

"

p. 3-13, last para. "If the termination condition is true, the sequencer fetches the next instruction after the end of the loop and pops the loop stack and PC stack."

p. 3-15, Figure 3.7 The One-Instruction Loop, Three Iterations table is modified in third, fourth, and fifth clock cycle columns.

p. 3-17, para. 1 ~~"For nested loops in which the outer loop's termination condition in not LCE, the end address of the outer loop must be at least two locations after the end address of the inner loop~~.

A non-counter-based loop is one in which the loop termination condition is something other than LCE. When a non-counter-based loop is the outer loop of a series of nested loops, the end address of the outer loop must be located at least two addresses after the end address of the inner loop.

The JUMP (LA) instruction is used to prematurely abort execution of a loop. When this instruction is located in the inner loop of a series of nested loops and the outer loop is non-counter-based, the address jumped to cannot be the last instruction of the outer loop. The address jumped to may, however, be the next-to-last instruction (or any earlier)."

p. 3-17, para. 2 "Non-counter-based short loops terminate in a special way because of the fetch-decode-execute instruction pipeline:"

p. 3-23 "• waitstates for memory accesses

• bus grant "

p. 3-23, last para. ~~"IRPTL is in an indeterminate state at reset. You should clear IRPTL by writing zeros to it before enabling interrupts or unmasking any interrupt.~~

IRPTL is cleared by a processor reset."

p. 3-27 ~~"If an RTI is specified as delayed, the two instructions following the RTI are executed after the status stack has been popped but before control returns to the main program. Any status bits read or written by those two instructions reflect the context of the main program, not the service routine.~~"

p. 3-28, para. 5 "The STKY register maintains stack ~~overflow/full and underflow flags for the PC stack, the status stack and the loop stacks~~ full and stack empty flags for the PC stack as well as overflow and empty flags for the status stack and loop stack. Unlike other STKY bits, ~~the stack overflow/full and underflow flags~~ several of these flag bits are not "sticky." They are set by the occurrence of the condition they indicate and are cleared when the condition is changed (by a push, pop or processor reset).

Bit	*Name*	*Definition*	*Sticky/Not Sticky*	*Cleared By*
21	PCFL	PC stack full	Not sticky	Pop
22	PCEM	PC stack empty	Not sticky	Push
23	SSOV	Status stack overflow	Sticky	RESET
24	SSEM	Status stack empty	Not sticky	Push
25	LSOV	Loop stacks overflow*	Sticky	RESET
26	LSEM	Loop stacks empty*	Not sticky	Push

* Loop address stack and loop counter stack

p. 3-30, para. 1 — "On return from the interrupt, execution continues at the instruction after the IDLE instruction."

p. 4-4, para. 6 — "The L register and modulo logic do not affect a pre-modified address—pre-modify addressing is always linear, not circular."

p. 4-6, para. 3 — "Circular buffer addressing must use M registers for post-modify of I registers, not pre-modify; for example:

```
F1=DM(I0,M0);     Use post-modify addressing for
                  circular buffers,
F1=DM(M0,I0);     not pre-modify. "
```

p. 4-8 — New section added: "4.3.2.3 Circular Buffer Overflow Interrupts"

p. 4-11, para. 2 — "For certain instruction sequences involving transfers to and from DAG registers, an extra (NOP) cycle is either automatically inserted by the processor **(1, 2)** or must be inserted in code by the programmer **(3)**. Certain other sequences cause incorrect results and are not allowed by the ADSP-21020 Assembler **(4)**."

p. 4-12, para. 1

"(Note that because the DAG2 registers are used to fetch instructions or access data in every cycle, a write to a program memory control register will *always* require an extra cycle to be inserted.)

Each of the following instruction sequences, for example,

~~PMWAIT=0x080000;~~
~~R15=PM(I10,M12);~~

```
PMWAIT=0x080000;  or   DMBANK1=0x10000000;
NOP;                   R15=DM(I0,M1);
```

"

p. 4-12, para. 3

"**3.)** An instruction that writes any L or M register of DAG2 (L8-L15, M8-M15), immediately followed by an instruction that reads the corresponding I register will result in incorrect data being read from the I register. The following instruction sequence, for example,

```
L8=24;
R0=I8;
```

will cause incorrect data to be read from I8. To prevent this, add a NOP to your program between the two instructions (i.e. the L or M register write and the I register read):

```
L8=24;
NOP;
R0=I8;
```

"

p. 6-2

"PMD_{47-0} Program Memory Data. The ADSP-21020 inputs and outputs data and instructions on these pins. 32-bit fixed-point data and 32-bit single-precision floating-point data is transferred over bits 47-16 of the PMD bus."

"~~PMS_0 — Program Memory Select 0. This pin is asserted to select bank 0 of program memory. Memory banks are user-defined in memory control registers.~~

~~$\overline{PMS}_1$ – Program Memory Select 1. This pin is asserted to select bank 1 of program memory. Memory banks are user-defined in memory control registers.~~"

"$\overline{PMS}_{1-0}$ Program Memory Select lines 1 & 0. These pins are asserted as chip selects for the corresponding banks of program memory. Memory banks must be defined in the processor's memory control registers. These pins are decoded program memory address lines and provide an early indication of a possible bus cycle."

"PMACK Program Memory Acknowledge. ~~An external device asserts this ADSP-21020 input to terminate a memory access. This is one method of creating wait states. This input is normally tied high for zero wait-state operation~~. An external device deasserts this input to add wait states to a memory access."

"$\overline{PMTS}$ Program Memory Three-State Control. ~~An active signal on this input places program memory address, data and control signals in a high-impedance state and locks out the PMACK input, without halting the processor~~. $\overline{PMTS}$ places the program memory address, data, selects, and strobes in a high-impedance state. If $\overline{PMTS}$ is asserted while a PM access is in progress, the processor will halt and the memory access will not be completed. PMACK must be asserted for at least one cycle when $\overline{PMTS}$ is deasserted to allow any pending memory access to complete properly. $\overline{PMTS}$ should only be asserted (low) during an active memory access cycle."

p. 6-3

"DMD_{39-0} Data Memory Data. The ADSP-21020 inputs and outputs data on these pins. 32-bit fixed-point data and 32-bit single-precision floating-point data is transferred over bits 39-8 of the DMD bus."

