

A WORD ABOUT TRADEMARKS . . .

The following products mentioned in this manual are identified with registered AT&T trademarks:

WE[®] 32001 Processor Module

WE[®] 32100 Microprocessor

WE[®] 32200 Microprocessor

WE[®] 32201 Memory Management Unit

WE[®] 32206 Math Acceleration Unit

UNIX[®] Operating System

AT&T reserves the right to make changes to the product(s), including any hardware, software, and/or firmware contained therein, described herein without notice. No liability is assumed as a result of the use or application of this product(s). No rights under any patent accompany the sale of any such product(s).

Copyright © 1987 AT&T. All Rights Reserved.

Printed in United States of America



November 1987

**WE[®] 32200 Microprocessor
Information Manual**

PRELIMINARY

Select Code 307-705

COMCODE 105311815

ACKNOWLEDGEMENTS

Published by
The AT&T Documentation Management Organization

for the

Microsystems Product Management Organization
AT&T Information Systems

and the

62401343 Microsystems Systems Peripheral Development Department
AT&T Information Systems, Holmdel

FOREWORD

This manual contains information on the *WE 32200* Microprocessor that is essential to computer designers, software architects, and system design engineers. Additional information is available in the form of data sheets, application notes, and other documentation. For details concerning warranty or to obtain additional information, contact your AT&T Account Manager or call:

1-800-372-2447

To obtain additional copies of this manual, Select Code 307-705, call:

□ **1-800-432-6600**

**WE® 32200 MICROPROCESSOR
INFORMATION MANUAL**

CONTENTS

CHAPTER 1. INTRODUCTION

1. INTRODUCTION	1-1
1.1 DEVELOPMENT	1-2
1.2 ARCHITECTURE.....	1-2
1.3 INSTRUCTION SET	1-3
1.4 OPERATING SYSTEM SUPPORT.....	1-3
1.5 CHAPTER SUMMARIES.....	1-4
1.5.1 Chapter 2. Registers, Data, and Instruction Formats	1-4
1.5.2 Chapter 3. Signal Descriptions	1-4
1.5.3 Chapter 4. Bus Operations	1-4
1.5.4 Chapter 5. Instruction Set and Addressing Modes	1-4
1.5.5 Chapter 6. Operating System Considerations	1-4

CHAPTER 2. REGISTERS, DATA, AND INSTRUCTION FORMATS

2. REGISTERS, DATA, AND INSTRUCTION FORMATS	2-1
2.1 USER REGISTERS	2-1
2.2 CONVENTIONAL REGISTER SET	2-4
2.2.1 Stack Pointer (SP)	2-4
2.2.2 Program Counter (PC)	2-5
2.3 HIGH-LEVEL LANGUAGE SUPPORT GROUP	2-5
2.3.1 Frame Pointer (FP)	2-5
2.3.2 Argument Pointer (AP)	2-6
2.4 OPERATING SYSTEM SUPPORT GROUP	2-6
2.4.1 Processor Status Word (PSW)	2-6
2.4.2 Process Control Block Pointer (PCBP)	2-6
2.4.3 Interrupt Stack Pointer (ISP)	2-6
2.5 DATA TYPES	2-7
Bit Fields	2-5
2.5.1 Floating-Point Data	2-1
2.6 SIGN AND ZERO EXTENSION	2-1
2.7 DATA STORAGE IN MEMORY	2-1
2.8 ARBITRARY BYTE ALIGNMENT	2-1
2.9 DYNAMIC BUS SIZING	2-1
2.10 REGISTER DATA STORAGE	2-1
2.11 INSTRUCTION SET	2-1
2.12 INSTRUCTION STORAGE IN MEMORY	2-1
2.13 CONDITION FLAGS	2-2
2.14 TRACE MECHANISM	2-2

CHAPTER 3. SIGNAL DESCRIPTIONS

3. SIGNAL DESCRIPTIONS	3-1
3.1 ADDRESS AND DATA SIGNALS	3-2
Address (ADDR00—ADDR31)	3-2
Data (DATA00—DATA31)	3-2
3.2 INTERRUPT SIGNALS	3-2
Autovector ($\overline{\text{AVEC}}$)	3-2
Interrupt Option ($\overline{\text{INTOPT}}$)	3-2
Interrupt Priority Level (IPL0—IPL3)	3-2
Nonmaskable Interrupt ($\overline{\text{NMINT}}$)	3-3
3.3 INTERFACE AND CONTROL SIGNALS	3-3
Address Strobe ($\overline{\text{AS}}$)	3-3
Cycle Initiate ($\overline{\text{CYCLE}}$)	3-3
Coprocessor Done ($\overline{\text{DONE}}$)	3-3
Data Ready ($\overline{\text{DRDY}}$)	3-3
Data Strobe ($\overline{\text{DS}}$)	3-3
Data Transfer Acknowledge ($\overline{\text{DTACK}}$)	3-4
Dynamic 16-Bit Port Acknowledge($\overline{\text{DYN16}}$)	3-4
Synchronous Ready ($\overline{\text{SRDY}}$)	3-4
3.4 BUS ARBITRATION SIGNALS	3-4
Bus Arbiter ($\overline{\text{BARB}}$)	3-5
Bus Request ($\overline{\text{BUSRQ}}$)	3-5
Bus Request Acknowledge ($\overline{\text{BRACK}}$)	3-5
3.5 BUS EXCEPTION SIGNALS	3-5
Access Abort ($\overline{\text{ABORT}}$)	3-6
Data Bus Shadow ($\overline{\text{DSHAD}}$)	3-6
Fault ($\overline{\text{FAULT}}$)	3-6
Reset Acknowledge ($\overline{\text{RESET}}$)	3-6
Reset Request ($\overline{\text{RESETR}}$)	3-6
Retry ($\overline{\text{RETRY}}$)	3-6
Relinquish and Retry Request Acknowledge ($\overline{\text{RRRACK}}$)	3-6
Relinquish and Retry Request ($\overline{\text{RRREQ}}$)	3-7
3.6 ACCESS STATUS SIGNALS	3-7
Block (Double-Word) Fetch ($\overline{\text{BLKFTCH}}$)	3-7
Data Size (DSIZE0—DSIZE2)	3-7
Read/Write (R/ $\overline{\text{W}}$)	3-8
Access Status Codes (SAS0—SAS3)	3-8
Virtual Address ($\overline{\text{VAD}}$)	3-9
Execution Mode (XMD0—XMD1)	3-9
3.7 DEVELOPMENT SYSTEM SUPPORT SIGNALS	3-10
High Impedance ($\overline{\text{HIGHZ}}$)	3-10
Instruction Queue Status (IQS0—IQS1)	3-10
Start of Instruction ($\overline{\text{SOI}}$)	3-10
Stop ($\overline{\text{STOP}}$)	3-10
3.8 CLOCK SIGNALS	3-10
Clock (CLK23)	3-10

Clock (CLK34)	3-11
3.9 PIN ASSIGNMENTS	3-11

CHAPTER 4. BUS OPERATION

4. BUS OPERATION	4-1
4.1 SIGNAL SAMPLING POINTS.....	4-1
4.2 READ AND WRITE OPERATIONS	4-1
4.2.1 Synchronous Read Transaction	4-4
4.2.2 Asynchronous Read Transaction	4-6
4.2.3 Synchronous Read Transaction with Wait Cycle	4-10
4.2.4 Asynchronous Read Transaction with Two Wait Cycles	4-11
4.2.5 Synchronous Write Transaction	4-12
4.2.6 Asynchronous Write Transaction	4-12
4.2.7 Write Transaction with Wait Cycle Using $\overline{\text{SRDY}}$	4-12
4.2.8 Write Transaction with Wait Cycle Using $\overline{\text{DTACK}}$	4-18
4.3 READ INTERLOCKED OPERATION	4-20
4.4 COMPARE AND SWAP INTERLOCKED TRANSACTION.....	4-20
4.5 BLOCKFETCH OPERATION.....	4-23
4.5.1 Blockfetch Transaction Using $\overline{\text{SRDY}}$	4-23
4.5.2 Blockfetch Transaction Using $\overline{\text{DTACK}}$	4-25
4.5.3 Blockfetch Transaction Using $\overline{\text{DTACK}}$ with Wait Cycle On Second Word ..	4-26
4.5.4 Blockfetch Transaction Using $\overline{\text{DTACK}}$ with Wait Cycles On Both Words ...	4-27
4.6 BUS EXCEPTIONS.....	4-28
4.6.1 Faults	4-29
$\overline{\text{FAULT}}$ with $\overline{\text{SRDY}}$	4-31
$\overline{\text{FAULT}}$ After $\overline{\text{DTACK}}$	4-33
4.6.2 Retry.....	4-33
4.6.3 Relinquish and Retry	4-33
4.7 BLOCKFETCH SPECIAL CASES	4-36
4.7.1 Fault on First Word of Blockfetch with Status Code Other Than Prefetch ...	4-36
4.7.2 Fault on First Word of Blockfetch with Status of Prefetch	4-36
4.7.3 Retry on First Word of Blockfetch	4-36
4.7.4 Retry on Second Word of Blockfetch	4-36
4.7.5 Relinquish and Retry on Blockfetch.....	4-41
4.8 INTERRUPTS.....	4-41
4.8.1 Interrupt Acknowledge	4-43
4.8.2 Autovector Interrupt	4-46
4.8.3 Nonmaskable Interrupt	4-46
4.9 BUS ARBITRATION	4-49
4.9.1 Bus Request During a Bus Transaction	4-49
4.9.2 DMA Operation	4-51
4.10 RESET.....	4-52
4.10.1 System Reset	4-52
4.10.2 Internal Reset	4-52
4.10.3 Reset Sequence	4-54
4.11 ABORTED MEMORY ACCESSES.....	4-54

4.11.1 Aborted Access on PC Discontinuity with Instruction Cache Hit	4-55
4.11.2 Alignment Fault Bus Activity	4-56
4.12 SINGLE-STEP OPERATION	4-57
4.13 COPROCESSOR OPERATIONS	4-58
4.13.1 Coprocessor Broadcast	4-58
4.13.2 Coprocessor Operand Fetch	4-63
4.13.3 Coprocessor Status Fetch	4-64
4.13.4 Coprocessor Data Write	4-65
4.14 SUPPLEMENTARY PROTOCOL DIAGRAMS	4-66

CHAPTER 5. INSTRUCTION SET AND ADDRESSING MODES

5. INSTRUCTION SET AND ADDRESSING MODES	5-1
5.1 REGISTERS	5-1
5.2 ADDRESSING MODES	5-3
5.2.1 Format 1 Addressing Modes	5-5
Register Mode	5-9
Register Deferred Mode	5-11
Displacement Mode	5-12
Deferred Displacement Mode	5-14
Immediate Mode	5-15
Absolute Mode	5-16
Absolute Deferred Mode	5-16
Expanded Operand Mode	5-18
5.2.2 Format 2 Addressing Modes	5-20
Auto Pre/Post Increment/Decrement	5-20
Indexed Register Modes	5-21
Format 1 Addressing Mode for Registers 16-31	5-25
5.3 INSTRUCTION SET FUNCTIONAL GROUPS	5-27
5.3.1 Data Transfer Instructions	5-28
5.3.2 Arithmetic Instructions	5-29
5.3.3 BCD Instructions	5-31
5.3.4 Logical Instructions	5-32
5.3.5 Program Control Instructions	5-34
5.3.6 Coprocessor Instructions	5-40
5.3.7 Stack and Miscellaneous Instructions	5-41
5.4 INSTRUCTION SET LISTINGS	5-42
5.4.1 Notation	5-42
5.4.2 Instruction Set Summary by Mnemonic	5-44
5.4.3 Instruction Set Summary by Opcode	5-49
5.4.4 Instruction Set Descriptions	5-54
Add (ADDB2, ADDH2, ADDW2)	5-55
Add, 3 Address (ADDB3, ADDH3, ADDW3)	5-56
Add Packed Decimal (ADDPB2)	5-57
Add Packed Decimal, 3 Address (ADDPB3)	5-58
Arithmetic Left Shift (ALSW3)	5-59
And (ANDB2, ANDH2, ANDW2)	5-60

And, 3 Address (ANDB3, ANDH3, ANDW3).....	5-61
Arithmetic Right Shift (ARSB3, ARSH3, ARSW3)	5-62
Branch on Carry Clear (BCCB, BCCH).....	5-63
Branch on Carry Set (BCSB, BCSH).....	5-64
Branch on Equal (BEB, BEH)	5-65
Branch on Greater Than (Signed) (BGB, BGH).....	5-66
Branch on Greater Than or Equal (Signed) (BGEB, BGEH)	5-67
Branch on Greater Than or Equal (Unsigned) (BGEUB, BGEUH).....	5-68
Branch on Greater Than (Unsigned) (BGUB, BGUH).....	5-69
Bit Test (BITB, BITH, BITW)	5-70
Branch on Less Than (Signed) (BLB, BLH)	5-71
Branch on Less Than or Equal (Signed) (BLEB, BLEH).....	5-72
Branch on Less Than or Equal (Unsigned) (BLEUB, BLEUH).....	5-73
Branch on Less Than (Unsigned) (BLUB, BLUH)	5-74
Branch on Not Equal (BNEB, BNEH)	5-75
Breakpoint Trap (BPT)	5-76
Branch (BRB, BRH)	5-77
Branch to Subroutine (BSBB, BSBH).....	5-78
Branch on Overflow Clear (BVCB, BVCH).....	5-79
Branch on Overflow Set (BVSF, BVSH).....	5-80
Call Procedure (CALL)	5-81
Compare and Swap Word Interlock (CASWI)	5-82
Cache Flush (CFLUSH).....	5-83
Clear (CLRB, CLRH, CLRW)	5-84
Clear X Bit (CLRX)	5-85
Compare (CMPB, CMPH, CMPW)	5-86
Decrement (DECB, DECH, DECW)	5-87
Divide (DIVB2, DIVH2, DIVW2).....	5-88
Divide, 3 Address (DIVB3, DIVH3, DIVW3)	5-89
Decrement and Test (DTB, DTH).....	5-90
Extract Field (EXTFB, EXTFH, EXTFW).....	5-91
Extended Opcode (EXTOP)	5-92
Increment (INCB, INCH, INCW).....	5-93
Insert Field (INSFB, INSFH, INSFW)	5-94
Jump (JMP).....	5-96
Jump to Subroutine (JSB)	5-97
Logical Left Shift (LLSB3, LLSH3, LLSW3)	5-98
Logical Right Shift (LRSW3)	5-99
Move Complemented (MCOMB, MCOMH, MCOMW).....	5-100
Move Negated (MNEGB, MNEGH, MNEGW).....	5-101
Modulo (MODB2, MODH2, MODW2).....	5-102
Modulo, 3 Address (MODB3, MODH3, MODW3)	5-103
Move Address (Word) (MOVAW)	5-104
Move (MOVB, MOVH, MOVW).....	5-105
Move Block (MOVBLW)	5-107
Multiply (MULB2, MULH2, MULW2)	5-109
Multiply, 3 Address (MULB3, MULH3, MULW3).....	5-110
Move Version Number (MVERNO).....	5-111

No Operation (NOP, NOP2, NOP3)	5-112
OR (ORB2, ORH2, ORW2).....	5-113
OR, 3 Address (ORB3, ORH3, ORW3)	5-114
Pack BCD Halfword	5-115
Pop (Word) (POPW).....	5-116
Push Address (Word) (PUSHAW)	5-117
Push (Word) (PUSHW).....	5-118
Return on Carry Clear (RCC)	5-119
Return on Carry Set (RCS)	5-119
Return on Equal (REQL, REQLU)	5-120
Restore Registers (RESTORE)	5-121
Return from Procedure (RET)	5-122
Return on Greater Than or Equal (Signed) (RGEQ)	5-123
Return on Greater Than or Equal (Unsigned) (RG EQU).....	5-123
Return on Greater Than (Signed) (RGTR)	5-124
Return on Greater Than (Unsigned) (RGTRU)	5-124
Return on Less Than or Equal (Signed) (RLEQ)	5-125
Return on Less Than or Equal (Unsigned) (RLEQU).....	5-125
Return on Less Than (Signed) (RLSS).....	5-126
Return on Less Than (Unsigned) (RLSSU).....	5-126
Return on Not Equal (RNEQ, RNEQU)	5-127
Rotate (ROTW)	5-128
Return from Subroutine (RSB)	5-129
Return on Overflow Clear (RVC).....	5-129
Return on Overflow Set (RVS)	5-130
Save Registers (SAVE).....	5-131
Set X Bit (SETX).....	5-132
Coprocessor Operation (no operands) (SPOP)	5-133
Coprocessor Operation Read (SOPRS, SOPRD, SOPRT).....	5-134
Coprocessor Operation, 2 Address (SOPS2, SOPD2, SPOPT2)	5-135
Coprocessor Operation Write (SOPWS, SPOPWD, SPOPWT)	5-137
String Copy (STRCPY).....	5-138
String End (STREND).....	5-140
Subtract (SUBB2, SUBH2, SUBW2).....	5-141
Subtract, 3 Address (SUBB3, SUBH3, SUBW3)	5-142
Subtract Packed Decimal (SUBPB2).....	5-143
Subtract Packed Decimal, 3 Address (SUBPB3).....	5-144
Swap (Interlocked) (SWAPBI, SWAPHI, SWAPWI)	5-145
Test Equal, Decrement, and Test (TEDTB, TEDTH).....	5-146
Test Greater than, Decrement, and Test (TGDTB, TGDTH)	5-147
Test Greater than or Equal, Decrement, and Test (TGEDTB, TGEDTH)..	5-148
Test not Equal, Decrement, and Test (TNEDTB, TNEDTH)	5-149
Test (TSTB, TSTH, TSTW).....	5-150
Unpack BCD Byte (UNPACKB).....	5-151
Exclusive Or (XORB2, XORH2, XORW2).....	5-152
Exclusive Or, 3 Address (XORB3, XORH3, XORW3)	5-153

CHAPTER 6. OPERATING SYSTEM CONSIDERATIONS

6. OPERATING SYSTEM CONSIDERATIONS	6-1
6.1 FEATURES OF THE OPERATING SYSTEM	6-1
6.1.1 Memory Management Considerations for Virtual Memory Systems	6-4
6.2 STRUCTURE OF A PROCESS	6-5
6.2.1 Execution Privilege	6-5
6.2.2 Execution Stack	6-6
6.2.3 Process Control Block	6-7
Initial Context for a Process	6-11
Saved Context for a Process	6-12
Memory Specifications	6-12
6.2.4 Processor Status Word	6-13
6.3 SYSTEM CALL	6-13
6.3.1 Gate Mechanism	6-17
Pointer Table	6-17
Handling-Routine Tables	6-17
6.3.2 GATE Instruction	6-18
First Entry Point GATE Instruction Entry	6-18
Second Entry Point—The Gate Mechanism	6-19
6.3.3 Return-from-Gate (RETG) Instruction	6-21
6.4 PROCESS SWITCHING	6-21
6.4.1 Context-Switching Strategy	6-22
R-Bit	6-22
AR-Bit	6-22
I-Bit	6-23
6.4.2 Call Process (CALLPS) Instruction	6-27
6.4.3 Return-to-Process (RETPS) Instruction	6-28
6.4.4 User Call Process (UCALLPS) Instruction	6-30
6.5 INTERRUPTS	6-30
6.5.1 Interrupt-Handler Model	6-30
6.5.2 Interrupt Mechanism	6-31
Full-Interrupt Handler's PCB	6-33
Interrupt Stack and ISP	6-33
Interrupt-Vector Table	6-35
6.5.3 On-Interrupt Microsequence	6-35
6.5.4 Returning From an Interrupt	6-36
Full-interrupts	6-36
Quick-Interrupts	6-37
6.6 EXCEPTIONS	6-37
6.6.1 Levels of Exception Severity	6-37
6.6.2 Exception Handler	6-37
6.6.3 Exception Microsequences	6-38
Normal Exceptions	6-39
Stack Exceptions	6-42
Process Exceptions	6-43
Reset Exceptions	6-44
6.7 MEMORY MANAGEMENT FOR VIRTUAL MEMORY SYSTEMS	6-44

6.7.1	Virtual Address Fields	6-45
6.7.2	Initializing the Memory Management Unit	6-48
	Defining Virtual Memory	6-48
	Peripheral Mode	6-48
6.7.3	MMU Interactions	6-48
	MMU Exceptions	6-49
	Flushing	6-49
	Translation Probe	6-49
6.7.4	Efficient Mapping Strategies	6-49
6.7.5	Indirect Segment Descriptors	6-50
6.7.6	Using the Cacheable Bit	6-50
6.7.7	Physical Data Cache	6-50
	Selective Caching	6-51
	Flush Data Cache Register (FDCR)	6-51
	Reset	6-51
	Disabling the Cache	6-51
6.7.8	Using the Page-Write Fault	6-51
6.7.9	Access Protection	6-52
6.7.10	Using the Software Bits	6-52
6.8	OPERATING SYSTEM INSTRUCTIONS	6-52
6.8.1	Notation	6-52
6.8.2	Privileged Instructions	6-53
	Call Process (CALLPS)	6-55
	Disable Virtual Pin and Jump (DISVJMP)	6-57
	Enable Virtual Pin and Jump (ENBVJMP)	6-58
	Interrupt Acknowledge (INTACK)	6-59
	Return to Process (RETPS)	6-60
	Return From Quick-Interrupt (RETQINT)	6-63
	Wait (WAIT)	6-65
6.8.3	Nonprivileged Instructions	6-66
	Gate (GATE)	6-67
	Move Translated Word (MOVTRW)	6-71
	Return from Gate (RETG)	6-73
	User Call Process (UCALLPS)	6-75
6.8.4	Microsequences	6-78
	On-Normal Exception	6-79
	On-Stack Exception	6-83
	On-Process Exception	6-85
	On-Reset Exception	6-87
	On-Interrupt	6-89
	XSWITCH Microsequence	6-94

GLOSSARY

ACRONYMS

INDEX

LIST OF FIGURES

Figure 1-1. The WE 32200 Microprocessor	1-1
Figure 2-1. WE 32200 Microprocessor Block Diagram	2-1
Figure 2-2. Programmer's Model for User Register	2-3
Figure 2-3. Conventional Register Set.....	2-4
Figure 2-4. The WE 32200 Microprocessor Stack	2-5
Figure 2-5. High-Level Language Register Support Group	2-6
Figure 2-6. Operating System Register Support Group.....	2-7
Figure 2-7. Byte Data	2-8
Figure 2-8. Halfword Data	2-8
Figure 2-9. Word Data	2-8
Figure 2-10. Packed BCD Byte	2-8
Figure 2-11. Unpacked BCD Bytes	2-9
Figure 2-12. Extraction of a Bit Field	2-10
Figure 2-13. Floating-Point Data Types	2-11
Figure 2-14. Extending Data to 32 Bits.....	2-12
Figure 2-15. Word Storage in Memory.....	2-13
Figure 2-16. Instruction Storage in Memory	2-20
Figure 2-17. Operand Format.....	2-20
Figure 2-18. Word Storage Within an Instruction	2-21
Figure 2-19. Condition Flags	2-22
Figure 3-1. Signal Grouping Diagram	3-1
Figure 3-2. WE 32200 Microprocessor 133-Pin Square, Ceramic PGA Package Bottom View	3-11
Figure 4-1. Signal Sampling Points	4-2
Figure 4-2. Synchronous Read Transaction	4-5
Figure 4-3. Asynchronous Read Transaction	4-7
Figure 4-4. Asynchronous Read Transaction - $\overline{\text{DYN16}}$ Asserted After $\overline{\text{DTACK}}$...	4-8
Figure 4-5. Asynchronous Read Transaction Only	4-9
Figure 4-6. Synchronous Read Transaction with One Wait Cycle	4-10
Figure 4-7. Asynchronous Read Transaction with Two Wait Cycles.....	4-11
Figure 4-8. Synchronous Write Transaction.....	4-13
Figure 4-9. Asynchronous Write Transaction.....	4-14
Figure 4-10. Asynchronous Write Transaction - $\overline{\text{DYN16}}$ Asserted After $\overline{\text{DTACK}}$	4-15
Figure 4-11. Asynchronous Write Transaction - $\overline{\text{DYN16}}$ Only.....	4-16
Figure 4-12. Write Transaction with Two Wait Cycles - $\overline{\text{SRDY}}$ Only.....	4-17
Figure 4-13. Write Transaction with Wait Cycle - $\overline{\text{DTACK}}$ Only	4-18
Figure 4-14. Read-Interlocked Transaction - $\overline{\text{DTACK}}$ Only.....	4-20
Figure 4-15. Compare and Swap Interlocked Transaction - $\overline{\text{DTACK}}$ Only.....	4-21
Figure 4-16. Blockfetch Transaction - $\overline{\text{SRDY}}$ Only.....	4-23
Figure 4-17. Blockfetch Transaction - $\overline{\text{DTACK}}$ Only	4-24
Figure 4-18. Blockfetch Transaction - $\overline{\text{DTACK}}$ Only, with Wait Cycle on Second Word.....	4-25

Figure 4-19. Blockfetch Transaction - $\overline{\text{SRDY}}$ Only, with Wait Cycle on Both Words.....	4-26
Figure 4-20. Asynchronous Fault without $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, and $\overline{\text{DYN16}}$ (Read Transaction).....	4-28
Figure 4-21. $\overline{\text{FAULT}}$ with $\overline{\text{SRDY}}$ (Synchronous Fault)	4-29
Figure 4-22. $\overline{\text{FAULT}}$ After Assertion of $\overline{\text{DTACK}}$ (Write Transaction is Shown) ..	4-30
Figure 4-23. Retry of Transaction (Read Transaction is Shown).....	4-32
Figure 4-24. Relinquish and Retry.....	4-33
Figure 4-25. Fault on First Word of Blockfetch Transaction with Access Status Code Other than Prefetch	4-35
Figure 4-26. Fault on First Word of Blockfetch Transaction with Access Status Code of Prefetch	4-36
Figure 4-27. Retry on First Word of Blockfetch Transaction.....	4-37
Figure 4-28. Retry on Second Word of Blockfetch Transaction.....	4-38
Figure 4-29. Interrupt Acknowledge	4-42
Figure 4-30. Autovector Interrupt Acknowledge	4-45
Figure 4-31. Nonmaskable Interrupt Acknowledge	4-46
Figure 4-32. Bus Request During a Transaction	4-48
Figure 4-33. Reset Sequence	4-52
Figure 4-34. Aborted Access on Instruction-Cache Hit with PC Discontinuity	4-53
Figure 4-35. Alignment fault Bus Activity (Write Translation is Shown).....	4-54
Figure 4-36. Start of Single-Step Operation	4-55
Figure 4-37. Single-Step Operation.....	4-56
Figure 4-38. Coprocessor Command and ID Transfer (Sheet 1 of 3)	4-57
Figure 4-39. Coprocessor Command and ID Transfer (No Coprocessor Present) ..	4-60
Figure 4-40. Coprocessor Operand Fetch.....	4-61
Figure 4-41. Coprocessor Status Fetch Using $\overline{\text{SRDY}}$	4-62
Figure 4-42. Coprocessor Data Write	4-63
Figure 4-43. Read Transaction Followed by a Read Transaction - $\overline{\text{DTACK}}$ Only... ..	4-65
Figure 4-44. Read Transaction Followed by a Write Transaction - $\overline{\text{DTACK}}$ Only ..	4-66
Figure 4-45. Write Transaction Followed by a Write Transaction - $\overline{\text{DTACK}}$ Only..	4-67
Figure 4-46. Write Transaction Followed by a Write Transaction - $\overline{\text{DYN16}}$ and $\overline{\text{DTACK}}$	4-68
Figure 4-47. Write Transaction Followed by a Read Transaction	4-69
Figure 4-48. Double-Word Instruction Fetch Without Blockfetch Transaction - $\overline{\text{DTACK}}$ Only	4-70
Figure 4-49. Bus Arbitration During Relinquish and Retry	4-71
Figure 4-50. Faulted Coprocessor Status Read.....	4-72
Figure 4-51. Timing of $\overline{\text{IQS}}$ and $\overline{\text{SOI}}$ Signals	4-73
Figure 4-52. Assertion of $\overline{\text{STOP}}$ After $\overline{\text{SOI}}$	4-73
Figure 4-53. Assertion of $\overline{\text{STOP}}$ with $\overline{\text{SOI}}$	4-74
Figure 5-1. Instruction Format	5-4
Figure 5-2. Operand Format	5-5
Figure 5-3. Descriptor Byte Format	5-5
Figure 5-4. Register Mode Example	5-10
Figure 5-5. Deferred Addressing Using a Pointer	5-11

Figure 5-6. Register Deferred Mode Example.....	5-12
Figure 5-7. Example of MOVE 0x30(% r2),% r3	5-12
Figure 5-8. A Displacement Mode Source Operand	5-13
Figure 5-9. Deferred Displacement Addressing	5-14
Figure 5-10. A Deferred Displacement Mode Source Operand	5-14
Figure 5-11. A 32-Bit Immediate Source Operand.....	5-15
Figure 5-12. An Absolute Mode Source Operand	5-17
Figure 5-13. An Absolute Deferred Mode Source Operand.....	5-17
Figure 5-14. Expanded-Operand Mode Descriptor Bytes	5-18
Figure 5-15. Expanded-Operand Mode Example	5-19
Figure 5-16. Auto Post-Decrement Mode Example.....	5-21
Figure 5-17. Indexed Register (Byte Displacement) Mode Example	5-23
Figure 5-18. Indexed Register (Halfword Displacement) Mode Example	5-24
Figure 5-19. Example of Indexed Register Mode with Scaling	5-25
Figure 5-20. Stack After CALL-SAVE Sequence	5-39
Figure 6-1. Memory Map.....	6-8
Figure 6-2. Typical Process Control Block (PSW<R> == 1 and PSW<AR>).....	6-10
Figure 6-3. Typical Process Control Block (PSW<R> == 1 and PSW<AR> == 1).....	6-11
Figure 6-4. Pointer and Handling Table Indexing	6-20
Figure 6-5. A PCB on an Initial Process Switch to a Process	6-24
Figure 6-6. A PCB on a Process Switch During Execution of a Process	6-26
Figure 6-7. Interrupt Stacks (Quick- and Full-Interrupts).....	6-34
Figure 6-8. On-Normal (Gate-like) Exception Indexing	6-40
Figure 6-9. Virtual Address Fields for 2K Page Size	6-44
Figure 6-10. Virtual Address Fields for 4K Page Size	6-44
Figure 6-11. Virtual Address Fields for 8K Page Size	6-44
Figure 6-12. Virtual to Physical Address Translation	6-47

LIST OF TABLES

Table 2-1. Read Accesses for Aligned and Nonaligned Data (32-Bit Port)	2-14
Table 2-2. Write Accesses for Aligned and Nonaligned Data (32-Bit Port)	2-14
Table 2-3. Replication Patterns for Write Accesses	2-14
Table 2-4. Read Accesses for Aligned and Nonaligned Data (16-Bit Port)	2-16
Table 2-5. Write Accesses for Aligned and Nonaligned Data (16-Bit Port)	2-17
Table 2-6. 16-Bit Instructions Fetches.....	2-18
Table 2-7. Trace Mechanism.....	2-23
Table 3-1. WE 32200 Microprocessor Pin Descriptions	3-12
Table 4-1. Simultaneously Asserted Exception Conditions	4-27
Table 4-2. Interrupt Acknowledge Summary	4-40
Table 4-3. Interrupt Level Code Assignment	4-43
Table 4-4. Output Signal States After DMA Request is Acknowledged	4-49
Table 4-5. Output States on Reset	4-51

Table 5-1. Register Set	5-2
Table 5-2. Addressing Modes	5-6
Table 5-3. Addressing Modes by Type	5-8
Table 5-4. Options for Type in Expanded-Operand Mode	5-19
Table 5-5. Auto Pre/Post Increment/Decrement Addressing Modes	5-20
Table 5-6. Indexed Register Addressing Modes	5-22
Table 5-7. Format 1 Modes for r16—r31	5-26
Table 5-8. Condition Flag Code Assignments	5-27
Table 5-9. Data Transfer Instruction Group	5-29
Table 5-10. Arithmetic Instruction Group	5-30
Table 5-11. BCD Instruction Group	5-32
Table 5-12. Logical Instruction Group	5-33
Table 5-13. Program Control Instruction Group	5-35
Table 5-14. Coprocessor Instruction Group	5-41
Table 5-15. Stack Miscellaneous Instruction Group	5-41
Table 5-16. Assembly Language Operators and Symbols	5-43
Table 5-17. Instruction Set Summary by Mnemonic	5-44
Table 5-18. Instruction Set Summary by Opcode	5-50
Table 6-1. Operating System Instructions	6-2
Table 6-2. Processor Status Word Fields	6-13
Table 6-3. Severity Levels for Exceptions	6-38
Table 6-4. Normal Exception	6-41
Table 6-5. Stack Exceptions	6-43
Table 6-6. Process Exceptions	6-43
Table 6-7. Reset Exceptions	6-44
Table 6-8. Option Bits for Microsequences	6-94

Chapter 1

Introduction

CHAPTER 1. INTRODUCTION

CONTENTS

1. INTRODUCTION.....	1-1
1.1 DEVELOPMENT.....	1-2
1.2 ARCHITECTURE.....	1-2
1.3 INSTRUCTION SET.....	1-3
1.4 OPERATING SYSTEM SUPPORT.....	1-3
1.5 CHAPTER SUMMARIES.....	1-4
1.5.1 Chapter 2. Registers, Data, and Instruction Formats.....	1-4
1.5.2 Chapter 3. Signal Descriptions.....	1-4
1.5.3 Chapter 4. Bus Operations.....	1-4
1.5.4 Chapter 5. Instruction Set and Addressing Modes.....	1-4
1.5.5 Chapter 6. Operating System Considerations.....	1-4

1. INTRODUCTION

The *WE 32200* Microprocessor (CPU) is a high-performance, single-chip, 32-bit central processing unit designed for efficient operation in a high-level language environment. This CPU represents a state-of-the-art concept in microprocessor architecture, providing one of the most powerful and extensive instruction sets available with any microprocessor.

The *WE 32200* Microprocessor is implemented in 1-micron CMOS technology and is packaged in a 133-pin square, ceramic pin grid array. It is available in 24-MHz and higher frequency versions.

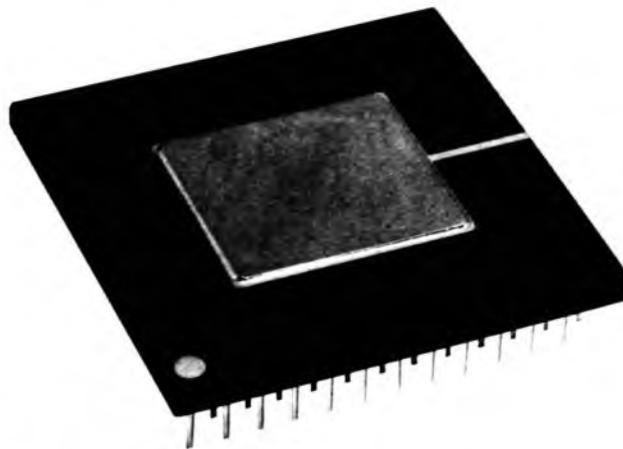


Figure 1-1. The WE[®] 32200 Microprocessor

The system memory space is addressed over a full 32-bit address bus by using either physical or virtual addresses. The 32-bit address bus produces a memory space of more than four billion bytes (4 Gbytes), which increases the flexibility of memory organization and provides ample space for program and data storage. Information can be read or written over the separate 32-bit data bus in byte (8-bit), halfword (16-bit), or word (32-bit) lengths (which can be aligned on any byte boundary via arbitrary byte alignment). In addition, dynamic bus sizing allows the *WE 32200* Microprocessor to communicate with 16- and 32-bit memories in the same system.

The *WE 32200* Microprocessor is an efficient execution vehicle for operating systems and high-level languages. Extensive addressing modes result in a symmetric, versatile, and powerful instruction set. The instructions included for the operating system establish an environment that permits process switching and interrupt handling with a minimum of operating system support. Other instructions allow the use of coprocessors and provide the necessary signals for interfacing with the *WE 32201* Memory Management Unit (MMU) for virtual memory systems.

INTRODUCTION

Chapter Summaries

A discussion of the techniques for efficient operating system design using the *WE 32200* Microprocessor and the use of the *WE 32201* Memory Management Unit (MMU) in a virtual memory operating system is provided in Chapter 6.

1.5 CHAPTER SUMMARIES

This section provides a brief summary of the remaining chapters in this manual. For a detailed description of the electrical, clock, timing, and thermal requirements refer to the *WE 32200* Microprocessor data sheet.

1.5.1 Chapter 2. Registers, Data, and Instruction Formats

Chapter 2 describes in detail the thirty-two 32-bit registers (r0—r31) of the *WE 32200* Microprocessor. Also described are the data types supported by the microprocessor (i.e., byte, halfword, word, and bit field), and how they are stored in or accessed from memory. This includes aligned and nonaligned data (i.e., arbitrary byte alignment) and dynamic bus sizing. A brief description of the microprocessor instruction set containing information about the syntax of the instructions is also included.

1.5.2 Chapter 3. Signal Descriptions

The *WE 32200* Microprocessor input and output signals are described in this chapter.

1.5.3 Chapter 4. Bus Operations

This chapter covers the bus protocol for the *WE 32200* Microprocessor. Among the topics discussed are signal sampling points, read and write operations, read-interlocked operation, blockfetch operation, bus exceptions, blockfetch special cases, interrupts, bus arbitration, reset, aborted memory accesses, single-step operation, and coprocessor operations. Additional protocol diagrams are also provided.

1.5.4 Chapter 5. Instruction Set and Addressing Modes

Chapter 5 describes the *WE 32200* Microprocessor instruction set. A listing of the syntax, opcodes, addressing modes, condition flags, exceptions, and examples are provided for each instruction.

1.5.5 Chapter 6. Operating System Considerations

This chapter presents the operating system considerations important for the system designer to make full use of the power of the *WE 32200* Microprocessor as an execution vehicle for today's efficient process-oriented operating systems.

Chapter 2

**Registers,
Data,
and
Instruction Formats**

CHAPTER 2. REGISTERS, DATA, AND INSTRUCTION FORMATS

CONTENTS

2. REGISTERS, DATA, AND INSTRUCTION FORMATS	2-1
2.1 USER REGISTERS	2-1
2.2 CONVENTIONAL REGISTER SET	2-4
2.2.1 Stack Pointer (SP)	2-4
2.2.2 Program Counter (PC)	2-5
2.3 HIGH-LEVEL LANGUAGE SUPPORT GROUP	2-5
2.3.1 Frame Pointer (FP)	2-5
2.3.2 Argument Pointer (AP)	2-6
2.4 OPERATING SYSTEM SUPPORT GROUP	2-6
2.4.1 Processor Status Word (PSW)	2-6
2.4.2 Process Control Block Pointer (PCBP)	2-6
2.4.3 Interrupt Stack Pointer (ISP)	2-6
2.5 DATA TYPES	2-7
Bit Fields	2-9
2.5.1 Floating-Point Data	2-10
2.6 SIGN AND ZERO EXTENSION	2-11
2.7 DATA STORAGE IN MEMORY	2-12
2.8 ARBITRARY BYTE ALIGNMENT	2-13
2.9 DYNAMIC BUS SIZING	2-16
2.10 REGISTER DATA STORAGE	2-18
2.11 INSTRUCTION SET	2-19
2.12 INSTRUCTION STORAGE IN MEMORY	2-19
2.13 CONDITION FLAGS	2-21
2.14 TRACE MECHANISM	2-23

2. REGISTERS, DATA, AND INSTRUCTION FORMATS

The *WE 32200* Microprocessor has separate 32-bit address and data buses. Using either physical or virtual addresses, the 32-bit address bus can access 4 Gbytes (2^{32} bytes) of system memory or peripherals. Data is read or written over the 32-bit bidirectional data bus in either byte (8-bit), halfword (16-bit), or word (32-bit) lengths and is processed internally over 32-bit internal data paths.

A block diagram of the *WE 32200* Microprocessor illustrating its four major sections – main controller, fetch unit, execution unit, and bus interface control – is shown on Figure 2-1.

The main controller is responsible for directing the actions of the fetch and execute controllers as instructions are executed.

The fetch unit is responsible for fetching all instructions and data. Although the operation of this unit is transparent to the microprocessor user, it contains unique features that significantly enhance the performance of the *WE 32200* Microprocessor. One of these features is a 64-word instruction cache that stores prefetched instructions from memory. The prefetched instructions are read from memory at the same time instructions are executed, a technique known as pipelining. Thus, the normal suspension of execution, while the processor waits for an instruction to be fetched, is avoided since the next instruction is available in the cache.

The execution unit provides all of the user-accessible features of the microprocessor. This unit performs all arithmetic, logical, data-movement, and program control instructions. Contained in the execution unit are the thirty-two 32-bit user-accessible registers, consisting of seventeen general-purpose (r0—r8 and r16—r23), seven special-purpose (r9—r15), and eight kernel-privileged (r24—r31) registers.

2.1 USER REGISTERS

Figure 2-2 shows the programming model for the microprocessor's thirty-two 32-bit registers (r0—r31). This register set is designed for efficient support of high-level language program execution. All of these registers, except the processor status word (r11), process control block pointer (r13), interrupt stack pointer (r14), program counter (r15), and the upper eight general-purpose registers (r24—r31), may be accessed in any addressing mode. The processor status word, process control block pointer, interrupt stack pointer, and the upper eight general-purpose registers are privileged registers. These may be read at any time, but may be written only when the microprocessor is in kernel mode (i.e., the operating system is in control). The other registers may be read or written in any of the four execution levels.

The *WE 32200* Microprocessor registers support an extremely powerful assembly language. These registers were also designed for the efficient support of procedure-oriented high-level languages, such a C, and process-oriented operating systems, such as the *UNIX* Operating System.

Although all of the thirty-two registers are available to assembly language programmers, it is useful to separate the registers set into three groups: the conventional register set, the high-level language registers, and the operating system registers. These groups are illustrated on Figure 2-2.

REGISTERS, DATA, AND INSTRUCTION FORMATS

User Registers

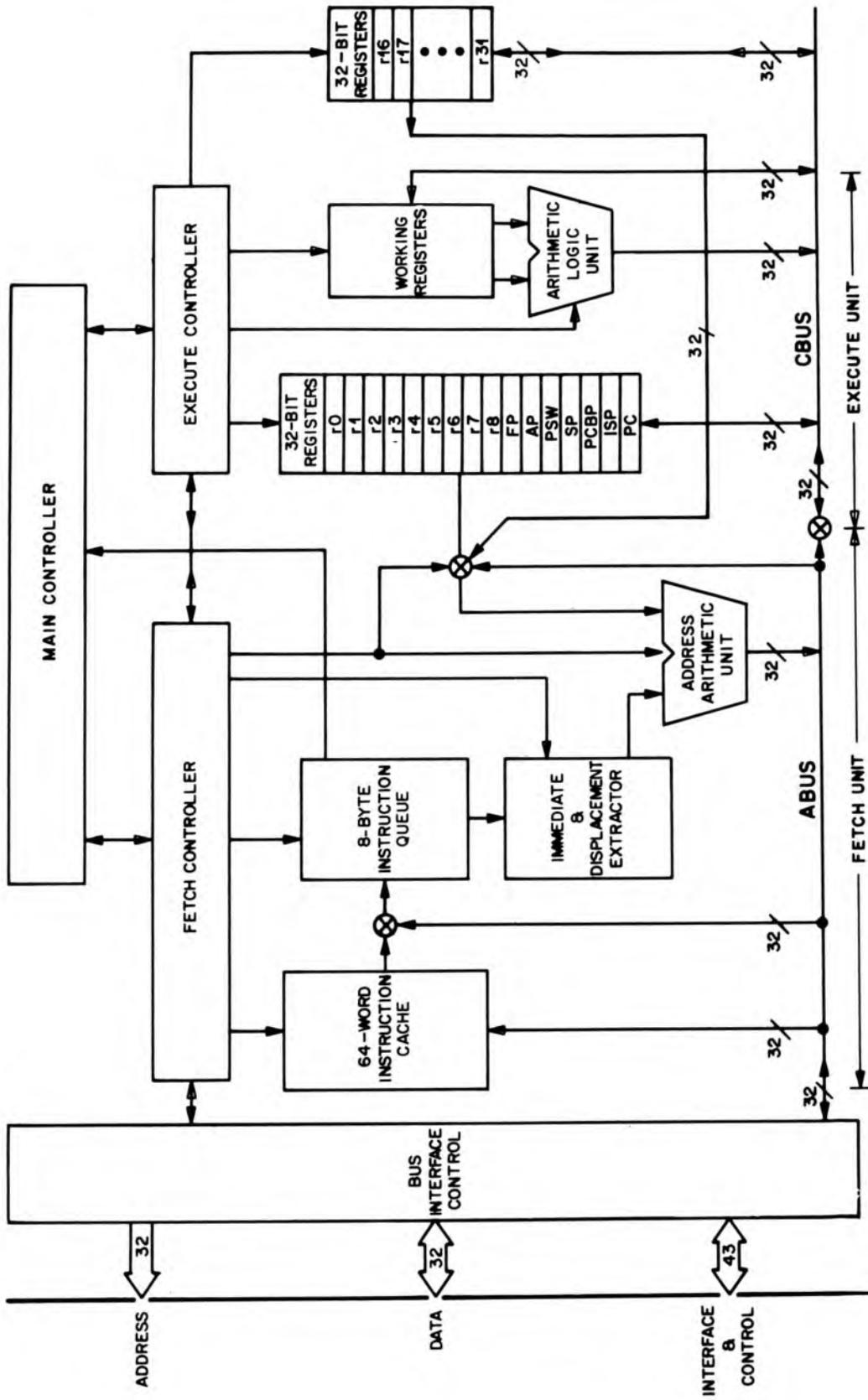


Figure 2-1. WE® 32200 Microprocessor Block Diagram

REGISTERS, DATA, AND INSTRUCTION FORMATS

User Registers

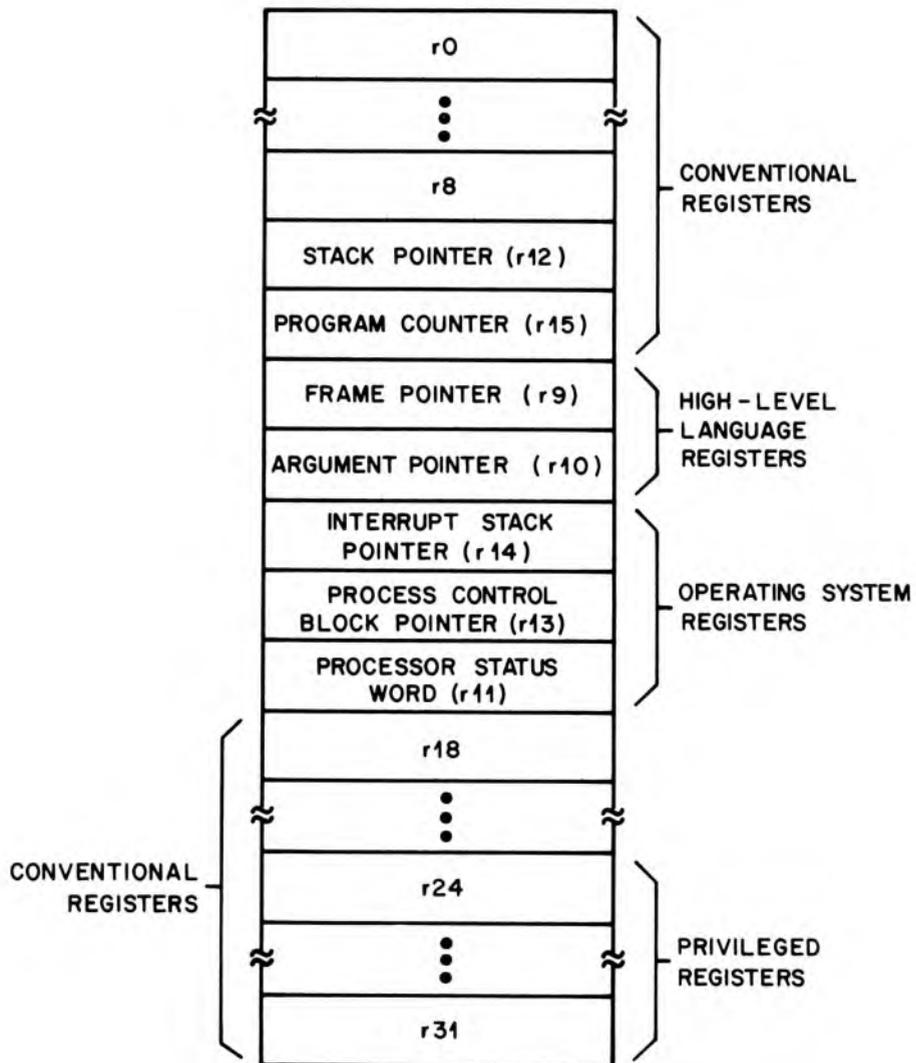


Figure 2-2. Programmer's Model for User Registers

REGISTERS, DATA, AND INSTRUCTION FORMATS

Conventional Register Set

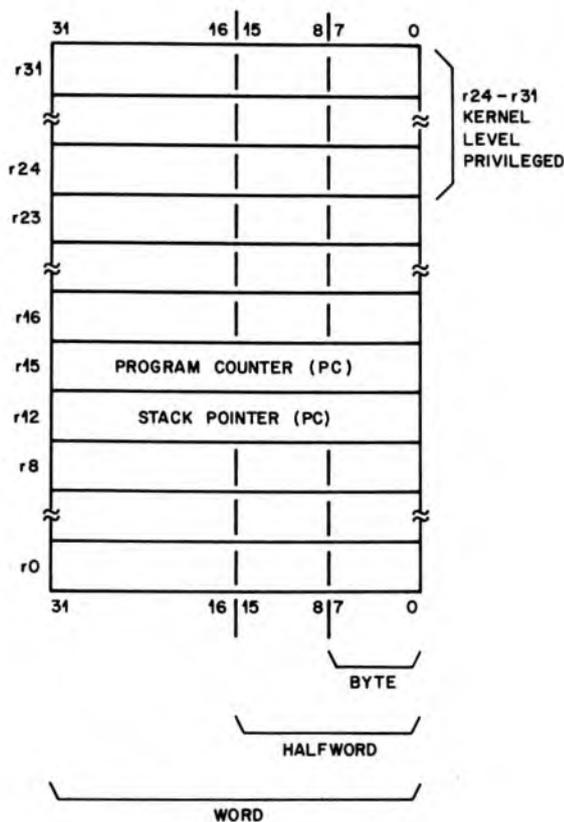


Figure 2-3. Conventional Register Set

2.2 CONVENTIONAL REGISTER SET

The conventional registers, shown on Figure 2-3, consist of the twenty-five general-purpose registers, the stack pointer, and the program counter. This set, including condition flags, is typically associated with an assembly language programming model of a microprocessor. In the WE 32200 Microprocessor the condition flags, indicators of the processor's current status, are contained within the processor status word register.

The twenty-five general-purpose registers, r0 through r8 and r16 through r31, can be used with all arithmetic, data transfer, logical, and program control assembly instructions. Additionally, registers r0, r1, and r2 are used in both string manipulation and transfer instructions and, by convention, for returning values from a called C-language program. The string manipulation and transfer instructions using registers r0, r1, and r2 include the block move (MOVBLW), string copy (STRCPY), and string end (STREND) instructions (see Chapter 5).

2.2.1 Stack Pointer (SP)

The stack pointer, r12, contains the 32-bit address of the top of the current execution stack. As illustrated on Figure 2-4, the stack pointer points to the next available

memory location. A PUSH instruction immediately stores its operand at the current memory address contained in the stack pointer. The stack pointer is then incremented by the size of the pushed operand. Thus, the stack "grows" into increasing memory address space. A POP instruction first decrements the stack pointer by the size of the POPed operand (to point to the last pushed operand) and then fetches the data from the top of the stack.

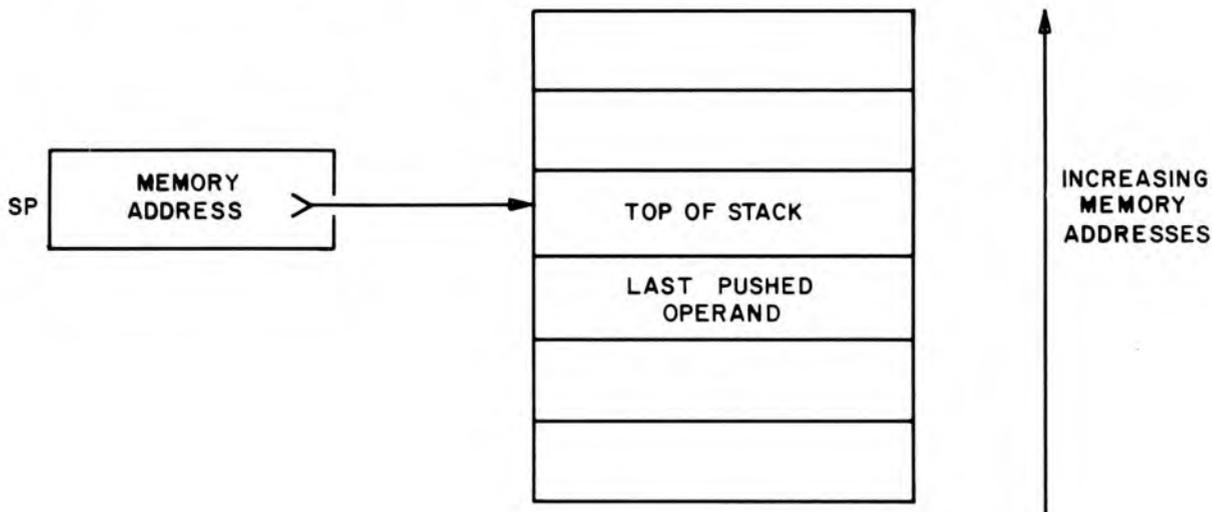


Figure 2-4. The WE[®] 32200 Microprocessor Stack

2.2.2 Program Counter (PC)

The program counter, r15, contains the 32-bit memory address of the currently executing instruction or, on completion, the starting address of the next instruction. The PC is referenced by all program control instructions, including all function calls and returns.

2.3 HIGH-LEVEL LANGUAGE SUPPORT GROUP

The frame pointer and argument pointer constitute the high-level language support group register set shown on Figure 2-5. Although these two registers may be accessed and used as general-purpose assembler registers, they are typically used in association with registers r0, r1, and r2 for passing, holding, and returning high-level language variables and arguments.

2.3.1 Frame Pointer (FP)

The frame pointer, r9, points to the beginning stack location in which the local variables of the currently running program, procedure, or function are stored. The

REGISTERS, DATA, AND INSTRUCTION FORMATS

Argument Pointer (AP)

frame pointer is implicitly changed by the save (SAVE) and restore (RESTORE) registers assembly instructions.

2.3.2 Argument Pointer (AP)

The argument pointer, r10, points to the beginning stack location in which arguments passed into the currently running program, procedure, or function have been pushed. The AP is implicitly affected by procedure call (CALL) and return (RET) assembly instructions.

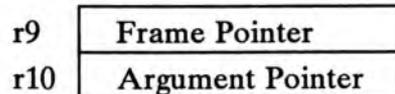


Figure 2-5. High-Level Language Register Support Group

2.4 OPERATING SYSTEM SUPPORT GROUP

The processor status word, process control block pointer, and interrupt stack pointer were designed to facilitate an efficient operating system interface. These three registers, therefore, are referred to as the operating system support group.

2.4.1 Processor Status Word (PSW)

The processor status word, r11, contains status information about the microprocessor and the current process. Additionally, the PSW contains five condition code flags (see section 2.13) used by assembly language transfer-of-control instructions. In general, the PSW changes as a whole only when a process switch occurs, and can be written only by the operating system (i.e., kernel mode).

2.4.2 Process Control Block Pointer (PCBP)

The process control block pointer, r13, points to the starting address of the process control block for the current process. The process control block is a data structure in external memory containing the hardware context of a process when the process is not running. This context consists of the initial and current contents of the PSW, PC, and SP; the last contents of registers r0 through r10 (in some cases, also r16 through r23); boundaries for an execution stack; and block move specifications (and possible memory specifications) for the process. The PCBP may be written only when the microprocessor is in the kernel mode (see Chapter 6).

2.4.3 Interrupt Stack Pointer (ISP)

The interrupt stack pointer, r14, contains the 32-bit memory address of the top of the interrupt stack. This stack is used when an interrupt request is received. In addition, the stack is used by the call process (CALLPS), user call process (UCALLPS), return to process (RETPS), and return from quick interrupt (RETQINT) instructions. The ISP may be written only when the microprocessor is in kernel mode.

r11	Processor Status Word
r13	Process Control Block Pointer
r14	Interrupt Stack Pointer

Figure 2-6. Operating System Register Support Group

2.5 DATA TYPES

The data types supported by the *WE 32200* Microprocessor are byte, halfword, word, binary-coded decimal (BCD), and bit field data. Support for floating-point data types (single, double, double-extended, and decimal) is provided only when the *WE 32206* Math Acceleration Unit (MAU) is used with the *WE 32200* Microprocessor. The instruction set provides that bytes, halfwords, and words can be interpreted as either signed or unsigned quantities.

A byte is an 8-bit quantity that may appear at any address. Bits are numbered from right to left within a byte, starting with zero, the least significant bit (LSB), and ending with 7, the most significant bit (MSB), as illustrated on Figure 2-7.

A halfword is a 16-bit quantity that may appear at any address if the arbitrary byte alignment enabled (EA) bit of the PSW is set. If the EA bit of the PSW is not set (arbitrary byte alignment disabled), then the halfword must be halfword aligned. An address is halfword aligned if it is divisible by 2. Bits are numbered from right to left starting with zero, the LSB, and ending with 15, the MSB, as illustrated on Figure 2-8. The address of a halfword is the address of its most significant byte. Accesses to halfwords that are aligned on an address divisible by two are more efficient than unaligned accesses.

A word is a 32-bit quantity that may appear at any address, providing the EA bit of the PSW is set. If the EA bit is not set (arbitrary byte alignment disabled), then the word must be word aligned. An address is word aligned if it is divisible by 4. Bits are numbered from right to left starting with zero, the LSB, and ending with 31, the MSB, as illustrated on Figure 2-9. The address of a word is the address of its most significant byte and halfword. Accesses to words that are aligned on an address divisible by four are more efficient than unaligned accesses.

A packed BCD operand is a byte containing two binary-coded decimal BCD digits. A BCD digit is a 4-bit field, whose value is within the range of 0 to 9, encoded as binary 0000 to 1001, respectively. Figure 2-10 illustrates a packed BCD byte.

An unpacked BCD operand is a halfword containing two encoded BCD digits; the lower nibble of each byte is a BCD digit. Figure 2-11 shows the unpacked form of the packed BCD byte shown on Figure 2-10 in ASCII encoding (i.e., the value 3 is used in the upper nibbles).

REGISTERS, DATA, AND INSTRUCTION FORMATS

Data Types

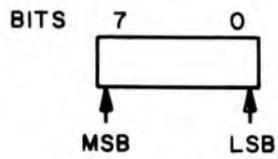


Figure 2-7. Byte Data

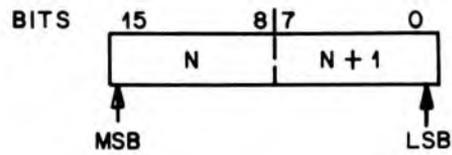


Figure 2-8. Halfword Data

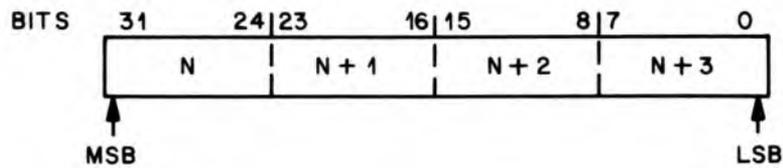


Figure 2-9. Word Data

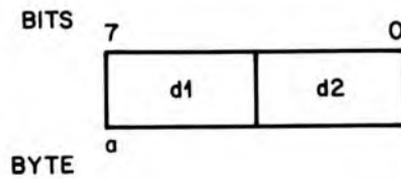


Figure 2-10. Packed BCD Byte

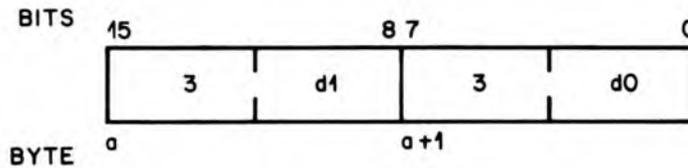


Figure 2-11. Unpacked BCD Bytes

Bytes, halfwords, and words can be interpreted as either signed or unsigned quantities for the purposes of arithmetic and logical operations. If arbitrary byte alignment is disabled and a word or halfword data is specified at an improper address, the WE 32200 Microprocessor generates an external memory fault internally.

Packed BCD numbers are stored as unsigned byte quantities. Unpacked BCD numbers are stored as unsigned halfword quantities.

Bit Fields

A bit field is a sequence of 1 to 32 bits extracted from byte, halfword, or word data. The bit field is specified by the address of the data containing the field, an offset, and a width. The offset (from 0 to 31) identifies the starting bit in the word containing the bit field. This bit becomes the LSB of the selected field. The width (also a number from 0 to 31) specifies the size of the field. The number of bits in the extracted field is one more than the width value. Figure 2-12 illustrates a bit field extracted from a word using an offset of six and a width of nine. Notice that the extracted field contains ten bits, one more than the width.

Bit fields do not extend across word boundaries. If the selected width requires bits beyond the MSB of the word being used, the extraction of bits continues by wrapping around to the LSB.

REGISTERS, DATA, AND INSTRUCTION FORMATS

Floating-Point Data

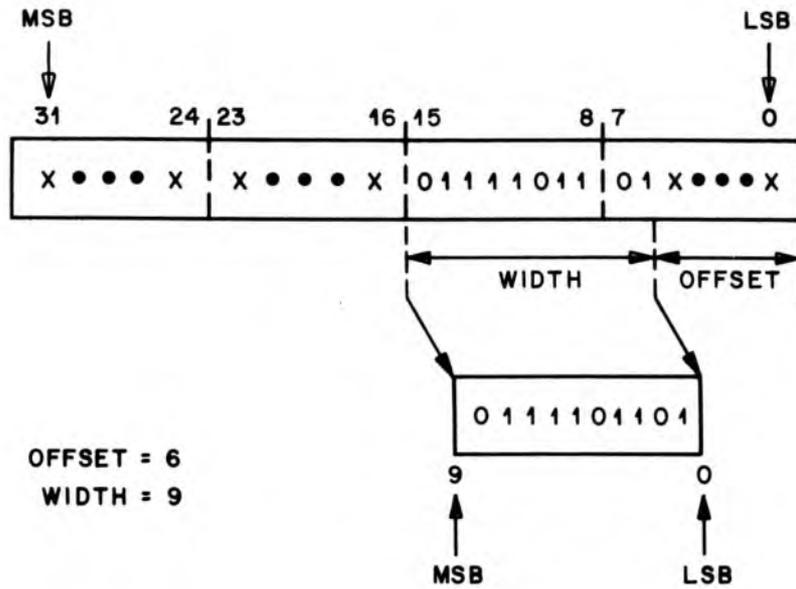


Figure 2-12. Extraction of a Bit Field

2.5.1 Floating-Point Data

Floating-point data types can appear at any address in memory divisible by four. Figure 2-13 illustrates the floating-point data types supported when the WE 32206 Math Acceleration Unit (MAU) is part of the system. For more information on floating-point data types, refer to the WE® 32206 Math Acceleration Unit Information Manual.

REGISTERS, DATA, AND INSTRUCTION FORMATS
Sign and Zero Extension

Bit	31	30 23	22 0
Field	Sign	Exponent	Fraction

A. Single Precision Floating-Point Data Type

Bit	63	62 52	51 0
Field	Sign	Exponent	Fraction

B. Double Precision Floating-Point Data Type

Bit	95 80	79	78 64	63	62 0
Field	Unused	Sign	Exponent	J	Fraction

C. Double-Extended Floating-Point Data Type

Bit	95 76	75 72	71 8	7 4	3 0
Field	Unused	d17	—	d0	Sign

D. Decimal Data Type

Figure 2-13. Floating-Point Data Types

2.6 SIGN AND ZERO EXTENSION

All CPU operations are performed on 32-bit quantities even though an instruction may specify a byte or halfword operand. The WE 32200 Microprocessor reads in the correct number of bits for the operand and extends the data automatically to 32 bits. It uses sign extension when reading signed bytes or halfwords and zero extension when reading unsigned bytes or halfwords (or bit fields that contain less than 32 bits). The data type of the source operand determines how many bits are fetched and what type of extension is applied. The type of extension applied can be changed by using the expanded-operand type mode described in Expanded-Operand Type Mode under section 5.2.1. For sign extension, the value of the MSB is replicated to fill the

REGISTERS, DATA, AND INSTRUCTION FORMATS

Data Storage in Memory

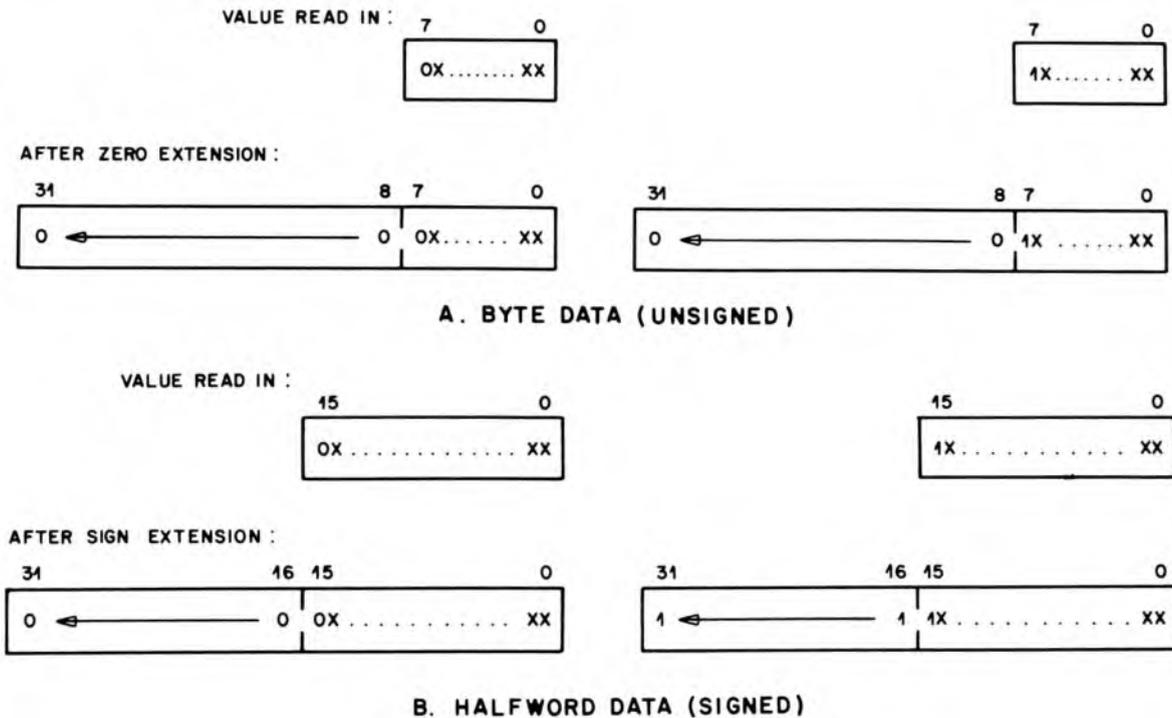


Figure 2-14. Extending Data to 32 Bits

high-order bits to form a 32-bit value. In zero extension, zeros fill the high-order bits. The microprocessor automatically extends a byte or halfword to 32 bits before performing an operation. Figure 2-14 illustrates sign and zero extension.

An arithmetic, logical, data transfer, or bit field operation always yields an intermediate result that is 32 bits in length. If the result is to be stored in a register, the processor writes all 32 bits to that register. The processor automatically strips any surplus high-order bits from a result when writing bytes or halfwords to memory.

2.7 DATA STORAGE IN MEMORY

Memory locations consist of a series of 8-bit (byte) locations for storing data. Halfwords occupy two consecutive memory locations and words occupy four consecutive memory locations. See section 2.5 for information concerning boundary restrictions that apply to the starting locations of halfwords and words. The microprocessor generates a fault if these boundaries are violated when arbitrary byte alignment is turned off (EA bit is not set).

Figure 2-15 illustrates the storage of word data in memory. As illustrated, the hexadecimal word, 0x12345678, is stored with the lower-order bytes at higher-order addresses. All data stored in memory follows this format. For example, the hexadecimal halfword data, 0xEEFF, would be stored in memory with the lower-order byte, 0xFF, at the next higher-byte address than the location containing the byte, 0xEE.

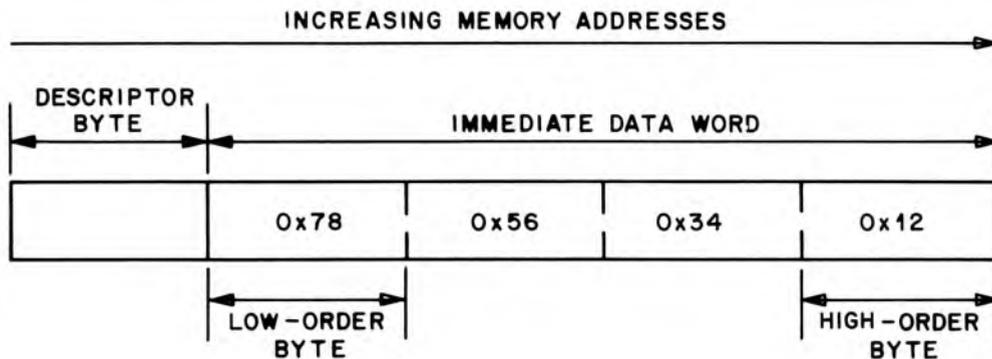


Figure 2-15. Word Storage in Memory

2.8 ARBITRARY BYTE ALIGNMENT

The arbitrary byte alignment feature enables the microprocessor to handle nonaligned memory accesses efficiently for both reads and writes. The microprocessor automatically generates multiple accesses when nonaligned data types cross word boundaries. In illustrating how this is accomplished (Tables 2-1 — 2-6), the following notation is used:

- n = most significant 30 bits of the address
- x = second LSB of the address
- y = LSB of the address

Note that $n01$ stands for a 32-bit address with the two LSBs being 01 ($x = 0, y = 1$), and $(n+1)xy$ stands for address $nxy + 100$ (binary arithmetic).

For example, reading a word (four 8-bit bytes) from memory location $n01$ would cross a word boundary (see Table 2-1). This requires two read transaction cycles: the first a 3-byte read from location $n01$; the second, a byte read from memory location $(n+1)00$.

Similar cases exist for data writes (see Table 2-2). For virtual memory mode, probes occur on nonaligned data writes to guarantee restartability during memory faults. In physical memory mode, no probes exist and data writes are the same as read transactions. On byte and aligned halfword 32-bit bus writes (halfword aligned), bytes are replicated and output onto the 32-bit data bus as shown in Table 2-3. In Table 2-3, all active bytes are identified as A_x , where x is used to differentiate active bytes. All replicated bytes are identified as R_x , where x indicates that the active byte A_x is being replicated. For any transfer, the address associated with the access is the

REGISTERS, DATA, AND INSTRUCTION FORMATS
Arbitrary Byte Alignment

address of the most significant active byte. XXX in the replication pattern field indicates that the byte may be driven on writes but the value is undefined. XX in the operand address field indicates a "don't care" condition.

Table 2-1. Read Accesses for Aligned and Nonaligned Data (32-Bit Port)

Operand Address	Operand Size	Transaction Number	Transaction Address	Transaction Data Size
n00	word	1	n00	word
n01	word	1 2	n01 (n+1)00	3 bytes byte
n10	word	1 2	n10 (n+1)00	halfword halfword
n11	word	1 2	n11 (n+1)00	byte 3 bytes
n00	halfword	1	n00	halfword
n01	halfword	1	n01	halfword
n10	halfword	1	n10	halfword
n11	halfword	1 2	n11 (n+1)00	byte byte
n00	byte	1	n00	byte
n01	byte	1	n01	byte
n10	byte	1	n10	byte
n11	byte	1	n11	byte

REGISTERS, DATA, AND INSTRUCTION FORMATS
Arbitrary Byte Alignment

Table 2-2. Write Accesses for Aligned and Nonaligned Data (32-Bit Port)

Operand Address	Operand Size	Transaction Number	Transaction Address	Transaction Data Size
n00	word	1	n00	word
n01	word	1	n01	0 byte (probe)
		2	(n+1)00	byte
		3	n01	3 bytes
n10	word	1	n10	0 byte (probe)
		2	(n+1)00	halfword
		3	n10	halfword
n11	word	1	n11	0 byte (probe)
		2	(n+1)00	3 bytes
		3	n11	byte
n00	halfword	1	n00	halfword
n01	halfword	1	n01	halfword
n10	halfword	1	n10	halfword
n11	halfword	1	n11	0 byte (probe)
		2	(n+1)00	byte
		3	n11	byte
n00	byte	1	n00	byte
n01	byte	1	n01	byte
n10	byte	1	n10	byte
n11	byte	1	n11	byte

Table 2-3. Replication Patterns for Write Accesses

Operand Address	DSIZE	Replication Pattern			
		D31—D24	D23—D16	D15—D8	D7—D0
XX	0 byte	XXX	XXX	XXX	XXX
n00	byte	A1	R1	R1	R1
n01	byte	R1	A1	R1	R1
n10	byte	R1	R1	A1	R1
n11	byte	R1	R1	R1	A1
n00	halfword	A1	A2	R1	R2
n01	halfword	XXX	A1	A2	XXX
n10	halfword	R1	R2	A1	A2
n00	3 bytes	A1	A2	A3	XXX
n01	3 bytes	XXX	A1	A2	A3
n00	word	A1	A2	A3	A4

REGISTERS, DATA, AND INSTRUCTION FORMATS
Dynamic Bus Sizing

2.9 DYNAMIC BUS SIZING

Dynamic bus sizing allows communication with both 16- and 32-bit memories and peripherals. For example, if the CPU is trying to read a word of data from memory and encounters a 16-bit port acknowledge (i.e., the CPU received only half the word), it automatically generates a second access to read the other half of the word.

Dynamic bus sizing also works with data writes and instruction fetches. The 16-bit port is defined as the upper half of the data bus. The CPU always initially assumes that it is communicating with a 32-bit port. When a 16-bit port is accessed, the CPU generates additional memory accesses if the size of the data is word, three bytes, or halfword, and the data resides in the middle of a word boundary. A very useful application of dynamic bus sizing is in boot ROMs. A 16-bit boot ROM can be substituted for a 32-bit boot ROM; reducing the board space requirements. Tables 2-4 and 2-5 show the read and write accesses for 16-bit aligned and nonaligned data. Table 2-6 shows the 16-bit instruction fetches.

Table 2-4. Read Accesses for Aligned and Nonaligned Data (16-Bit Port)				
Operand Address	Operand Size	Transaction Number	Transaction Address	Transaction Data Size
n00	word	1	n00	halfword
		2	n10	halfword
n01	word	1	n01	byte
		2	n10	halfword
		3	(n+1)00	byte
n10	word	1	n10	halfword
		2	(n+1)00	halfword
n11	word	1	n11	byte
		2	(n+1)00	halfword
		3	(n+1)10	byte
n00	halfword	1	n00	halfword
n01	halfword	1	n01	byte
		2	n10	byte
n10	halfword	1	n10	halfword
n11	halfword	1	n11	byte
		2	(n+1)00	byte
n00	byte	1	n00	byte
n01	byte	1	n01	byte
n10	byte	1	n10	byte
n11	byte	1	n11	byte

Table 2-5. Write Accesses for Aligned and Nonaligned Data (16-Bit Port)				
Operand Address	Operand Size	Transaction Number	Transaction Address	Transaction Data Size
n00	word	1	n00	halfword
		2	n10	halfword
n01	word	1	n01	0 byte (probe)
		2	(n+1)00	byte
		3	n01	byte
		4	n10	halfword
n10	word	1	n10	0 byte (probe)
		2	(n+1)00	halfword
		3	n10	halfword
n11	word	1	n11	0 byte (probe)
		2	(n+1)00	halfword
		3	(n+1)10	byte
		4	n11	byte
n00	halfword	1	n00	halfword
n01	halfword	1	n01	byte
		2	n10	byte
n10	halfword	1	n10	halfword
n11	halfword	1	n11	0 byte (probe)
		2	(n+1)00	byte
		3	n11	byte
n00	byte	1	n00	byte
n01	byte	1	n01	byte
n10	byte	1	n10	byte
n11	byte	1	n11	byte

REGISTERS, DATA, AND INSTRUCTION FORMATS

Register Data Storage

Table 2-6. 16-bit Instruction Fetches

Instruction Address	Instruction Size	Transaction Number	Transaction Address	Transaction Data Size	Bytes Transferred
n00	double-word	1	n00	double-word	2
		2	n10	halfword	2
n01	double-word	1	n01	double-word	2
		2	n11	halfword	2
n10	double-word	1	n10	double-word	2
		2	n00	halfword	2
n11	double-word	1	n11	double-word	2
		2	n01	halfword	2
n00	word	1	n00	word	2
		2	n10	halfword	2
n01	word	1	n01	word	2
		2	n11	halfword	2
n10	word	1	n10	word	2
		2	n00	halfword	2
n11	word	1	n11	word	2
		2	n01	halfword	2

2.10 REGISTER DATA STORAGE

All data stored in a register is a full 32 bits, regardless of the instruction or data type. For all CPU operations, including register storage, the *WE 32200* Microprocessor reads in the correct number of bits for the operand and extends the data automatically to 32 bits. Halfword operands, assumed to be signed data, are sign extended to 32 bits. Byte operands are assumed to be unsigned, and zero extension is used.

Intermediate results of all operations in the CPU are always 32 bits. If the results of an operation are stored in a register, the processor writes all 32 bits to the register.

When a register is specified as the source of a byte operand, the low-order 8 bits (bits 0—7) of the register are fetched and zero extended to 32 bits. The zero extension may be changed to a sign extension by using an expanded operand type addressing mode (this addressing mode is described in Chapter 5). When a register is used as the source of a halfword operand, the low-order 16 bits (bits 0—15) of the register are fetched and sign extended to 32 bits. Again, the extension may be changed to zero by using an expanded operand type addressing mode.

2.11 INSTRUCTION SET

The *WE 32200* Microprocessor has a powerful instruction set, described in Chapter 5, that includes the standard data transfer, arithmetic, and logical operations for microprocessors, and some unique operating system operations. Its program control instructions (branch, jump, return) provide flexibility for altering the sequence in which instructions are executed. Some of these instructions check the setting of the processor's condition flags before execution. For operating systems, the processor has instructions to establish an environment that permits other processes to take control of the CPU. The special instructions dedicated to operating system use are discussed in Chapter 6.

The microprocessor instructions are mnemonic-based assembly language statements. A C-language compiler is also available for those wishing to write programs for the *WE 32200* Microprocessor in a high-level language.

An instruction consists of a one- or two-byte opcode followed by zero or up to four operands. In assembly language, the mnemonic replaces the opcode and is followed by its operands. This is represented as

mnemonic opnd1,opnd2,opnd3,opnd4

where the mnemonic is separated from the operands by a white space and commas are used to separate operands.

2.12 INSTRUCTION STORAGE IN MEMORY

Instructions may appear at any byte address in memory and are stored as a one- or two-byte opcode followed by up to four operands. Figure 2-16 illustrates the general format of an assembly instruction as it is stored in memory. Each individual operand shown on Figure 2-16 consists of a descriptor byte, followed by up to four bytes of data (see Figure 2-17).

The descriptor byte defines an operand's addressing mode and register fields, which are discussed in Chapter 5. Immediate data stored within an instruction is stored with lower-order bytes located at lower-order addresses. For example, the hexadecimal value, 0x12345678, would be stored within an instruction as illustrated on Figure 2-18.

REGISTERS, DATA, AND INSTRUCTION FORMATS

Instruction Storage in Memory

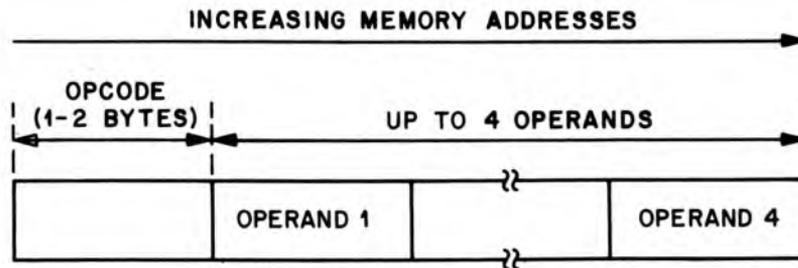


Figure 2-16. Instruction Storage in Memory

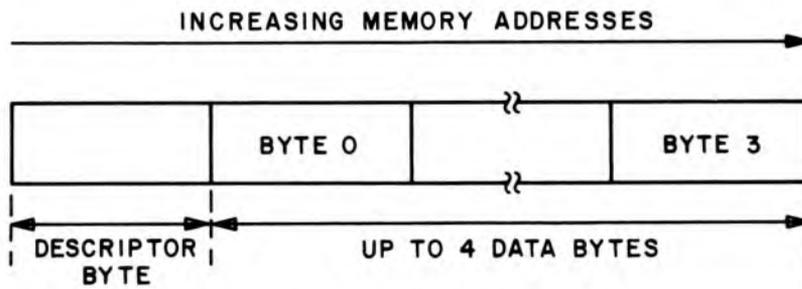


Figure 2-17. Operand Format

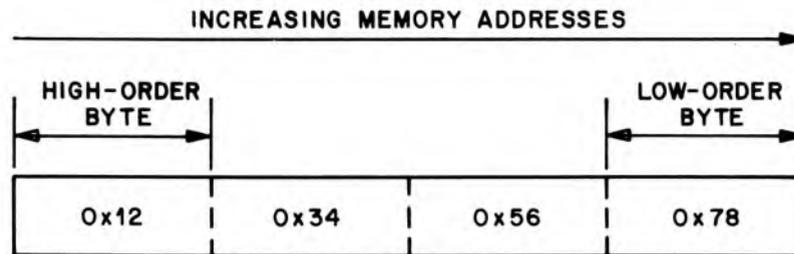


Figure 2-18. Word Storage Within an Instruction

Notice that data storage within an instruction, as shown on Figure 2-18, is the reverse of the data storage within a memory location shown on Figure 2-15.

2.13 CONDITION FLAGS

Bits 18 to 21, and 26 of the processor status word, r11, contain five condition flags (X, N, Z, V, and C) that are altered by most instructions. The arrangement of these bits in the register is shown on Figure 2-19. The conditional program-control instructions check one or more of these flags before executing the branch, jump, or return instructions. In general, these flags reflect the result of the most recently executed instruction. Most instructions alter the flags according to standard criteria. The following terms are used in the discussion of those criteria.

- **Result** – refers to the internal result of the operation as if it were performed in an infinite-precision machine. The microprocessor operates on 32-bit data internally and uses a 33-bit space for the internal result. Bytes and halfwords read in are extended to 32 bits before the operation. The destination operand determines the type (signed or unsigned) and size (byte, halfword, or word) of this result.
- **Output Value** – refers to the data written to the destination location. The size of this data, 8-, 16-, or 32-bits, corresponds to the data type of the destination operand: byte, halfword, or word, respectively.
- **Requested Representation** – refers to the actual data written into the desired location. This depends on the size requested (byte, halfword, or word) and whether it is signed or unsigned.

The following conditions cause the appropriate flag bit to be altered:

- X **Extended Carry/Borrow (PSW bit 26)** – The extended carry/borrow bit can be set and cleared only by BCD instructions. The bit is set if there is a carry or borrow from a BCD arithmetic operation on two BCD digits. The bit is cleared if there is no carry or borrow from such BCD operations. It can also be set and

REGISTERS, DATA, AND INSTRUCTION FORMATS

Condition Flags

cleared by two BCD flag setting instructions, SETX and CLRX, respectively, or by writing to the PSW in kernel mode.

- N **Negative** (PSW bit 21) – The negative flag is set if the sign of the infinite-precision result is negative for nonlogical instructions. Zero is considered positive in this case. For logical instructions, the negative flag is the most significant bit (bit 31 for words; bit 15 for halfwords; bit 7 for bytes) of the requested representation. Note that this bit is valid even in unsigned operations.
- Z **Zero** (PSW bit 20) – For nonlogical instructions, the zero flag is set if the infinite-precision result is equal to zero. For logical instructions, it is set if the requested representation is equal to zero.
- V **Overflow** (PSW bit 19) – For arithmetic operations with signed destination, the V bit is set if any truncated bit of the infinite precision result is different from the sign bit, which is the MSB, of the requested representation. For arithmetic operations with an unsigned destination, the V bit is set if any truncated bit is nonzero. For arithmetic left-shift operations, bits are not shifted past bit 31 of the result, so the V bit is set only if there is a truncation error. For logical instructions with a signed destination, the V bit is set if any truncated bit of the 32-bit result is different from the sign bit of the requested representation. For logical instructions with an unsigned destination, the V bit is set if any truncated bit is nonzero.
- C **Carry/Borrow** (PSW bit 18) – The carry bit is the carry into the 33rd bit position (bit 32) for word operations, the 17th bit position (bit 16) for halfword operations, or the 9th bit position (bit 8) for byte operations. For subtracts, negates, decrements, and compares, it is the borrow from the 33rd, 17th, or 9th bit position. For example, given $A-B$, the carry flag is set if the 32-bit value (a sign-extended or zero-extended representation of A) evaluated as an unsigned number, is less than B (also a 32-bit value, sign- or zero-extended) which is also evaluated as an unsigned number. For logical instructions, the carry is cleared. For BCD arithmetic operations, the carry is set if there is a carry or a borrow from a BCD arithmetic operation on two 2-BCD digits.

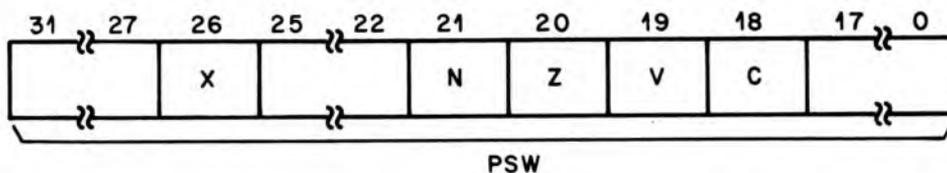


Figure 2-19. Condition Flags

Note: If a memory-write fault occurs, the flags (except the X flag) are altered as if the instruction were completed normally.

The instruction descriptions in Chapter 5 include the effect that each instruction has on the condition flags.

2.14 TRACE MECHANISM

A feature that is useful in user applications development is the *WE 32200* Microprocessor trace mechanism. It is implemented with two bits of the PSW, the trace enable (TE) bit and the trace mask (TM) bit. The truth table for the operation of the trace mechanism is shown in Table 2-7.

TE-end	TM-end	Trap
0	0/1	No
1	0	No
1	1	Yes

The TE bit must be set before an instruction is executed. If the TE bit is not cleared by the instruction and the TM bit remains set at the end of the instruction, a trace trap is taken. The TE-end column of Table 2-7 corresponds to the value of the TE bit at the time the trap is detected. The TM-end column corresponds to the value of the TM bit at the same instant. The trace trap is detected before the next instruction is started and has priority over interrupts.

The TM bit cannot be set by software, however, the CPU changes the TM bit automatically as follows:

- It is set to 1 at the beginning of every instruction
- It is cleared to 0 as a part of every microsequence that does a context switch
- It is cleared to 0 as part of the return from gate microsequence
- It is cleared to 0 when any exception or interrupt is detected and responded to.

The effect of the TM bit is to mask the TE bit for the duration of one instruction. Thus, the trace trap handler software would use this bit to prevent a trap from occurring when it returns control by using the return from gate instruction. Similarly, the *WE 32200* CPU uses this bit to prevent a trace trap from occurring in the context of a newly switched process while the previous process was traced.

REGISTERS, DATA, AND INSTRUCTION FORMATS

Trace Mechanism

The values of the TE and TM bits at the end of the instruction can be changed by one of the following:

- A non-microsequence instruction writing to the PSW changes the TE bit (Table 2-7 is not valid for non-microsequence instructions which write to the PSW. This method of changing the TE bit causes inconsistent trace behavior and should be avoided.).
- A PSW restored from the stack because of the return from gate instruction.
- A PSW restored from the PCB because of a context switch to the process.

Because of the behavior of the TE and TM bits, the following instructions cannot be traced:

- Return from process (RETPTS)
- Call process (CALLPS)
- Return from gate (RETG)
- User call process (UCALLPS)
- Return from quick interrupt (RETQINT)

In addition, a breakpoint trap (BPT) instruction causes a breakpoint trap and then a trace trap if trace is enabled.

When a trace trap condition is detected, the CPU performs an on normal exception operation. The current PSW (the TE bit corresponds to its end values) is pushed along with the PC, which points to the next instruction. Because of this, the trace trap handler has to maintain the PC of the instruction being traced as a static instruction. This is before any write to a memory location or register has occurred as part of the instruction. Therefore, the PSW saved as part of the interrupt sequence has the TE bit corresponding to the beginning value.

A coprocessor instruction that is being traced terminates if an interrupt occurs and the instruction flow changes to the trace handler. Since the instruction has not completed, the PC still points to the beginning of the coprocessor instruction. If no interrupt occurs and the instruction completes normally, it traces just like any other instruction.

Chapter 3

**Signal
Descriptions**

CHAPTER 3. SIGNAL DESCRIPTIONS

CONTENTS

3. SIGNAL DESCRIPTIONS	3-1
3.1 ADDRESS AND DATA SIGNALS	3-2
Address (ADDR00—ADDR31)	3-2
Data (DATA00—DATA31)	3-2
3.2 INTERRUPT SIGNALS	3-2
Autovector ($\overline{\text{AVEC}}$)	3-2
Interrupt Option ($\overline{\text{INTOPT}}$)	3-2
Interrupt Priority Level (IPL0—IPL3)	3-2
Nonmaskable Interrupt ($\overline{\text{NMINT}}$)	3-3
3.3 INTERFACE AND CONTROL SIGNALS	3-3
Address Strobe ($\overline{\text{AS}}$)	3-3
Cycle Initiate ($\overline{\text{CYCLE}}$)	3-3
Coprocessor Done ($\overline{\text{DONE}}$)	3-3
Data Ready ($\overline{\text{DRDY}}$)	3-3
Data Strobe ($\overline{\text{DS}}$)	3-3
Data Transfer Acknowledge ($\overline{\text{DTACK}}$)	3-4
Dynamic 16-Bit Port Acknowledge ($\overline{\text{DYN16}}$)	3-4
Synchronous Ready ($\overline{\text{SRDY}}$)	3-4
3.4 BUS ARBITRATION SIGNALS	3-4
Bus Arbiter ($\overline{\text{BARB}}$)	3-5
Bus Request ($\overline{\text{BUSRQ}}$)	3-5
Bus Request Acknowledge ($\overline{\text{BRACK}}$)	3-5
3.5 BUS EXCEPTION SIGNALS	3-5
Access Abort ($\overline{\text{ABORT}}$)	3-6
Data Bus Shadow ($\overline{\text{DSHAD}}$)	3-6
Fault ($\overline{\text{FAULT}}$)	3-6
Reset Acknowledge ($\overline{\text{RESET}}$)	3-6
Reset Request ($\overline{\text{RESETR}}$)	3-6
Retry ($\overline{\text{RETRY}}$)	3-6
Relinquish and Retry Request Acknowledge ($\overline{\text{RRRACK}}$)	3-6
Relinquish and Retry Request ($\overline{\text{RRREQ}}$)	3-7
3.6 ACCESS STATUS SIGNALS	3-7
Block (Double-Word) Fetch ($\overline{\text{BLKFTCH}}$)	3-7
Data Size (DSIZE0—DSIZE2)	3-7
Read/Write (R/ $\overline{\text{W}}$)	3-8
Access Status Codes (SAS0—SAS3)	3-8
Virtual Address ($\overline{\text{VAD}}$)	3-9
Execution Mode (XMD0—XMD1)	3-9
3.7 DEVELOPMENT SYSTEM SUPPORT SIGNALS	3-10
High Impedance ($\overline{\text{HIGHZ}}$)	3-10
Instruction Queue Status (IQS0—IQS1)	3-10
Start of Instruction ($\overline{\text{SOI}}$)	3-10
Stop ($\overline{\text{STOP}}$)	3-10
3.8 CLOCK SIGNALS	3-10
Clock (CLK23)	3-10
Clock (CLK34)	3-11
3.9 PIN ASSIGNMENTS	3-11

3. SIGNAL DESCRIPTIONS

The WE 32200 Microprocessor input and output signals are described in this chapter. The signals are organized in functional groups and are shown on Figure 3-1.