~~"$\overline{DMS}_0$ — Data Memory Select 0. This pin is asserted to select bank 0 of data memory. Memory banks are user-defined in memory control registers.~~

~~$\overline{DMS}_1$ — Data Memory Select 1. This pin is asserted to select bank 1 of data memory. Memory banks are user-defined in memory control registers.~~

~~$\overline{DMS}_2$ — Data Memory Select 2. This pin is asserted to select bank 2 of data memory. Memory banks are user-defined in memory control registers.~~

~~$\overline{DMS}_3$ — Data Memory Select 3. This pin is asserted to select bank 3 of data memory. Memory banks are user-defined in memory control registers."~~

"$\overline{DMS}_{3-0}$ Data Memory Select lines 0, 1, 2, & 3. These pins are asserted as chip selects for the corresponding banks of data memory. Memory banks must be defined in the processor's memory control registers. These pins are decoded data memory address lines and provide an early indication of a possible bus cycle."

"$\overline{DMACK}$ Data Memory Acknowledge. ~~An external device asserts this ADSP-21020 input to terminate a memory access. This is one method of creating wait states. This input is normally tied high for zero wait-state operation~~. An external device deasserts this input to add wait states to a memory access."

"$\overline{DMTS}$ Data Memory Three-State Control. ~~An active signal on this input places program memory address, data and control signals in a high-impedance state and locks out the DMACK input, without halting the processor~~. $\overline{DMTS}$ places the data memory address, data, selects, and strobes in a high-impedance state. If $\overline{DMTS}$ is asserted while a DM access is in progress, the processor will halt and the memory

	access will not be completed. DMACK must be asserted for at least one cycle when $\overline{\text{DMTS}}$ is deasserted to allow any pending memory access to complete properly. $\overline{\text{DMTS}}$ should only be asserted (low) during an active memory access cycle."
p. 6-4, para. 3	"1. The ADSP-21020 drives the read address and asserts a memory select signal to indicate the selected bank. A memory select signal is not deasserted between successive accesses of the same memory bank. 2. The ADSP-21020 asserts the read strobe (unless the memory access is aborted because of a conditional instruction)."
p. 6-5, para. 2	"1. The ADSP-21020 drives the write address and asserts a memory select signal to indicate the selected bank. A memory select signal is not deasserted between successive accesses of the same memory bank. ~~2. The ADSP-21020 asserts the write strobe~~. ~~3. The ADSP-21020 drives the data~~. 2. The ADSP-21020 asserts the write strobe and drives the data (unless the memory access is aborted because of a conditional instruction)."
p. 6-7, para. 1	"... the acknowledge should be deasserted (low) in the same cycle ~~after~~ that the three-state enable is deasserted in."
p. 6-7, para. 3	"~~DMACK~~"
p. 6-7, last para.	"~~PMACK~~"
p. 6-14	Figure 6.4, "Bus Request/Bus Grant Timing," is revised to show possible multicycle instruction execution completion before buses are granted.

p. 6-16, last para.

"To write a 48-bit word to a program memory location named *Port1*, for example, the following instructions would be used:

```
R0=0x9A00;      /* load R0 with 16 LSBs */
R1=0x12345678; /* load R1 with 32 MSBs */
PX1=R0;
PX2=R1;
PM(Port1)=PX;  /* write 16 LSBs to PM bits 15-0 */
          /* and 32 MSBs to PM bits 47-16 */
```

p. 7-4, para. 2

"An example is:

```
L2=8;
DM(I0,M1)=R1; "
```

p. 7-4, para. 3

"(Note that because the DAG2 registers are used to fetch instructions or access data in every cycle, a write to a program memory control register will *always* require an extra cycle to be inserted.)

Each of the following instruction sequences, for example, cause the ADSP-21020 to insert an extra cycle between the two instructions:

```
PMWAIT=0x080000;  or   DMBANK1=0x10000000;
NOP;                   R15=DM(I0,M1); "
```

An instruction that writes any L or M register of DAG2 (L8-L15, M8-M15), immediately followed by an instruction that reads the corresponding I register will result in incorrect data being read from the I register. The following instruction sequence, for example, will cause incorrect data to be read from I8:

```
L8=24;
R0=I8;
```

To prevent this, add a NOP between the two instructions:

```
L8=24;
NOP;
R0=I8; "
```

p. 7-14 — Table 7-14 is simplified (but no changes made in instruction definitions).

p. 7-19 — Instruction type 10 deleted.

p. 8-6 — In Listing 8.1, several segment addresses are corrected as follows:

```
.SEGMENT /RAM  /BEGIN=0x000100 /END=~~0x007FFF~~ /PM     pm_code;
.SEGMENT /RAM  /BEGIN=~~0x800000~~ /END=~~0xFFFFFF~~ /PM     pm_data;
.SEGMENT /RAM  /BEGIN=0x00000000 /END=~~0x007FFFFF~~ /DM   dm_data;

.SEGMENT /RAM  /BEGIN=0x000100 /END=0x0007FF /PM     pm_code;
.SEGMENT /RAM  /BEGIN=0x000800 /END=0x000FFF /PM     pm_data;
.SEGMENT /RAM  /BEGIN=0x00000000 /END=0x00007FF /DM   dm_data;
```

p. 8-20 — The next-to-last instruction of the `initial setups:` portion of Listing 8.4 is deleted:

```
~~bit set irptl 0;  {RESET doesn't clear this}~~
```

p. 8-27, para. 2 — ~~`bit set irptl 0;`~~

p. 8-27, para. 4 — ~~`bit set irptl 0;`~~

p. 8-28, para. 2 — ~~`bit set irptl 0;`~~

p. 8-34

In Listing 8.8, the

~~`f12=f12 - f12`~~

portion of the second instruction is replaced by:

`r8=r8 xor r8`

p. 9-4, para. 2

Therefore, to ensure recognition of an asynchronous input, it must be asserted for at least one full processor cycle plus setup and hold time (except for RESET, which must be asserted for at least four processor cycles).

p. 9-4, para. 4

Table 9.2 shows the states of outputs ~~after~~ during reset (i.e. while RESET is low).

p. 9-4

New section added: "9.4 RCOMP Pin"

p. 9-5, Table 9.1

The following reset values are corrected:

PC	~~Unchanged~~	0x0008
LCNTR	~~Unchanged~~	0x0000 (cleared)
IRPTL	~~Unchanged~~	0x0000 (cleared)
IMASK	~~0~~	0x0003
STKY	~~0~~	0x0540 0000
MODE2	~~0~~	0x*n*000 0000 (bits 28-31 are the device identification field, identifying the silicon revision #)
ASTAT	~~0~~	0x00*nn* 0000 (bits 19-22 are equal to the values of the FLAG0-3 input pins; the flag pins are configured as inputs after reset)

p. 9-6, para. 1

"During the first two memory accesses, which have ~~eight~~ seven wait states each due to the default value of the PMWAIT register, ..."

p. 9-6, Fig. 9.2

One additional CLKIN cycle is added between rising edge of RESET and start of first instruction fetch (0x000008 driven onto PMA bus).

p. 9-9, para. 2	"No pullup or pulldown resistors are needed on these unused pins—this is taken care of on-chip."		
p. 9-9, Fig. 9.4	Data bus lines corrected:	~~DMD0-31~~	DMD39-8
	Faster ADSP-21020 device:	~~50 ns~~	30 ns
	Memories relabeled:	~~7C196-35~~	SRAM 64K x 4 15 ns
p. 9-10, Fig. 9.5	Faster ADSP-21020 device:	~~50 ns~~	30 ns
	Memories relabeled:	~~7C199-35~~	SRAM 32K x 8 15 ns
p. 9-11, para. 2	"I/O devices should be connected to the 32-bit integer field (the *upper* 32 bits) of the DMD or PMD data buses—bits 39-8 of the DMD bus, and bits 47-16 of the PMD bus."		
p. 9-11, Fig. 9.6	Faster ADSP-21020 device:	~~50 ns~~	30 ns
	Memories relabeled:	~~7C199-35~~	SRAM 32K x 8 15 ns
p. 9-13, Fig. 9.8	Data bus lines corrected:	~~DMD31-0~~	DMD39-8
p. 9-15, Fig. 9.10	Data bus lines corrected:	~~DMD31-0~~	DMD39-8
p. 9-16, Fig. 9.11	Data bus lines corrected:	~~DMD31-0~~	DMD39-8
p. 9-19, Fig. 9.13	Data bus lines corrected:	~~DMD0-39~~	DMD39-8
p. 9-18 (1st Ed)	Section 9.5.2.2 "Shared Single-Port Memory" deleted.		

p. 9-30

Section 9.8 "EZ-ICE EMULATOR CONSIDERATIONS" is revised. In Figure 9.20 "Target Board Connector For EZ-ICE Probe" the signals BTRST and TRST are now shown as active low: BTRST TRST

p. A-8

LCE Loop Cntr Expired (loop term) CURLCNTR = ~~0~~ 1
NOT LCE Loop Cntr Expired (condition) CURLCNTR ≠ ~~0~~ 1

p. A-23

Instruction type 10 deleted.

p. A-28,29 (1st Ed)

Instruction type 10 deleted.

p. A-30

"The end address can be either a label for an absolute 24-bit program memory address, or a PC-relative, 24-bit twos-complement address."

p. A-32, Do Until

"The end address can be either a label for an absolute 24-bit program memory address, or a PC-relative, 24-bit twos-complement address."

"**Examples:**

```
DO ~~0x2F00~~ end UNTIL FLAG1_IN;
                {end is a program label}

DO (PC, ~~end~~ 7) UNTIL AC;
                ~~{end is user-defined label}~~ "
```

p. A-46, Idle

"On return from the interrupt, execution continues at the instruction following the IDLE instruction."

p. B-2, para.1

"The CU (computation unit) field is defined as follows:

CU=00 ALU operations
CU=01 Multiplier operations
CU=10 Shifter operations"

p. B-39, para.2

"The following code performs floating-point division using an iterative convergence algorithm.* The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). ~~This code executes in 8 cycles.~~ It requires these inputs: F0=numerator, F12=denominator, F11=2.0. It returns the quotient in F0. (The two highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F0=RECIPS F12, F7=F0;       {Get 8 bit seed R0=1/D}
F12=F0*F12;                 {D' = D*R0}
F7=F0*F7, F0=F11-F12;       {F0=R1=2-D', F7=N*R0}
F12=F0*F12;                 {F12=D'-D'*R1}
F7=F0*F7, F0=F11-F12;       {F7=N*R0*R1, F0=R2=2-D'}

~~RTS (DB)~~ F12=F0*F12;          {F12=D'=D'*R2}
F7=F0*F7, F0=F11-F12;       {F7=N*R0*R1*R2, F0=R3=2-D'}

F0=F0*F7;                   {F7=N*R0*R1*R2*R3}
```

Note that this code segment can be made into a subroutine by adding an `RTS(DB)` clause to the third-to-last instruction."

p. B-40, para.2

"The following code calculates a floating-point reciprocal square root ($1/\sqrt{x}$) using a Newton-Raphson iteration algorithm.* The result is accurate to one LSB in whichever format mode, 32-bit or 40-bit, is set (32-bit only for ADSP-21010). To calculate the square root, simply multiply the result by the original input. ~~This code executes in 13 cycles~~. It requires these inputs: F0=input, F8=3.0, F1=0.5. It returns the result in F4. (The four highlighted instructions can be removed if only a ±1 LSB accurate single-precision result is necessary.)