The term *asserted* means that signal is driven active (true) either by the microprocessor (outputs) or an external device (inputs). The term *negated* means a signal is driven inactive (false). A bar over a signal name (e.g., \overline{AS}) indicates an active low signal.

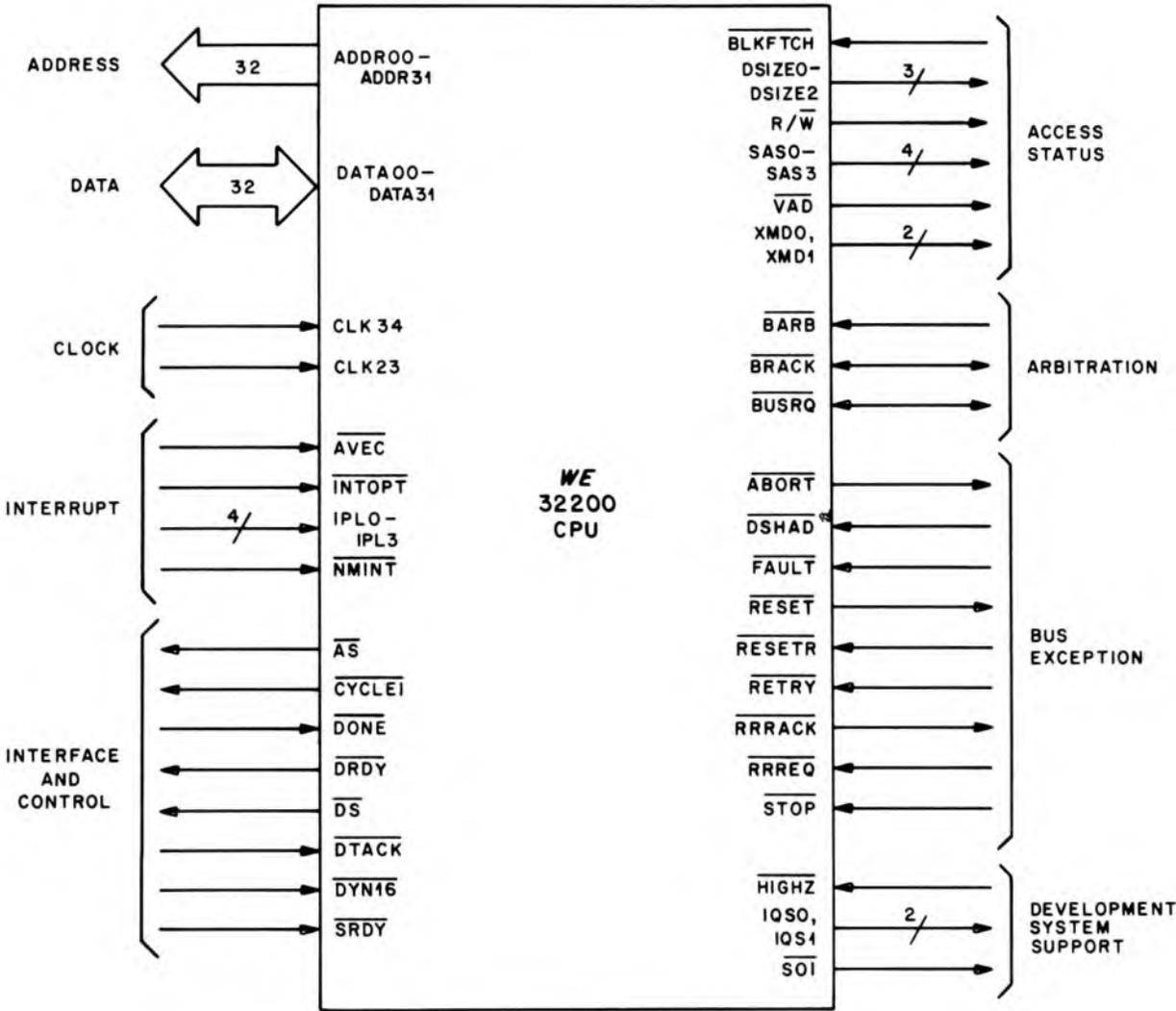


Figure 3-1. Signal Grouping Diagram

SIGNAL DESCRIPTIONS

Address and Data Signals

3.1 ADDRESS AND DATA SIGNALS

Separate 32-bit address and data buses are provided to the external system, eliminating the need for external multiplexing/demultiplexing of address and data buses.

Address (ADDR00—ADDR31)

Memory or peripherals mapped into the system memory space are accessed by this 32-bit output address bus. Address bits 2 through 6 convey the interrupt acknowledge level during an interrupt acknowledge operation. ADDR00 is the least significant address bit (LSB) of the address.

Data (DATA00—DATA31)

Data is read to or written from the microprocessor over this 32-bit bidirectional data bus. During a normal interrupt, the microprocessor uses data bits 0 through 7 to fetch the interrupt vector number from an interrupting device. DATA00 is the LSB.

3.2 INTERRUPT SIGNALS

The microprocessor's interrupt structure is flexible enough to handle a wide variety of applications.

Autovector ($\overline{\text{AVEC}}$)

If the autovector ($\overline{\text{AVEC}}$) input is active (low) during an interrupt request, the microprocessor will not fetch a vector number from the interrupting device. Instead, the WE 32200 Microprocessor generates its own vector number by concatenating the inverted $\overline{\text{INTOPT}}$ input with the inverted interrupt priority level inputs (IPL0—IPL3).

Interrupt Option ($\overline{\text{INTOPT}}$)

During an interrupt acknowledge transaction this asynchronous input is latched along with the interrupt priority level inputs (IPL0—IPL3). The value of $\overline{\text{INTOPT}}$ is inverted and transmitted on bit 6 of the address bus.

Interrupt Priority Level (IPL0—IPL3)

An interrupt request can be made by placing an interrupt request value on these asynchronous inputs. The code is based on a decreasing priority scheme, with 0000 having the highest priority and 1110 the lowest. Level 1111 indicates that no interrupts are pending. This type of interrupt request can be masked by using the interrupt priority level field (bits 13—16) of the PSW. To be acknowledged, the interrupt request value inverted must be greater than the present IPL field priority level. The exception to this is a nonmaskable interrupt, which can interrupt the microprocessor regardless of the present IPL field priority level.

Nonmaskable Interrupt (\overline{NMINT})

When asserted, this asynchronous input indicates that a nonmaskable interrupt is being requested. As previously mentioned, a nonmaskable interrupt can interrupt the microprocessor regardless of the current priority level in the IPL field. The interrupt is treated by the microprocessor as an autovector interrupt with vector number 0. All address bus bits are driven low during the acknowledge cycle; this distinguishes the nonmaskable from all other interrupts.

3.3 INTERFACE AND CONTROL SIGNALS

The bus protocol required for the efficient transfer of data is provided by the following group of signals.

Address Strobe (\overline{AS})

When asserted, this output signal indicates the presence of a valid physical address on the address pins. If the address is virtual, the falling edge of \overline{AS} indicates a valid address; the virtual address will be 3-stated subsequent to the falling edge of \overline{AS} .

Cycle Initiate (\overline{CYCLEI})

This output signal is asserted during the first clock state (one clock state is half a clock cycle) to indicate the beginning of a bus transaction to external devices. \overline{CYCLEI} is negated at the end of the fourth clock state (second clock cycle); it is asserted in both the read and write halves of an interlocked read transaction.

Coprocessor Done (\overline{DONE})

This signal is recognized during a coprocessor instruction. A coprocessor done indicates that it has finished its operation by asserting this microprocessor input signal.

Data Ready (\overline{DRDY})

When active, this output signal indicates that the microprocessor (acting as bus master) has not detected any bus exceptions (\overline{FAULT} , \overline{RETRY} , and \overline{RRREQ} signals) during the current bus transaction. This signal may be tied common to other devices that have this signal as an output (e.g., MMU, DMAC, etc.). However, only one device (bus master) can be in control of this signal at any one time. For example, when the MMU data shadow signal \overline{DSHAD} is inactive, the MMU is not the bus master and its \overline{DRDY} is 3-stated. \overline{DRDY} 's trailing edge marks the end of a bus cycle that has no bus exceptions.

Data Strobe (\overline{DS})

When asserted during a read operation, this output signal indicates that a coprocessor or slave device can place data on the data bus. When asserted during a write

SIGNAL DESCRIPTIONS

Data Transfer Acknowledge (\overline{DTACK})

operation this signal indicates that the microprocessor has placed valid data on the data bus.

Data Transfer Acknowledge (\overline{DTACK})

This signal is used for handshaking between the microprocessor and a coprocessor or slave device. During a read operation, the microprocessor latches data present on the data bus and terminates the bus transaction one cycle after \overline{DTACK} is driven low by the coprocessor or slave device. During a write operation, the transaction is terminated when the coprocessor or slave device drives \overline{DTACK} low. If \overline{DTACK} is high, wait states are inserted in the current cycle. \overline{DTACK} is ignored if \overline{DSHAD} is asserted. \overline{DTACK} is an asynchronous input and is latched at two different times (i.e., double-latched) to maintain stability.

Dynamic 16-Bit Port Acknowledge ($\overline{DYN16}$)

This input signal informs the CPU that it is writing to or reading from a 16-bit port (the upper 16 bits of the data bus, DATA16—DATA31). The memory system has four different options when asserting $\overline{DYN16}$:

- $\overline{DYN16}$ asserted with \overline{DTACK} . The $\overline{DYN16}$ signal can be qualified with the asynchronous acknowledge to indicate a 16-bit port. The signal is internally double latched. Assertion of $\overline{DYN16}$ looks identical to the assertion of \overline{DTACK} .
- $\overline{DYN16}$ asserted after \overline{DTACK} . This is similar to the case described above except that $\overline{DYN16}$ may be asserted at the synchronous sampling point.
- $\overline{DYN16}$ alone. The $\overline{DYN16}$ can be used as an acknowledge by itself (i.e., it can terminate a memory transaction). The signal is internally double-latched. $\overline{DYN16}$ must be asserted at the asynchronous sampling point and looks exactly like a \overline{DTACK} assertion.
- $\overline{DYN16}$ asserted with \overline{SRDY} . The $\overline{DYN16}$ signal can be qualified with the synchronous acknowledge to indicate a 16-bit port. The assertion of $\overline{DYN16}$ looks exactly like the assertion of \overline{SRDY} .

$\overline{DYN16}$ is ignored if \overline{DSHAD} is asserted.

Synchronous Ready (\overline{SRDY})

This signal is a synchronous input that begins the termination of a read or write operation when asserted. It is sampled only once on the leading edge of the fifth clock state during read and write operations. If \overline{SRDY} is not asserted at this time and \overline{DTACK} was not asserted during the previous clock state, then wait-state cycles are inserted until one of these signals is asserted. \overline{SRDY} is ignored if the \overline{DSHAD} signal was previously asserted.

3.4 BUS ARBITRATION SIGNALS

When two or more processors share the system bus, a mechanism to arbitrate control of the bus must exist. This is accomplished through arbitration signals.

Bus Arbiter ($\overline{\text{BARB}}$)

When this input is strapped low, the microprocessor is arbiter of the bus. In this mode, the microprocessor need not request the bus to obtain access to it. When this signal is strapped high, an external device (e.g., DMAC) is the bus arbiter and the microprocessor must request the bus to obtain access to it. When the microprocessor is not the bus arbiter, the following outputs are 3-stated until the CPU performs a bus transaction:

$\overline{\text{ABORT}}$	ADDR00—ADDR31
$\overline{\text{DS}}$	DSIZE0—DSIZE2
$\overline{\text{AS}}$	R/ $\overline{\text{W}}$
$\overline{\text{CYCLEI}}$	SAS0—SAS3
$\overline{\text{VAD}}$	DATA00—DATA31
$\overline{\text{DRDY}}$	XMD0—XMD1

Bus Request ($\overline{\text{BUSRQ}}$)

This bidirectional signal is an input when the microprocessor is the bus arbiter and an output when it is not. As an input, this signal indicates that an external device (e.g., DMAC) is requesting the bus. It is an asynchronous input and double-latched to maintain stability. As an output, this signal indicates that the microprocessor is requesting the bus.

Bus Request Acknowledge ($\overline{\text{BRACK}}$)

Unlike $\overline{\text{BUSRQ}}$, this signal is an output when the microprocessor is acting as bus arbiter and an input when it is not. As an output, this signal indicates that $\overline{\text{BUSRQ}}$ has been recognized, and the microprocessor has 3-stated the bus for the requesting bus master. The bus signals that are 3-stated when the $\overline{\text{BRACK}}$ is issued are:

$\overline{\text{ABORT}}$	ADDR00—ADDR31
$\overline{\text{DS}}$	DSIZE0—DSIZE2
$\overline{\text{AS}}$	R/ $\overline{\text{W}}$
$\overline{\text{CYCLEI}}$	SAS0—SAS3
$\overline{\text{VAD}}$	DATA00—DATA31
$\overline{\text{DRDY}}$	XMD0—XMD1

As an input, assertion of this signal indicates that the microprocessor's bus request has been recognized and the microprocessor may take possession of the bus.

3.5 BUS EXCEPTION SIGNALS

This group of signals can cause the termination of the current access and results when an access retry is required or when an exception occurs during an access.

SIGNAL DESCRIPTIONS

Access Abort ($\overline{\text{ABORT}}$)

Access Abort ($\overline{\text{ABORT}}$)

This output signal is asserted on an access that is ignored by the memory system. The two cases in which the CPU aborts a memory access are:

- An instruction cache hit on program counter discontinuity.
- An alignment fault.

Data Bus Shadow ($\overline{\text{DSHAD}}$)

This input signal is used by the MMU to remove the microprocessor from the data bus. The $\overline{\text{DATA00—DATA31}}$, $\overline{\text{DRDY}}$, $\overline{\text{DSIZE0—DSIZE2}}$, and $\overline{\text{R/W}}$ are 3-stated and the $\overline{\text{DTACK}}$, $\overline{\text{DYN16}}$, $\overline{\text{SRDY}}$, and $\overline{\text{FAULT}}$ inputs are ignored when this input is asserted.

Fault ($\overline{\text{FAULT}}$)

This input notifies the microprocessor that a fault condition has occurred. It is a double-latched, asynchronous input if asserted prior to the $\overline{\text{DTACK}}$ assertion. It is a synchronous signal if asserted after $\overline{\text{DTACK}}$ assertion (latched once). The $\overline{\text{FAULT}}$ signal is ignored if $\overline{\text{DSHAD}}$ is asserted.

Reset Acknowledge ($\overline{\text{RESET}}$)

This output signal indicates that the microprocessor has recognized an external reset request or that it has generated an internal reset (e.g., reset exception). The microprocessor executes its reset routine once it negates $\overline{\text{RESET}}$.

Reset Request ($\overline{\text{RESETR}}$)

This asynchronous input signal is used to reset the microprocessor. $\overline{\text{RESETR}}$ is sampled once per clock cycle and must be active for two consecutive clock cycles in order to be acknowledged. The microprocessor acknowledges the request by immediately asserting $\overline{\text{RESET}}$.

Retry ($\overline{\text{RETRY}}$)

When this input signal is asserted, the microprocessor terminates the current bus transaction and retries it when $\overline{\text{RETRY}}$ is negated.

Relinquish and Retry Request Acknowledge ($\overline{\text{RRRACK}}$)

This output signal is asserted in response to a relinquish and retry bus exception when the microprocessor has relinquished (3-stated) the bus. It is negated upon retry of the bus transaction that was preempted by the relinquish and retry bus exception.

Relinquish and Retry Request ($\overline{\text{RRREQ}}$)

This input signal is used to preempt a bus transaction so that the microprocessor bus may be used. This signal causes the microprocessor to terminate the current bus transaction and to 3-state the following signals:

$\overline{\text{ABORT}}$	ADDR00—ADDR31
$\overline{\text{DS}}$	DSIZE0—DSIZE2
$\overline{\text{AS}}$	R/ $\overline{\text{W}}$
$\overline{\text{CYCLEI}}$	SAS0—SAS3
$\overline{\text{VAD}}$	DATA00—DATA31
$\overline{\text{DRDY}}$	XMD0-XMD1

In response to this input, the microprocessor asserts $\overline{\text{RRRACK}}$. During the 3-state phase, the external device (bus master) requesting the relinquish and retry may take possession of the bus. No external bus arbitration signals are acknowledged during the assertion of a $\overline{\text{RRREQ}}$. When $\overline{\text{RRREQ}}$ is negated, the preempted bus transaction is retried.

3.6 ACCESS STATUS SIGNALS

This group of signals relates the access status to external devices.

Block (Double-Word) Fetch ($\overline{\text{BLKFTCH}}$)

This input signal, when active, tells the microprocessor that the memory is designed to handle a double-word blockfetch. Thus, the memory provides two words of instruction code for every address generated by the CPU. In this instance, the data size (DSIZE0—DSIZE2) signals will show a double-word access on all instruction fetches. If the input is negated, the microprocessor fetches a block of instructions with two consecutive reads.

Note: The $\overline{\text{BLKFTCH}}$ input cannot be asserted for the same bus transaction as the $\overline{\text{DYN16}}$ signal.

Data Size (DSIZE0—DSIZE2)

This 3-bit output is used to indicate the size of the data in the current bus transaction. These signals are used to indicate whether the CPU is transferring byte, halfword, three bytes, word, or double-word pieces of data. On all instruction fetches the DSIZE signals will show a double-word access.

SIGNAL DESCRIPTIONS

Read/Write

For prefetches, the DSIZE signals have the value of either a double-word or word access. The DSIZE signals are interpreted as follows:

DSIZE2	DSIZE1	DSIZE0	Transaction Size
0	0	0	Three-byte transaction
0	0	1	0 byte transaction (probe)
0	1	0	Reserved
0	1	1	Reserved
1	0	0	Word transaction
1	0	1	Double-word transaction
1	1	0	Halfword transaction
1	1	1	Byte transaction

Read/Write (R/\overline{W})

This output signal indicates whether the bus transaction is a read or write. When low, the operation is a write; when high, it is a read. This signal is valid during the time the \overline{AS} is active.

Access Status Code (SAS0—SAS3)

These outputs identify the type of bus cycle being executed through the following codes. SAS0 is the LSB of the code.

SAS3	SAS2	SAS1	SAS0	Description
0	0	0	0	Move translated word
0	0	0	1	Coprocessor data write
0	0	1	0	Autovector interrupt acknowledge
0	0	1	1	Coprocessor data fetch
0	1	0	0	Stop acknowledge
0	1	0	1	Coprocessor broadcast
0	1	1	0	Coprocessor status fetch
0	1	1	1	Read interlocked
1	0	0	0	Address fetch
1	0	0	1	Operand fetch
1	0	1	0	Write
1	0	1	1	Interrupt acknowledge
1	1	0	0	Instruction fetch after PC discontinuity
1	1	0	1	Instruction prefetch
1	1	1	0	Instruction fetch
1	1	1	1	No operation

Virtual Address (\overline{VAD})

When asserted, this output indicates that the address is virtual. When negated, the address is physical. When the address is virtual, the ADDR00—ADDR31 signals are 3-stated in the middle of the third clock state. The \overline{VAD} is asserted by the execution of the enable virtual pin and jump (ENBVJMP) instruction and negated by execution of the disable virtual pin and jump (DISVJMP) instruction. \overline{VAD} is a level rather than a pulsed signal.

Execution Mode (XMD0—XMD1)

These two outputs indicate the present execution mode of the microprocessor. XMD0 is the LSB of the execution mode code. The execution mode signals are interpreted as shown below:

XMD1	XMD0	Modes
0	0	Kernel
0	1	Execution
1	0	Supervisor
1	1	User

The XMD0—XMD1 signals are level signals. If an MMU is present in the system, it may latch and use a spurious execution mode value if XMD0—XMD1 changes during an access. XMD0—XMD1 reflect the value of the current execution level (CM) bits of the PSW; changes to the CM field via non-microsequence instructions must be avoided.

SIGNAL DESCRIPTIONS

Development System Support Signals

3.7 DEVELOPMENT SYSTEM SUPPORT SIGNALS

These signals aid in the design and testing of the system.

High Impedance ($\overline{\text{HIGHZ}}$)

When asserted, this input signal places all output pins on the microprocessor into the high-impedance state. This signal is intended for testing purposes.

Instruction Queue Status (IQS0—IQS1)

These outputs identify the activity on the microprocessor instruction queue by the following code:

IQS1	IQS0	Description
0	0	Discard 4 bytes
0	1	Discard 1 byte
1	0	Discard 2 bytes
1	1	No discard this cycle

IQS0 is the LSB of the instruction queue status code.

Start of Instruction ($\overline{\text{SOI}}$)

When asserted, this output signal indicates that the microprocessor's internal control has fetched the opcode for the next instruction from the internal instruction queue. Since the instructions are pipelined, it does not mean the end of the previous instruction execution.

Stop ($\overline{\text{STOP}}$)

When asserted, this asynchronous input signal halts the execution of any further instructions beyond those already started. Before the microprocessor comes to a halt, there may be one or two instructions executed beyond the instruction during which $\overline{\text{STOP}}$ was asserted. This is a result of pipelining. The $\overline{\text{STOP}}$ signal should be used for development purposes only.

3.8 CLOCK SIGNALS

The microprocessor requires two clock inputs, both operating at the same frequency.

Clock (CLK23)

This clock input leads CLK34 by 90°.

Clock (CLK34)

The falling edge of this input signifies the beginning of a machine cycle.

3.9 PIN ASSIGNMENTS

The WE 32200 Microprocessor is available in a 133-pin square, ceramic pin grid array (PGA) package. The package has 109 active pins, 8 power pins, and 16 ground pins. Figure 3-2 and Table 3-1 describe the pin assignments.

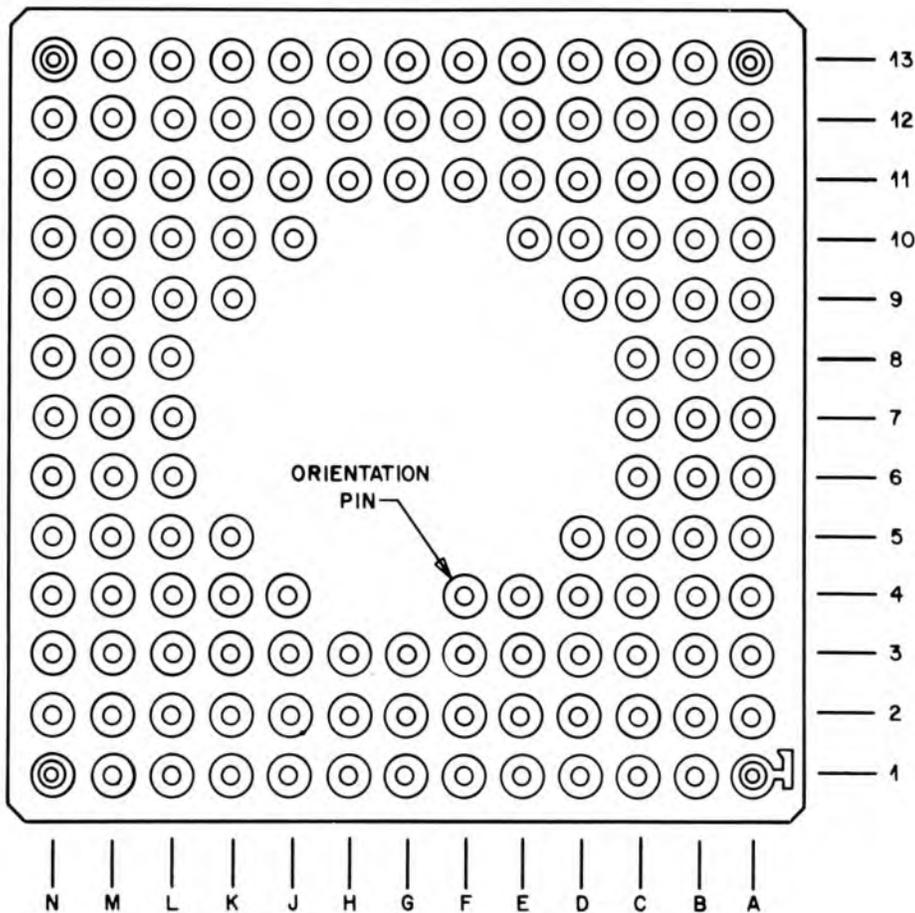


Figure 3-2. WE[®] 32200 Microprocessor 133-Pin Square, Ceramic PGA Package Bottom View

SIGNAL DESCRIPTIONS

Pin Assignments

Table 3-1. WE [®] 32200 Microprocessor Pin Descriptions			
Pin	Name	Type	Description
A1	DATA23	I/O	Microprocessor Data 23
A2	DATA24	I/O	Microprocessor Data 24
A3	DATA26	I/O	Microprocessor Data 26
A4	ADDR28	O	Microprocessor Address 28
A5	IQS0	O	Instruction Queue Status 0
A6	ADDR30	O	Microprocessor Address 30
A7	XMD1	O	Execution Mode 1
A8	IQS1	O	Instruction Queue Status 1
A9	SAS3	O	Access Status Code 3
A10	XMD0	O	Execution Mode 0
A11	SAS2	O	Access Status Code 2
A12	DSIZE1	O	Data Size 1
A13	SAS1	O	Access Status Code 1
B1	DATA22	I/O	Microprocessor Data 22
B2	DATA28	I/O	Microprocessor Data 28
B3	DATA29	I/O	Microprocessor Data 29
B4	DATA31	I/O	Microprocessor Data 31
B5	ADDR29	O	Microprocessor Address 29
B6	ADDR31	O	Microprocessor Address 31
B7	SOI	O	Start of Instruction
B8	BRACK	I/O	Bus Request Acknowledge
B9	BUSRQ	I/O	Bus Request
B10	SAS0	O	Access Status Code 0
B11	R/W	O	Read/Write
B12	DSIZE0	O	Data Size 0
B13	RRREQ	I	Relinquish and Retry Request
C1	ADDR19	O	Microprocessor Address 19
C2	ADDR26	O	Microprocessor Address 26
C3	DATA25	I/O	Microprocessor Data 25
C4	ADDR27	O	Microprocessor Address 27
C5	DATA30	I/O	Microprocessor Data 30
C6	+5V	—	Microprocessor Power
C7	GND	—	Microprocessor Ground
C8	+5V	—	Microprocessor Power
C9	GND	—	Microprocessor Ground
C10	DONE	I	Coprocessor Done
C11	HIGHZ	I	High Impedance
C12	RRRACK	O	Relinquish and Retry Request Acknowledge
C13	FAULT	I	Fault
D1	ADDR24	O	Microprocessor Address 24

Table 3-1. WE[®] 32200 Microprocessor Pin Descriptions (Continued)			
Pin	Name	Type	Description
D2	ADDR20	O	Microprocessor Address 20
D3	ADDR25	O	Microprocessor Address 25
D4	DATA27	I/O	Microprocessor Data 27
D5	GND	—	Microprocessor Ground
D9	DSIZE2	O	Data Size 2
D10	$\overline{\text{SRDY}}$	I	Synchronous Ready
D11	$\overline{\text{RETRY}}$	I	Retry
D12	BARB	I	Bus Arbiter
D13	$\overline{\text{RESET}}$	O	Reset Acknowledge
E1	ADDR18	O	Microprocessor Address 18
E2	ADDR23	O	Microprocessor Address 23
E3	+5V	—	Microprocessor Power
E4	GND	—	Microprocessor Ground
E10	GND	—	Microprocessor Ground
E11	+5V	—	Microprocessor Power
E12	$\overline{\text{DTACK}}$	I	Data Transfer Acknowledge
E13	$\overline{\text{BLKFTCH}}$	I	Block (Double-Word) Fetch
F1	ADDR21	O	Microprocessor Address 21
F2	ADDR22	O	Microprocessor Address 22
F3	GND	—	Microprocessor Ground
F4	GND	—	Microprocessor Ground
F11	CLK23	I	Input Clock 23
F12	$\overline{\text{DSHAD}}$	I	Data Bus Shadow
F13	$\overline{\text{RESETR}}$	I	Reset Request
G1	DATA21	I/O	Microprocessor Data 21
G2	ADDR17	O	Microprocessor Address 17
G3	DATA20	I/O	Microprocessor Data 20
G11	CLK34	I	Input Clock 34
G12	$\overline{\text{CYCLEI}}$	O	Cycle Initiate
G13	$\overline{\text{STOP}}$	I	Microprocessor Stop
H1	ADDR15	O	Microprocessor Address 15
H2	DATA18	I/O	Microprocessor Data 18
H3	GND	—	Microprocessor Ground
H11	GND	—	Microprocessor Ground
H12	$\overline{\text{AS}}$	O	Address Strobe
H13	$\overline{\text{DS}}$	O	Data Strobe
J1	DATA19	I/O	Microprocessor Data 19
J2	DATA17	I/O	Microprocessor Data 17
J3	+5V	—	Microprocessor Power
J4	GND	—	Microprocessor Ground

SIGNAL DESCRIPTIONS

Pin Assignments

Table 3-1. WE [®] 32200 Microprocessor Pin Descriptions (Continued)			
Pin	Name	Type	Description
J10	GND	—	Microprocessor Ground
J11	+5V	—	Microprocessor Power
J12	$\overline{\text{ABORT}}$	O	Access Abort
J13	$\overline{\text{DRDY}}$	O	Data Ready
K1	ADDR16	O	Microprocessor Address 16
K2	ADDR13	O	Microprocessor Address 13
K3	DATA15	I/O	Microprocessor Data 15
K4	DATA12	I/O	Microprocessor Data 12
K5	GND	—	Microprocessor Ground
K9	GND	—	Microprocessor Ground
K10	DATA00	I/O	Microprocessor Data 00
K11	GND	—	Microprocessor Ground
K12	$\overline{\text{DYN16}}$	I	Dynamic 16-bit Port Acknowledge
K13	$\overline{\text{NMINT}}$	I	Nonmaskable Interrupt
L1	ADDR14	O	Microprocessor Address 14
L2	DATA16	I/O	Microprocessor Data 16
L3	DATA11	I/O	Microprocessor Data 11
L4	ADDR11	O	Microprocessor Address 11
L5	GND	—	Microprocessor Ground
L6	ADDR08	O	Microprocessor Address 08
L7	+5V	—	Microprocessor Power
L8	GND	—	Microprocessor Ground
L9	+5V	—	Microprocessor Power
L10	ADDR00	O	Microprocessor Address 00
L11	$\overline{\text{INTOPT}}$	I	Interrupt Option
L12	IPL3	I	Interrupt Priority Level 3
L13	$\overline{\text{AVEC}}$	I	Autovector
M1	ADDR12	O	Microprocessor Address 12
M2	DATA14	I/O	Microprocessor Data 14
M3	DATA13	I/O	Microprocessor Data 13
M4	DATA08	I/O	Microprocessor Data 08
M5	ADDR07	O	Microprocessor Address 07
M6	DATA05	I/O	Microprocessor Data 05
M7	ADDR03	O	Microprocessor Address 03
M8	ADDR05	O	Microprocessor Address 05
M9	DATA03	I/O	Microprocessor Data 03
M10	DATA02	I/O	Microprocessor Data 02
M11	DATA01	I/O	Microprocessor Data 01
M12	IPL2	I	Interrupt Priority Level 2
M13	$\overline{\text{VAD}}$	O	Virtual Address

SIGNAL DESCRIPTIONS
Pin Assignments

Table 3-1. WE[®] 32200 Microprocessor Pin Descriptions (Continued)			
Pin	Name	Type	Description
N1	DATA10	I/O	Microprocessor Data 10
N2	DATA09	I/O	Microprocessor Data 09
N3	ADDR10	O	Microprocessor Address 10
N4	ADDR09	O	Microprocessor Address 09
N5	ADDR06	O	Microprocessor Address 06
N6	DATA07	I/O	Microprocessor Data 07
N7	ADDR02	O	Microprocessor Address 02
N8	DATA06	I/O	Microprocessor Data 06
N9	ADDR01	O	Microprocessor Address 01
N10	ADDR04	O	Microprocessor Address 04
N11	DATA04	I/O	Microprocessor Data 04
N12	IPL0	I	Interrupt Priority Level 0
N13	IPL1	I	Interrupt Priority Level 1

Chapter 4

Bus Operation

CHAPTER 4. BUS OPERATION

CONTENTS

4. BUS OPERATION	4-1
4.1 SIGNAL SAMPLING POINTS.....	4-1
4.2 READ AND WRITE OPERATIONS	4-1
4.2.1 Synchronous Read Transaction	4-4
4.2.2 Asynchronous Read Transaction	4-6
4.2.3 Synchronous Read Transaction with Wait Cycle	4-10
4.2.4 Asynchronous Read Transaction with Two Wait Cycles	4-11
4.2.5 Synchronous Write Transaction	4-12
4.2.6 Asynchronous Write Transaction	4-12
4.2.7 Write Transaction with Wait Cycle Using $\overline{\text{SRDY}}$	4-12
4.2.8 Write Transaction with Wait Cycle Using $\overline{\text{DTACK}}$	4-18
4.3 READ INTERLOCKED OPERATION	4-19
4.4 COMPARE AND SWAP INTERLOCKED TRANSACTION.....	4-19
4.5 BLOCKFETCH OPERATION.....	4-22
4.5.1 Blockfetch Transaction Using $\overline{\text{SRDY}}$	4-22
4.5.2 Blockfetch Transaction Using $\overline{\text{DTACK}}$	4-24
4.5.3 Blockfetch Transaction Using $\overline{\text{DTACK}}$ with Wait Cycle on Second Word	4-25
4.5.4 Blockfetch Transaction Using $\overline{\text{DTACK}}$ with Wait Cycles on Both Words	4-26
4.6 BUS EXCEPTIONS.....	4-27
4.6.1 Faults	4-28
$\overline{\text{FAULT}}$ with $\overline{\text{SRDY}}$	4-29
$\overline{\text{FAULT}}$ After $\overline{\text{DTACK}}$	4-31
4.6.2 Retry.....	4-31
4.6.3 Relinquish and Retry.....	4-31
4.7 BLOCKFETCH SPECIAL CASES	4-34
4.7.1 Fault on First Word of Blockfetch with Status Code Other Than Prefetch ...	4-34
4.7.2 Fault on First Word of Blockfetch with Status of Prefetch	4-34
4.7.3 Retry on First Word of Blockfetch	4-34
4.7.4 Retry on Second Word of Blockfetch	4-34
4.7.5 Relinquish and Retry on Blockfetch.....	4-39
4.8 INTERRUPTS.....	4-39
4.8.1 Interrupt Acknowledge	4-41
4.8.2 Autovector Interrupt	4-44
4.8.3 Nonmaskable Interrupt	4-44
4.9 BUS ARBITRATION.....	4-47
4.9.1 Bus Request During a Bus Transaction	4-47
4.9.2 DMA Operation	4-49
4.10 RESET.....	4-50
4.10.1 System Reset	4-50
4.10.2 Internal Reset	4-50
4.10.3 Reset Sequence.....	4-52
4.11 ABORTED MEMORY ACCESSES.....	4-52
4.11.1 Aborted Access on PC Discontinuity with Instruction Cache Hit	4-53

4.11.2 Alignment Fault Bus Activity	4-54
4.12 SINGLE-STEP OPERATION	4-55
4.13 COPROCESSOR OPERATIONS	4-56
4.13.1 Coprocessor Broadcast	4-56
4.13.2 Coprocessor Operand Fetch	4-61
4.13.3 Coprocessor Status Fetch	4-62
4.13.4 Coprocessor Data Write	4-63
4.14 SUPPLEMENTARY PROTOCOL DIAGRAMS	4-64



4. BUS OPERATION

This chapter discusses bus protocol for the *WE 32200* Microprocessor.

4.1 SIGNAL SAMPLING POINTS

The *WE 32200* Microprocessor utilizes two phase-shifted input clocks (CLK23 and CLK34), as shown on Figure 4-1. The CPU samples all inputs at the points indicated on this figure. This figure can be used as a reference for the protocol diagrams in the sections that follow.

The bus transactions that are described in the upcoming sections share the following attributes. The read/write (R/\overline{W}) output remains in its mode (high, logic 1 for read transactions and low, logic 0 for write transactions) for the entire transaction. The cycle initiate (\overline{CYCLEI}) output goes active for two clock cycles at the beginning of each transaction. The CPU asserts the data ready (\overline{DRDY}) output at the end of the transaction if there are no bus exceptions (fault (\overline{FAULT}), retry (\overline{RETRY}), or relinquish and retry (\overline{RRREQ})) during the transaction.

The address bus (ADDR00—ADDR31) is driven for the entire transaction if the CPU is operating in physical mode. In virtual mode, the CPU drives the address bus only during the first and second clock states (one clock state is half a clock cycle). The CPU 3-states its address bus during the third clock state so that the MMU can drive the translated physical address onto the bus.

The data size bits (DSIZE0—DSIZE2) indicate the size of the transaction — 0 byte, 1 byte, 2 bytes (halfword), 3 bytes, 4 bytes (word), or 8 bytes (double word) and are driven for the entire transaction. The access status code (SAS0—SAS3) is driven one clock cycle before the transaction starts and remains active for two additional cycles during the transaction. This 4-bit code indicates the type of transaction being performed. The leading edge of \overline{CYCLEI} can be used to latch the access status code.

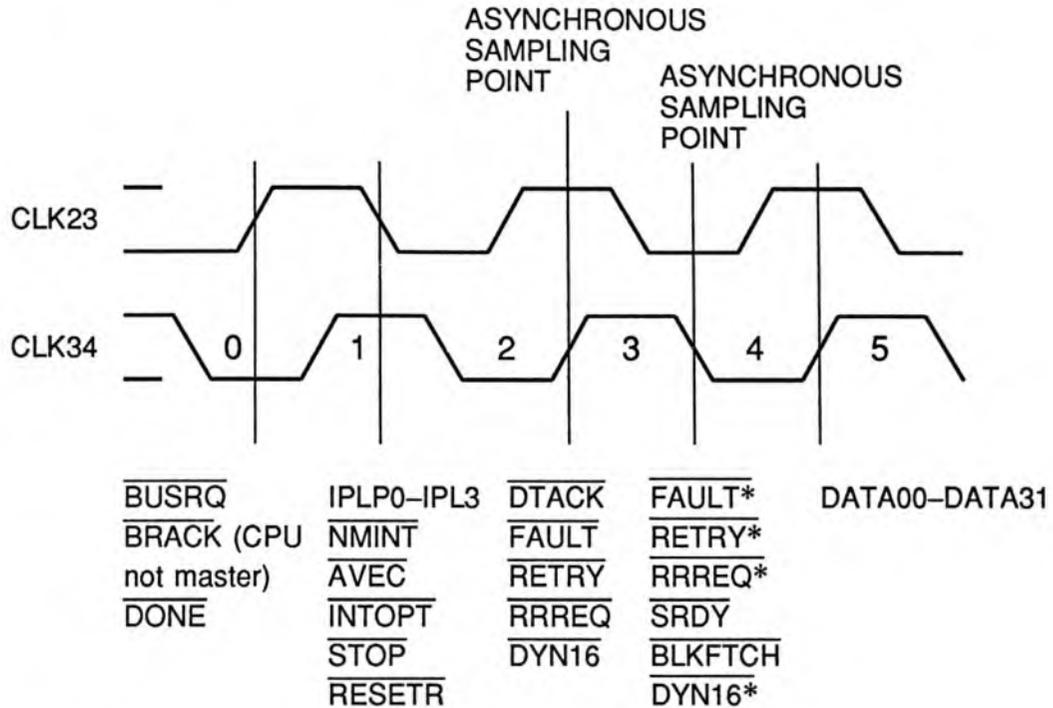
4.2 READ AND WRITE OPERATIONS

The *WE 32200* Microprocessor performs zero wait-state read and write accesses in three clock cycles. These accesses are performed in two stages. The microprocessor outputs the address and the control signals necessary for the given operation. Once these signals have had time to settle, the data transfer takes place. All accesses are followed by an additional cycle to allow enough time for a memory management unit to release the shared address bus.

The arbitrary byte alignment feature discussed in section 2.8 enables the microprocessor to handle nonaligned memory accesses efficiently for both reads and writes. The CPU automatically generates multiple accesses when nonaligned data types cross word boundaries.

BUS OPERATION

Read & Write Operations



* Double-latched.

Notes:

1. $\overline{\text{BUSRQ}}$, $\overline{\text{BRACK}}$, $\overline{\text{IPLO}}$ – $\overline{\text{IPL3}}$, $\overline{\text{NMINT}}$, $\overline{\text{AVEC}}$, $\overline{\text{INTOPT}}$, $\overline{\text{STOP}}$, $\overline{\text{RESETR}}$ are sampled repetitively one CLK34 cycle apart (i.e., on every clock cycle).
2. After $\overline{\text{DTACK}}$ is asserted, $\overline{\text{DYN16}}$, $\overline{\text{FAULT}}$, $\overline{\text{RETRY}}$, $\overline{\text{RRREQ}}$ and $\overline{\text{BLKFTCH}}$ are sampled once at the synchronous sampling point. If $\overline{\text{DYN16}}$, $\overline{\text{FAULT}}$, $\overline{\text{RETRY}}$, or $\overline{\text{RRREQ}}$ are asserted prior to or at the same time as $\overline{\text{DTACK}}$, then they are sampled once and double latched. If $\overline{\text{SRDY}}$ is asserted, then $\overline{\text{DYN16}}$, $\overline{\text{FAULT}}$, $\overline{\text{RETRY}}$, $\overline{\text{RRREQ}}$ and $\overline{\text{BLKFTCH}}$ are sampled once at the synchronous sampling point.
3. $\overline{\text{BLKFTCH}}$ must remain asserted until $\overline{\text{DS}}$ is negated.
4. $\overline{\text{DSHAD}}$ is not latched and can be asserted at any time, subject to the following conditions: $\overline{\text{DSHAD}}$ should be asserted only during a CPU-initiated transaction while $\overline{\text{AS}}$ is active and $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, $\overline{\text{DYN16}}$, and $\overline{\text{FAULT}}$ are inactive. Unless $\overline{\text{RETRY}}$ or $\overline{\text{RRREQ}}$ is active, $\overline{\text{DSHAD}}$ should be negated only while $\overline{\text{AS}}$ is still active and $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, $\overline{\text{DYN16}}$, and $\overline{\text{FAULT}}$ are inactive. If $\overline{\text{RETRY}}$ or $\overline{\text{RRREQ}}$ is active, then $\overline{\text{DSHAD}}$ should be negated one cycle after $\overline{\text{AS}}$ is negated.
5. The sampling of $\overline{\text{DYN16}}$ is exactly the same as that of any of the bus exceptions.
6. The blockfetch input for instruction fetch cannot be asserted for the same bus transaction as the $\overline{\text{DYN16}}$ input. $\overline{\text{DYN16}}$ is ignored if $\overline{\text{DSHAD}}$ is asserted.

Figure 4-1. Signal Sampling Points

Three inputs that allow handshaking between the CPU and slow slave devices are provided. External devices can cause the CPU to insert wait cycles during a bus transaction through the use of the synchronous ready ($\overline{\text{SRDY}}$) input, the data transfer acknowledge ($\overline{\text{DTACK}}$) input, and the dynamic 16-bit port ($\overline{\text{DYN16}}$) input. Wait cycles prolong a bus transaction, which allows slave devices more time to place data on the bus during a read transaction and to pick up data from the bus during a write transaction.

During bus transactions, the CPU samples the $\overline{\text{SRDY}}$, $\overline{\text{DTACK}}$, and $\overline{\text{DYN16}}$ inputs at their respective sampling points, as shown on Figure 4-1. If any of these inputs is active (low) at its sampling point, no wait cycles are inserted and the transaction completes in three clock cycles. However, if none of the inputs is sampled active, wait cycles are inserted, and the CPU continues sampling until any input is sampled active. At this point no more new wait cycles are generated and the bus transaction completes one clock cycle after the completion of the current wait cycle.

When the dynamic 16-bit port ($\overline{\text{DYN16}}$) input is asserted, the CPU is informed that the port it is communicating with is a 16-bit port. This port uses the upper bits of the data bus (DATA16—DATA31).

For ease of system design, the memory system has several different options when asserting $\overline{\text{DYN16}}$:

- $\overline{\text{DYN16}}$ is asserted with $\overline{\text{SRDY}}$. The $\overline{\text{DYN16}}$ signal may be qualified with the $\overline{\text{SRDY}}$ signal to indicate a 16-bit port. The assertion of $\overline{\text{DYN16}}$ should be exactly the same as the assertion of $\overline{\text{SRDY}}$.
- $\overline{\text{DYN16}}$ is asserted with $\overline{\text{DTACK}}$. The $\overline{\text{DYN16}}$ signal can be qualified with the asynchronous acknowledge to indicate a 16-bit port. The signal is double-latched internally. The assertion of $\overline{\text{DYN16}}$ should be exactly the same as the assertion of $\overline{\text{DTACK}}$.
- $\overline{\text{DYN16}}$ is asserted after $\overline{\text{DTACK}}$. This is the same as the second option above except that $\overline{\text{DYN16}}$ can be asserted at the synchronous sampling point.
- Only $\overline{\text{DYN16}}$ is asserted. The $\overline{\text{DYN16}}$ signal can be used as an acknowledge by itself (i.e., can terminate a memory transaction). The signal is double-latched internally. The $\overline{\text{DYN16}}$ input must be asserted at the asynchronous sampling point and should look exactly like a $\overline{\text{DTACK}}$ assertion.

Dynamic bus sizing can also be used with instruction fetches. When the CPU fetches an instruction from memory (data size of word or double word) and receives a 16-bit port acknowledge, the CPU generates a second access which has a data size of a halfword, in which the second least significant bit of the address is complemented. Therefore, on the second access the CPU always fetches the other half of the instruction.

In the following read and write operation descriptions, the term *asserted* means that a signal is driven to its active state either by the microprocessor (outputs) or by an external device (inputs). The term *negated* means that the signal is driven to its

BUS OPERATION

Synchronous Read Transaction

inactive state. A bar over a signal name (e.g., \overline{AS}) indicates that the signal is active low, logic 0.

4.2.1 Synchronous Read Transaction

Figure 4-2 illustrates a synchronous read transaction with zero wait cycles (3-cycle access) that uses \overline{SRDY} to terminate the access. The read transaction begins when the CPU issues an address ($ADDR00$ — $ADDR31$) and data size ($DSIZE0$ — $DSIZE2$), negates the R/\overline{W} output, and asserts the \overline{CYCLEI} output at the beginning of clock state zero.