```
F4=RSQRTS F0;              {Fetch 4-bit seed}
F12=F4*F4;                 {F12=X0^2}
F12=F12*F0;                {F12=C*X0^2}
F4=F1*F4, F12=F8-F12;      {F4=.5*X0, F12=3-C*X0^2}
F4=F4*F12;                 {F4=X1=.5*X0(3-C*X0^2)}
F12=F4*F4;                 {F12=X1^2}
F12=F12*F0;                {F12=C*X1^2}
F4=F1*F4, F12=F8-F12;      {F4=.5*X1, F12=3-C*X1^2}

F4=F4*F12;                 {F4=X2=.5*X1(3-C*X1^2)}
F12=F4*F4;                 {F12=X2^2}
RTS (DB) F12=F12*F0;       {F12=C*X2^2}
F4=F1*F4, F12=F8-F12;      {F4=.5*X2, F12=3-C*X2^2}

F4=F4*F12;                 {F4=X3=.5*X2(3-C*X2^2)}
```

Note that this code segment can be made into a subroutine by adding an `RTS(DB)` clause to the third-to-last instruction."

p. B-46 — In Table B.4, the following Mod2 options are deleted:

~~(SSIR) —11 0—1~~
~~(SUIR) —01 0—1~~
~~(USIR) —10 0—1~~
~~(UUIR) —00 0—1~~

p. B-52 — MR Register Transfer instruction is moved to this page (from p. B-81 in 1st Edition).

p. B-55 to B-69, B-72, 73 — "SS ~~Is not affected~~ Is cleared"

p. B-64, 66, 68, 69 — "The floating-point extension field of Rn (bits 7-0 of the 40-bit word) is set to all 0s."

p. B-64, 66, 68 — New figures added for FDEP, FEXT instructions.

p. B-71

In the shifter operation Rn=EXP Rx (EX), the definition of the SS status flag is changed:

"SS ~~Is set if the fixed-point operand in Rx is negative (bit 31 is a 1), otherwise cleared~~"

"SS Is set if the exclusive OR of the AV status bit and the sign bit (bit 31) of the fixed-point operand in Rx is equal to 1, otherwise cleared"

p. C-3, Table C.1

Instruction Bits 1 2 3 4	Name	Register (Serial Path)	Type
~~0 1 x x~~	~~Reserved for emulation~~		~~Private~~
0 1 0 0	Reserved for emulation		Private
x x 1 0	Reserved for emulation		Private

p. C-9, 10

Scan Position	Latch Type	Signal Name
234	~~Output~~ Input	FLAG3 Input Latch
235	~~Input~~ Output	FLAG3 Output Latch
236	~~Output~~ Input	FLAG2 Input Latch
237	~~Input~~ Output	FLAG2 Output Latch
238	~~Output~~ Input	FLAG1 Input Latch
239	~~Input~~ Output	FLAG1 Output Latch
240	~~Output~~ Input	FLAG0 Input Latch
241	~~Input~~ Output	FLAG0 Output Latch

p. C-11

†† 1 = Drive the associated signals during the EXTEST and INTEST instructions

0 = Tristate the associated signals during the EXTEST and INTEST instructions

p. D-1, para.1

"The ADSP-21020 and ADSP-21010 support ~~two single-precision floating-point data formats~~ the 32-bit single-precision floating-point data format defined in the IEEE Standard 754/854. In addition, the ADSP-21020 supports an extended-precision version of the same format with eight additional bits in the mantissa (40 bits total). Both the ADSP-21020 and ADSP-21010 also support 32-bit fixed-point formats—fractional and integer—which can be signed (twos-complement) or unsigned.

p. D-2, para.5

"The IEEE single-precision floating-point data types supported by the ADSP-21020 and ADSP-21010 and their interpretations are summarized in Table D.1.

Type	*Exponent*	*Fraction*	*Value*
Normal	$1 \le e \le 254$	Any	$(-1)^s$ ~~$(0.f)$~~ 2^{e-127}
Normal	$1 \le e \le 254$	Any	$(-1)^s$ $(1.f_{22\text{-}0})$ 2^{e-127}

p. E-1, Table E.1

Register values after reset are specified.

p. E-2, para.1

"~~At reset, all system registers except IRPTL, USTAT1, and USTAT2 are cleared~~."

p. E-2, para.5

"~~BTF is read-only~~."

p. E-3

In the MODE1 Register bit definitions, the following text is added:

Bit	*Name*	*Definition*
8-9		Reserved
16	RND32	1=Round floating-point data to 32 bits; 0=Round to 40 bits *(must be set to 1 for ADSP-21010)*

p. E-5

In the ASTAT Register bit definitions, the following text is deleted:

Bit	*Name*	*Definition*
18	BTF	Bit test flag for system registers ~~(Read only)~~

and the following footnote is added:

* Bit 17 (DAG1 circular buffer 7 overflow) and Bit 18 (DAG2 circular buffer 15 overflow) indicate the occurrence of a circular buffer overflow. Rather then remaining set until explicitly cleared, however, these bits are cleared by the next subsequent memory access that uses the corresponding I register (I7, I15). Circular buffer interrupts, therefore, should be used instead of these STKY register bits. See Section 4.3.2.3, "Circular Buffer Overflow Interrupts," in Chapter 4.

Index

A

Abbreviations A–2
ABS B–15, B–31, B–77
Absolute value B–15, B–23, B–26, B–27, B–31, B–44
AC flag 2–8, 3–8, A–8, B–6, B–7, B–10, B–11
Accumulator 2–12
.ACH file 8–5, 8–8
Acknowledge; see also DMACK
  and PMACK 6–9, 9–12
Add with carry B–6
Addition B–4, B–6, B–12, B–24, B–26, B–75, B–77
Address decoding 8–7
AF flag 2–9
AI flag 2–9
AIS flag 2–7
Alternate registers 1–8, 2–28, 4–3
ALU 1–4, A–8
  Input operands 2–5
  Opcodes B–2, B–3
  Operations 2–5, 2–10, 7–13, B–2, B–3
  Status 2–7
ALU saturation 2–6
ALUSAT bit 2–6
AN flag 2–8, B–9, B–29
AND B–17
AOS flag 2–8
Architecture description file 8–5, 8–8, 8–16
Arithmetic shift B–57, B–58
Arithmetic status 3–7
AS flag 2–9
ASHIFT B–57, B–58
Assembler 1–10, 8–15, 8–29
Assembly library/librarian 1–10
ASTAT register 2–3, 3–5, 3–21, 3–27
  Default value at reset 9–5
  Summary E–5
  Ureg address A–9
Asynchronous inputs 9–4
Asynchronous interrupts 3–28
AUS flag 2–8
Autowrap 8–29
AV flag 2–8, 3–8, A–8
Average B–8, B–28, B–77
AVS flag 2–8
AZ flag 2–8, B–9, B–29

B

B registers 4–1
  Default value at reset 9–5
  Ureg address A–9
Background MR register 2–12
Base register 4–1
BCLR B–60
$\overline{BG}$; see also Bus grant 6–13
Binary logarithm B–36
Bit operation 7–20, A–4, A–42, E–2
Bit test A–8, B–63, E–2
Bit-reverse 4–9, A–42
  Instruction 4–10, A–42
  Mode 4–9, 6–8, 7–8
Bit6 ..... 2–20, A–2
BITREV instruction 4–10, 6–10, 7–20, A–42
Booting 9–25
Borrow B–7, B–11
Boundary register C–4
Boundary scan C–1
BR0 bit 4–9
$\overline{BR}$; see also Bus request/bus grant 6–13, 9–3
Branch 3–6, 3–9, A–4, A–24, A–26
BSET B–61
BTF flag 3–7, A–42, E–2
BTGL B–62
BTST B–63
Buffer latches 9–20
Built-in self-test (BIST) C–11
Bus exchange (PX registers) 6–15
Bus grant 3–23
Bus request/bus grant
  ..... 1–8, 6–6, 6–13, 7–5, 9–2, 9–16
  Timing 6–14

C

C compiler 1–10
CACC 2–7, E–5
Cache, instruction 3–30
  Efficiency 3–32
  Enable/disable 3–32
  Freeze 3–32
Cache, external 6–6, 9–13
CADIS bit 3–32
CAFRZ bit 3–32
Call 3–6, 3–9, A–4, A–24, A–26
Capacitive loads 9–27

Index

Carry A–8, B–6, B–10
CB7I 3–24
CB15I 3–24
Circular buffers 4–1, 4–6, 7–8
Clear bit A–4, A–42, B–60
Clear MR 2–14, B–52
CLIP B–23, B–44
CLKIN 9–1, 9–3
CLR A–42
Compare accumulation;
see also CACC 2–9, B–9, B–29
Compare B–9, B–29
Complement sign B–14, B–30
Computation unit 1–4, 2–2, 7–9, A–4
Computation unit register A–3
Compute field B–1
Compute operation A–1, B–1
Conditions 3–7
Codes 3–8, 7–11, A–4, A–8
Mnemonics 3–8, A–8
Conditional branch 3–9
Conditional instruction A–1
Context switch 1–8, 2–28, 4–3
Conventions, notation A–2
Conversion
Fixed-point-to-floating-point B–38, B–77
Floating-point-to-fixed-point B–37, B–77
COPYSIGN B–41
Counter A–4, A–5
Counter-based loops 3–15, 7–3, 8–31
CURLCNTR 3–5, 3–7, 3–18, 7–6, 7–8
Default value at reset 9–5
Ureg address A–9

D

DADDR 3–5
Default value at reset 9–5
Ureg address A–9
Data address generator 1–6, 4–1
DAG architecture 4–2
DAG register transfers 4–10
DAG registers 4–1, 7–4, 7–8, 8–12
DAG restrictions 4–11
DAG1 ..... 4–1, A–4
DAG2 4–1
Data memory access 6–4, 6–5, A–12, A–14, A–16, A–20, A–34, A–35, A–36
Data memory address hold time 6–10
Data memory interface 6–3
Data memory read cycle 6–4
Data memory write cycle 6–5
DB; see also Delayed branch 3–10, A–5, A–24, A–26
Decode cycle 3–2
Decrement B–13
DEF21020.H 8–21, 8–22
Delayed branch 3–9, 3–11, 3–22, 7–5, 8–39, A–5
Denormal numbers 2–3
Deposit field B–64, B–65, B–66, B–67
Development software 1–9, 8–1
Device identification register C–11
Direct branch 3–9, A–24
Direct memory access (DMA) 9–16
Division B–39
DMA bus 1–7
DMA controller 9–16
DMA31-0 ..... 6–3, 6–7, 9–1, 9–4
DMACK 6–3, 6–7, 6–9, 9–2, 9–12, 9–13, 9–14
DMADR 4–10, 4–12
Default value at reset 9–5
Ureg address A–10