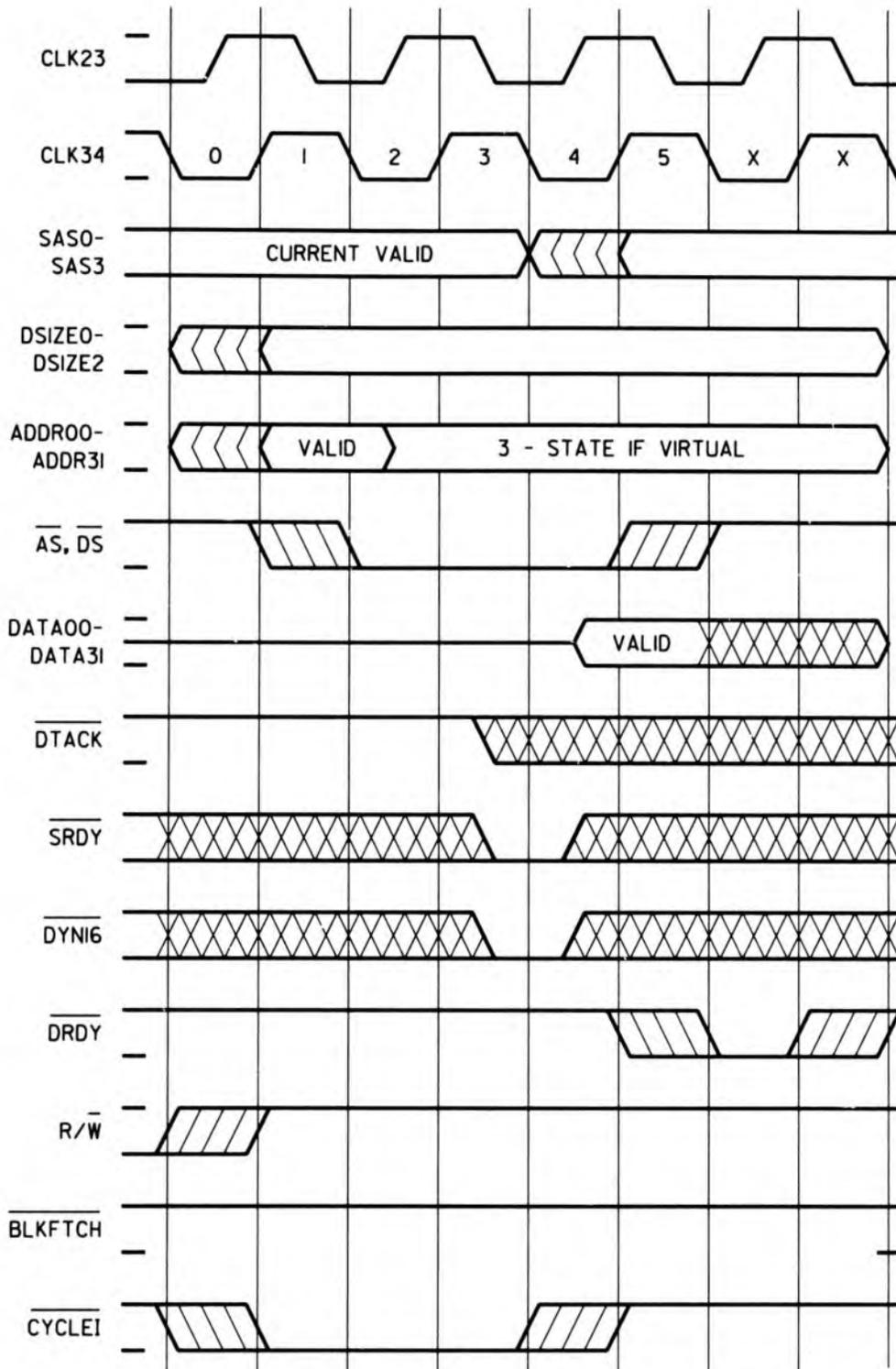
For read operations, the address strobe (\overline{AS}) and data strobe (\overline{DS}) have the same timing. The CPU latches data driven onto the data bus by the addressed device at the end of clock state four, at which time the CPU negates \overline{AS} and \overline{DS} . Data can be driven onto the bus while \overline{AS} and \overline{DS} are active.

The transaction illustrated on Figure 4-2 is terminated by the assertion of \overline{SRDY} by the addressed device. \overline{SRDY} is the acknowledgement that the addressed device is putting the data onto the data bus and that the CPU can latch the data and terminate the transaction.

\overline{SRDY} is synchronously sampled at the end of clock state three. The read transaction depicted on Figure 4-2 completes in three clock cycles (zero wait cycles) because \overline{SRDY} is active when sampled at the end of the clock state three. In addition, Figure 4-2 illustrates a read transaction using $\overline{DYN16}$ qualified with \overline{SRDY} to indicate communications with a 16-bit port. $\overline{DYN16}$ is asserted at the same time as \overline{SRDY} .

BUS OPERATION

Synchronous Read Transaction



Note: Zero wait cycles.

* $\overline{\text{SRDY}}$ can be asserted by itself or qualified with $\overline{\text{DYN16}}$. When $\overline{\text{DYN16}}$ is used, only DATA16—DATA31 are valid.

Figure 4-2. Synchronous Read Transaction

BUS OPERATION

Asynchronous Read Transaction

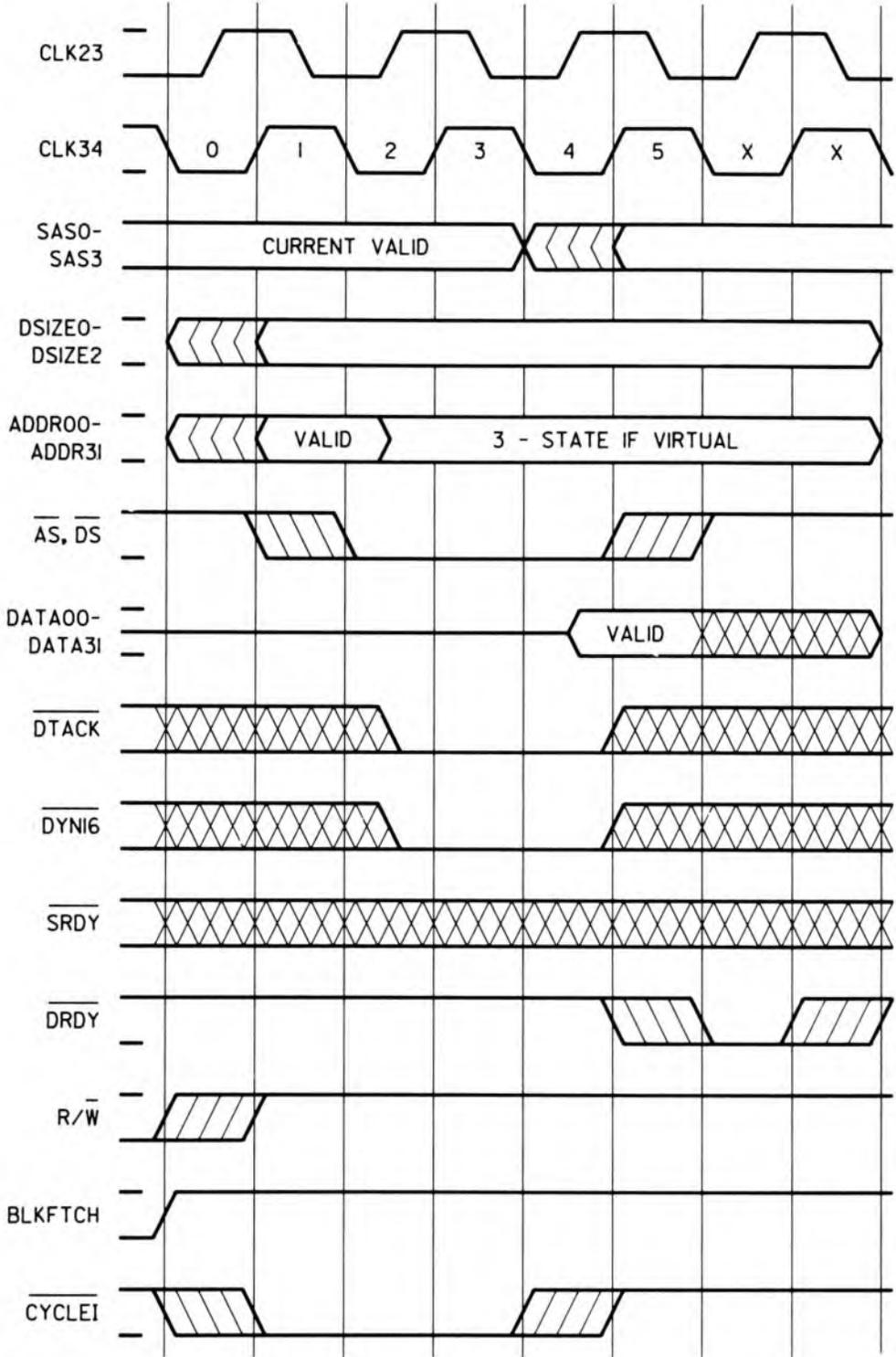
4.2.2 Asynchronous Read Transaction

The asynchronous read transaction using \overline{DTACK} is identical to that using \overline{SRDY} , except that the addressed device asserts \overline{DTACK} instead of \overline{SRDY} to acknowledge that it is putting data on the data bus (see Figure 4-3). \overline{DTACK} is asynchronously sampled at the end of clock state two and is double-latched to maintain stability.

The read transaction shown on Figure 4-3 completes in three clock cycles because the CPU samples \overline{DTACK} active at the end of clock state two. Upon sampling \overline{DTACK} active, the CPU latches the data and terminates the transaction.

When communicating with a 16-bit port, the $\overline{DYN16}$ signal can be asserted at the same time as \overline{DTACK} , after \overline{DTACK} , or by itself. Figure 4-3 shows $\overline{DYN16}$ being qualified with \overline{DTACK} . Figure 4-4 shows $\overline{DYN16}$ asserted at the synchronous sampling point after \overline{DTACK} was sampled at the asynchronous sample point. Figure 4-5 shows $\overline{DYN16}$ being used to terminate a read transaction. $\overline{DYN16}$ is sampled at the asynchronous sample point.

BUS OPERATION
Asynchronous Read Transaction



Note: Zero wait cycles.

* \overline{DTACK} can be asserted by itself or qualified with $\overline{DYN16}$. When $\overline{DYN16}$ is used, only DATA16—DATA31 are valid.

Figure 4-3. Asynchronous Read Transaction

BUS OPERATION

Asynchronous Read Transaction

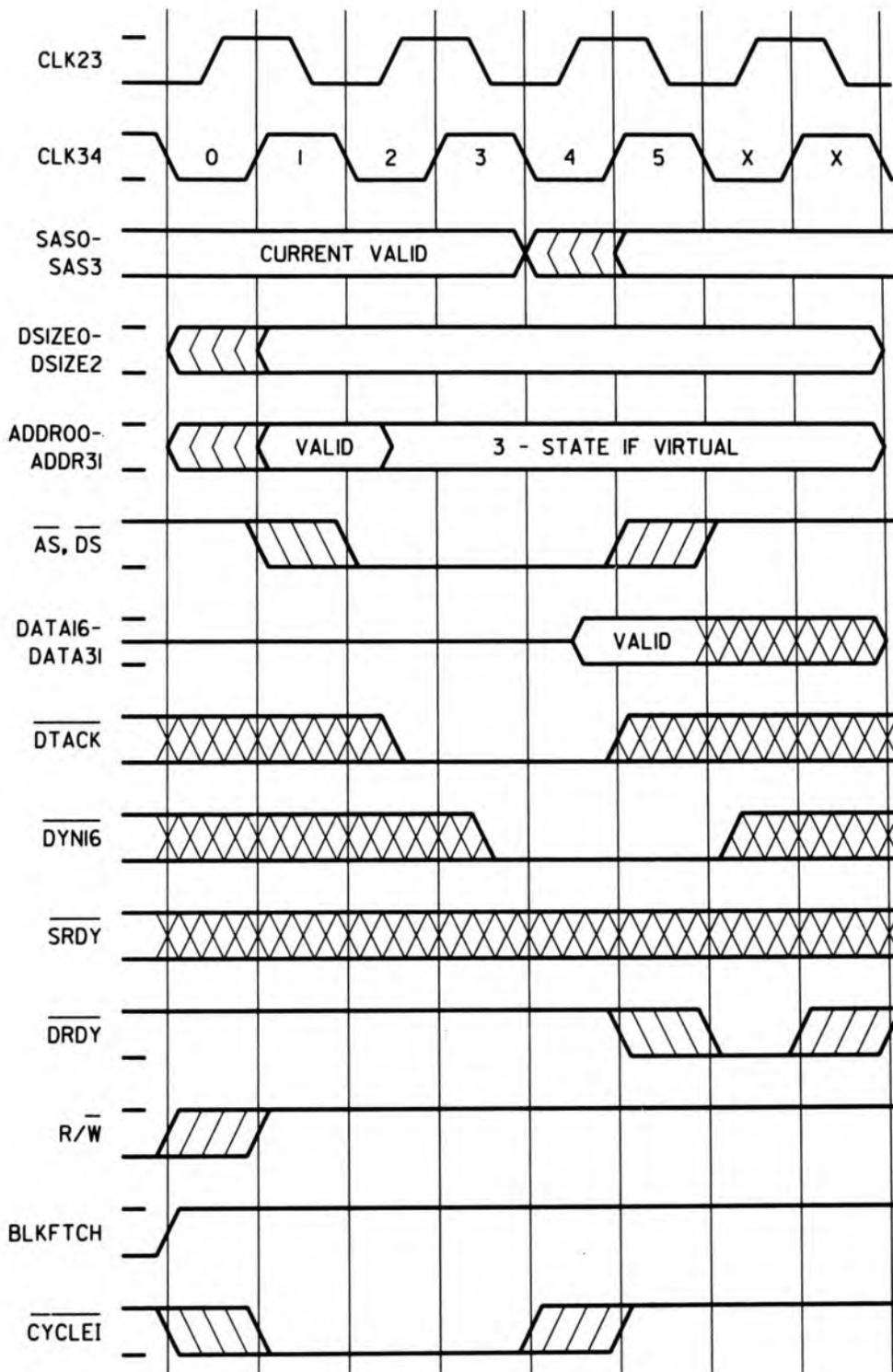


Figure 4-4. Asynchronous Read Transaction – $\overline{\text{DYN16}}$ Asserted After $\overline{\text{DTACK}}$

BUS OPERATION
Asynchronous Read Transaction

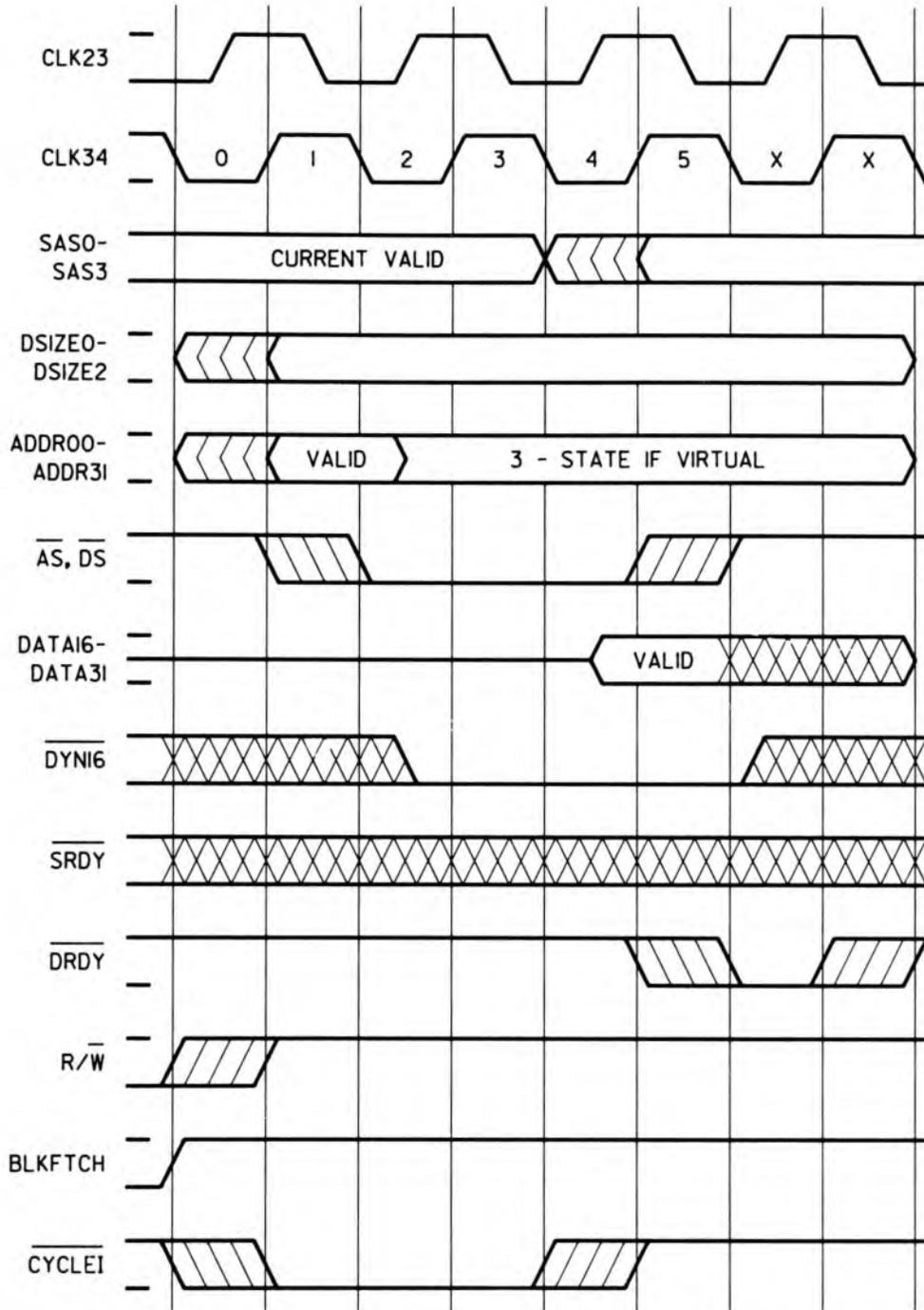


Figure 4-5. Asynchronous Read Transaction – $\overline{\text{DYN16}}$ Only

BUS OPERATION

Synchronous Read Transaction with Wait Cycle

4.2.3 Synchronous Read Transaction with Wait Cycle

The CPU inserts wait cycles during bus transactions if it does not sample $\overline{\text{DTACK}}$ active at the end of clock state two or $\overline{\text{SRDY}}$ active at the end of clock state three, no bus exceptions occur, and if $\overline{\text{DYN16}}$ is not asserted. As illustrated on Figure 4-6, the CPU inserts one wait cycle because $\overline{\text{DTACK}}$ is not active at the end of clock state two and $\overline{\text{SRDY}}$ is not active at the end of clock state three. Only one wait cycle is inserted during the transaction because $\overline{\text{SRDY}}$ is active when sampled at the end of the wait cycle. The CPU then latches the data and terminates the transaction.

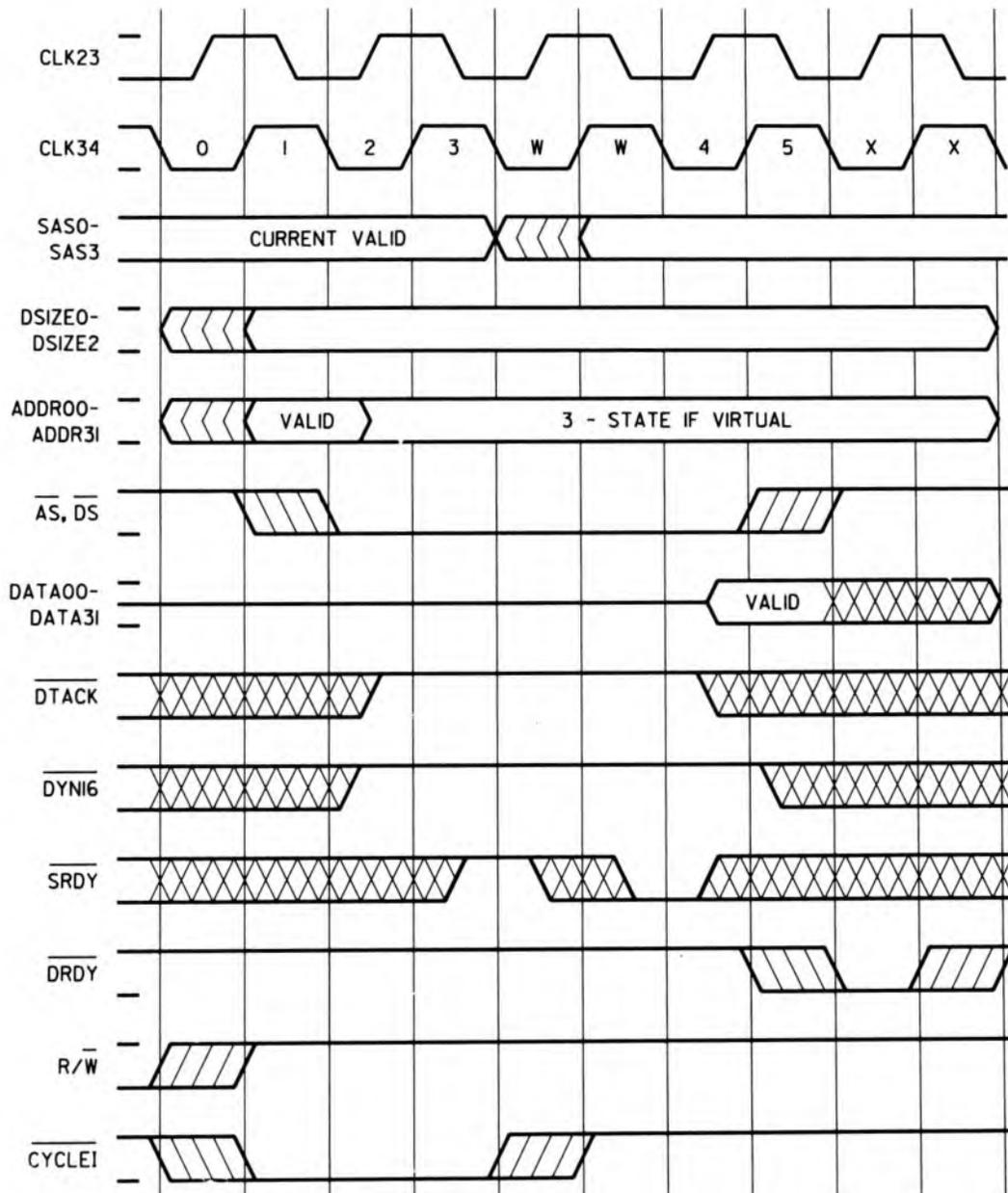


Figure 4-6. Synchronous Read Transaction with One Wait Cycle

4.2.4 Asynchronous Read Transaction with Two Wait Cycles

The CPU can insert multiple wait cycles during bus transactions, as illustrated on Figure 4-7. In this figure the CPU does not receive an acknowledge (\overline{DTACK} , $\overline{DYN16}$, or \overline{SRDY}) for two clock cycles. Neither \overline{DTACK} , $\overline{DYN16}$, or \overline{SRDY} is active during clock states two and three or the first wait cycle. \overline{DTACK} is sampled active in the middle of the second wait cycle, causing the termination of wait cycle generation. The CPU then latches the data and terminates the transaction.

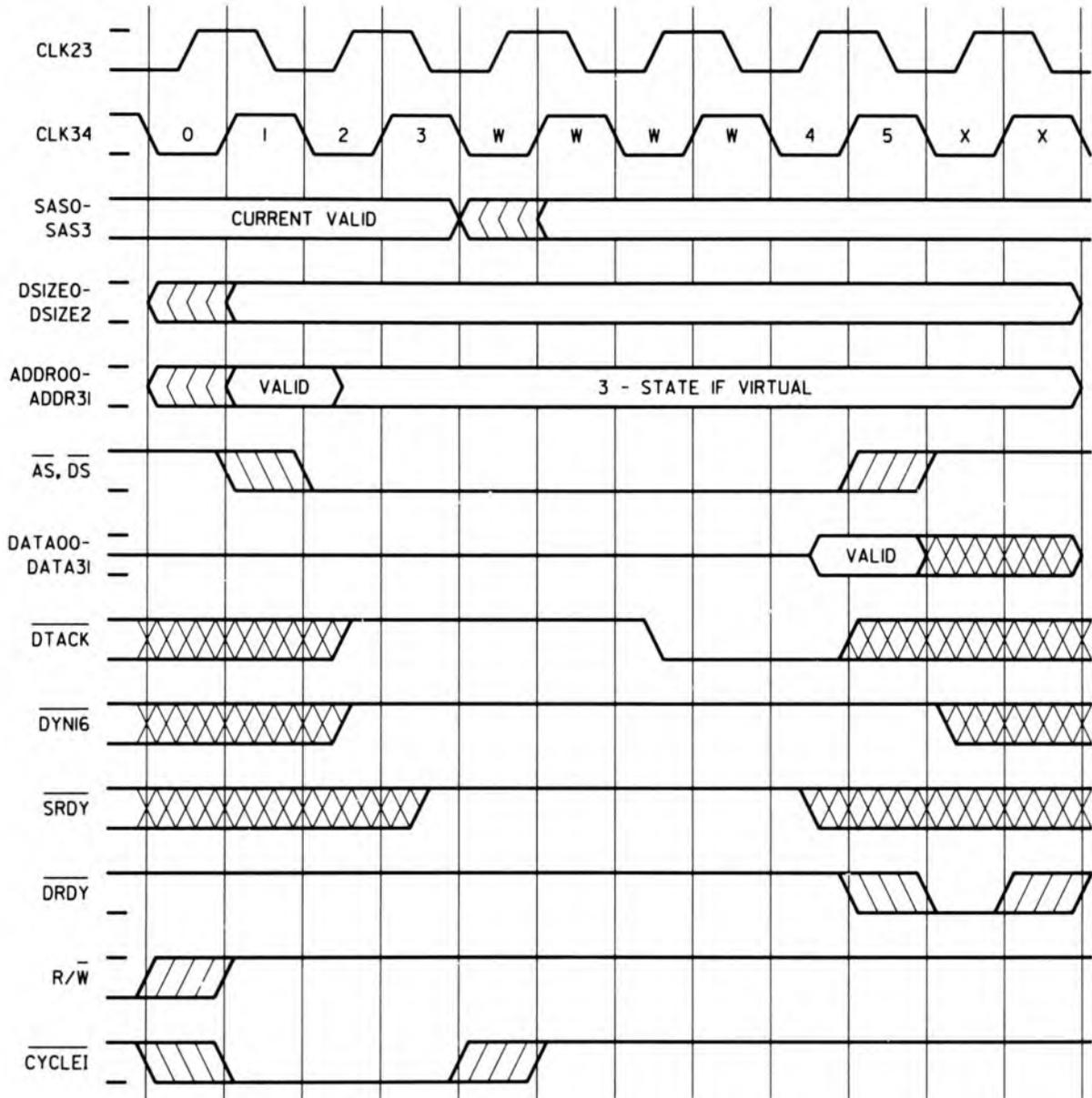


Figure 4-7. Asynchronous Read Transaction with Two Wait Cycles

BUS OPERATION

Synchronous Write Transaction

4.2.5 Synchronous Write Transaction

During synchronous write transactions, the R/\overline{W} output is held low (logic 0) for the entire transaction. The CPU drives the data bus with the data to be written from clock state two until the end of the transaction. The access status code at the beginning of a write transaction is write (SAS3—SAS0 = 1010).

Unlike read transactions in which \overline{AS} and \overline{DS} have the same timing, the CPU asserts \overline{DS} one cycle after it has asserted \overline{AS} , allowing the addressed device to latch the data with either the leading or trailing edge of \overline{DS} .

Figure 4-8 illustrates a synchronous write transaction in which the addressed device uses \overline{SRDY} as the acknowledgement. By asserting \overline{SRDY} , the addressed device indicates to the CPU that it is ready to latch the data that is on the data bus. \overline{SRDY} is synchronously sampled at the end of clock state three. On Figure 4-8, the CPU sampled \overline{DTACK} inactive at the end of clock state two; however, it sampled \overline{SRDY} active at the end of clock state three. As a result, the CPU terminated the transaction.

In addition, Figure 4-8 illustrates a write transaction using $\overline{DYN16}$ qualified with \overline{SRDY} to indicate communications with a 16-bit port. $\overline{DYN16}$ is asserted at the same time as \overline{SRDY} .

4.2.6 Asynchronous Write Transaction

The asynchronous write transaction that uses \overline{DTACK} is identical to that which uses \overline{SRDY} , except that the addressed device asserts \overline{DTACK} to indicate that it is ready to latch the data that is on the data bus. \overline{DTACK} is sampled asynchronously at the end of clock state two. On Figure 4-9, the CPU samples \overline{DTACK} active at the end of clock state two and proceeds to terminate the transaction.

When communicating with a 16-bit port, the $\overline{DYN16}$ signal can be asserted at the same time as \overline{DTACK} , after \overline{DTACK} , or without \overline{DTACK} . Figure 4-9 shows $\overline{DYN16}$ being qualified with the asynchronous acknowledge \overline{DTACK} . Figure 4-10 shows $\overline{DYN16}$ being asserted at the synchronous sampling point. Figure 4-11 shows a write transaction using $\overline{DYN16}$ without \overline{DTACK} .

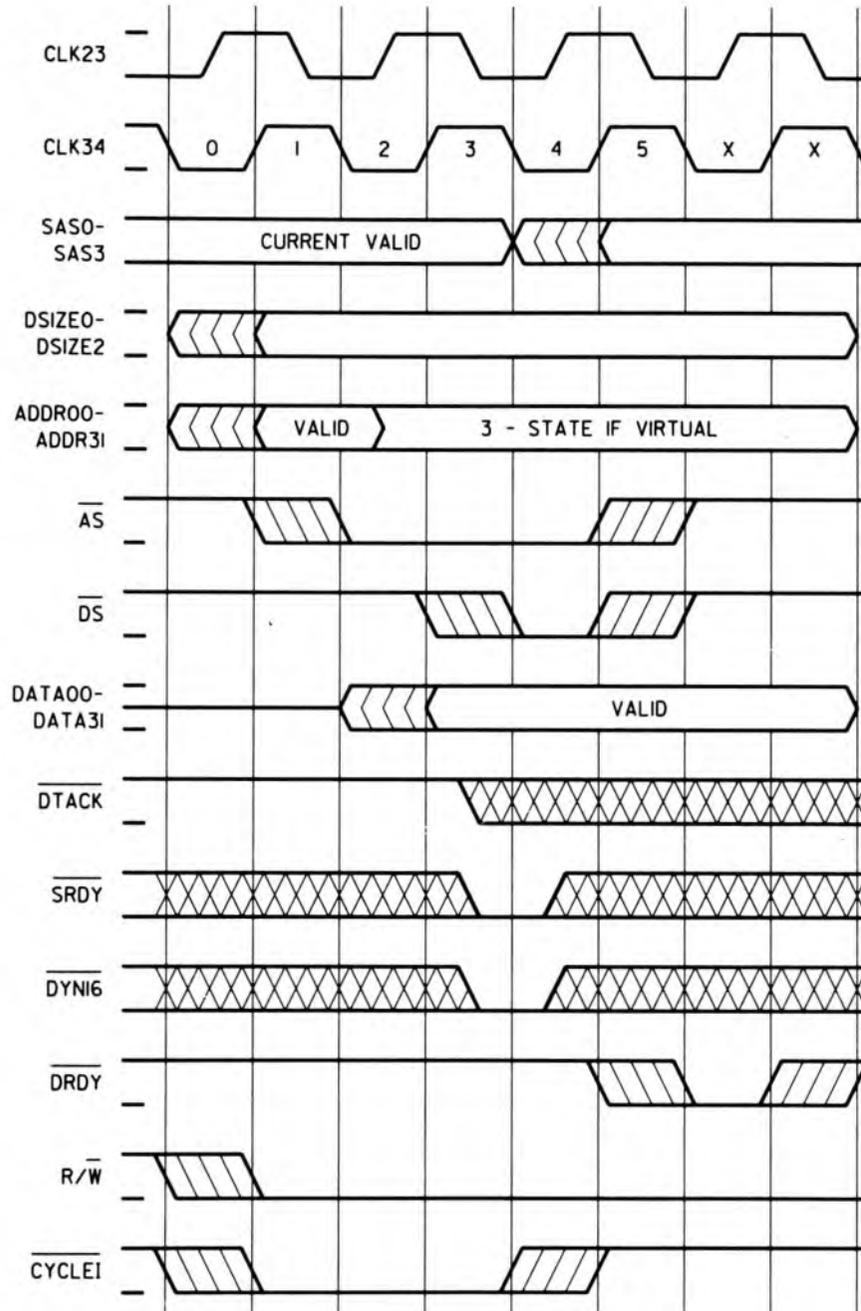
4.2.7 Write Transaction with Wait Cycle – \overline{SRDY} Only

A wait cycle insertion for write transactions is similar to a wait cycle insertion for read transactions. As in read transactions, the CPU inserts wait cycles if \overline{DTACK} is not active when sampled at the end of clock state two, \overline{SRDY} is not active when sampled at the end of clock state three, no bus exceptions occurred, and $\overline{DYN16}$ is not asserted.

BUS OPERATION

Synchronous Write Transaction

Figure 4-12 illustrates a write transaction with two wait cycles. The CPU begins wait cycle insertion because \overline{DTACK} is not active at the end of state two and \overline{SRDY} is not active at the end of state three. A second wait cycle is inserted because, again, neither input was active when sampled during the first wait cycle. The addressed device finally asserts \overline{SRDY} at the end of the second wait cycle, and the CPU terminates the transaction.

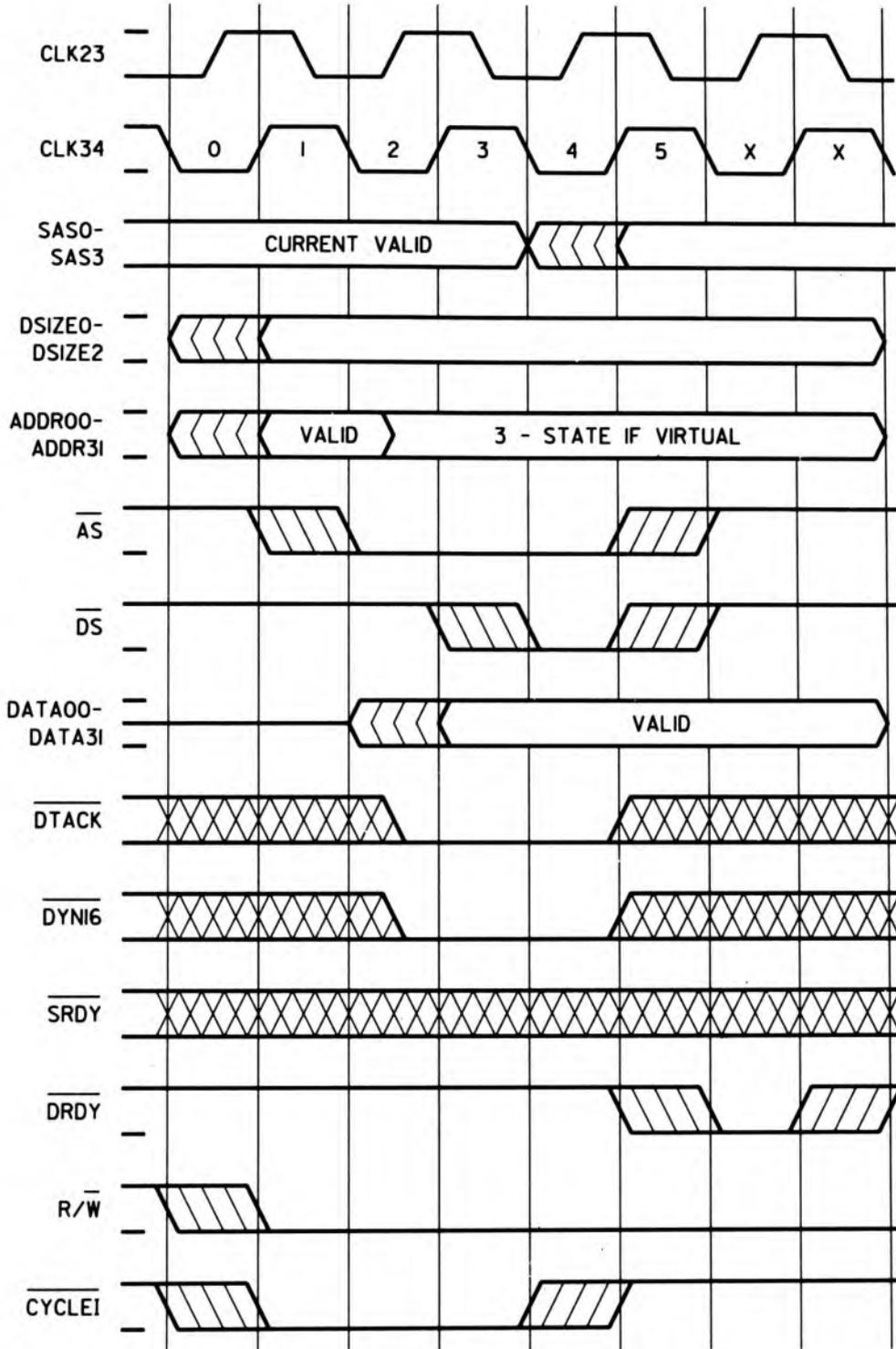


Note: Zero wait cycles.

* \overline{SRDY} can be asserted by itself or qualified with $\overline{DYN16}$.
When $\overline{DYN16}$ is used, only DATA16—DATA31 are valid.

Figure 4-8. Synchronous Write Transaction

BUS OPERATION
Asynchronous Write Transaction



Note: Zero wait cycles.

* \overline{DTACK} can be asserted by itself or qualified with $\overline{DYN16}$. When $\overline{DYN16}$ is used, only DATA16—DATA31 are valid.

Figure 4-9. Asynchronous Write Transaction

BUS OPERATION
Asynchronous Write Transaction

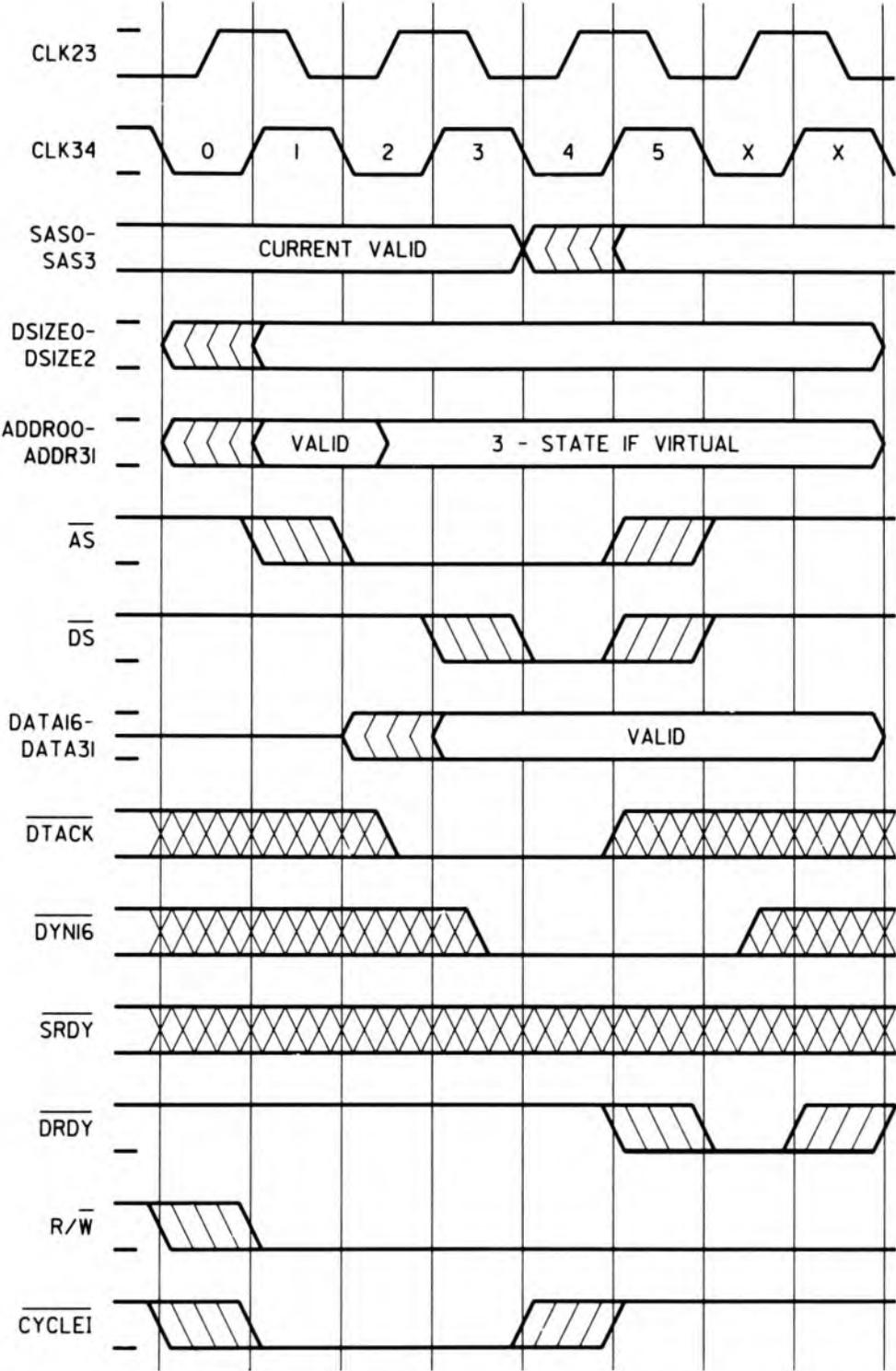


Figure 4-10. Asynchronous Write Transaction – $\overline{\text{DYN16}}$ Asserted After $\overline{\text{DTACK}}$

BUS OPERATION
Asynchronous Write Transaction

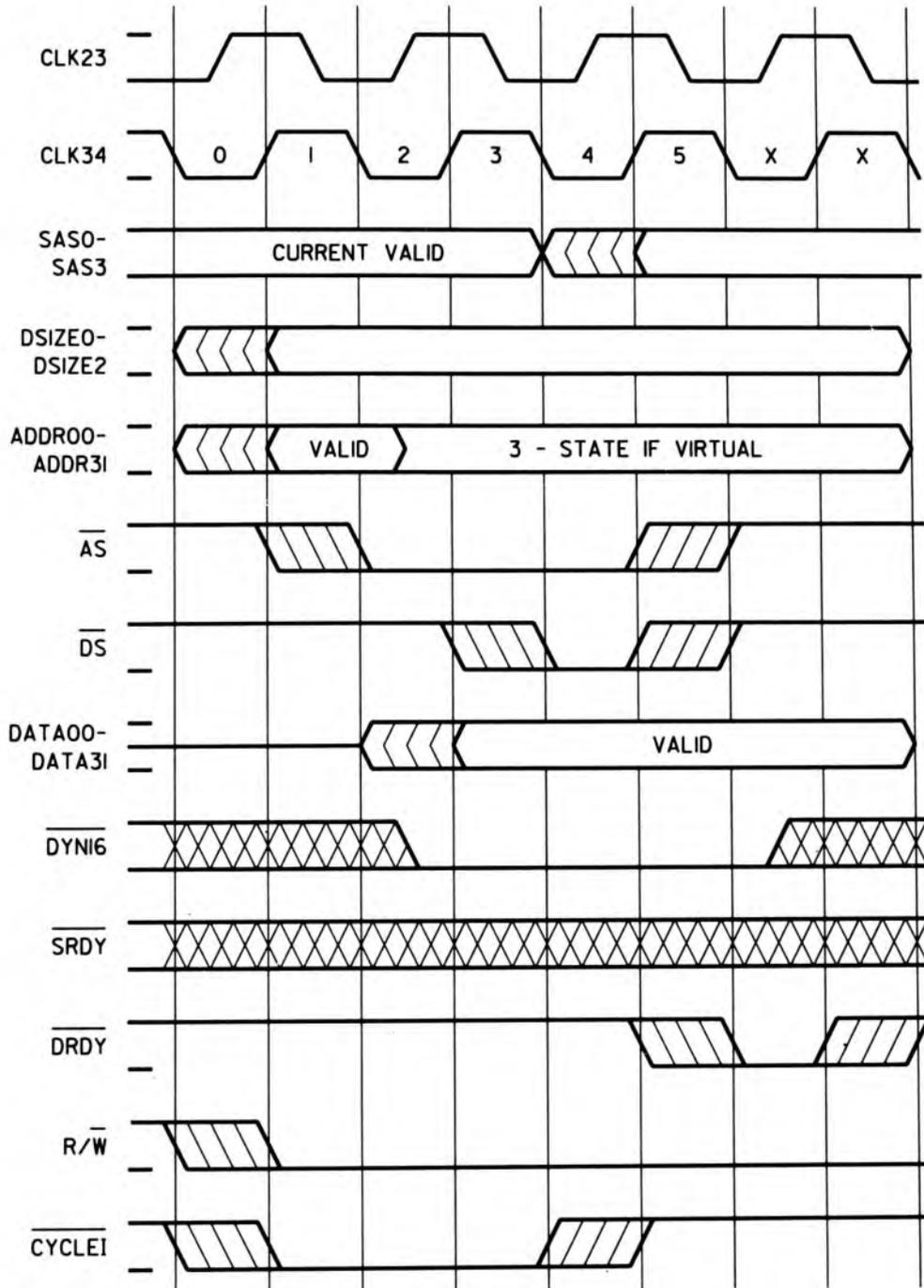


Figure 4-11. Asynchronous Write Transaction – $\overline{\text{DYN16}}$ Only

BUS OPERATION

Write Transaction

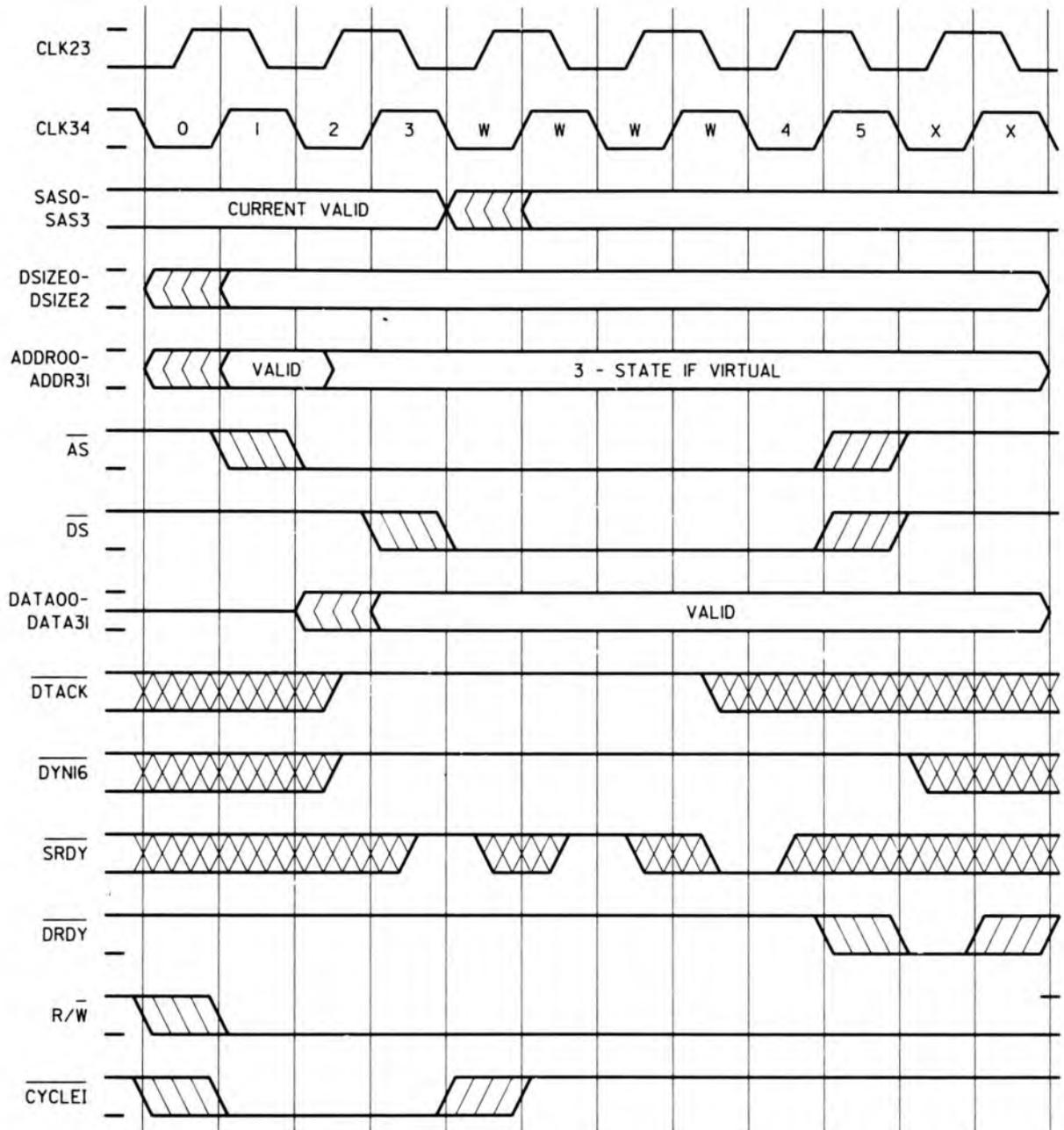


Figure 4-12. Write Transaction with Two Wait Cycles – $\overline{\text{SRDY}}$ Only

BUS OPERATION

Write Transaction

4.2.8 Write Transaction with Wait Cycle Using $\overline{\text{DTACK}}$

The write transaction shown on Figure 4-13 is another example of wait cycle insertion. In this transaction, the addressed device asserts $\overline{\text{DTACK}}$ to indicate that it is ready to latch the data and that, therefore, no more wait cycles are to be inserted.

Neither $\overline{\text{DTACK}}$ nor $\overline{\text{SRDY}}$ is active at its initial sampling point and, as a result, the CPU inserts a wait cycle. When the CPU samples $\overline{\text{DTACK}}$ a second time during the wait cycle, $\overline{\text{DTACK}}$ is active. The CPU can then terminate the transaction.

Additional protocol diagrams for read and write operations are provided in section 4.14.

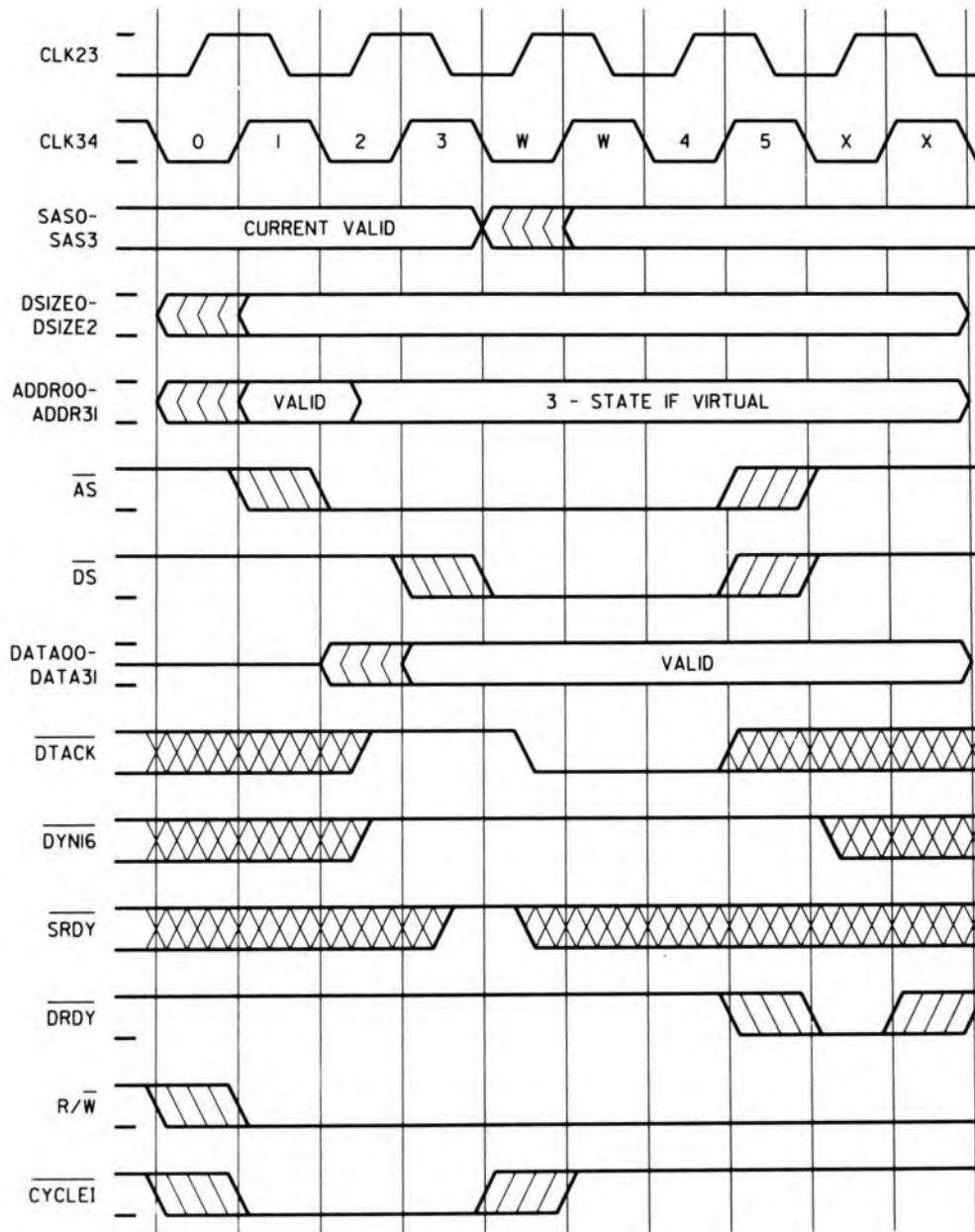


Figure 4-13. Write Transaction with Wait Cycle – $\overline{\text{DTACK}}$ Only

4.3 READ-INTERLOCKED OPERATION

A read-interlocked operation consists of a memory fetch (read access) and one or more internal microprocessor operations, followed by a write access to the same memory location. Once the read access has been completed, the read-interlocked operation may not be preempted except by a reset. This prevents another process from altering data in memory that is being operated on by the current process. If a fault occurs during the read access, the read-interlocked operation terminates without going through the write access.

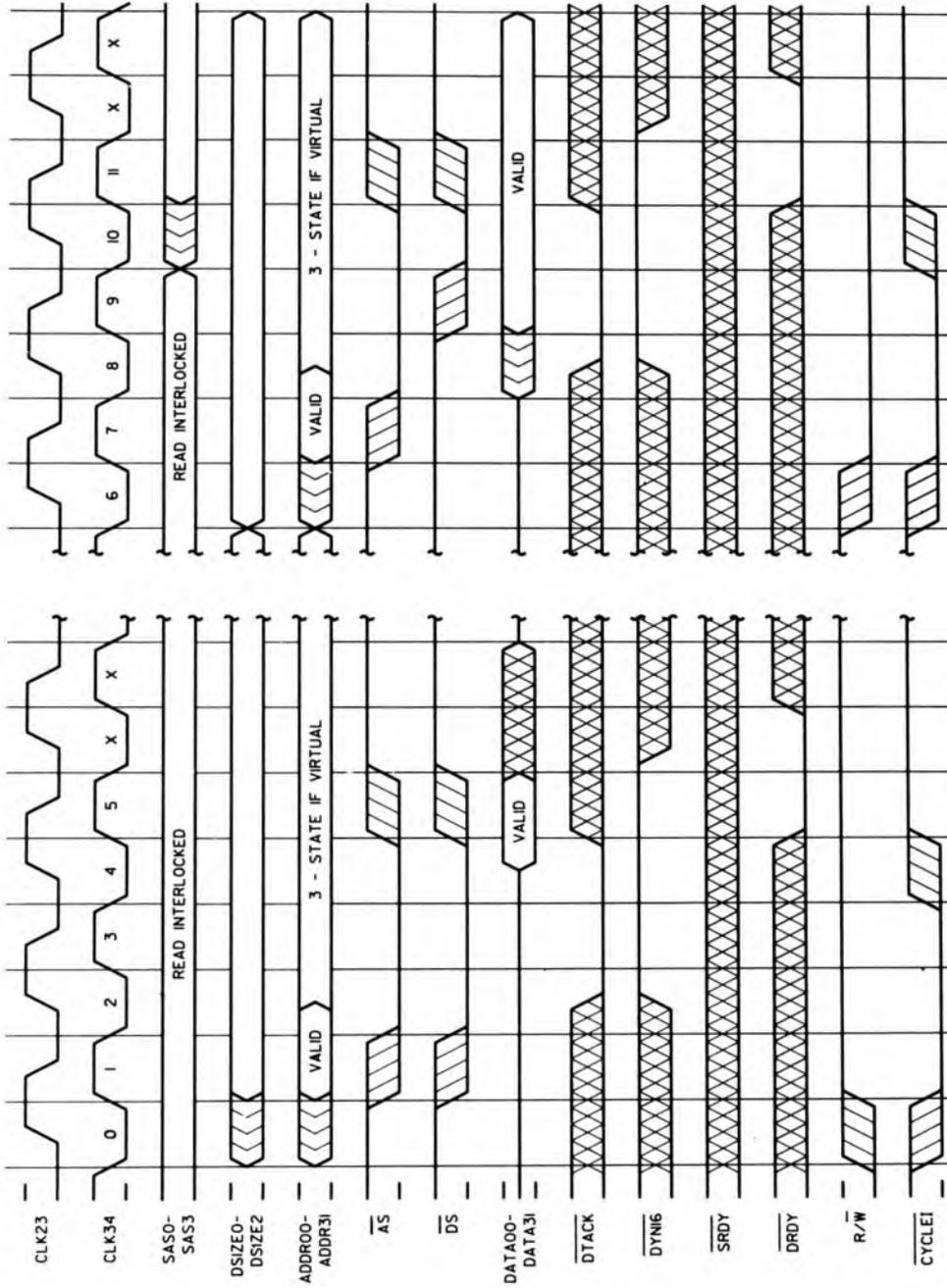
Figure 4-14 illustrates a read-interlocked transaction. Note that the access status code is read-interlocked (SAS3—SAS0 = 0111) for both transactions and that the address remains the same for both transactions. The read portion and the write portion of the transaction are standard read and write transactions.

4.4 COMPARE AND SWAP-INTERLOCKED TRANSACTION

The compare and swap interlocked transaction is the same as the read-interlocked transaction, except that it is dependent on a comparison. A write may or may not occur, depending upon the result of the comparison (see Figure 4-15).

BUS OPERATION

Compare and Swap-Interlocked Transaction



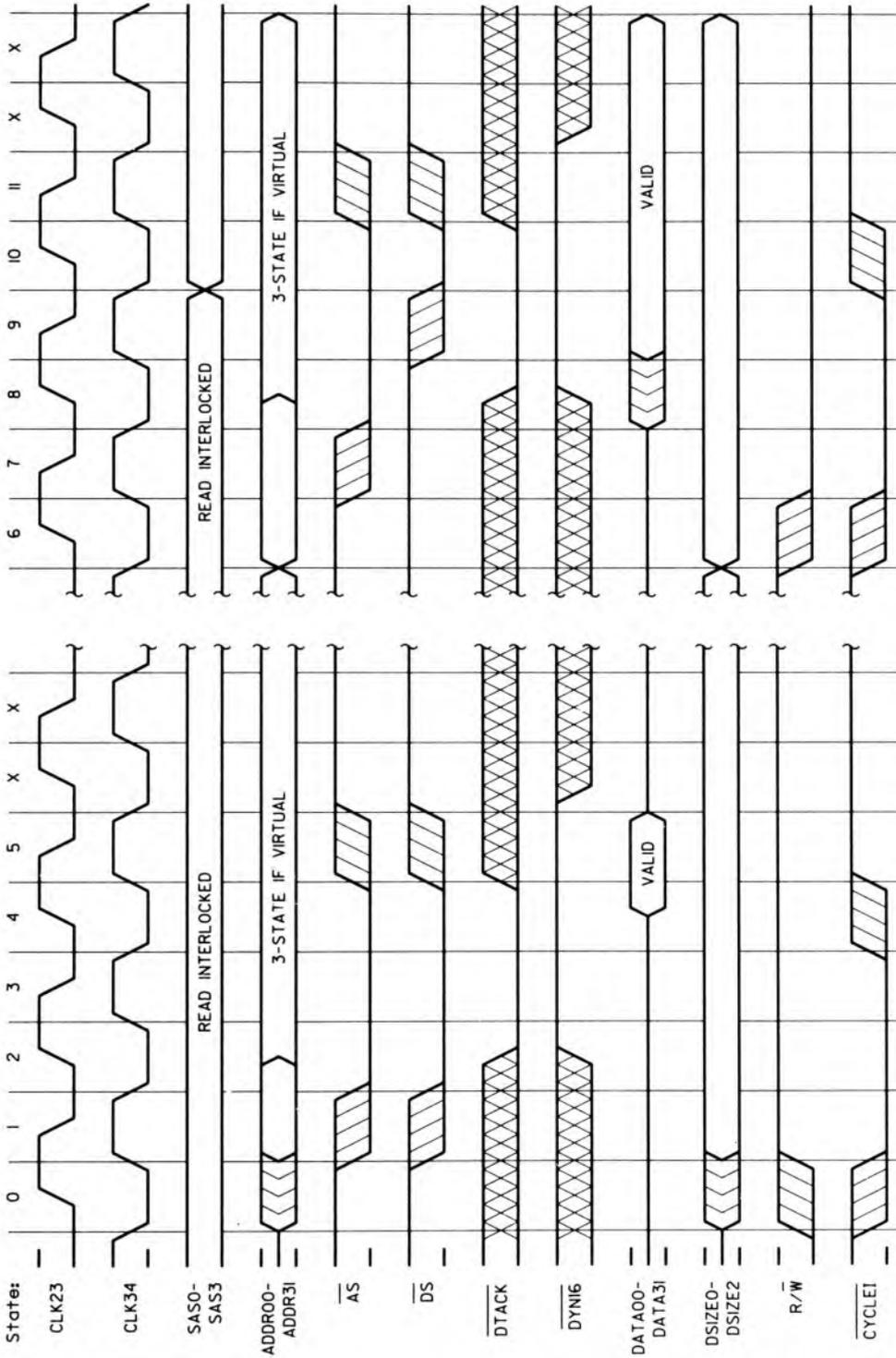
Notes:

Number of cycles between the read transaction and write transaction is four for swap-word interlocked (SWAPWI), and six for swap-halfword interlocked (SWAPHI) and swap-byte interlocked (SWAPBI) instructions.
Zero wait cycles.

Figure 4-14. Read-Interlocked Transaction – DTACK Only

BUS OPERATION

Compare and Swap-Interlocked Transaction



Notes:

Number of cycles between the read transaction and write transaction is four for swap-word interlocked (SWAPWI), and six for swap-halfword interlocked (SWAPHI) and swap-byte interlocked (SWAPBI) instructions. Zero wait cycles.

Figure 4-15. Compare and Swap-Interlocked Transaction – \overline{DTACK} Only

BUS OPERATION

Blockfetch Operation

4.5 BLOCKFETCH OPERATION

The CPU can fetch two words of instruction code in one bus transaction via a blockfetch operation. The CPU generates one address, and the memory provides two words of instruction code. This reduces the number of cycles that it takes to fetch two words. The CPU starts the transaction with a data size (DSIZE0—DSIZE2) of double-word, which indicates that it is ready to perform a blockfetch.

If the memory is designed to handle blockfetch, it responds with a blockfetch ($\overline{\text{BLKFTCH}}$) signal and an acknowledge signal, either $\overline{\text{SRDY}}$ or $\overline{\text{DTACK}}$. The $\overline{\text{BLKFTCH}}$ signal cannot be asserted with the $\overline{\text{DYN16}}$ signal.

4.5.1 Blockfetch Transaction Using $\overline{\text{SRDY}}$

After the memory issues $\overline{\text{BLKFTCH}}$ and $\overline{\text{SRDY}}$, the CPU latches the data being sourced by the memory during clock state four, negates $\overline{\text{DS}}$, and keeps $\overline{\text{AS}}$ in the active state. One cycle later, the CPU reissues $\overline{\text{DS}}$ and is ready to latch the second word.

The memory drives the data bus with the second word and asserts $\overline{\text{SRDY}}$. The CPU samples $\overline{\text{SRDY}}$ at the end of clock state seven, and then latches the data during clock state eight and terminates the transaction. This operation is shown on Figure 4-16.

The $\overline{\text{AS}}$ signal stays low for both words fetched. The $\overline{\text{DS}}$ signal goes inactive for one cycle in between the first and second words. The data size (DSIZE0—DSIZE2) changes from double word to word at clock state six. The R/ $\overline{\text{W}}$ signal is held in the read mode for the entire transaction. Only one $\overline{\text{CYCLEI}}$ is issued for this transaction. Two $\overline{\text{DRDY}}$ s are issued, one for each word. The $\overline{\text{BLKFTCH}}$ signal is sampled only with the first $\overline{\text{SRDY}}$, and is not used during the second word. The access status code (SAS) for the first word can be instruction fetch, instruction fetch after PC discontinuity, or prefetch. SAS for the second word is always prefetch. If the memory does not issue a $\overline{\text{BLKFTCH}}$ with the acknowledgement on the first word, the CPU latches the data and terminates the transaction by removing both $\overline{\text{AS}}$ and $\overline{\text{DS}}$. It then proceeds to start up a second read with an SAS of prefetch and issues a new address.

For a blockfetch transaction, the CPU issues only one address. If it is in virtual mode, the CPU 3-states the address during clock state two, which allows the MMU to drive the physical address for fetching both words. Note also that the CPU fetches the two words from a double-word address block. The CPU asks for either the even or odd address first, as indicated by the value on the address bus. For the second word, it expects the memory to provide the data corresponding to the address location with ADDR02 complemented. This address is the other corresponding word from the double word block.

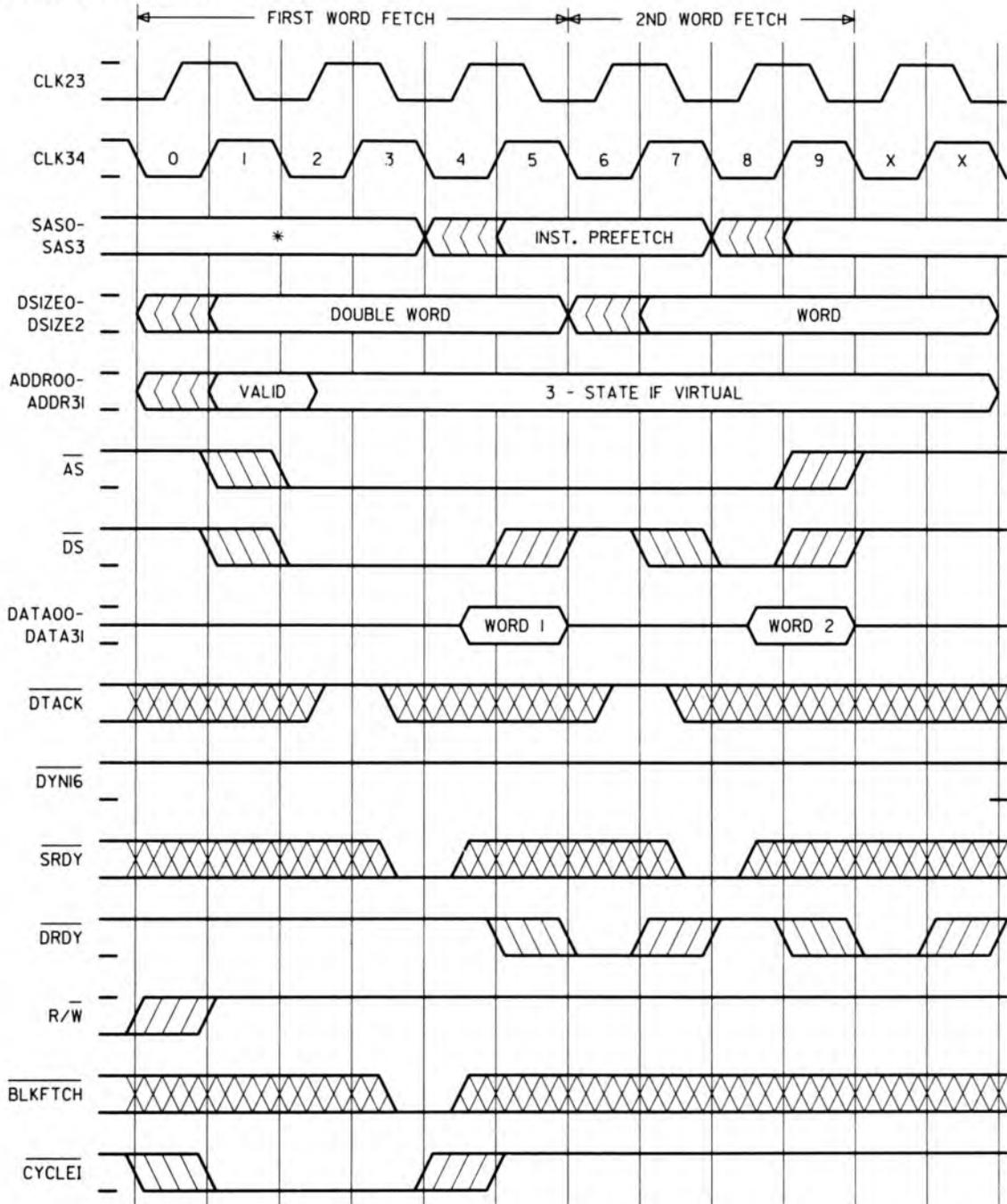
For example, assuming physical addressing mode, the CPU drives the address bus with 0003C000. Memory provides data for the first word corresponding to location

BUS OPERATION

Blockfetch Transactions

0003C000. Memory provides data for the second word corresponding to location 0003C004.

As another example of physical addressing mode, consider the CPU drives the address bus with 00078004. Memory provides data for the first word corresponding to location 00078004. Memory provides data for the second word corresponding to location 00078000.



Note: Zero wait cycles.

* Instruction fetch/prefetch or instruction fetch after PC discontinuity.

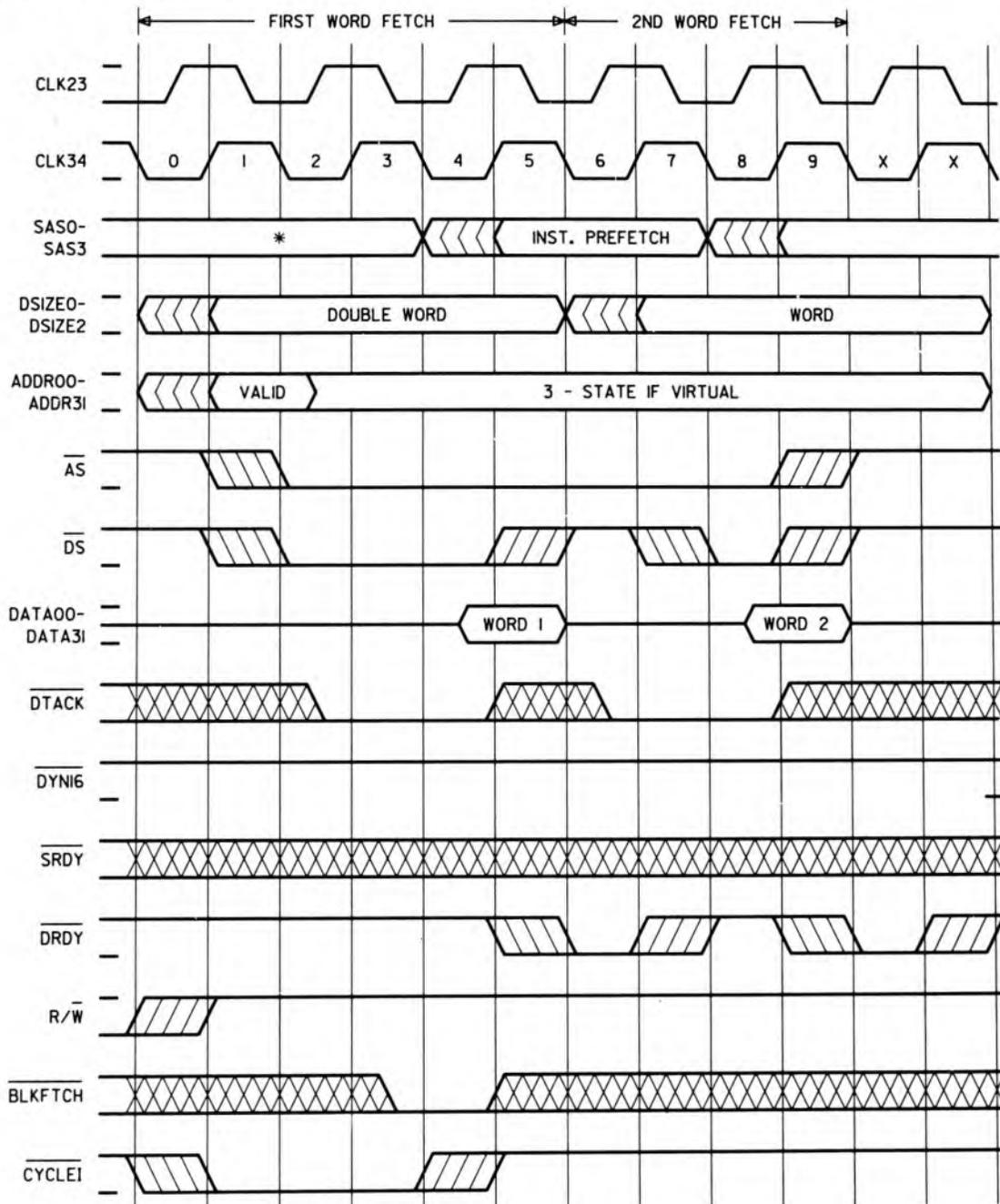
Figure 4-16. Blockfetch Transaction – $\overline{\text{SRDY}}$ Only

BUS OPERATION

Blockfetch Transactions

4.5.2 Blockfetch Transaction Using \overline{DTACK}

This transaction (see Figure 4-17) is the same as blockfetch using \overline{SRDY} , Figure 4-16, except that the acknowledgement used by the memory is \overline{DTACK} . On the first word, the CPU samples \overline{DTACK} at the end of clock state two. $\overline{BLKFTCH}$ is sampled at the end of clock state three, the same as on Figure 4-16. For the second word, \overline{DTACK} is sampled at the end of clock state six.



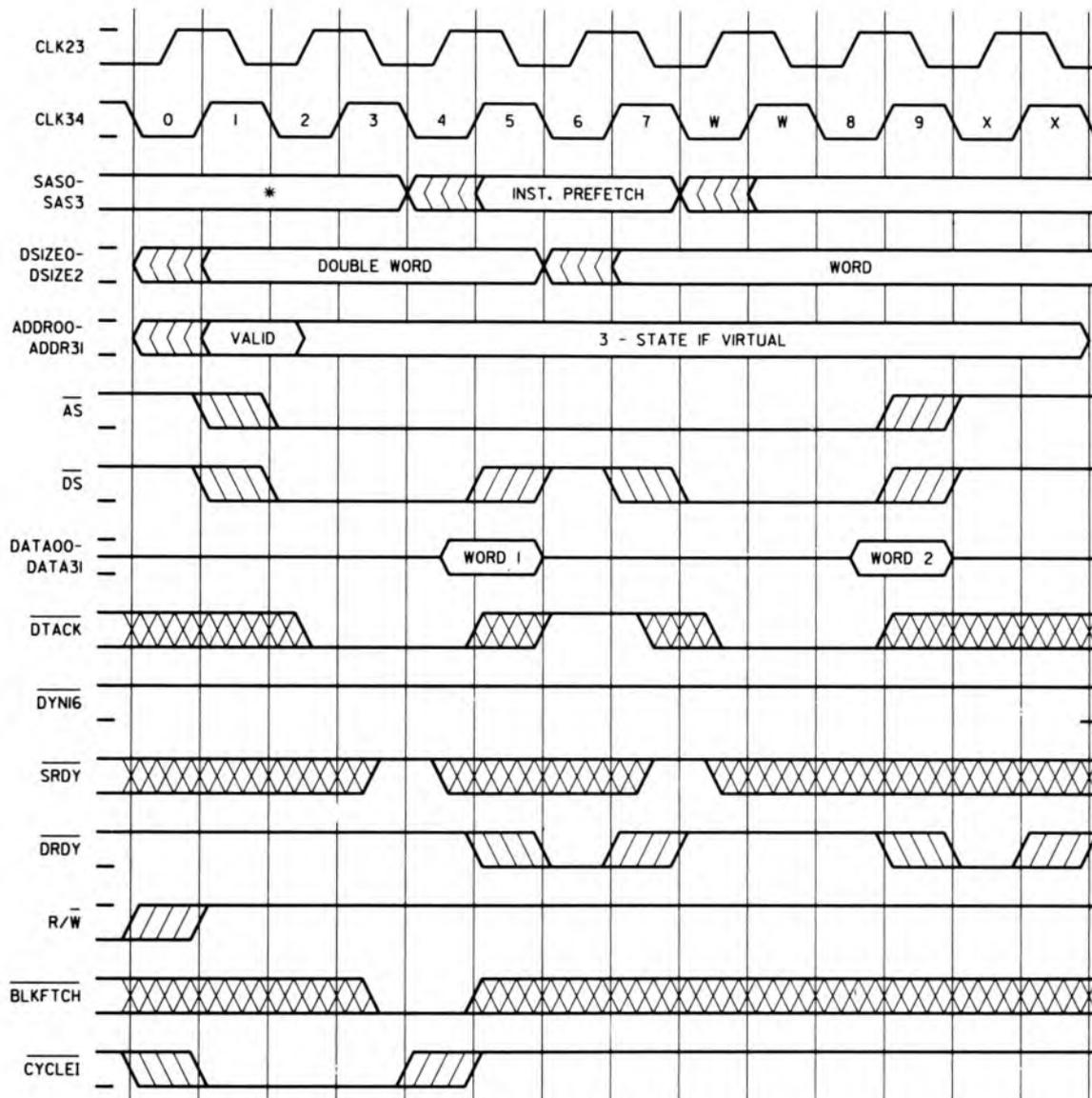
Note: Zero wait cycles.

* Instruction fetch/prefetch or instruction fetch after PC discontinuity.

Figure 4-17. Blockfetch Transaction – \overline{DTACK} Only

4.5.3 Blockfetch Transaction Using $\overline{\text{DTACK}}$ with Wait Cycle On Second Word

In this case (see Figure 4-18), during the fetch of the second word there was no $\overline{\text{DTACK}}$ during clock state six nor SRDY during clock state seven. Therefore, the CPU inserted a wait cycle. It sampled $\overline{\text{DTACK}}$ during the first wait state (W), and then latched the data and terminated the transaction.



Note: Zero wait cycles.

* Instruction fetch/prefetch or instruction fetch after PC discontinuity.

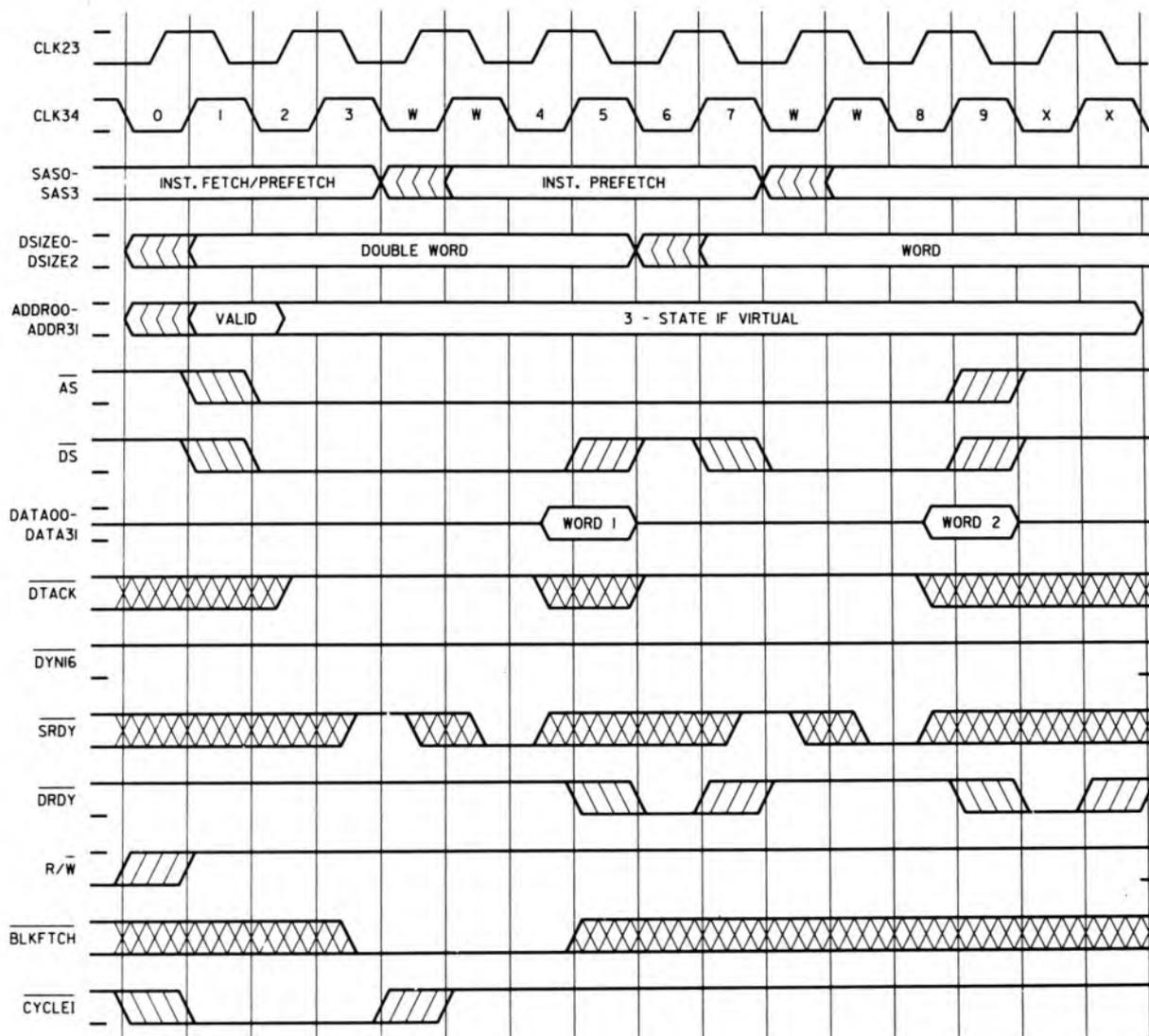
Figure 4-18. Blockfetch Transaction – $\overline{\text{DTACK}}$ Only, with Wait Cycle on Second Word

BUS OPERATION

Blockfetch Transactions

4.5.4 Blockfetch Transaction Using $\overline{\text{SRDY}}$ with Wait Cycles On Both Words

During the fetch of the first word in this transaction, the CPU did not sample $\overline{\text{DTACK}}$ active during clock state two or $\overline{\text{SRDY}}$ active during clock state three (see Figure 4-19); therefore, it inserted a wait cycle. During the second wait state (W), it sampled $\overline{\text{SRDY}}$ and $\overline{\text{BLKFTCH}}$, and then latched the data and terminated $\overline{\text{DS}}$. The CPU proceeded to the second fetch. During clock state six, it did not sample $\overline{\text{DTACK}}$ or $\overline{\text{SRDY}}$ active during clock state seven; therefore, it inserted a wait cycle. During the second wait state (W), it sampled $\overline{\text{SRDY}}$ active, and then latched the data and terminated the transaction.



Note: Wait cycle on both words.

Figure 4-19. Blockfetch Transaction – $\overline{\text{SRDY}}$ Only, with Wait Cycles on Both Words

4.6 BUS EXCEPTIONS

Bus exceptions cause the termination of the current memory access and result when an access retry is required or when a fault occurs during an access. The three bus exceptions are fault, retry, and relinquish and retry.

A fault is the result of an error condition during a bus cycle. An external device reports errors to the CPU (such as address translation and memory faults) by asserting the fault ($\overline{\text{FAULT}}$) input. This causes the CPU to terminate the access and possibly execute a fault handling routine. The WE 32201 Memory Management Unit uses the $\overline{\text{FAULT}}$ input when it detects that a virtual address corresponds to data that is not presently in physical memory. The MMU also generates a fault if it detects an error condition when it attempts to translate the virtual address. A retry causes the CPU to retry the access. An external device requests a retry by asserting the retry ($\overline{\text{RETRY}}$) input. A relinquish and retry request ($\overline{\text{RRREQ}}$) assertion causes the microprocessor to give up its bus and to retry the preempted access once the bus has been returned to its control. An external device requests a relinquish and retry by asserting the $\overline{\text{RRREQ}}$ input.

Table 4-1 describes how the microprocessor handles the simultaneous assertion of two or more bus exceptions. The term *negated* indicates the signal is driven to its inactive state.

Table 4-1. Simultaneously Asserted Exception Conditions*	
Simultaneously Asserted Signals	Behavior
$\overline{\text{RRREQ}}$, $\overline{\text{RETRY}}$, $\overline{\text{FAULT}}$	$\overline{\text{RRREQ}}$ is honored first. The microprocessor acknowledges this request by relinquishing the bus and then asserting the relinquish and retry request acknowledge ($\overline{\text{RRRACK}}$) output. The access is retried once $\overline{\text{RRREQ}}$ and $\overline{\text{RETRY}}$ are negated by the requesting devices. If the $\overline{\text{FAULT}}$ input is still asserted during the retried access, the fault is honored (recognized). The fault input is recognized only during the retried access.
$\overline{\text{RRREQ}}$, $\overline{\text{RETRY}}$	$\overline{\text{RRREQ}}$ is honored first. The microprocessor 3-states the appropriate signals and then asserts $\overline{\text{RRRACK}}$. The access is retried once $\overline{\text{RRREQ}}$ and $\overline{\text{RETRY}}$ are negated.
$\overline{\text{RRREQ}}$, $\overline{\text{FAULT}}$	Same as in behavior for $\overline{\text{RRREQ}}$, $\overline{\text{RETRY}}$, and $\overline{\text{FAULT}}$ simultaneously asserted.
$\overline{\text{RETRY}}$, $\overline{\text{FAULT}}$	The $\overline{\text{RETRY}}$ request is honored first. The $\overline{\text{FAULT}}$ is recognized on the retried access if it is still asserted.

* Table 4-1 applies only when the microprocessor is the bus master.

BUS OPERATION

Faults

4.6.1 Faults

A bus transaction can be terminated by a bus exception; in this case, $\overline{\text{FAULT}}$ without a $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, or $\overline{\text{DYN16}}$ (see Figure 4-20). On Figure 4-20, the CPU inserted two wait cycles because it did not receive an acknowledge or a bus exception. During the third wait state (W), the CPU asynchronously sampled $\overline{\text{FAULT}}$ and terminated the transaction. Note that if $\overline{\text{DTACK}}$ or $\overline{\text{DYN16}}$ was also sampled with the $\overline{\text{FAULT}}$ during the third wait state (W), the figure would not change.

Upon the faulted transaction, the CPU proceeds to the fault handler to process the exception. However, for a faulted prefetch, the CPU ignores the data, continues with its current instruction execution, and does not enter the fault handler. If the CPU needs this instruction later, it does an instruction fetch, and if this is also faulted, the CPU proceeds with the fault handler.

At the end of the transaction, $\overline{\text{DRDY}}$ is not issued starting with clock state five because the CPU sampled the $\overline{\text{FAULT}}$ signal. If this bus transaction is a write, the CPU samples $\overline{\text{FAULT}}$ in the same way as in a read. There are some differences for a blockfetch transaction. These special cases are discussed in section 4.7.

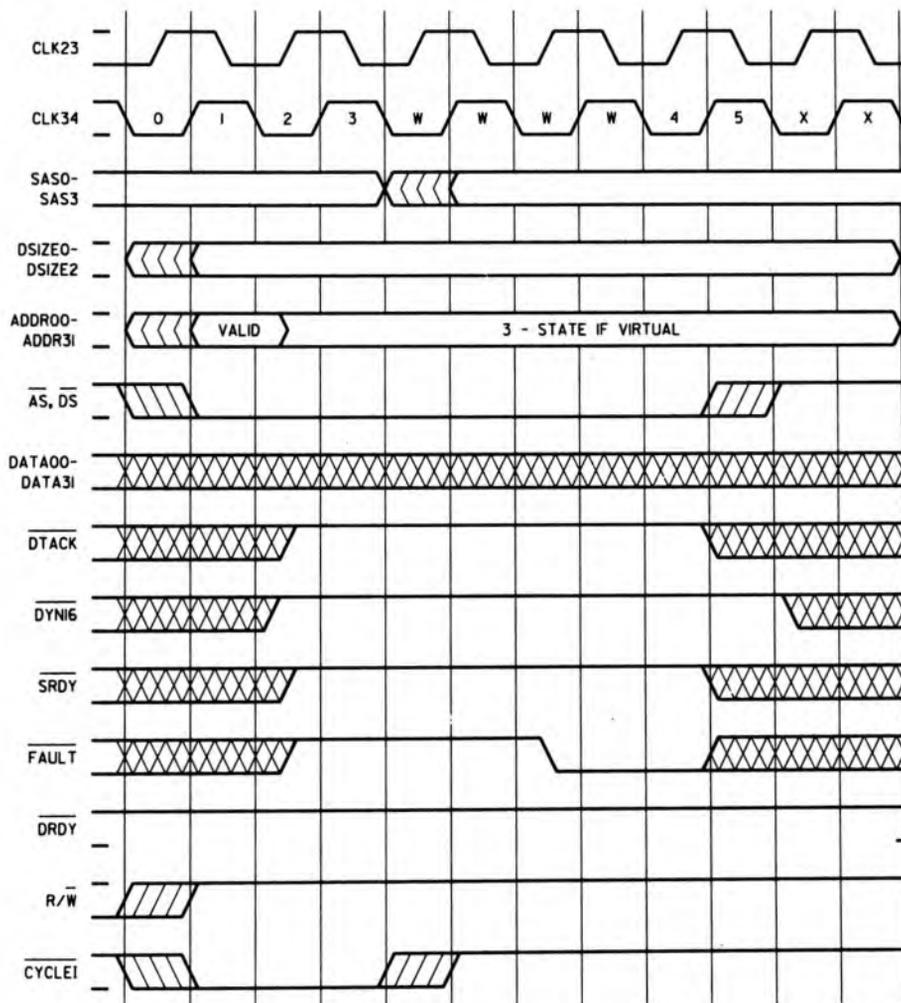


Figure 4-20. Asynchronous Fault without $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, and $\overline{\text{DYN16}}$ (Read Transaction)

FAULT With SRDY

The CPU can sample FAULT synchronously if both signals (FAULT and SRDY) are sampled active. Figure 4-21 shows both SRDY and FAULT being sampled during the last wait state (W). The CPU then terminates the transaction and does not issue a DRDY. For reads and writes, the CPU samples the SRDY and FAULT in the same way.