DMBANK1 ..... 4–12, 6–8, 8–12
Default value at reset 9–5
Ureg address A–10
DMBANK2 ..... 4–12, 6–8, 8–12
Default value at reset 9–5
Ureg address A–10
DMBANK3 ..... 4–12, 6–8, 8–12
Default value at reset 9–5
Ureg address A–10
DMD bus 1–7, 6–15
DMD39-0 ..... 6–3, 6–7, 9–1, 9–4
DMPAGE 6–3, 6–7, 6–12, 9–2, 9–4
$\overline{\text{DMRD}}$ 6–3, 6–7, 9–1, 9–4, 9–10, 9–31
$\overline{\text{DMS0}}$ 4–10, 7–8, 9–9
$\overline{\text{DMS3-0}}$ 6–3, 6–7, 6–8, 8–7, 9–1, 9–4
$\overline{\text{DMTS}}$ 6–3, 6–6, 9–2, 9–13, 9–14
DMWAIT 4–12, 6–10, 8–12, 8–23
Default value at reset 9–5
Summary E–11
Ureg address A–10
$\overline{\text{DMWR}}$ 6–3, 6–7, 9–1, 9–4, 9–9
DO UNTIL instruction 3–6, 3–13, 3–19, 8–14, 8–31
DRAM interface 9–14
Dreg A–2
Dual add/subtract 2–26, B–75, B–80
Dual-port RAM 9–24
Dynamic RAM (DRAM) 6–12, 9–2

E

Edge-sensitive interrupts 3–27
Effect latency 3–5, 7–7, E–2
Emulator 1–11
.ENDSYS directive 8–7
EQ 3–8, A–8
EX (extended exponent) B–71
Exceptions 2–3
Execute cycle 3–2
EXP 2–24, B–70, B–71
Exponent extraction B–36, B–70, B–71
Extended floating-point format 2–3, D–2
.EXTERN directive 8–37
External interrupts 3–20, 3–27
Extra cycle conditions 7–2
Extract field B–68, B–69
EZ-ICE® emulator 1–11, 9–30

Index

F

F3-0 ..... A–3
F7-4 ..... A–3
F11-8 ..... A–3
F15-12 ..... A–3
Fa A–2
FADDR 3–5
  Default value at reset 9–5
  Ureg address A–9
FDEP B–64, B–65, B–66, B–67
Fetch cycle 3–2
FEXT B–68, B–69
Field deposit B–64, B–65, B–66, B–67
Field extract B–68, B–69
FIFOs 9–22
Filter coefficients 8–35, 8–36
FIX B–37
Fixed-point format 2–4, D–3
Fixed-point-to-floating-point B–38
FIXI 3–24
Flags (FLAG3-0) 3–27, 9–3, 9–7, 9–21, A–8
  Direction 9–7
  Timing 9–8
  Value 9–7
FLAG0_IN 3–8, A–8
FLAG1_IN 3–8, A–8
FLAG2_IN 3–8, A–8
FLAG3_IN 3–8, A–8
FLG0 9–7
FLG0O 9–7
FLG1 9–7
FLG1O 9–7
FLG2 9–7
FLG2O 9–7
FLG3 9–7
FLG3O 9–7
FLOAT B–38
Floating-point format 1–3, D–1, D–2
Floating-point precision 2–3, D–2
Floating-point-to-fixed-point B–37
FLTII 3–24
FLTOI 3–24
FLTUI 3–24
Fm A–2
Fn A–2
Foreground MR register 2–12
FOREVER 3–8, A–8
Fractional format B–46, D–3
Fractional result 2–12, D–4
Fs A–2
Fx A–2
Fy A–2

G-H

GE 3–8, A–8
GENERIC.ACH 8–6
.GLOBAL directive 8–31, 8–37
GT 3–8, A–8
High-level programming language 1–2
Host processor 9–32

I

I registers 4–1
  Default value at reset 9–5
  Ureg address A–9
I/O devices 9–11
Ia A–3
Ic A–3
IDLE instruction 3–29, 7–20, 8–1, 8–14, 8–29, A–46
Idle state 3–1, 3–23, 3–29
IEEE 754/854 standard 2–2, D–1
IEEE 1149.1 specification 9–25, C–1
IF A–1
IIR filter 8–35
IIRCOEFS.DAT 8–36
IIRIRQ.ACH 8–17
IIRIRQ.ASM 8–19, 8–20
IIRMEM.ASM 8–10
IMASK 3–5, 3–25, 8–27
  Default value at reset 9–5
  Summary E–8
  Ureg address A–9
IMASKP 3–5, 3–21, 3–26
  Default value at reset 9–5
  Ureg address A–9
Immediate address 7–20, A–34
Immediate data 7–20, A–1, A–36, A–37
Immediate modify 7–20, 4–6, A–16, A–35, A–42
Immediate shift operation A–2, A–20, B–54
Increment B–12
Index register 4–1, A–4, A–6
Indirect addressing 4–1
Indirect branch 3–9, A–26
Inexact flags 2–2
Infinity D–2
Initial setups 8–8, 8–19
Input operands
  ALU 2–5
  Multiplier 2–11
  Shifter 2–19
Instruction cache 1–6, 3–6, 3–30
Instruction groups 7–1, A–1, A–11
Instruction pipeline
  ..... 1–6, 3–3, 3–5, 3–12, 3–13, 3–15, 3–30
Instruction register C–2
Instruction set 1–8, 7–1
Instruction type 7–1, A–1
Integer format B–46, D–3
Integer result 2–12, D–4
Interrupt-driven operation 8–16
Interrupt-driven data transfers 9–37

Index

Interrupts
..... 1–7, 3–1, 3–20, 6–14, 7–6, 8–11, 8–13, 8–26
External 3–20, 3–27
Latency 3–21
Masking 3–25
Priority 3–20, 3–21, 3–26
Sensitivity 3–27
Service routine 3–21
Vectors 3–21, 7–17, 8–11, A–5
IRPTEN bit 3–25
IRPTL 3–5, 3–21, 3–23, 3–24, 3–25, 7–7, 8–28
Default value at reset 9–5
Summary E–8
Ureg address A–9
IRQ0E 3–28
IRQ0I 3–24
IRQ1E 3–28
IRQ1I 3–24
IRQ2E 3–28
IRQ2I 3–24
IRQ3-0; see also Interrupts, external 9–3
IRQ3E 3–28
IRQ3I 3–24

J-K-L

JTAG C–1
Jump 3–1, 3–6, A–4, A–5, A–24, A–26
L registers 4–1, 8–12
Default value at reset 9–5