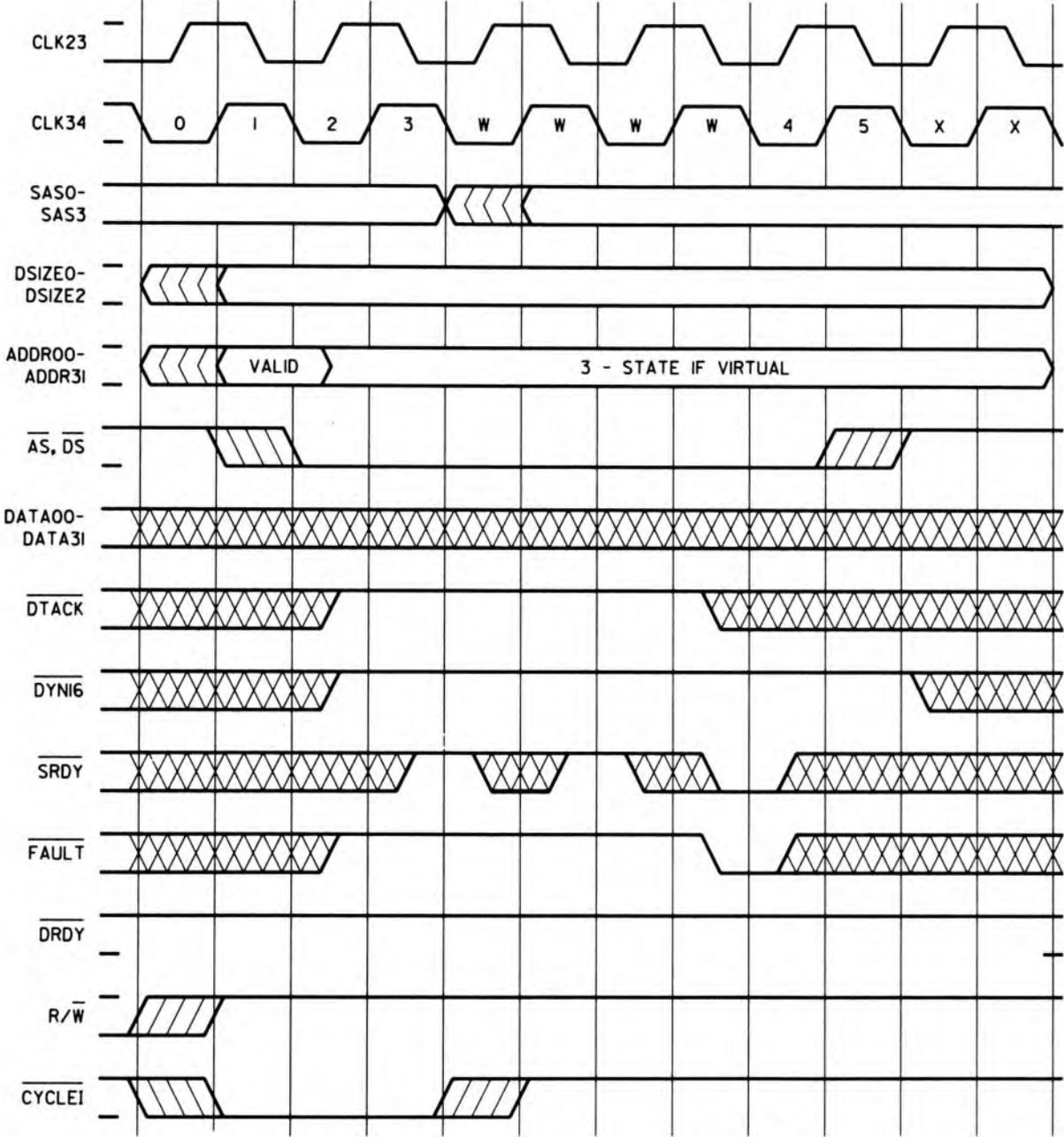
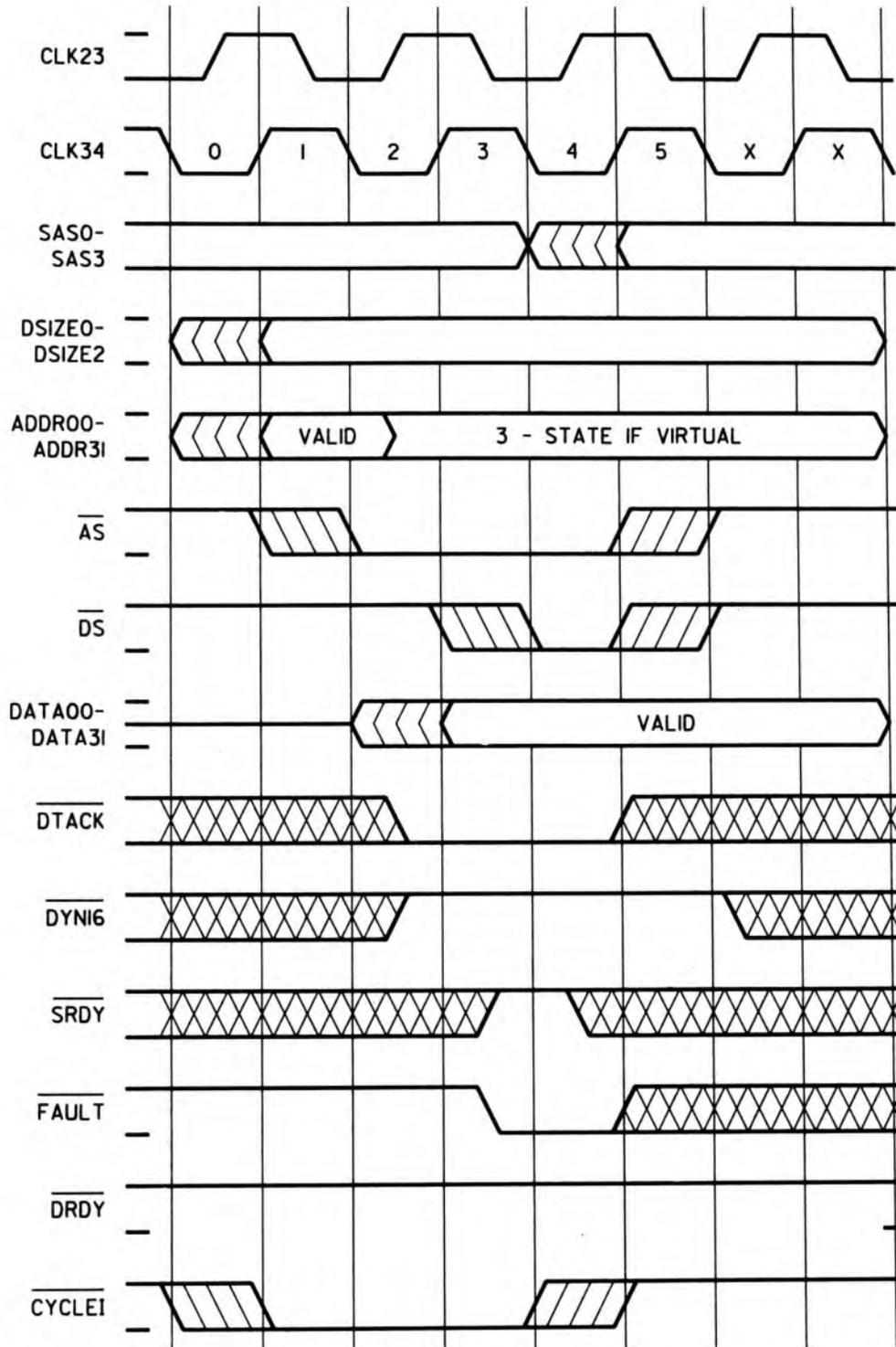


Figure 4-21. FAULT with SRDY (Synchronous Fault)

BUS OPERATION
FAULTS



Note: $\overline{\text{FAULT}}$ must meet set-up time with respect to CLK34 edge after assertion of $\overline{\text{DTACK}}$.

Figure 4-22. $\overline{\text{FAULT}}$ After Assertion of $\overline{\text{DTACK}}$ (Write Transaction is Shown)

$\overline{\text{FAULT}}$ After $\overline{\text{DTACK}}$

The CPU can also sample $\overline{\text{FAULT}}$ synchronously if it has sampled $\overline{\text{DTACK}}$ or $\overline{\text{DYN16}}$ asynchronously in the same clock cycle. Figure 4-22 shows $\overline{\text{DTACK}}$ or $\overline{\text{DYN16}}$ sampled during clock state three and $\overline{\text{FAULT}}$ sampled during clock state four. The CPU then terminates the bus transaction and does not issue $\overline{\text{DRDY}}$. This sampling of $\overline{\text{DTACK}}$, $\overline{\text{FAULT}}$, and $\overline{\text{DYN16}}$ is the same for reads.

4.6.2 Retry

The $\overline{\text{RETRY}}$ signal is sampled the same way the $\overline{\text{FAULT}}$ signal is sampled. $\overline{\text{RETRY}}$ can replace $\overline{\text{FAULT}}$ in Figures 4-20—4-22 for purposes of sampling. Figure 4-23 shows a retry for a read transaction.

When the CPU samples the $\overline{\text{RETRY}}$, it terminates the transaction and does not issue a $\overline{\text{DRDY}}$. The CPU continues to asynchronously sample $\overline{\text{RETRY}}$. After $\overline{\text{RETRY}}$ is removed, the CPU redoes the entire transaction. The SAS code, address, DSIZE, and R/ $\overline{\text{W}}$ signals are the same as in the first transaction.

$\overline{\text{RETRY}}$ operates on reads and writes in the same way. There are some differences if the transaction is a blockfetch. This is a special case discussed in section 4.7.

4.6.3 Relinquish and Retry

$\overline{\text{RRREQ}}$ is sampled the same way the other two bus exceptions ($\overline{\text{FAULT}}$ and $\overline{\text{RETRY}}$) are sampled. An example is shown on Figure 4-24.

When the CPU samples $\overline{\text{RRREQ}}$, it terminates the transaction and does not issue a $\overline{\text{DRDY}}$. After the second clock state (X), the CPU 3-states the address and data buses as well as most of the control signals in order to allow some other device to use the bus. A cycle later, the CPU issues $\overline{\text{RRRACK}}$. This indicates that the device that issued $\overline{\text{RRREQ}}$ can get onto the bus for a bus transaction. The CPU continues to asynchronously sample $\overline{\text{RRREQ}}$. When the device using the bus is finished, it should remove $\overline{\text{RRREQ}}$. When the CPU sees that $\overline{\text{RRREQ}}$ is removed, it takes back the bus and redoes the entire transaction. As with the $\overline{\text{RETRY}}$, the SAS code, address, DSIZE, and R/ $\overline{\text{W}}$ signals are the same on both transactions.

$\overline{\text{RRREQ}}$ operates on reads and writes in the same way. There are some differences if the transaction is a blockfetch, and these special case are described in section 4.7.

BUS OPERATION

Relinquish and Retry

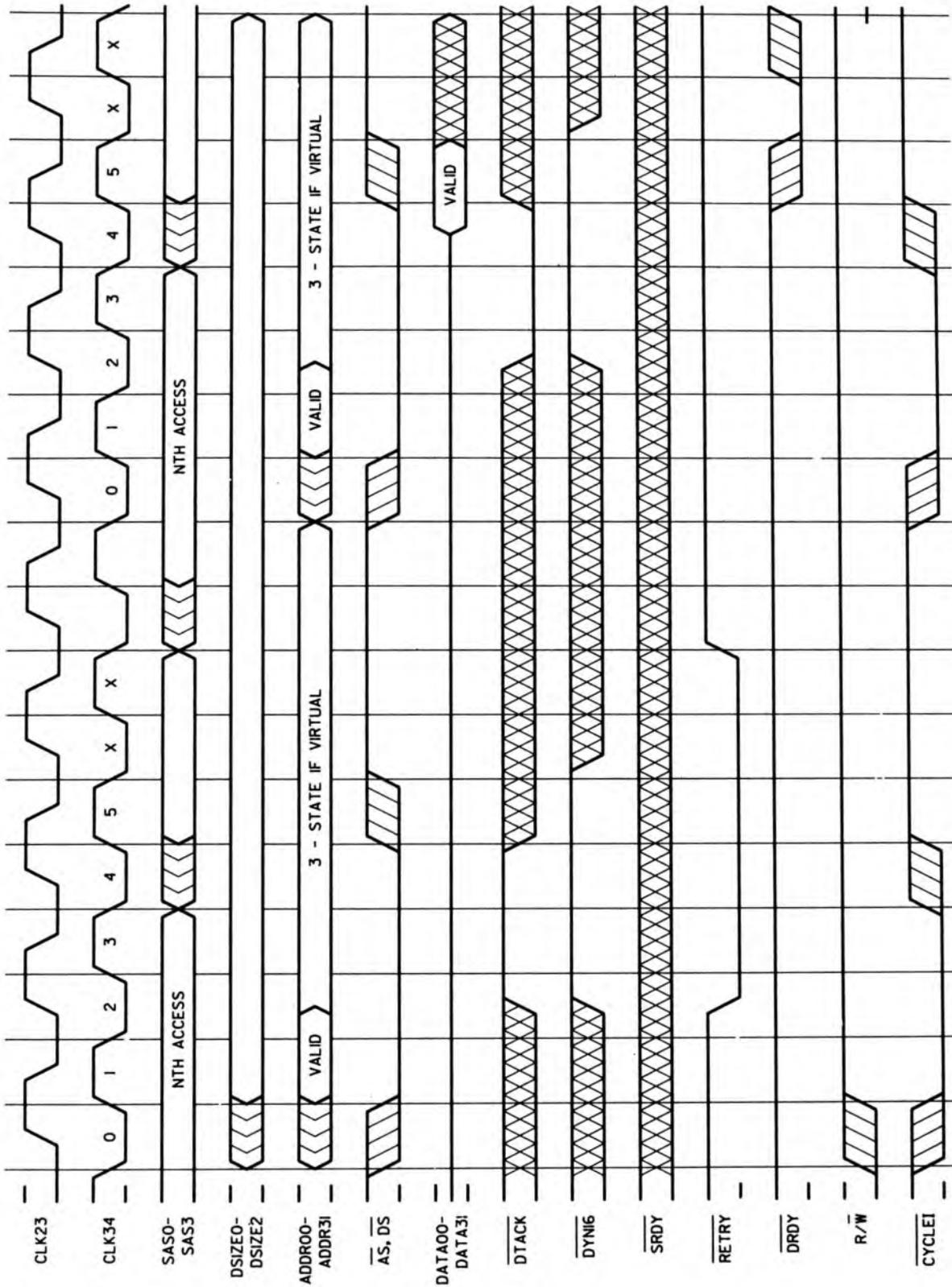


Figure 4-23. Retry of Transaction (Read Transaction is Shown)

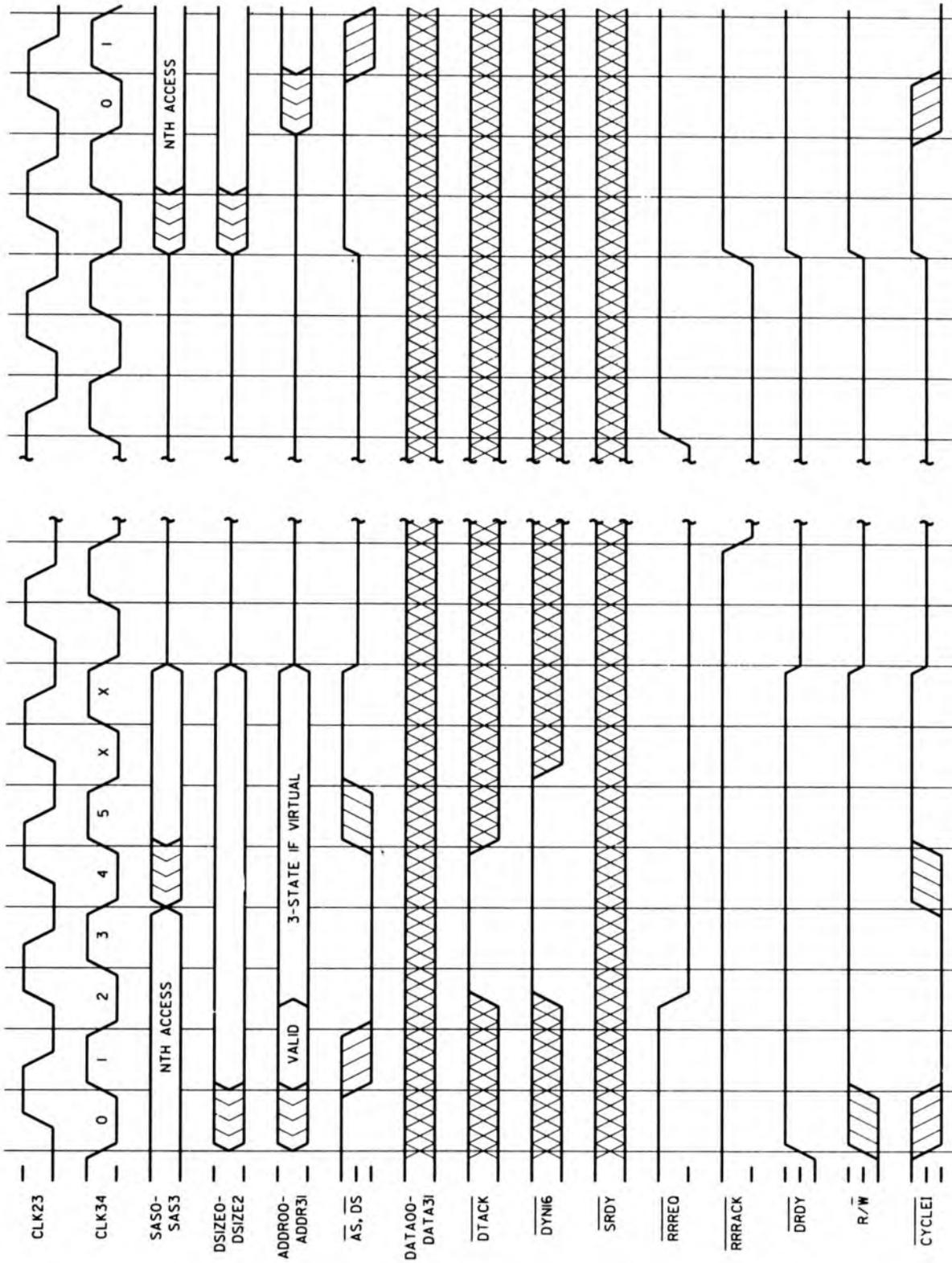


Figure 4-24. Relinquish and Retry

BUS OPERATION

Blockfetch Special Cases

4.7 BLOCKFETCH SPECIAL CASES

As indicated in the descriptions of the bus exceptions, a fault, retry, or relinquish and retry of a blockfetch transaction is a special case.

4.7.1 Fault on First Word of Blockfetch with Status Code Other than Prefetch

If the CPU samples $\overline{\text{FAULT}}$ on the first word of a blockfetch in which the SAS code is "instruction fetch" or "instruction fetch after PC discontinuity," the blockfetch transaction is altered as shown on Figure 4-25. The CPU sampled $\overline{\text{FAULT}}$ in clock state two and $\overline{\text{BLKFTCH}}$ in clock state three. Note that to sample $\overline{\text{BLKFTCH}}$, the CPU needs a $\overline{\text{DTACK}}$, $\overline{\text{SRDY}}$, or bus exception. Upon seeing $\overline{\text{BLKFTCH}}$ and $\overline{\text{FAULT}}$, the CPU removes $\overline{\text{DS}}$ and $\overline{\text{AS}}$ at clock state five. The address bus, if in physical mode, is driven until the second clock state (X). $\overline{\text{DRDY}}$ is not issued at all during this transaction.

4.7.2 Fault on First Word of Blockfetch with Status of Prefetch

As in other prefetch transactions, when the CPU is faulted it ignores the data and continues with its current execution (see Figure 4-26). On clock state two, the CPU samples $\overline{\text{FAULT}}$ and on clock state three, it samples $\overline{\text{BLKFTCH}}$. The CPU terminates the first word fetch by removing $\overline{\text{DS}}$ (not issuing $\overline{\text{DRDY}}$) and continuing to the second transaction. The second transaction operates normally.

4.7.3 Retry on First Word of Blockfetch

The CPU samples $\overline{\text{RETRY}}$ during clock state three and the $\overline{\text{BLKFTCH}}$ during clock state four. The CPU then terminates the transaction by removing $\overline{\text{DS}}$ and $\overline{\text{AS}}$ at clock state five. At this point, the CPU waits for $\overline{\text{RETRY}}$ to be negated. Then, the CPU retries the entire blockfetch transaction (see Figure 4-27).

4.7.4 Retry on Second Word of Blockfetch

In this case, the CPU samples $\overline{\text{BLKFTCH}}$ and $\overline{\text{DTACK}}$, clocks the data from the data bus, issues a $\overline{\text{DRDY}}$, and continues to the second word (see Figure 4-28). During the first wait state (W), the CPU samples $\overline{\text{RETRY}}$. It then terminates the transaction, does not issue a $\overline{\text{DRDY}}$, and waits for the $\overline{\text{RETRY}}$ to be negated. Since the second word of a blockfetch is always a prefetch, the CPU faults this transaction internally rather than retrying the entire blockfetch transaction. When the $\overline{\text{RETRY}}$ signal is negated, the CPU continues with its current execution. The CPU, if it wants to, proceeds with a new bus transaction since it does not retry the blockfetch.

BUS OPERATION

Blockfetch Special Cases

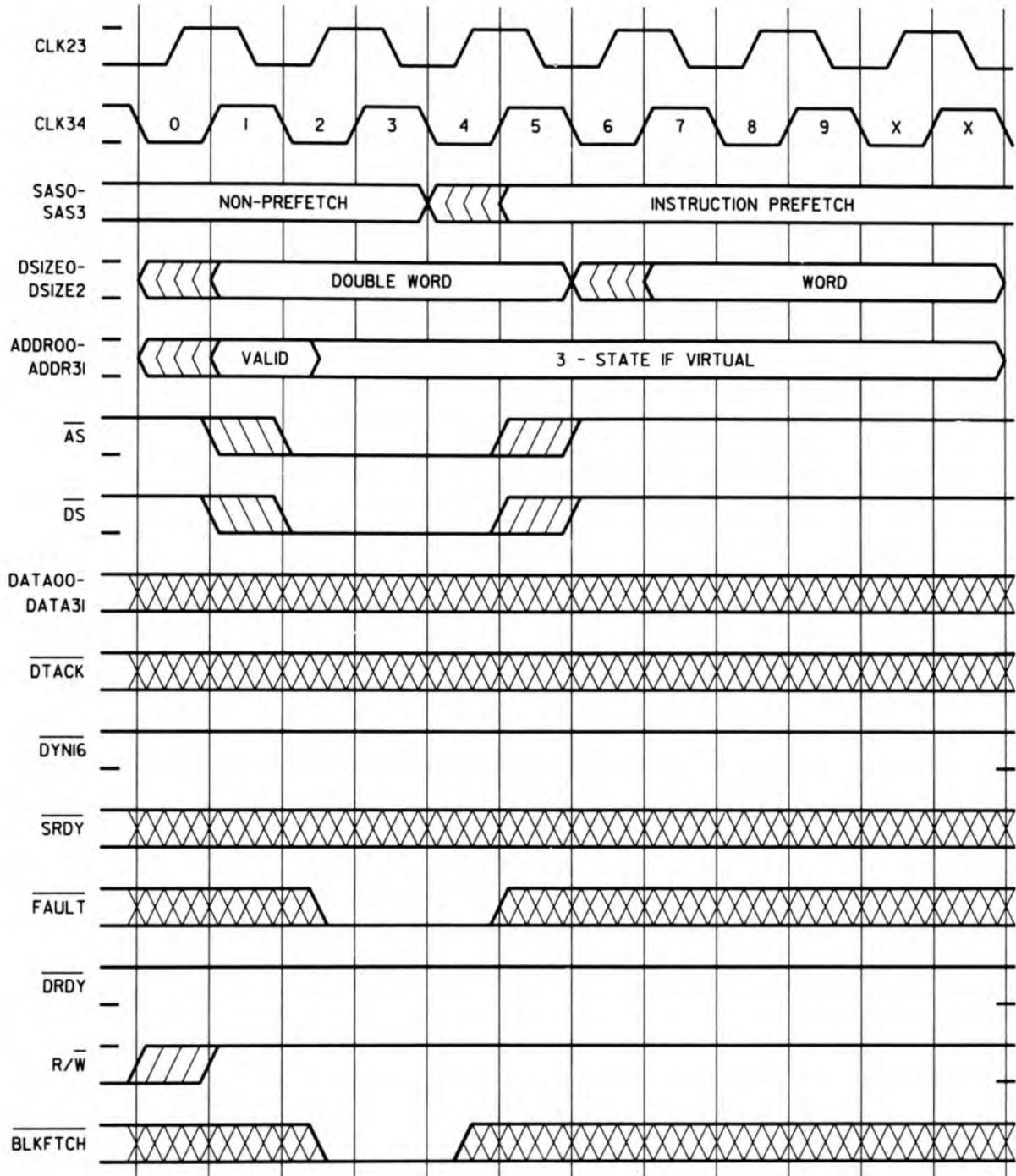


Figure 4-25. Fault on First Word of Blockfetch Transaction with Access Status Code Other than Prefetch

BUS OPERATION

Blockfetch Special Cases

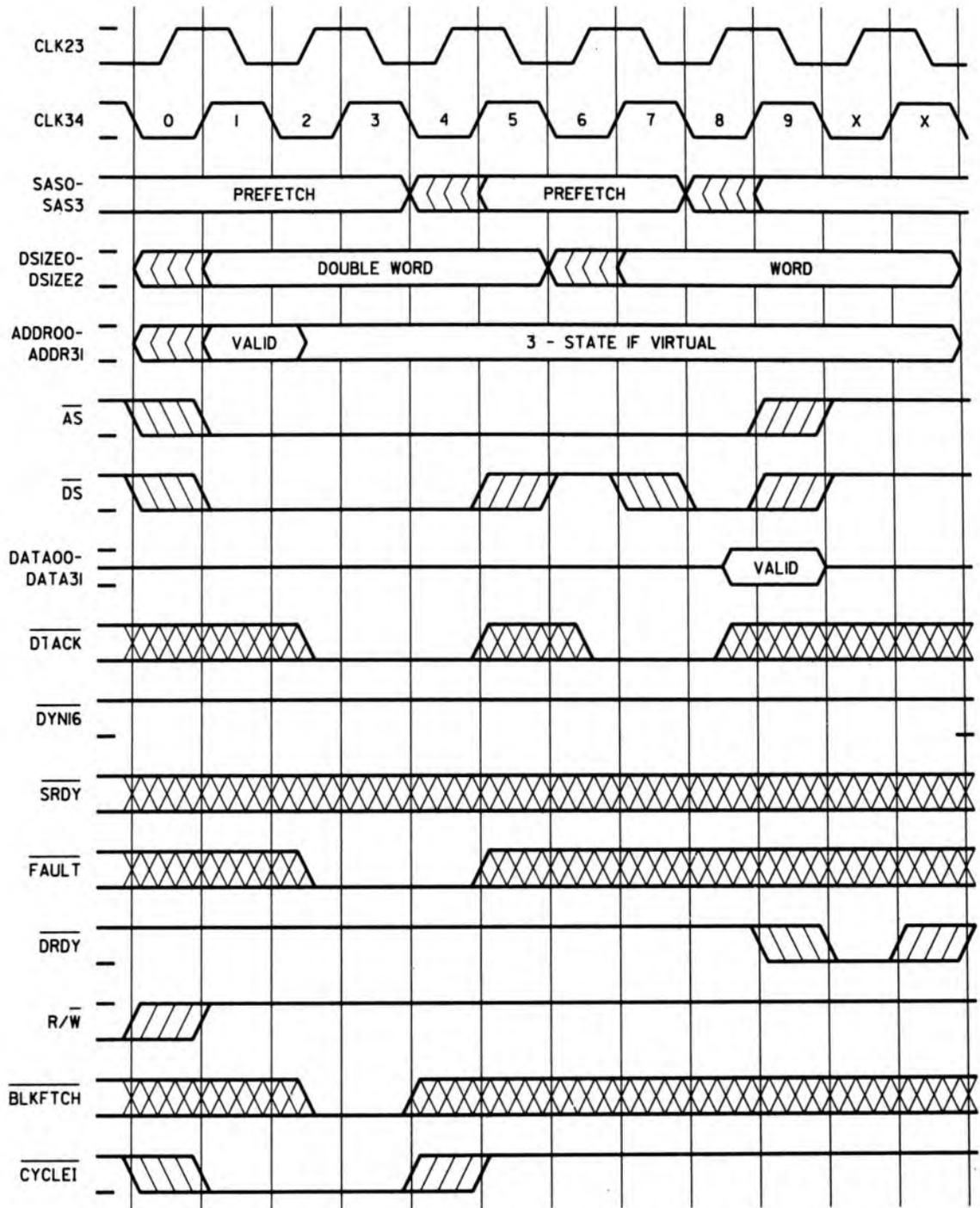
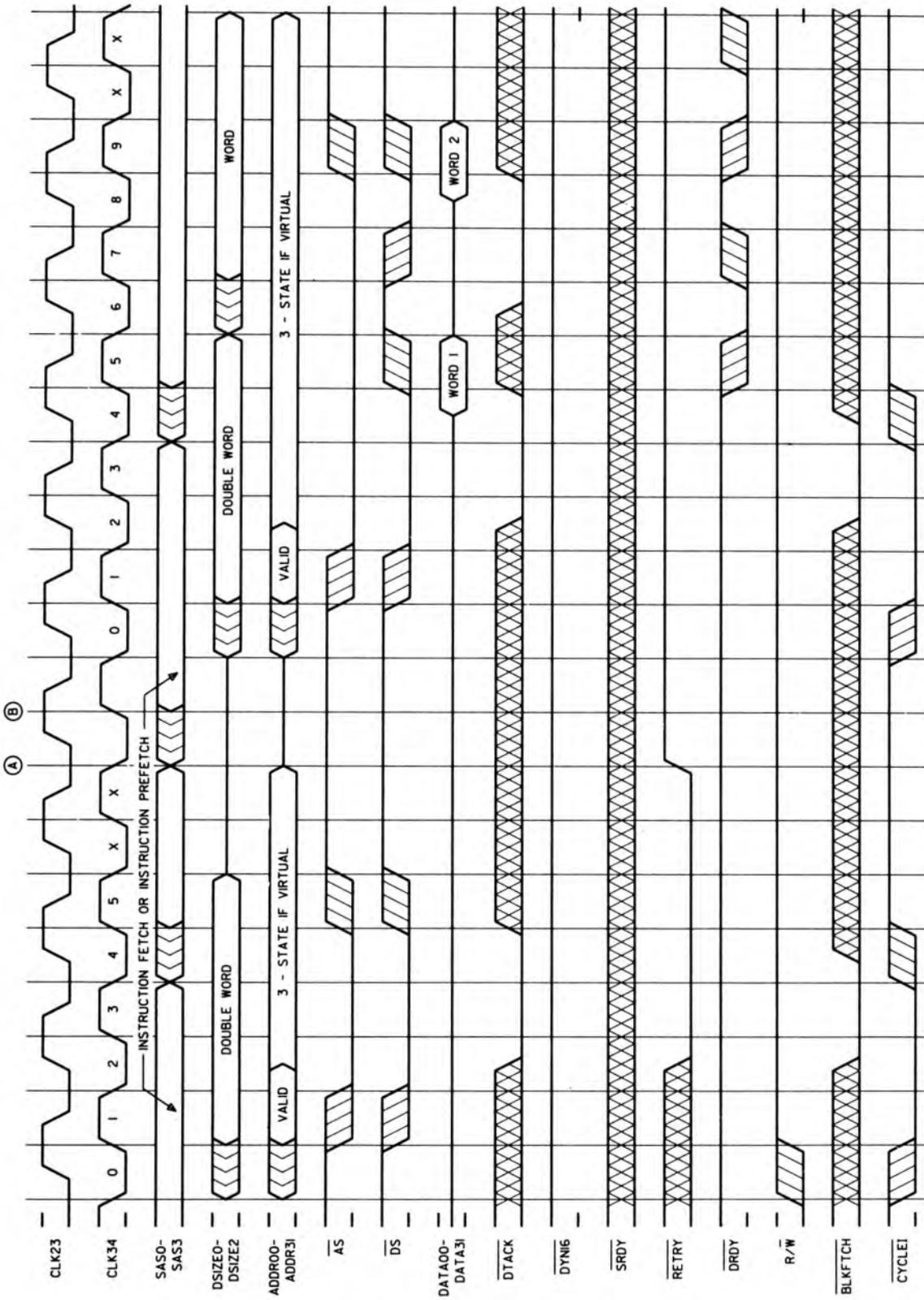


Figure 4-26. Fault on First Word of Blockfetch Transaction with Access Status Code of Prefetch

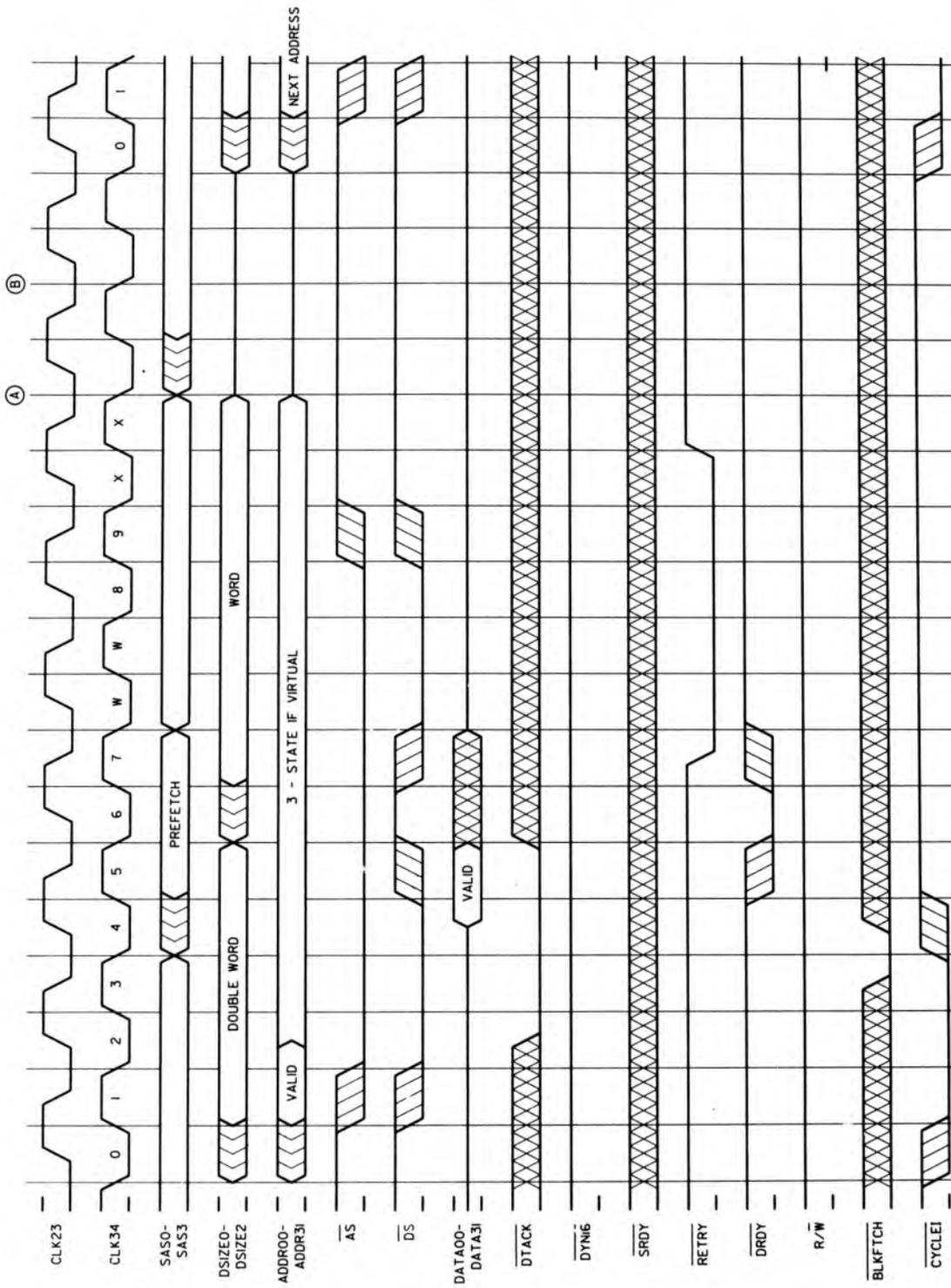


Note: If RRREQ is asserted instead of RETRY, the CPU 3-states the bus at A and RRRACK one clock cycle later at B.

Figure 4-27. Retry on First Word of Blockfetch Transaction

BUS OPERATION

Blockfetch Special Cases



Notes:
 On \overline{RRREQ} , the following signals are 3-stated between points A and B: ADDR00—ADDR31, DATA00—DATA31, \overline{AS} , \overline{CYCLEI} , \overline{DRDY} , \overline{DS} , DSIZEO—DSIZE2, R/W, SAS0—SAS3.
 If \overline{RRREQ} or \overline{RETRY} is asserted during the first word, the entire blockfetch access is retried.

Figure 4-28. Retry on Second Word of Blockfetch Transaction

4.7.5 Relinquish and Retry on Blockfetch

Figures 4-27 and 4-28 can be used to illustrate the $\overline{\text{RRREQ}}$ bus exception for the first and second word of a blockfetch.

The timing and bus transaction for Figure 4-27 looks the same if the bus exception is $\overline{\text{RRREQ}}$ rather than $\overline{\text{RETRY}}$. However, the CPU releases the bus before doing the retried transaction. Additionally, the CPU 3-states the bus at the end of the second clock state (X) indicated by A on the diagram. One cycle later an $\overline{\text{RRREQ}}$ is issued to tell the requesting device that it can now use the bus. When $\overline{\text{RRREQ}}$ is negated, the CPU continues with the retried transaction, starting at point B.

The same explanation applies for a $\overline{\text{RRREQ}}$ on the second word of a blockfetch (see Figure 4-28). As above, the CPU would 3-state the bus at point A and issue an $\overline{\text{RRRACK}}$. Once $\overline{\text{RRREQ}}$ is negated, the CPU continues with the next bus transaction, starting at point B, and does not retry the blockfetch.

4.8 INTERRUPTS

The microprocessor accepts fifteen levels of interrupts. An interrupt request is made to the microprocessor by placing an interrupt request value on the interrupt priority level (IPL0—IPL3) signals or by requesting a nonmaskable interrupt by asserting $\overline{\text{NMINT}}$. Pending interrupts are not acknowledged until the currently executing instructions are completed. The exceptions to this are multiply, divide, modulo, move block word, string copy, and string end instructions, which abort upon a pending interrupt.

The pending interrupt value input on IPL0—IPL3 is internally inverted and compared to the value contained in the interrupt priority level (IPL) field of the processor status word (PSW). In order for the pending interrupt to be acknowledged, its inverted value must be greater than the IPL field value. Pending interrupts whose inverted values are equal to or less than the IPL field value are ignored. However, if the pending interrupt is nonmaskable, it always interrupts the microprocessor, regardless of the IPL field value.

The microprocessor handles interrupts in one of two ways, full or quick interrupt. The full interrupt isolates the interrupt handler from all other processes, allowing the interrupt to be executed at any level or returned to a different process (i.e., the interrupt handler is a separate process). The quick interrupt enhances the performance of systems that do not require the functionality of the full interrupt since these interrupts are executed in the same process as the interrupted program. All interrupts are serviced via the quick interrupt facility if the quick interrupt enable (QIE) bit in the PSW is set (1). Table 4-2 summarizes how the microprocessor handles the various interrupt requests. See Chapter 6 for more information on full and quick interrupts.

BUS OPERATION
Interrupts

Table 4-2. Interrupt Acknowledge Summary					
Interrupt Priority	Interrupt Acknowledge	$\overline{\text{AVEC}}$	$\overline{\text{NMINT}}$	QIE	Result
Less than PSW IPL field priority	No	x	1	x	Interrupt is not acknowledged.
Equal to PSW IPL field priority	No	x	1	x	Interrupt is not acknowledged.
Greater than PSW IPL field priority	Yes	1	1	0	Interrupt is acknowledged and serviced via the full-interrupt sequence. Microprocessor fetches vector number from interrupting device.
Greater than PSW IPL field priority	Yes	0	1	0	Interrupt is acknowledged and serviced via the full-interrupt sequence. Microprocessor supplies the vector number.
Any level compared to PSW IPL field priority	Yes	x	0	0	Interrupt is acknowledged and serviced via the full-interrupt sequence. It is treated as an autovector at vector number 0. The address bus contains all zeros during the acknowledge.
Greater than PSW IPL field priority	Yes	1	1	1	Interrupt is acknowledged and serviced via quick-interrupt sequence. Microprocessor fetches vector number from interrupting device.

Table 4-2. Interrupt Acknowledge Summary (Continued)					
Interrupt Priority	Interrupt Acknowledge	$\overline{\text{AVEC}}$	$\overline{\text{NMINT}}$	QIE	Result
Greater than PSW IPL field priority	Yes	0	1	1	Interrupt is acknowledged and serviced via quick-interrupt sequence. Microprocessor supplies the vector number.
Any level compared to PSW IPL field priority	Yes	x	0	1	Interrupt is acknowledged and serviced via quick-interrupt sequence. It is treated as an autovector interrupt at vector number 0. The address bus contains all zeros during the acknowledge.

The microprocessor also provides autovector and nonmaskable interrupt facilities. The following sections describe these facilities.

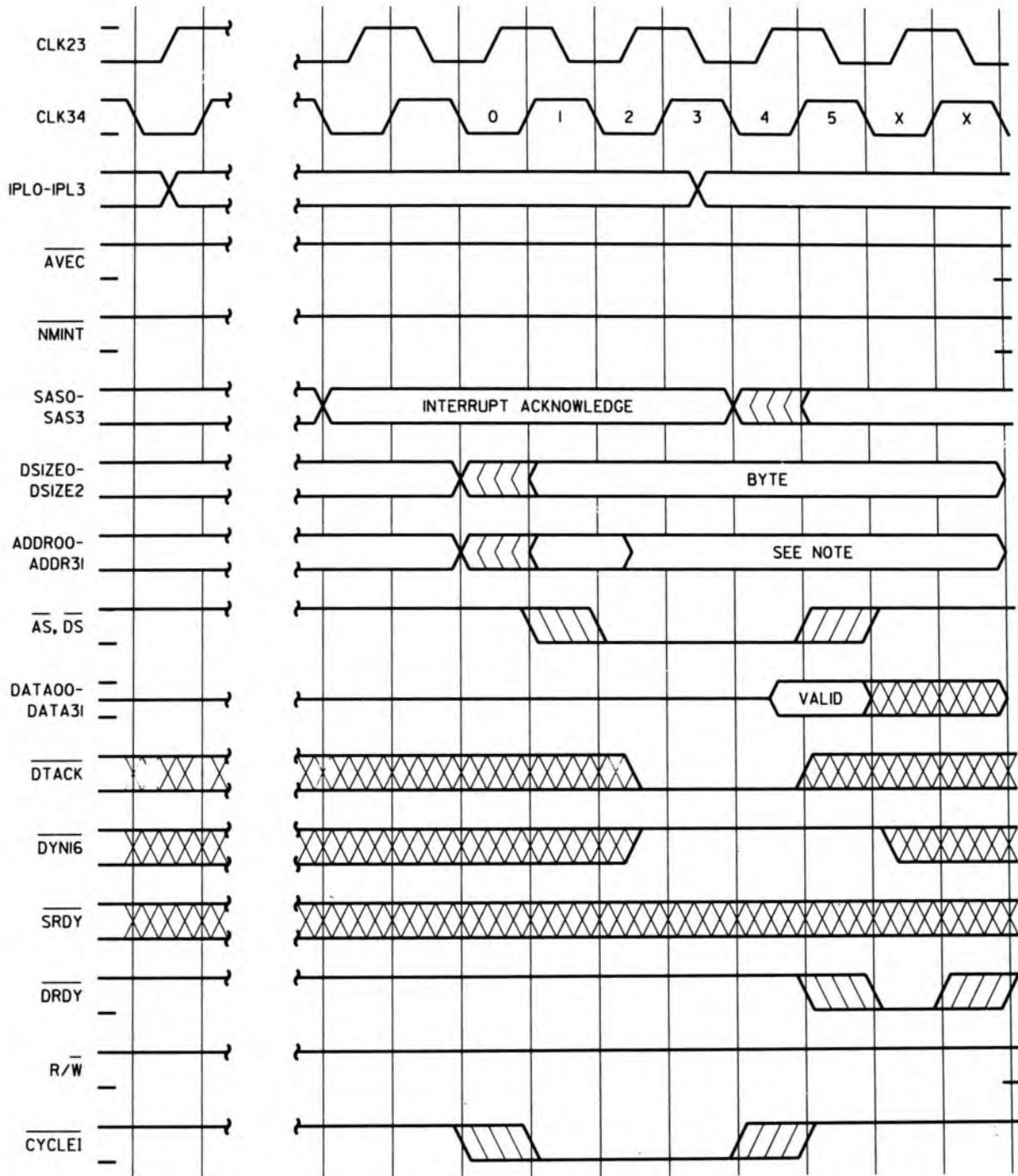
4.8.1 Interrupt Acknowledge

The microprocessor acknowledges an interrupt by transmitting the inverted interrupt value on bits 2 through 5 of the address bus. In addition, the value placed on the interrupt option ($\overline{\text{INTOPT}}$) pin is inverted and transmitted on bit 6 of the address bus. (The $\overline{\text{INTOPT}}$ input has no effect on the microprocessor; however, it could be used to indicate, for example, whether the interrupt was hardware- or software-generated.) The microprocessor then fetches the interrupt vector number from the interrupting device on bits 0 through 7 of the data bus and begins execution of the interrupt handling routine.

The interrupt acknowledge transaction illustrated on Figure 4-29, depicts the case in which a value placed on the IPL0—IPL3 inputs caused an interrupt. In this case, the interrupt acknowledge is issued in response to the application of the IPL0—IPL3 signals and $\overline{\text{INTOPT}}$ signal, without $\overline{\text{AVEC}}$ or $\overline{\text{NMINT}}$ active. During the interrupt acknowledge transaction, the CPU reads in an 8-bit offset provided by the interrupting device and uses it as an offset to a table. The SAS code is "interrupt acknowledge"; the DSIZE is a byte. The address corresponding to the interrupt acknowledge is indicated at the bottom of the figure. The interrupting device drives the data bus with the 8-bit offset and a memory acknowledge, in this case, a $\overline{\text{DTACK}}$ or $\overline{\text{DYN16}}$. The bus exceptions are accepted during this bus transaction. The IPL input values should be removed once the corresponding interrupt acknowledge has occurred.

BUS OPERATION

Interrupt Acknowledge



Note: During the interrupt acknowledge, the address bus (ADDR00—ADDR31) contains the following data:

31	7	6	5	4	3	2	1	0
0 0		INVERTED		INVERTED				1 1
		INTOPT	INPUT	IPL3	IPL2	IPL1	IPL0	

Figure 4-29. Interrupt Acknowledge

BUS OPERATION
Interrupt Acknowledge

Table 4-3 summarizes how the interrupt priority levels are to be interpreted and shows the corresponding acknowledge output for each level.