Ureg address A–9
LA; see also Loop abort 3–9, A–3, A–24, A–26
LADDR 3–5, 3–18
Default value at reset 9–5
Ureg address A–9
LCE 3–8, 3–19, 7–8, A–8
LCNTR 3–5, 3–19, 8–14, 8–31
Default value at reset 9–5
Ureg address A–9
LE 3–8, A–8
Leading ones B–73
Leading zeros B–72
LEFTO B–73
LEFTZ B–72
Len6 ..... 2–20, A–2
Length register 4–1, 8–12
Level-sensitive interrupts 3–27
Linker 1–10, 8–15, 8–29
Load variations 9–27
Loader program 9–26
LOGB B–36
Logical shift B–55, B–56
Loops 3–1, 3–6, 3–13, 7–2, 8–14, 8–31
Loop-back 3–14
Nesting 3–14, 3–17
Restrictions 3–14, 7–6
Stacks A–3, A–5, A–6, A–44
Termination 3–14
Loop abort 3–9, 3–14, 3–18, A–3
Loop address stack 3–17
Loop counter 3–18, A–8
Loop counter stack 3–18
Looped code 8–31
Low power A–46
LRU bit 3–30
LSEM 3–28
LSHIFT B–55, B–56
LSOV 3–28
LT 3–8, A–8

M

M registers 4–1
Default value at reset 9–5
Ureg address A–9
MANT B–35
Mantissa extraction B–35
Map 1 ..... A–9
Map 2 ..... A–10
MAX B–22, B–43, B–77
Maximum B–22, B–43
Mb A–3
Md A–3
Memory
Access A–4, A–5
Banks 6–8, 8–7, 9–9, 9–10
Configurations 9–8
Initialization 8–13, 8–25
Interface capacitive load 9–27
Maps 8–9, 8–18
Paging 6–12
Segments 8–7
Memory-mapped I/O 9–11
Memory-resident data 8–2
MI flag 2–17
MIN B–21, B–42, B–77
Minimum B–21, B–42
MIS flag 2–16
Miscellaneous instructions A–1, A–39
MN flag 2–17
Mod1 ..... B–46
Mod2 ..... B–46
MODE1 ..... 3–5, 3–21, 3–27, 8–27
Default value at reset 9–5
Summary E–4
Ureg address A–9
MODE2 ..... 3–5, 8–27
Default value at reset 9–5
Summary E–3
Ureg address A–9
Modify operation 7–20, A–22, A–42
Modify register 4–1, A–5, A–6
Modify, immediate A–16
Modulo addressing 4–6
MOS flag 2–17
MR clear B–52

Index

MR register 2–12, B–1
MR register transfer 2–13, A–1, B–52
MR rounding B–51
MR saturation B–50
MR0 register 2–12
MR0B A–3, B–52
MR0F A–3, B–52
MR1 register 2–12
MR1B A–3, B–52
MR1F A–3, B–52
MR2 register 2–12
MR2B A–3, B–52
MR2F A–3, B–52
MRB 2–12
MRF 2–12
MS flag 3–8, A–8
MU flag 2–16
Multifunction computations
.... 1–5, 1–8, 2–26, 7–16, 8–34, 8–40, A–7, B–1, B–74
Multiplication, fixed-point B–47
Multiplication, floating-point B–53
Multiplier 1–4, A–8
Input operands 2–11
Opcodes B–45
Operations 2–18, 7–14, B–45
Status 2–16
Multiplier result (MR) 2–12, A–1, B–52
Multiplier/ALU operation B–77
Multiply/accumulate 2–11, B–48, B–49
Multiport memory 9–18
Multiprocessor configurations 9–18
MUS flag 2–16
MV flag 2–17, 3–8, A–8
MVS flag 2–17

N

NAN (Not-A-Number) 2–2, D–2
NE 3–8, A–8
Negate B–14, B–30
Nested loops 3–21, 3–26, 7–6
NESTM bit 3–21, 3–26
Nondelayed branch 3–9, 3–10, 7–2, A–5
NOP 7–20, A–45
NOT AC 3–8, A–8
NOT AV 3–8, A–8
NOT B–20
NOT FLAG0_IN 3–8, A–8
NOT FLAG1_IN 3–8, A–8
NOT FLAG2_IN 3–8, A–8
NOT FLAG3_IN 3–8, A–8
NOT LCE 3–8, A–8
NOT MS 3–8, A–8
NOT MV 3–8, A–8
NOT SV 3–8, A–8
NOT SZ 3–8, A–8
NOT TF 3–8, A–8
Notation conventions A–2
Numerical C compiler 1–10

O-P-Q

Opcodes A–1, A–5
ALU B–2, B–3
Multiplier B–45
Notation A–3
Shifter B–54
OR B–18, B–56, B–58, B–65, B–67
Overflow A–8
Page boundary detection 6–12, 7–5
Page size 6–12, 6–13
Parallel memory accesses A–12
Parallel multiplier/ALU operation 2–26
PASS B–16, B–32
PC 3–3, 3–5, 9–5
PC register address A–9
PC stack 3–12, 3–13, 3–21, A–3
PC stack pointer 3–13
PC-relative address A–2, A–6, A–24, A–26
PC-relative branch 3–9
PCEM 3–28
PCFL 3–28
PCSTK (PC stack) 3–5, 3–12
Default value at reset 9–5
Ureg address A–9
PCSTKP (PC stack pointer) 3–5, 3–13
Default value at reset 9–5
Ureg address A–9
PMA bus 1–7
PMA23-0 ..... 6–2, 6–7, 9–1, 9–4
PMACK 6–2, 6–7, 6–9, 9–2
PMADR 4–12
Default value at reset 9–5
Ureg address A–10
PMBANK1 ..... 4–12, 6–8
Default value at reset 9–5
Ureg address A–10
PMD bus 1–7, 6–15
PMD47-0 ..... 6–2, 6–7, 9–1, 9–4
PMPAGE 6–2, 6–7, 6–12, 9–2, 9–6
$\overline{\text{PMRD}}$ 6–2, 6–7, 9–1, 9–4, 9–31
$\overline{\text{PMS1-0}}$ 6–2, 6–7, 6–8, 9–1, 9–6
$\overline{\text{PMTS}}$ 6–2, 6–6, 9–2
PMWAIT 4–12, 6–10, 8–23
Default value at reset 9–5
Summary E–10
Ureg address A–10
$\overline{\text{PMWR}}$ 6–2, 6–7, 9–1, 9–4