Table 4-3. Interrupt Level Code Assignments										
Interrupt Request Input IPL0—IPL3				Interrupt Option Input INTOPT	Interrupt Acknowledge Output ADDR02—ADDR06*					Priority Level
Bits					Bits					
3	2	1	0		06	05	04	03	02	
0	0	0	0	0	1	1	1	1	1	Highest Priority
0	0	0	0	1	0	1	1	1	1	
0	0	0	1	0	1	1	1	1	0	2nd
0	0	0	1	1	0	1	1	1	0	
0	0	1	0	0	1	1	1	0	1	3rd
0	0	1	0	1	0	1	1	0	1	
0	0	1	1	0	1	1	1	0	0	4th
0	0	1	1	1	0	1	1	0	0	
0	1	0	0	0	1	1	0	1	1	5th
0	1	0	0	1	0	1	0	1	1	
0	1	0	1	0	1	1	0	1	0	6th
0	1	0	1	1	0	1	0	1	0	
0	1	1	0	0	1	1	0	0	1	7th
0	1	1	0	1	0	1	0	0	1	
0	1	1	1	0	1	1	0	0	0	8th
0	1	1	1	1	0	1	0	0	0	
1	0	0	0	0	1	0	1	1	1	9th
1	0	0	0	1	0	0	1	1	1	
1	0	0	1	0	1	0	1	1	0	10th
1	0	0	1	1	0	0	1	1	0	
1	0	1	0	0	1	0	1	0	1	11th
1	0	1	0	1	0	0	1	0	1	
1	0	1	1	0	1	0	1	0	0	12th
1	0	1	1	1	0	0	1	0	0	
1	1	0	0	0	1	0	0	1	1	13th
1	1	0	0	1	0	0	0	1	1	
1	1	0	1	0	1	0	0	1	0	14th
1	1	0	1	1	0	0	0	1	0	
1	1	1	0	0	1	0	0	0	1	Lowest Priority
1	1	1	0	1	0	0	0	0	1	
1	1	1	1	0	x	x	x	x	x	No Interrupt Pending
1	1	1	1	1	x	x	x	x	x	

* The x signifies that no value is placed on address bus.

BUS OPERATION

Autovector Interrupt

4.8.2 Autovector Interrupt

If the autovector ($\overline{\text{AVEC}}$) input is active during an interrupt request, the microprocessor does not fetch a vector number from the interrupting device. Instead, the microprocessor provides the interrupt vector by treating the inverted $\overline{\text{INTOPT}}$ input, concatenated with the interrupt priority level (IPL0—IPL3) inputs, as a vector number. The autovector facility reduces hardware costs in smaller, less complex systems because the interrupt vector is supplied by the microprocessor instead of by external hardware.

Refer to Figure 4-30 for an illustration of the autovector interrupt acknowledge transaction. In this transaction, an autovector acknowledge is issued in response to the application of the IPL pins and $\overline{\text{INTOPT}}$ pin with $\overline{\text{AVEC}}$ active and no $\overline{\text{NMINT}}$. Since the CPU does not need to read in an external value, it does an autovector interrupt acknowledge without looking for a memory acknowledge or a bus exception. The transaction goes through the clock states without inserting wait cycles. This transaction is used to tell the interrupting device that it should remove the IPL and $\overline{\text{AVEC}}$ input values. No $\overline{\text{DRDY}}$ is issued because there is no latching of data.

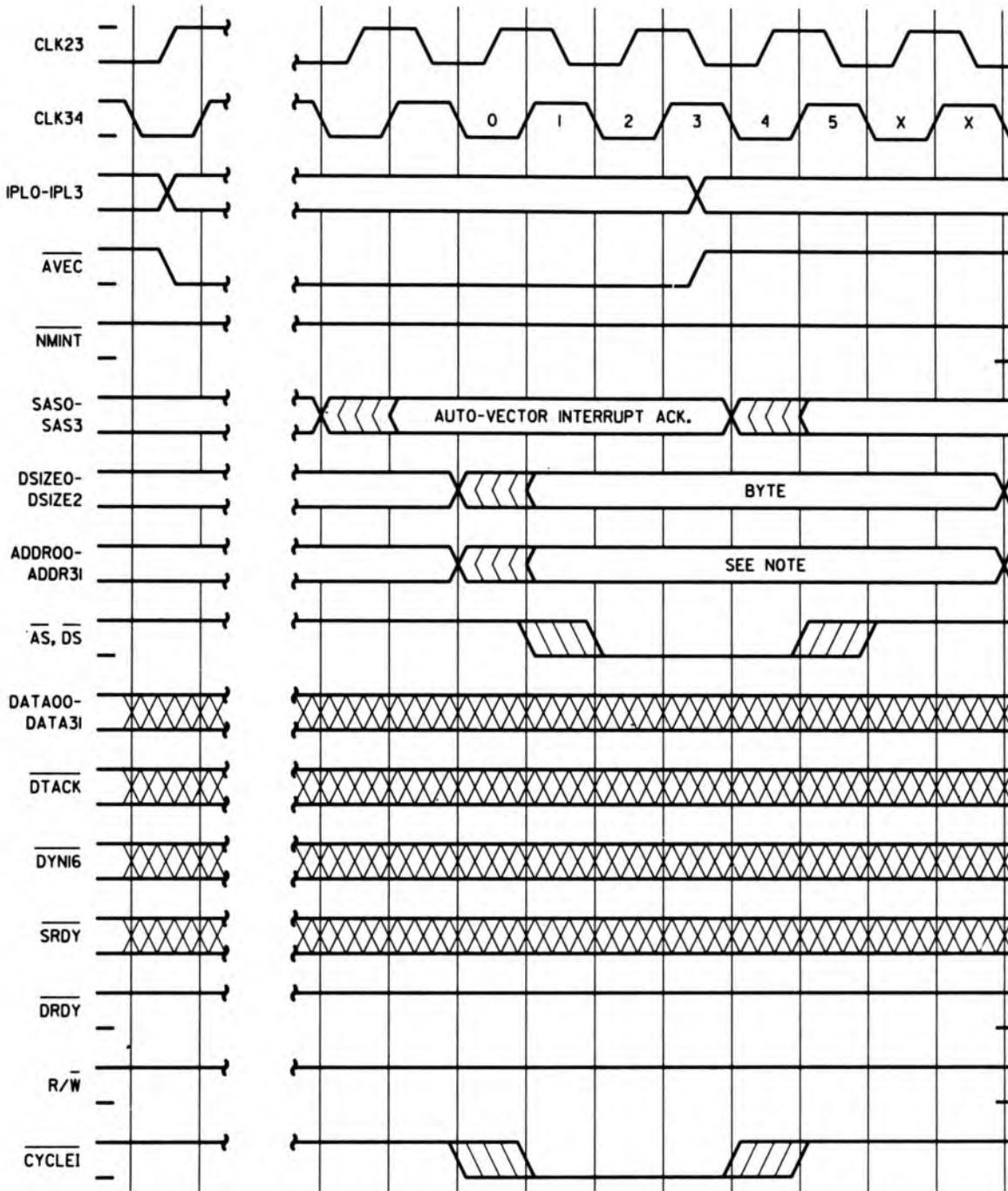
The $\overline{\text{AVEC}}$ and IPL signals are double-sampled and compared for greater noise immunity.

4.8.3 Nonmaskable Interrupt

The nonmaskable interrupt facility is provided to satisfy reliability and recoverability requirements of various systems. As previously mentioned, a nonmaskable interrupt can interrupt the microprocessor, regardless of the current priority level in the IPL field. A nonmaskable interrupt occurs if the nonmaskable interrupt ($\overline{\text{NMINT}}$) input is asserted. The interrupt is then treated as an autovector interrupt with vector number 0. During the interrupt acknowledge cycle of a nonmaskable interrupt, address bus bits ADDR00—ADDR31 contain zeros. This distinguishes a nonmaskable interrupt from all other interrupts.

Figure 4-31 illustrates the nonmaskable interrupt acknowledge transaction. Here, a nonmaskable interrupt acknowledge is issued in response to the application of the $\overline{\text{NMINT}}$ input. For a nonmaskable interrupt, the CPU uses an internal offset corresponding to an IPL of zero. Since the CPU does not need to read in data, it performs the transaction without looking for a memory acknowledge or a bus exception. The transaction goes through the clock states without inserting wait cycles. Again, the interrupting device should release $\overline{\text{NMINT}}$ when it sees the acknowledge. The SAS code is "autovector acknowledge," but the interrupt vector is 0. ADDR00 can be used to determine the difference between the $\overline{\text{AVEC}}$ and $\overline{\text{NMINT}}$ interrupts. It is a 1 for autovector and a 0 for nonmaskable interrupt.

BUS OPERATION Autovector Interrupt



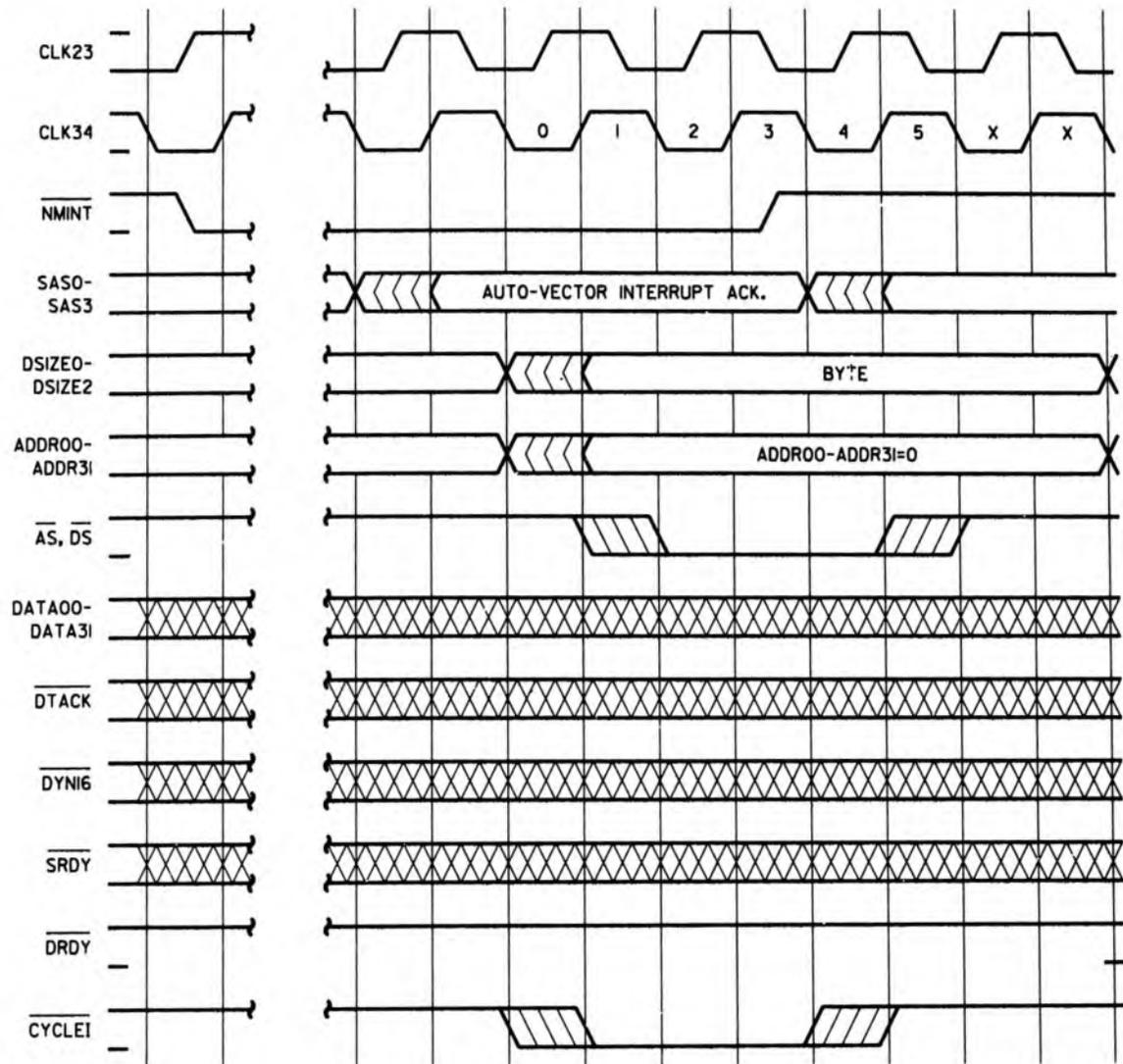
Note: During the interrupt acknowledge, the address bus (ADDR00—ADDR31) contains the following data:

31				7				6		5	4	3	2	1	0
0 0				INVERTED				INVERTED							
				INTOPT INPUT				IPL3	IPL2	IPL1	IPL0			1	1

Figure 4-30. Autovector Interrupt Acknowledge

BUS OPERATION

Nonmaskable Interrupt Acknowledge



Note: The address bus ADDR00—ADDR31 contains all zeros during the acknowledge of a nonmaskable interrupt.

Figure 4-31. Nonmaskable Interrupt Acknowledge

4.9 BUS ARBITRATION

The microprocessor bus may be requested in two ways. External devices may request the bus by asserting the \overline{RRREQ} input, as explained previously, or by asserting the \overline{BUSREQ} input.

The relinquish and retry request has priority over a bus request. The microprocessor acknowledges a relinquish and retry request only during bus transactions; however, it ignores the request during the write portion of a read-interlocked transaction.

A bus request during a CPU bus transaction is not acknowledged until the end of the bus transaction or until the end of the write portion of a read-interlocked transaction.

4.9.1 Bus Request During a Bus Transaction

\overline{BUSRQ} is sampled independently of bus transactions at the beginning of every clock cycle. On Figure 4-32 it is sampled for the first time at the beginning of clock state two. After sampling \overline{BUSRQ} , the CPU continues the current bus transaction. After the transaction is completed, the CPU 3-states the address and data buses and some control signals just after the last clock state (X). A cycle later, it issues the bus request acknowledge, \overline{BRACK} . At this point, the device requesting the bus can perform its operations. When finished, the device drops the \overline{BUSRQ} . After seeing this drop, the CPU removes \overline{BRACK} and takes back the bus. Note that, if the bus request occurred during an active retry request or relinquish and retry request, it would not be acknowledged until after the current transaction had been retried. Refer to section 4.14 for an example.

For a bus request that does not occur during a bus transaction, the CPU 3-states the bus a cycle after sampling \overline{BUSRQ} and issue \overline{BRACK} a cycle after that.

BUS OPERATION

Bus Request During a Bus Transaction

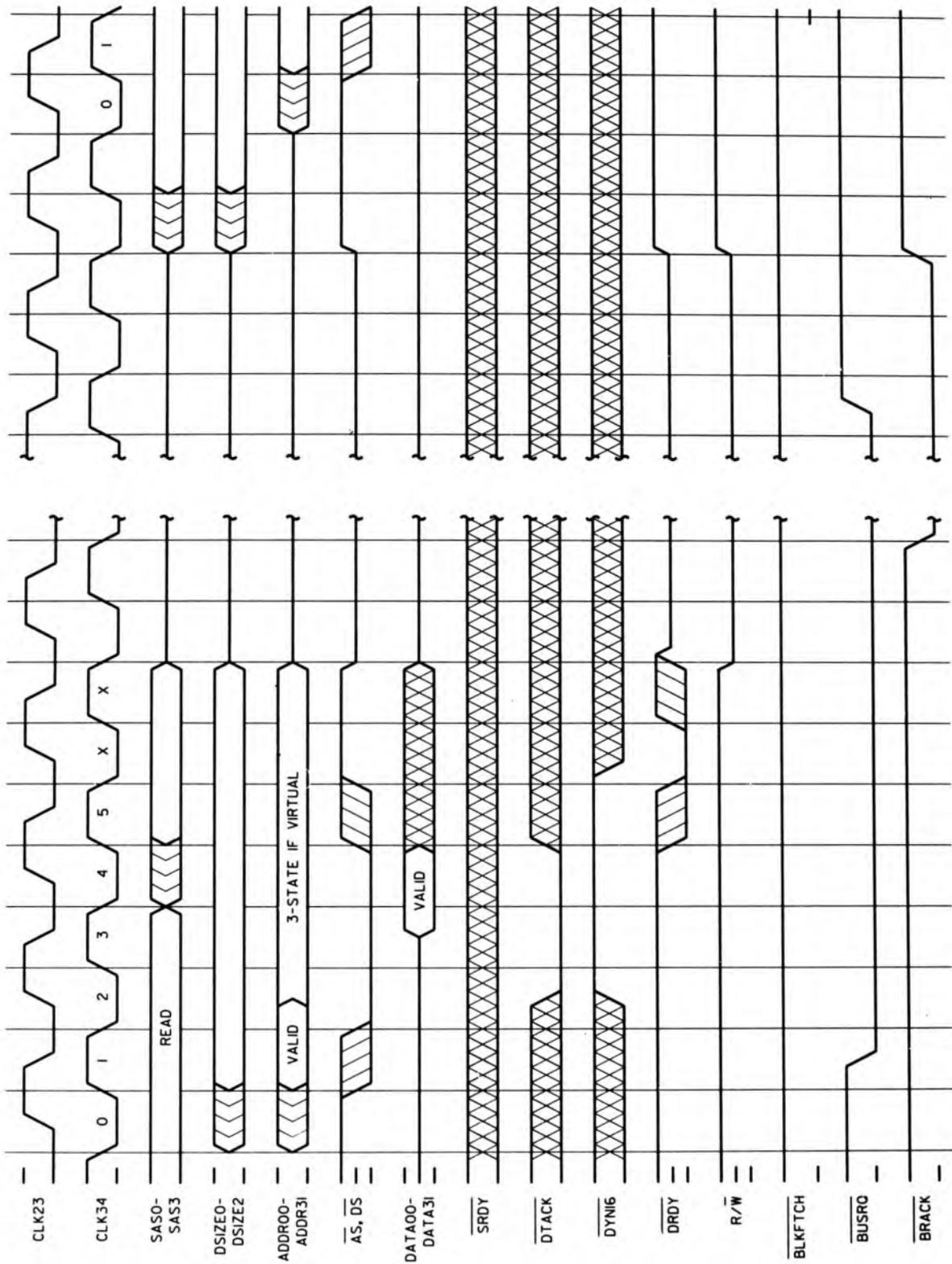


Figure 4-32. Bus Request During a Transaction

4.9.2 DMA Operation

The microprocessor provides support for direct memory access (DMA) and shares bus control responsibilities with the system DMA controller. To initiate a DMA operation, the controller requests the microprocessor bus by asserting $\overline{\text{BUSRQ}}$. Recall that this request is not acknowledged until the end of a bus transaction or until the end of the write portion of a read-interlocked transaction. However, if the CPU is not using the bus, the request is acknowledged immediately. Once the microprocessor recognizes the request, it 3-states the following signals:

$\overline{\text{ABORT}}$	DATA00—DATA31	R/ $\overline{\text{W}}$
ADDR00—ADDR31	$\overline{\text{DRDY}}$	SAS0—SAS3
$\overline{\text{AS}}$	$\overline{\text{DS}}$	$\overline{\text{VAD}}$
$\overline{\text{CYCLEI}}$	DSIZE0—DSIZE2	XMD0—XMD1

After the microprocessor has 3-stated the above signals, it acknowledges the DMA request by asserting the bus request acknowledge output. Table 4-4 summarizes the output signal states once the DMA has been acknowledged.

A DMA operation is terminated when the DMA controller removes the request by negating $\overline{\text{BUSRQ}}$ (drives the input high). The microprocessor then negates the acknowledge ($\overline{\text{BRACK}}$) and, finally, the 3-stated signals are returned to the microprocessor's control. The next operation may then begin.

Output Signal	Signal State	Output Signal	Signal State
$\overline{\text{ABORT}}$	Z*	DSIZE0—DSIZE2	Z
ADDR00—ADDR31	Z	R/ $\overline{\text{W}}$	Z*
$\overline{\text{AS}}$	Z*	$\overline{\text{RESET}}$	Logic 1
$\overline{\text{BRACK}}$	Logic 0	$\overline{\text{RRRACK}}$	Logic 1
$\overline{\text{CYCLEI}}$	Z*	SAS0—SAS3	Z*
DATA00—DATA31	Z	$\overline{\text{VAD}}$	Z
$\overline{\text{DRDY}}$	Z*	XMD0—XMD1	Z
$\overline{\text{DS}}$	Z*	—	—

Notes:

Z High impedance state.

Z* High impedance state held at logic 1 with external passive holding resistor.

BUS OPERATION

Reset

4.10 RESET

The microprocessor handles two types of reset requests: system and internal. A reset has the highest interrupt priority and preempts any ongoing microprocessor operation.

4.10.1 System Reset

A system reset is initiated when the system drives the reset request ($\overline{\text{RESETR}}$) input low. This double-latched input must be active on three consecutive latching before being recognized; this ensures noise immunity. After recognizing the reset request, the microprocessor sends a reset acknowledge to the system by asserting $\overline{\text{RESET}}$. All microprocessor outputs are then driven to a temporary state that prevents control signal and bus conflicts while the system responds to the reset acknowledge.

Once the system has responded to the acknowledge, it negates $\overline{\text{RESETR}}$. The microprocessor continues to hold $\overline{\text{RESET}}$ active for 128 clock cycles after $\overline{\text{RESETR}}$ has been negated, allowing the external system to go through its own initialization sequence. At the end of this period, the microprocessor negates $\overline{\text{RESET}}$ and begins executing the internal reset sequence. Table 4-5 indicates the states of the microprocessor's output signals once $\overline{\text{RESET}}$ is negated. During this sequence the microprocessor performs the following register initialization to restart operations:

- Changes to physical addressing mode.
- Fetches a word at location 80 hexadecimal and stores it in the process control block pointer (PCBP). This word is the beginning address of the reset process control block (PCB).
- Fetches a word at the PCB address and stores it in the processor status word (PSW).
- Fetches a word at the location four bytes from the PCB address and stores it in the program counter (PC). This word is the PC value for initial execution.
- Fetches a word at the location eight bytes from the initial PCB address and stores it in the stack pointer (SP).
- If the PSW I bit is set (1), the bit is cleared (0) and the PCBP incremented by 12.
- Begins execution at the address specified by the PC.

4.10.2 Internal Reset

An internal reset sequence is like a system reset sequence except there is no external reset request signal; the request is generated internally. Note that the $\overline{\text{RESET}}$ line still goes active for 128 clock cycles.

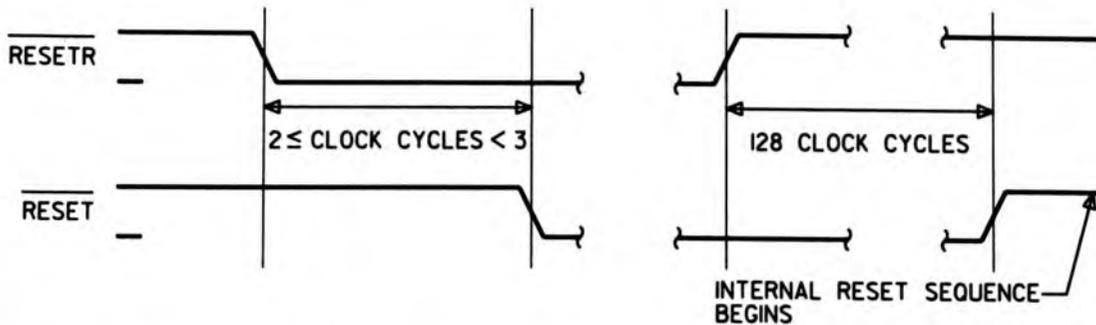
Table 4-5. Output States on Reset		
Output Signal	Signal State	
	CPU is Bus Arbiter	CPU is Not Bus Arbiter
$\overline{\text{ABORT}}$	Logic 1	Z
ADDR00—ADDR31	Z	Z
$\overline{\text{AS}}$	Logic 1	Z
$\overline{\text{BRACK}}$	Logic 1	—
$\overline{\text{BUSRQ}}$	—	Logic 1
$\overline{\text{CYCLEI}}$	Logic 1	Z
DATA00—DATA31	Z	Z
$\overline{\text{DRDY}}$	Logic 1	Z
$\overline{\text{DS}}$	Logic 1	Z
DSIZE0—DSIZE2	Logic 0	Z
IQS0—IQS1	Logic 1	Logic 1
R/ $\overline{\text{W}}$	Logic 1	Z
$\overline{\text{RRRACK}}$,	Z*	Z*
SAS0—SAS3	Logic 1	Z
$\overline{\text{SOI}}$	Logic 1	Logic 1
$\overline{\text{VAD}}$	**	Z
XMD0—XMD1	†	Z

Note: Z = High impedance state.

- * Open drain output not actively driven under this condition.
- ** Not guaranteed to be logic 1 (i.e., physical address) until approximately 66 clock cycles after $\overline{\text{RESET}}$ is negated.
- † Not guaranteed to be in kernel mode until approximately 18 clock cycles after $\overline{\text{RESET}}$ is negated.

BUS OPERATION

Reset Sequence



Note: $\overline{\text{RESETR}}$ must be asserted for at least four clock cycles to be recognized on power-up; otherwise, two cycles. $\overline{\text{RESET}}$ is negated 128 clock cycles after negation of $\overline{\text{RESETR}}$.

Figure 4-33. Reset Sequence

4.10.3 Reset Sequence

The reset sequence is depicted on Figure 4-33. As previously stated, after $\overline{\text{RESETR}}$ is sampled for at least two consecutive clock cycles, the CPU issues the reset acknowledge. While $\overline{\text{RESETR}}$ is active, the CPU holds $\overline{\text{RESET}}$ active. Once $\overline{\text{RESETR}}$ is removed by the requesting device, the CPU counts 128 clock cycles and then removes $\overline{\text{RESET}}$. At this point, the CPU enters the internal reset sequence (see Chapter 6).

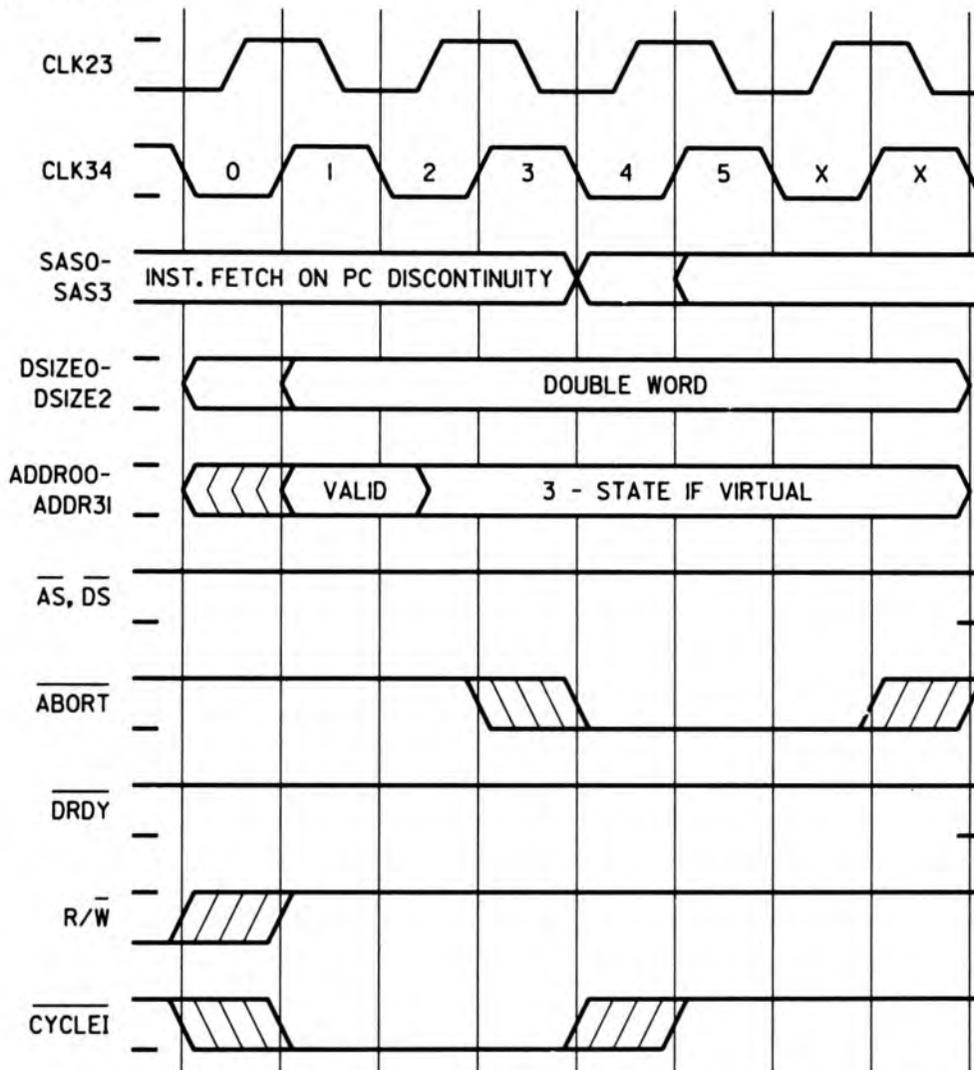
Note that, if the CPU receives a fault during certain high-level bus transactions, it can enter a reset exception. This internal reset sequence goes through a simulated system reset and includes issuing $\overline{\text{RESET}}$ for 128 clock cycles (see section 4.10.2).

4.11 ABORTED MEMORY ACCESSES

There are two events that cause the CPU to abort a memory access: when the CPU has a PC discontinuity with an instruction cache hit, and when an alignment fault occurs.

4.11.1 Aborted Access on PC Discontinuity with Instruction Cache Hit

Figure 4-34 illustrates the protocol associated with this event. When the CPU does a PC discontinuity, it starts to fetch the next instruction word from memory. The SAS code is "instruction fetch after PC discontinuity." If there is a hit in the cache for this instruction fetch, the CPU cancels the external instruction fetch by terminating the transaction. The CPU ignores memory acknowledges and bus exceptions during this transaction. To indicate that it is terminating the transaction, the CPU issues $\overline{\text{ABORT}}$ for two cycles, starting with clock state four. No $\overline{\text{DRDY}}$ is issued and the CPU ignores the data bus. The CPU uses the instruction word that it obtained from the instruction cache. Note that $\overline{\text{AS}}$ and $\overline{\text{DS}}$ are not issued; therefore, a memory system activated by $\overline{\text{DS}}$ will not notice this aborted transaction. The same is true for alignment faults.



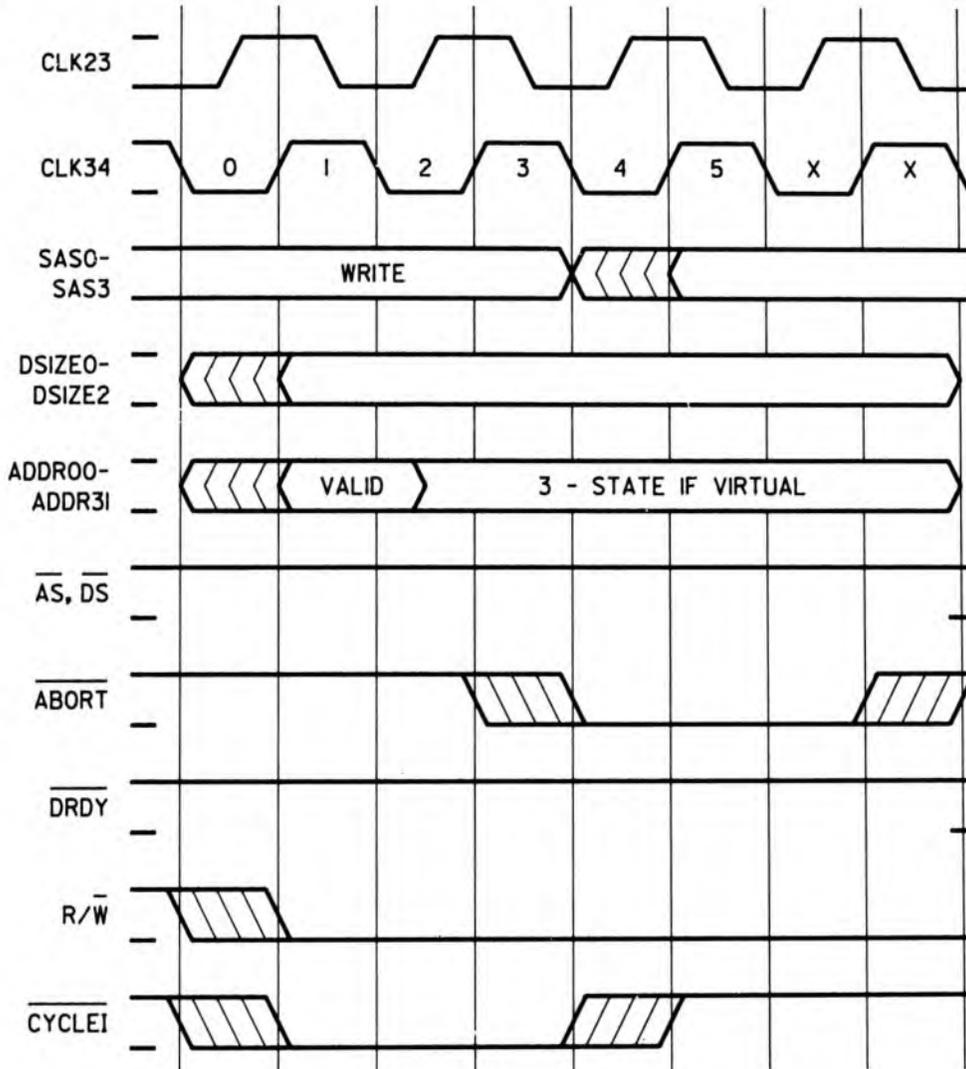
Note: $\overline{\text{BLKFTCH}}$, $\overline{\text{DATA00}}-\overline{\text{DATA31}}$, $\overline{\text{DTACK}}$, $\overline{\text{FAULT}}$, $\overline{\text{RETRY}}$, $\overline{\text{RRREQ}}$, $\overline{\text{SRDY}}$, and $\overline{\text{DYN16}}$ are ignored.

Figure 4-34. Aborted Access on Instruction-Cache Hit with PC Discontinuity

BUS OPERATION
Alignment Fault Bus Activity

4.11.2 Alignment Fault Bus Activity

If the CPU detects an alignment fault on an intended CPU-generated bus transaction (when arbitrary byte alignment is disabled), it terminates the transaction and proceeds to the fault handler. The write transaction on Figure 4-35 starts with the address bus, as well as the DSIZE, SAS, R/W, and $\overline{\text{CYCLEI}}$, being driven by the CPU. The CPU detects the alignment fault and does not issue $\overline{\text{AS}}$ and $\overline{\text{DS}}$. It issues $\overline{\text{ABORT}}$, starting at clock state three, to indicate that it is terminating the transaction. The CPU ignores memory acknowledges, and bus exceptions during this time. $\overline{\text{DRDY}}$ is not issued.



Notes:

DATA00—DATA31, $\overline{\text{DTACK}}$, $\overline{\text{FAULT}}$, $\overline{\text{RETRY}}$, $\overline{\text{RRREQ}}$, $\overline{\text{SRDY}}$, and $\overline{\text{DYN16}}$ are ignored.

Protocol is the same for a read transaction

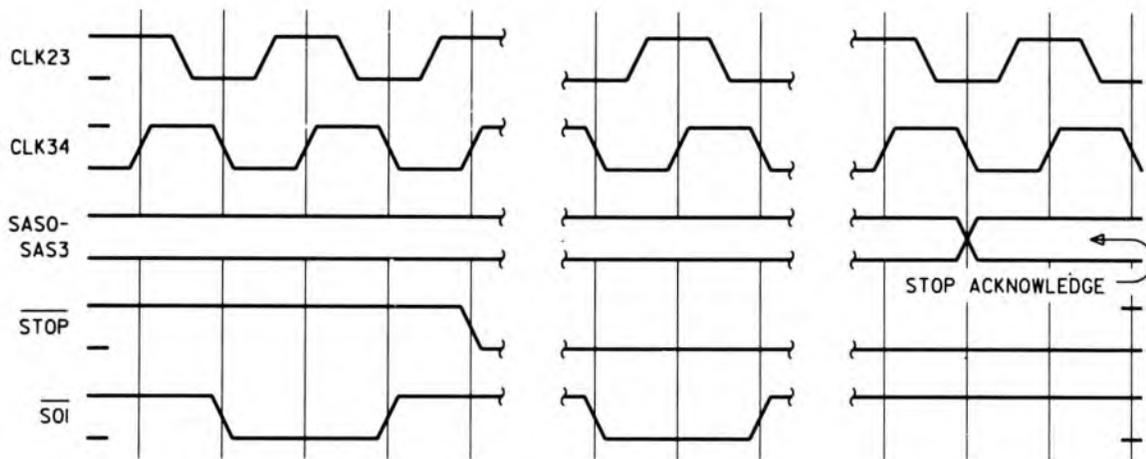
Figure 4-35. Alignment Fault Bus Activity (Write Transaction is Shown)

4.12 SINGLE-STEP OPERATION

Hardware single-step can be performed by use of the stop ($\overline{\text{STOP}}$) input. This input halts the execution of instructions beyond the ones already started by the microprocessor. Because of the pipelined architecture, the CPU may execute, at most, one more instruction beyond the instruction during which $\overline{\text{STOP}}$ was asserted. The microprocessor then remains in a halt state until the $\overline{\text{STOP}}$ input is released.

A bus request is honored while the microprocessor is halted. Additionally, interrupts are acknowledged upon release of $\overline{\text{STOP}}$, but not while $\overline{\text{STOP}}$ remains asserted.

Figure 4-36 illustrates the start of single-step operation. The operation is started by the assertion of $\overline{\text{STOP}}$. The CPU completes the current instruction and executes, at most, one more instruction. After this the CPU stops execution and issues the SAS code "stop acknowledge." The CPU remains in this state until $\overline{\text{STOP}}$ is released.



Notes:

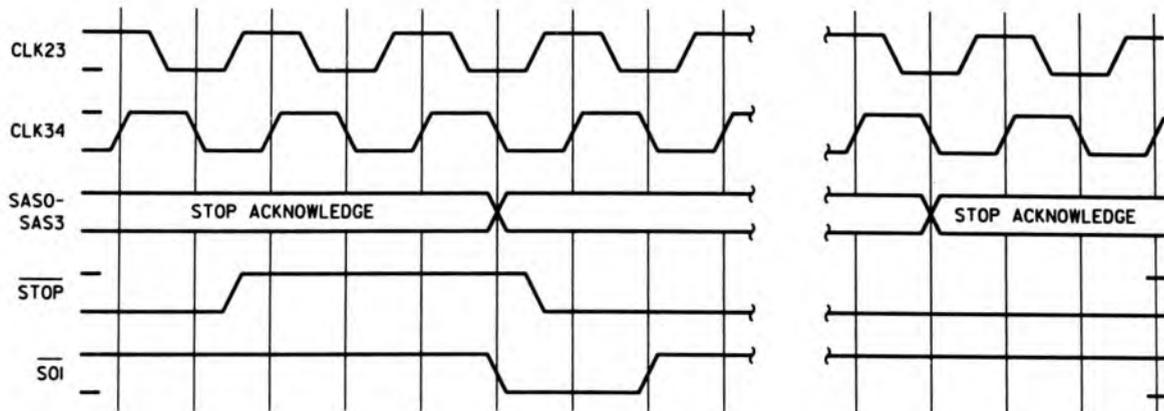
At most, one full assertion of $\overline{\text{SOI}}$ may appear before $\overline{\text{STOP}}$ is acknowledged.

$\overline{\text{BARB}} = 0$ and $\overline{\text{BRACK}} = 1$ in order to see stop acknowledge access status code.

Figure 4-36. Start of Single-Step Operation

BUS OPERATION

Coprocessor Operations



Note: $\overline{BARB} = 0$ and $\overline{BRACK} = 1$ in order to see stop acknowledge access status code.

Figure 4-37. Single-Step Operation

After the CPU has stopped and until a start of instruction (\overline{SOI}) output is issued, instruction by instruction execution can be performed by releasing \overline{STOP} and keeping it released until an \overline{SOI} output is issued. At this point, immediate application and holding of \overline{STOP} prevents a second instruction from starting. With \overline{STOP} asserted, the CPU completes the instruction and issues the stop acknowledge SAS code. To resume normal execution, \overline{STOP} must be completely released. The single-step operation is shown on Figure 4-37.

4.13 COPROCESSOR OPERATIONS

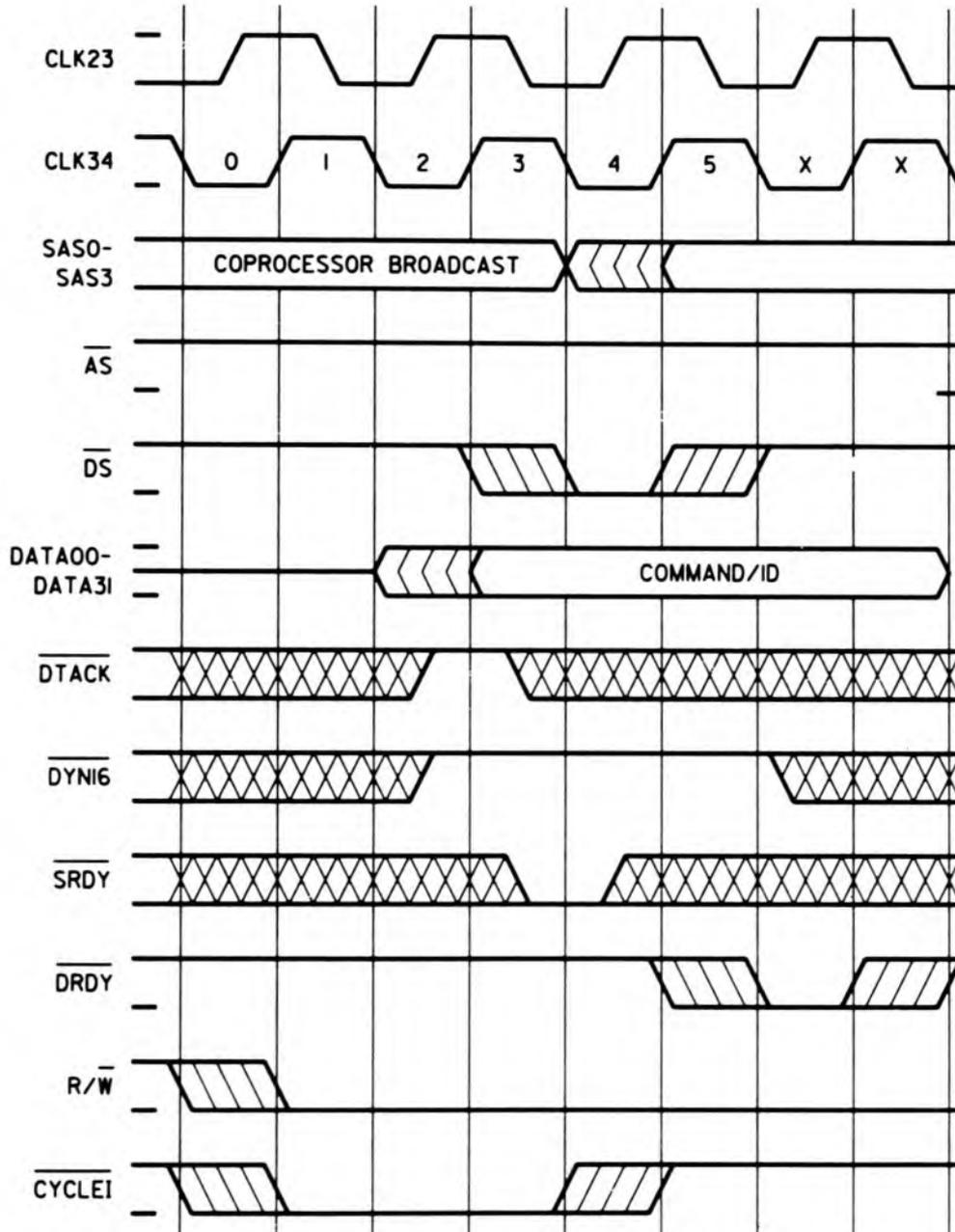
The WE 32200 Microprocessor provides a coprocessor interface consisting of ten instructions and the associated pinout and bus transactions. The coprocessor interface assures high performance and system throughput. When a coprocessor instruction is executed by the CPU, a series of bus transactions occur.

4.13.1 Coprocessor Broadcast

This transaction notifies the coprocessor of the action the CPU wants performed. To prevent memory from being selected, \overline{AS} is not issued during this transaction. Since this is a write operation, R/\overline{W} is in write mode and the timing of \overline{DS} is for a write. The CPU drives the data bus with the information that it wants to send to the coprocessor.