Pointers 4–1
Pop loop stack 3–18
Pop stack 7–20, A–44
Post-modify 4–4
Powerup 9–4, 9–25
Pre-modify 4–4, A–26
Private instructions C–11
Probe connector 9–30
Program counter (PC) 3–3
Program flow 3–2, 7–19, A–1, A–23

Index

Program memory access 6–4, 6–5, A–12, A–14, A–16, A–20, A–34, A–35, A–36
Program memory boot 9–25
Program memory data access 3–6, 3–23, 3–30, 7–2
Program memory interface 6–2
Program memory read cycle 6–4
Program memory write cycle 6–5
Program sequencer 1–6, 3–1
  Architecture 3–3
  Registers 3–5
Programmable wait states 6–9
Programming 7–2, 8–1, 8–37
PROM splitter 1–10
Pullup resistors 9–9, 9–13
Push loop stack 3–18
Push stack 7–20, A–44
PX 6–15
  Default value at reset 9–5
  Ureg address A–10
PX1 6–15
  Default value at reset 9–5
  Ureg address A–10
PX2 6–15
  Default value at reset 9–5
  Ureg address A–10

R

R3-0 ..... A–2
R7-4 ..... A–2
R11-8 ..... A–2
R15-12 ..... A–3
Ra A–2
Read latency 3–5, 7–7
Reciprocal seed B–39
Reciprocal square root B–40
RECIPS B–39
Register file 2–1, 2–27, 7–9, 9–5, A–5, A–6
Register transfers 1–8, 4–10, 5–4, 6–15, A–18, B–52
RESET 9–1, 9–3, 9–4
Reset 6–14, 8–8, 9–4, 9–25
Return 3–6, 3–9
Return address 3–21
Rm A–2
Rn A–2
RND B–33
RND32 bit 2–3, 2–7, 2–15
Rolling loops 8–32
ROT B–59
Rotate B–59
Round MR 2–14
Rounding B–33, B–46, B–51
  Boundary 2–15
  Modes 2–4, 2–6, 2–15
Rs A–2
RSQRTS B–40
RSTI 3–24
Rx A–2
RXA A–7, B–78, B–79
RXM A–7, B–78, B–79
Ry A–2
RYA A–7, B–78, B–79
RYM A–7, B–78, B–79

S

Saturate MR 2–14
Saturation 2–6, 2–14, B–50
SCALB B–34
Scaling B–34
Scope of variables 8–37
SE (sign extension) B–66, B–67, B–69
Segments 8–7
Serial data flow 9–20
Serial scan path 1–3, C–4
Set bit A–4, A–40, B–61
SFT0I 3–24
SFT1I 3–24
SFT2I 3–24
SFT3I 3–24
SFT4I 3–24
SFT5I 3–24
SFT6I 3–24
SFT7I 3–24
Shf8 2–20
Shifter 1–4, A–8
  Fields 2–20
  Input operands 2–19
  Opcodes B–54
  Operations 2–25, 7–15, B–54
Shifter immediate operation A–2, A–20, B–54
Shiftimm A–2, A–20
Short loops 3–14, 7–3
Signed format B–46, D–3, D–4
Simulator 1–10, 8–15, 8–29
Single-function operation B–1
Software interrupts 3–25
SOVFI 3–24
Square root B–40
SRCU bit 2–13
SRD1H bit 4–4
SRD1L bit 4–4
SRD2H bit 4–4
SRD2L bit 4–4
Sreg; see also System register A–2, A–6, A–40
SRRFH bit 2–28
SRRFL bit 2–28
SS flag 2–24
SSEM 3–28
SSOV 3–28
Stack flags 3–28, 7–9
Stack operation A–44
Stack overflow 3–18, 3–28
Static RAM (SRAM) 9–9
Status flags 2–3
Status stack 3–21, 3–26, A–5, A–6, A–44

STKY register 2–4, 3–5, 3–23, 3–25
Default value at reset 9–5
Summary E–6
Ureg address A–9
Subroutines 3–1
Subtract with borrow B–7
Subtraction B–5, B–7, B–25, B–27, B–75, B–77
SV flag 2–24, 3–8, A–8
Synchronization delay 9–3
Syntax A–1
Syntax notation 7–10, A–2
.SYSTEM directive 8–7
System registers
..... 3–5, 7–12, A–2, A–6, A–9, A–40, E–1
SZ flag 2–24, 3–8, A–8, B–63

T

TCK 9–3, 9–30, C–1, C–2
TCOUNT 5–1
Default value at reset 9–5
Ureg address A–10
TDI 9–3, 9–30, C–2
TDO 9–3, 9–4, 9–30, C–2
Termination address (for loops) 3–17
Termination code (for loops) 3–17, 7–11
Test access port (TAP) C–2
Test bit A–4, A–40
TF 3–8, A–8
TGL A–40
Three-state enable 1–8, 6–6, 7–5, 7–9
TIMEN bit 5–2, 8–27
Timer 1–7, 3–30, 5–1, 8–25
Enable/disable 5–1
Interrupt 3–25, 5–4
TIMEXP 5–1, 9–3, 9–4
TMS 9–3, 9–30, C–2
TMZHI 3–24, 5–4, 8–27
TMZLI 3–24, 5–4, 8–27
Toggle bit A–4, A–40, B–62
TPERIOD 5–1
Default value at reset 9–5
Ureg address A–10
TRST 9–3, 9–30, C–2
TRUE 3–8, A–8
TRUNC bit 2–4, 2–6, 2–15
TST A–40

U-V

Universal register
..... 1–8, 7–12, A–2, A–6, A–9, A–18, A–37
Unsigned format B–46, D–3, D–4
Ureg; see also Universal register A–2, A–6,
User status registers E–2
USTAT1 ..... 3–5, E–2
Default value at reset 9–5
Ureg address A–9
USTAT2 ..... 3–5, E–2
Default value at reset 9–5
Ureg address A–9
Valid bit 3–30
.VAR directive 8–37
Vector A–5

W-X-Y-Z

Wait state modes 6–9
Wait states 6–7, 6–9, 6–13, 7–5, 7–9, 8–23, 9–13
XOR bit A–4, A–40
XOR B–19
Zero D–2

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

42-000538-02

E1798a–5–5/93