BUS OPERATION Coprocessor Broadcast

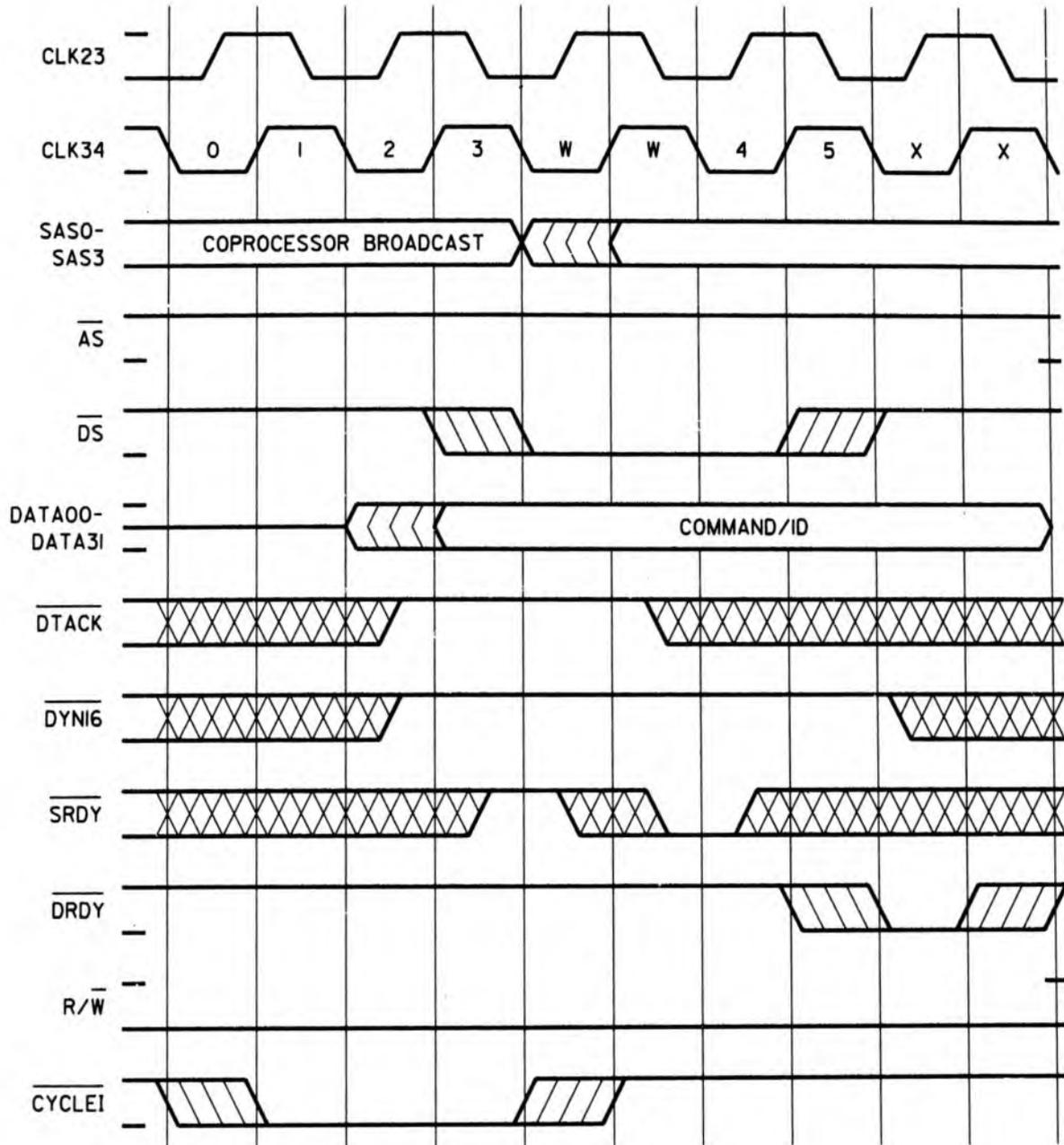
The coprocessor responds with a memory acknowledge. The CPU then terminates the transaction and goes on to the next one. The CPU inserts up to two wait cycles while it waits for the memory acknowledge from the coprocessor. This gives the coprocessor a limited time to respond to this transaction. Figure 4-38 shows the zero, one, and two wait cycle cases before the coprocessor responds with a memory acknowledge (in this case, \overline{SRDY}).



Note: Zero wait cycles.

Figure 4-38. Coprocessor Command and ID Transfer (Sheet 1 of 3)

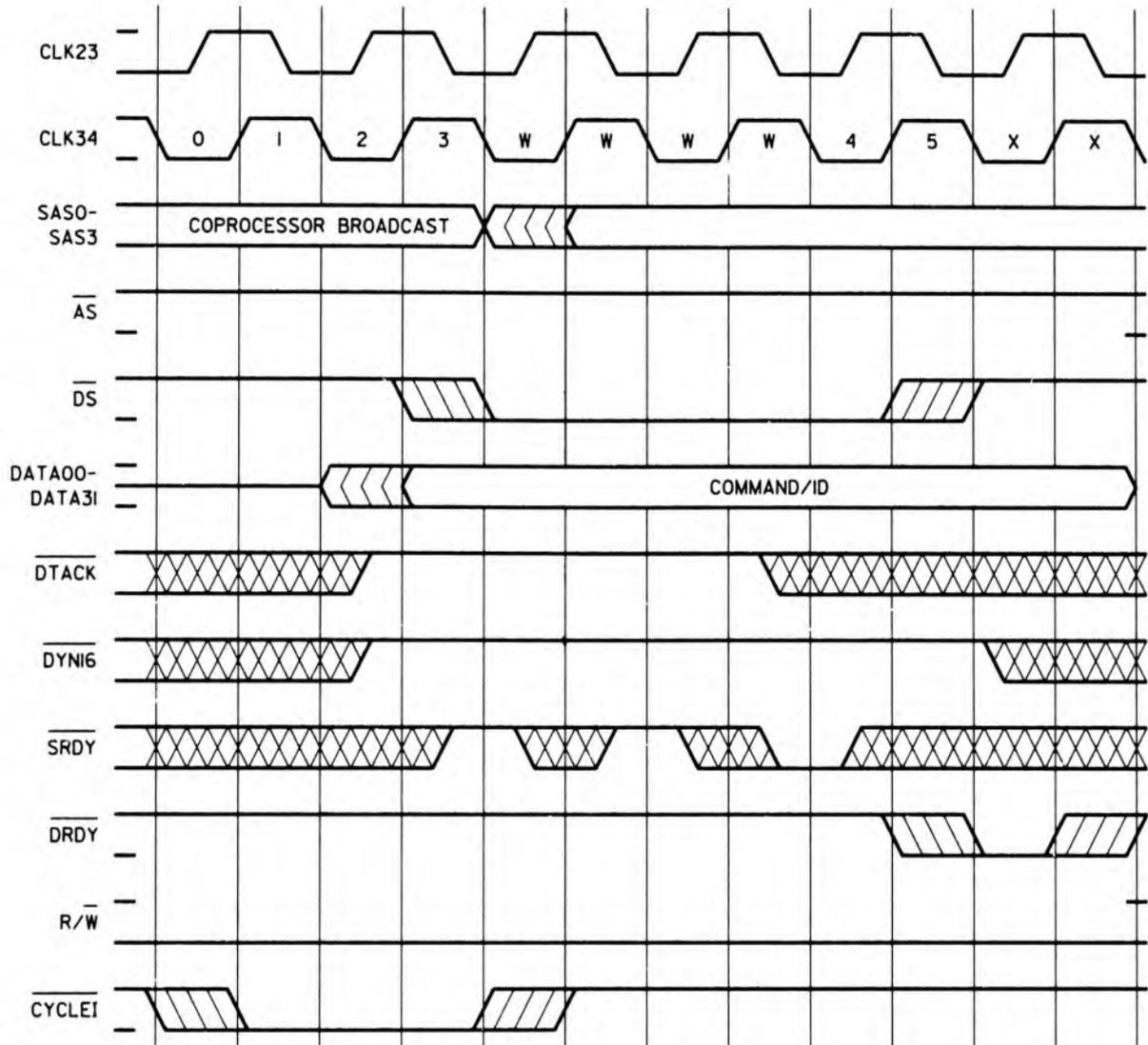
BUS OPERATION
Coprocessor Broadcast



Note: One wait cycle.

Figure 4-38. Coprocessor Command and ID Transfer (Sheet 2 of 3)

BUS OPERATION Coprocesor Broadcast



Notes:

Two wait cycles.

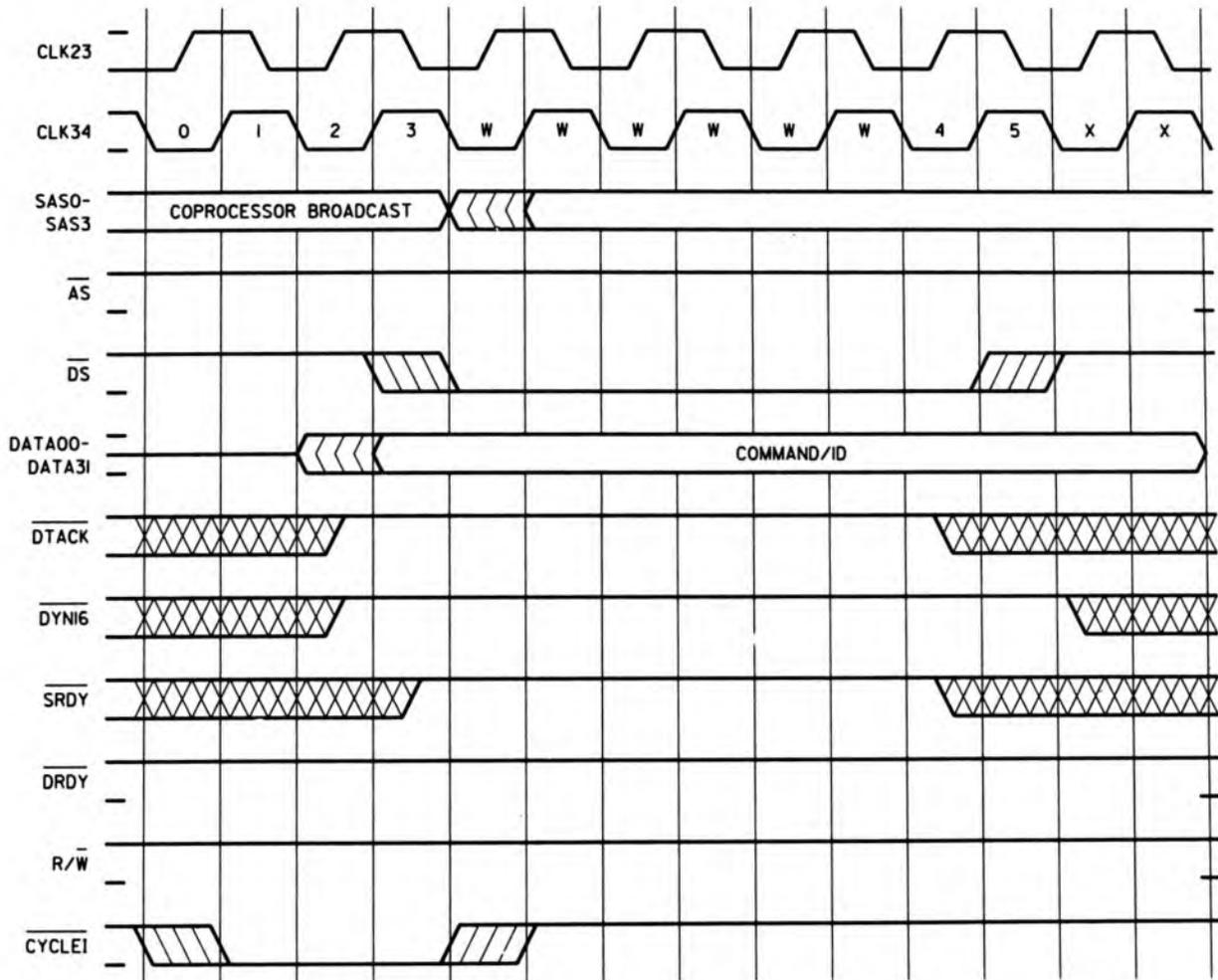
Greater than two wait cycles causes internal CPU memory fault.

Figure 4-38. Coprocessor Command and ID Transfer (Sheet 3 of 3)

BUS OPERATION

Coprocessor Broadcast

If the coprocessor does not respond to the coprocessor broadcast transaction, the CPU generates an internal fault. This is because the coprocessor function can generally be done in software if the hardware is not in the system. The fault takes the CPU to a fault handler, which then should proceed to execute the software version of the coprocessor function. Figure 4-39 depicts a case in which the CPU has not seen a memory acknowledge during 0, 1, or 2 wait cycles. The CPU then internally faults this transaction and proceeds to the fault handler. \overline{DRDY} is not issued.



Notes:

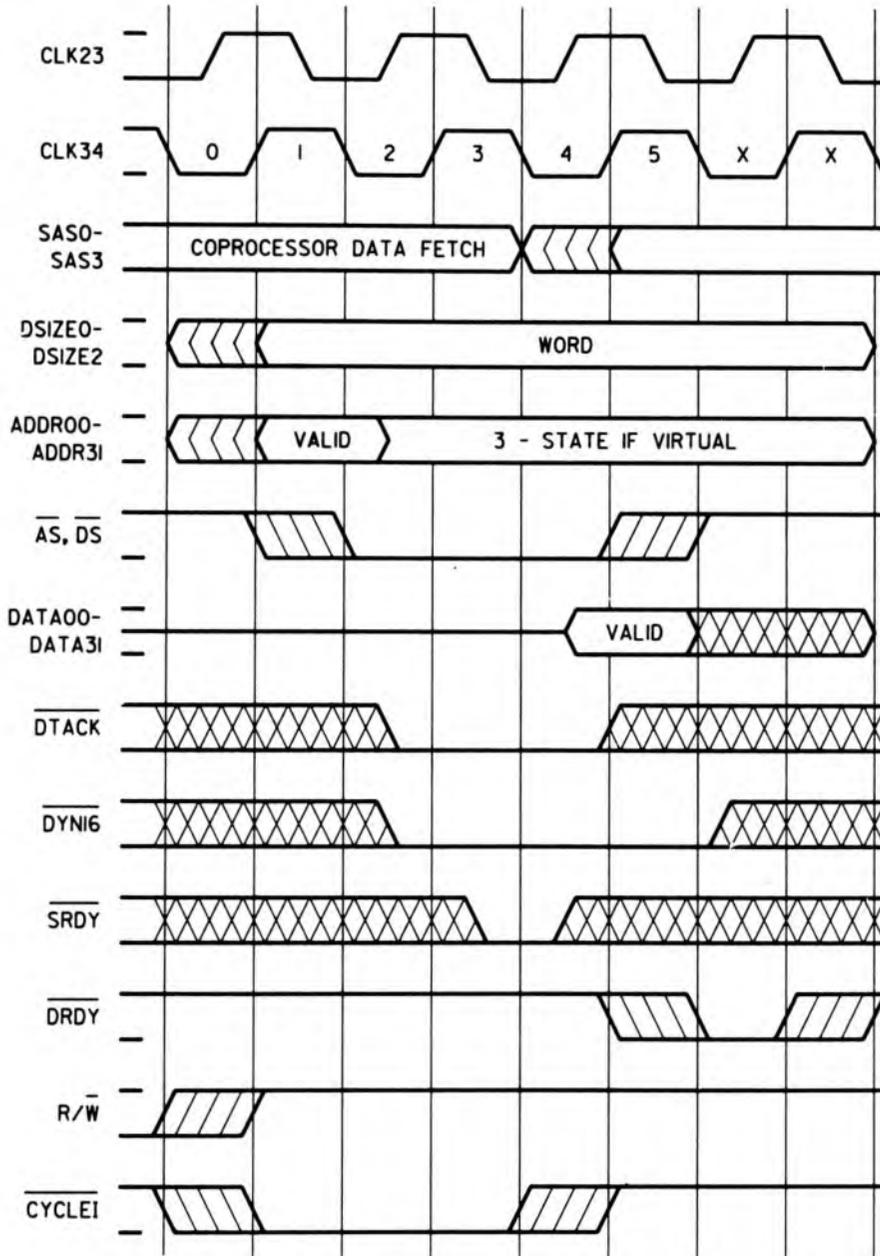
$\overline{RRREQ} = 1$, $\overline{RETRY} = 1$, $\overline{FAULT} = 1$.

No acknowledge (\overline{DTACK} , \overline{SRDY} , $\overline{DYN16}$) and no bus exceptions (\overline{RETRY} , \overline{RRREQ} , \overline{FAULT}).

Figure 4-39. Coprocessor Command and ID Transfer (No Coprocessor Present)

4.13.2 Coprocessor Operand Fetch

After doing a broadcast, the CPU performs from zero to three coprocessor operand fetch transactions, depending on what coprocessor instruction is being executed. For this transaction, the CPU goes through the motions of doing a read from the memory, but the coprocessor latches the data on the bus. The SAS is "coprocessor data fetch." The memory issues the acknowledge for this transaction. Figure 4-40 shows the protocol for a single coprocessor operand fetch.



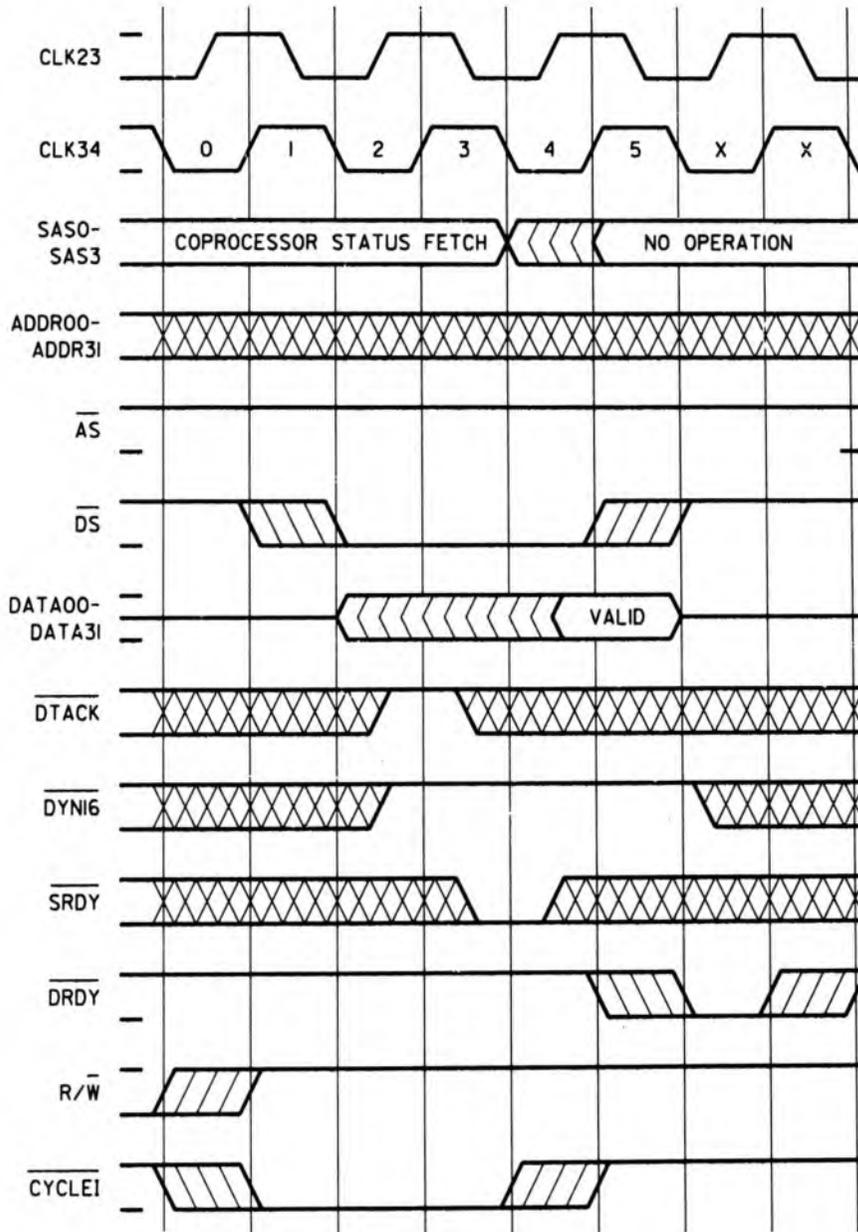
Note: Zero wait cycles use of \overline{DTACK} , $\overline{DYN16}$, or \overline{SRDY} .

Figure 4-40. Coprocessor Operand Fetch

BUS OPERATION
Coprocessor Status Fetch

4.13.3 Coprocessor Status Fetch

After processing the data latched during the coprocessor operand fetch transaction, the coprocessor indicates that it is finished by asserting the coprocessor done (\overline{DONE}) input of the CPU. Approximately two clock cycles later, the CPU initiates the coprocessor status fetch transaction (shown on Figure 4-41). This is a read type transaction in which the coprocessor drives the data bus with status information. There is no \overline{AS} issued to prevent memory from being accessed. The SAS code is "coprocessor status fetch."

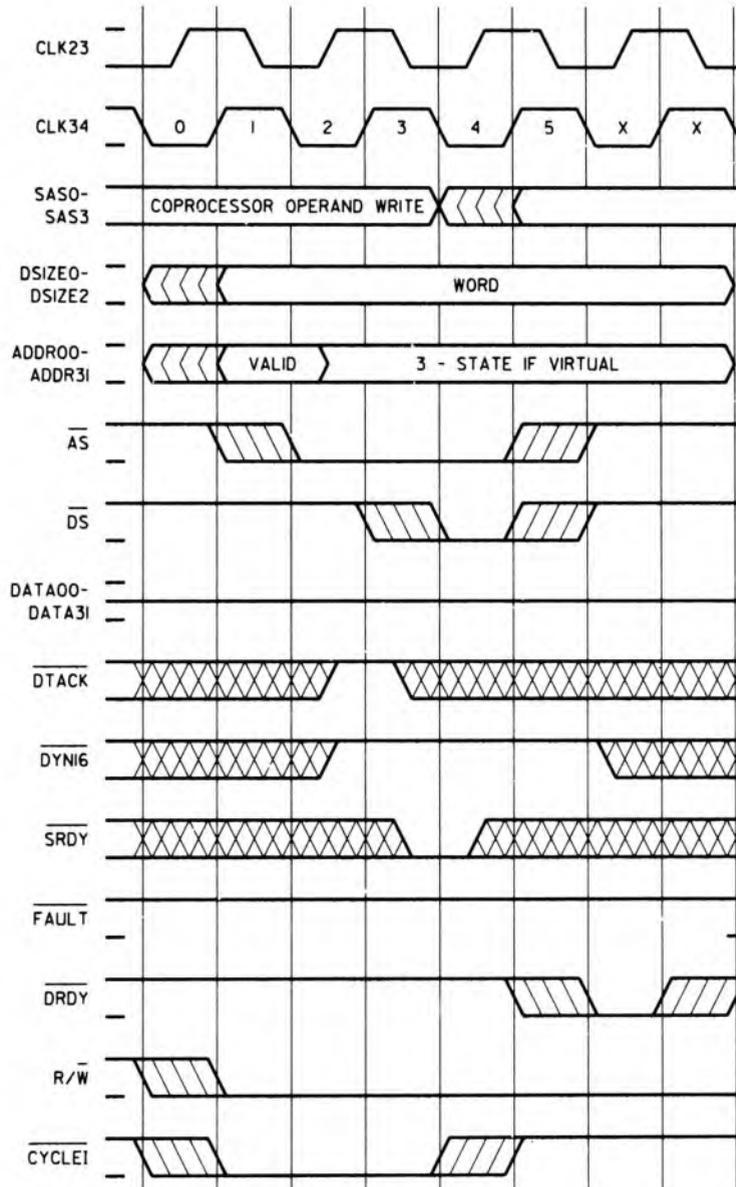


Note: Coprocessor status fetch begins approximately two clock cycles after the microprocessor's coprocessor done (\overline{DONE}) input has been asserted.

Figure 4-41. Coprocessor Status Fetch Using \overline{SRDY}

4.13.4 Coprocessor Data Write

After the coprocessor status fetch, the CPU performs zero to three coprocessor data write transactions, depending on the coprocessor instruction executed. For this transaction the CPU goes through the motions of a write to memory, but the coprocessor drives the data bus with the results for memory. The CPU does not drive the data bus during this transaction. The SAS is "coprocessor data write." The memory issues the acknowledge for this transaction. Figure 4-42 shows the protocol for a single coprocessor data write.



Note: Zero wait cycles using $\overline{\text{SRDY}}$.
 * DATA00—DATA31 supplied by coprocessor.

Figure 4-42. Coprocessor Data Write

BUS OPERATION

Supplementary Protocol Diagrams

4.14 SUPPLEMENTARY PROTOCOL DIAGRAMS

The following supplementary protocol diagrams are provided:

Figure 4-43. Read Transaction Followed by a Read Transaction – $\overline{\text{DTACK}}$ Only.

Figure 4-44. Read Transaction Followed by a Write Transaction – $\overline{\text{DTACK}}$ Only.

Figure 4-45. Write Transaction Followed by a Write Transaction – $\overline{\text{DTACK}}$ Only.

Figure 4-46. Write Transaction Followed by a Write Transaction – $\overline{\text{DYN16}}$ and $\overline{\text{DTACK}}$.

Figure 4-47. Write Transaction Followed by a Read Transaction.

Figure 4-48. Double-Word Instruction Fetch Without Blockfetch Transaction – $\overline{\text{DTACK}}$ Only.

Figure 4-49. Bus Arbitration During Relinquish and Retry.

Figure 4-50. Faulted Coprocessor Status Read

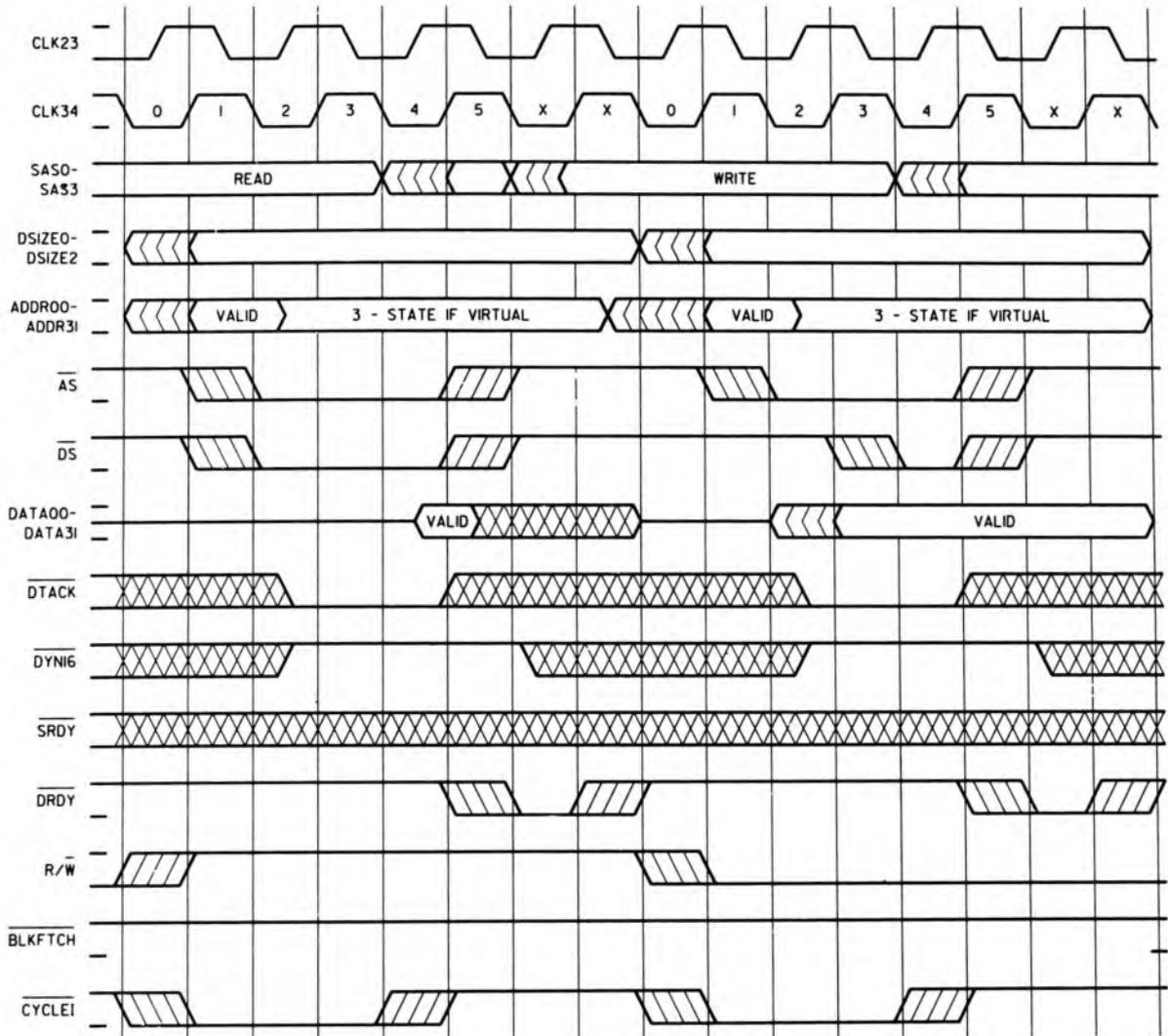
Figure 4-51. Timing of IQS and $\overline{\text{SOI}}$ Signals

Figure 4-52. Assertion of $\overline{\text{STOP}}$ After $\overline{\text{SOI}}$

Figure 4-53. Assertion of $\overline{\text{STOP}}$ With $\overline{\text{SOI}}$

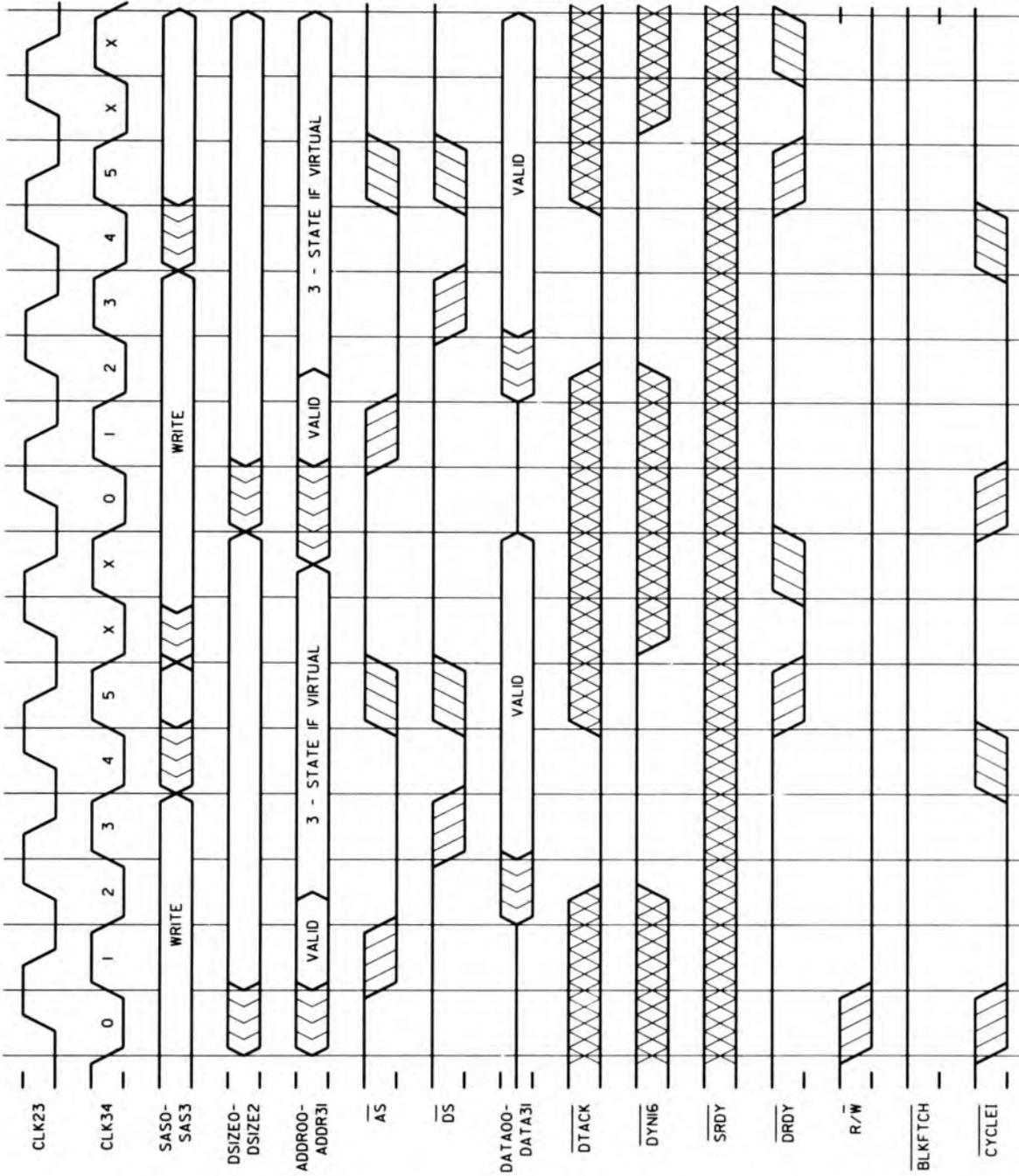
BUS OPERATION

Supplementary Protocol Diagrams



Note: Zero wait cycles.

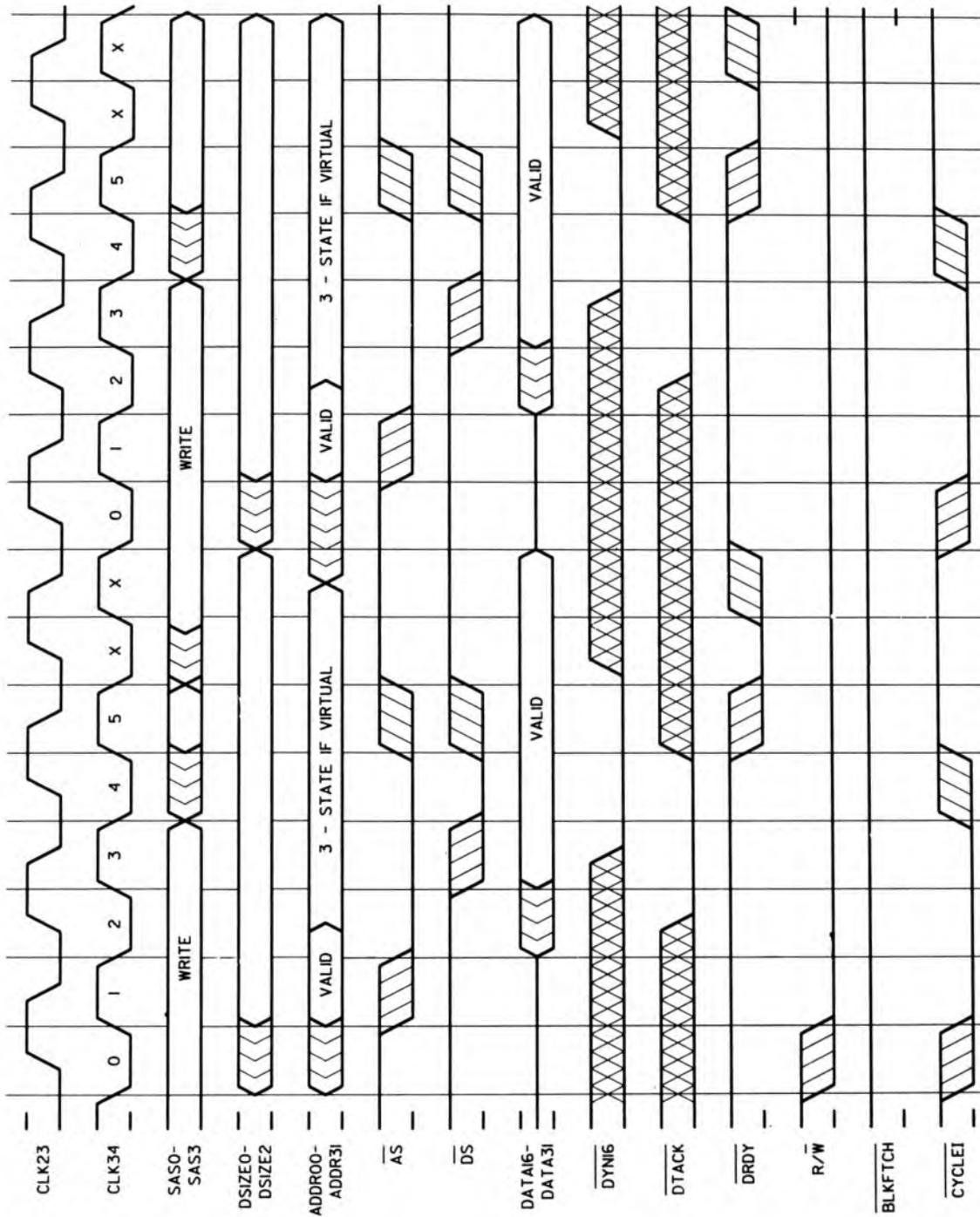
Figure 4-44. Read Transaction Followed by a Write Transaction – \overline{DTACK} Only



Note: Zero wait cycles.

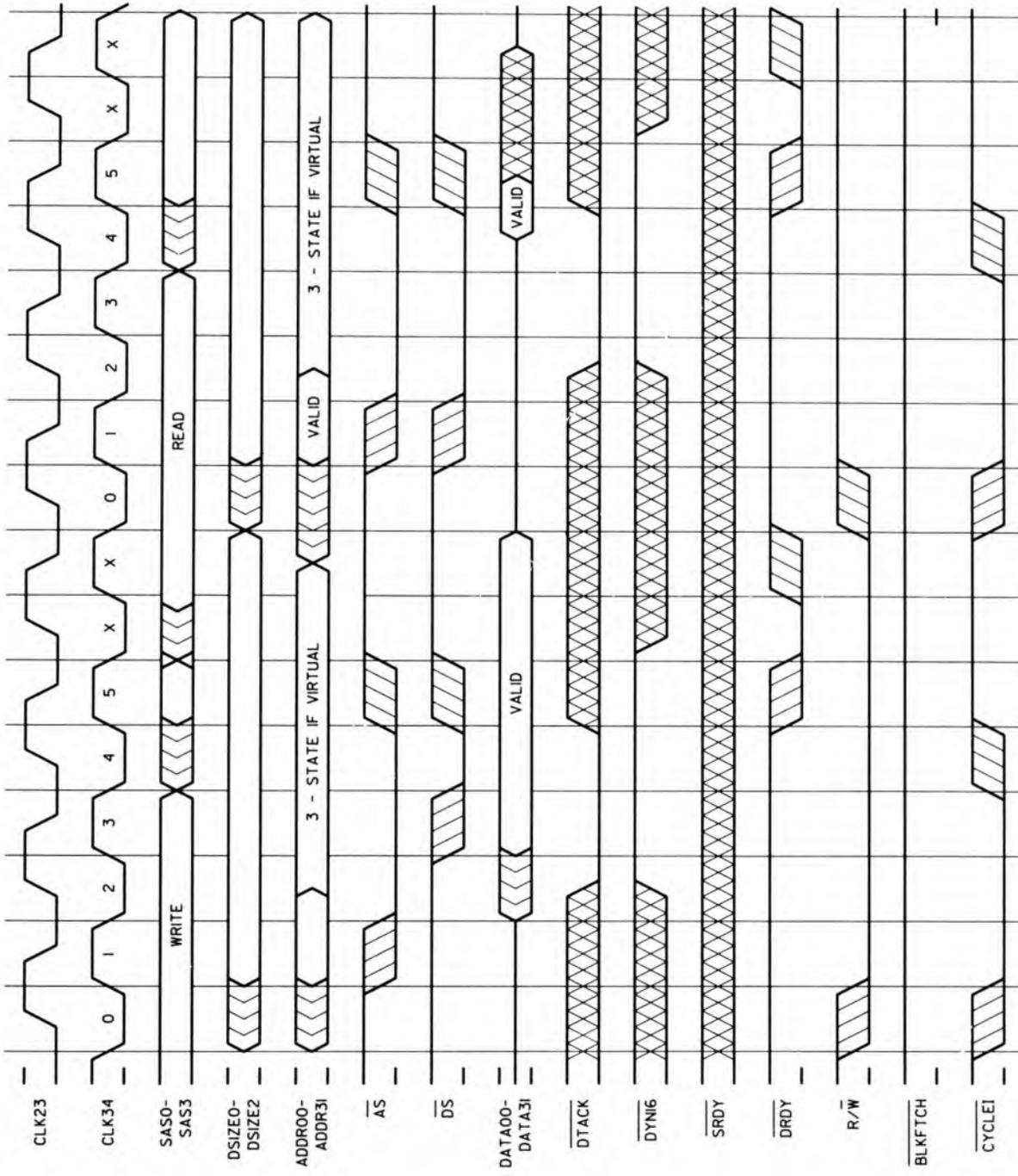
Figure 4-45. Write Transaction Followed by a Write Transaction — \overline{DTACK} Only

BUS OPERATION
Supplementary Protocol Diagrams



Note: Zero wait cycles.

Figure 4-46. Write Transaction Followed by a Write Transaction — $\overline{\text{DYN16}}$ and $\overline{\text{DTACK}}$

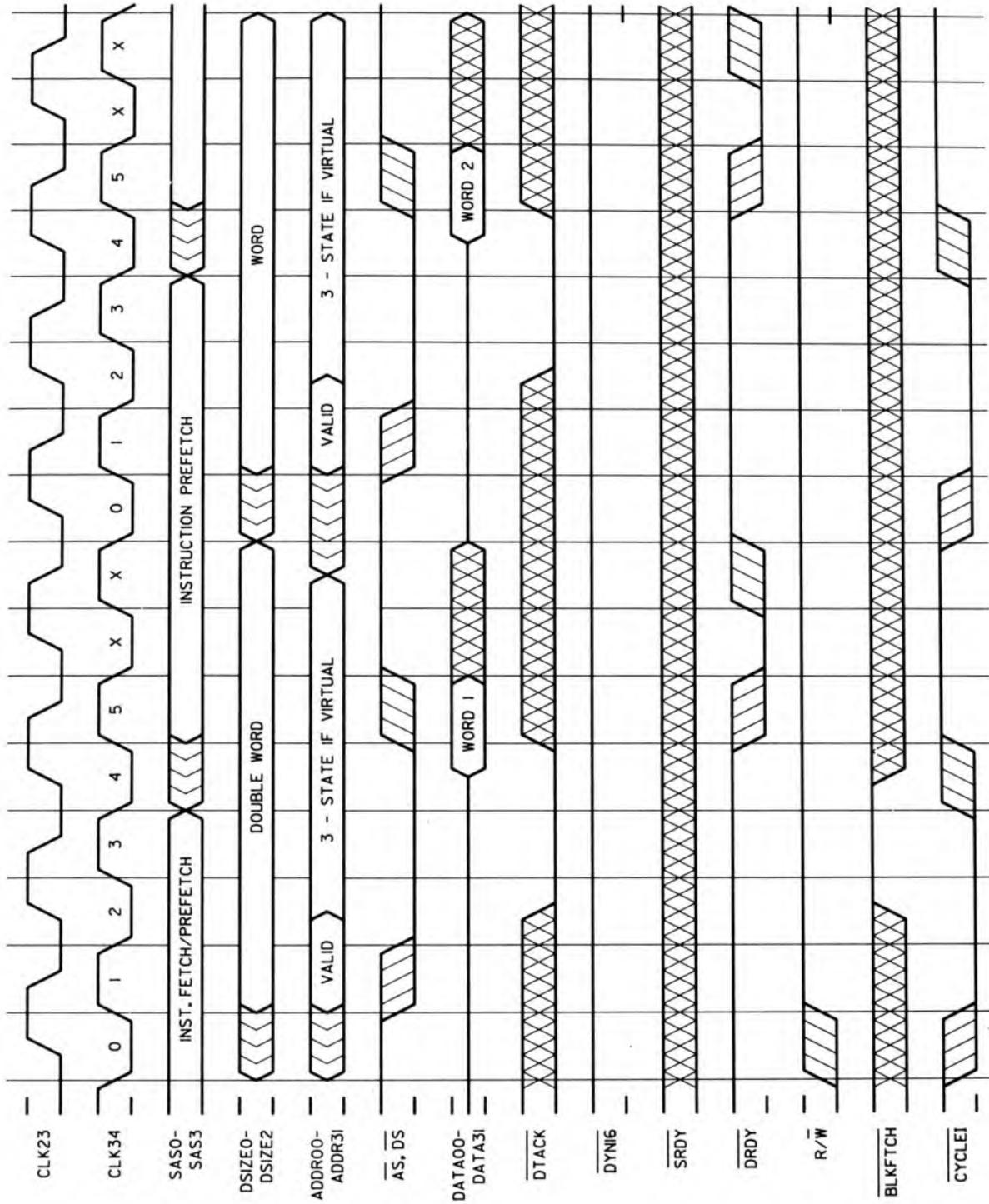


Note: Zero wait cycles.

Figure 4-47. Write Transaction Followed by a Read Transaction

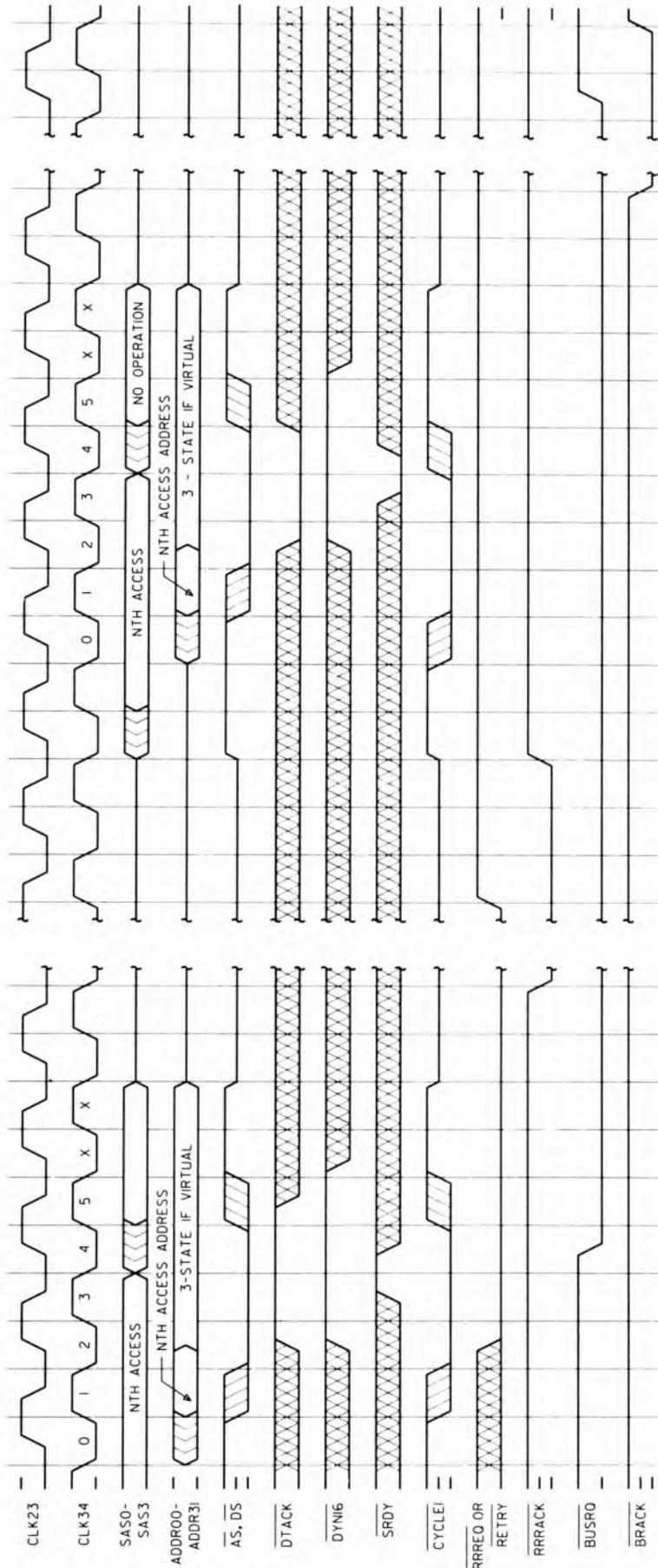
BUS OPERATION

Supplementary Protocol Diagrams



Note: Zero wait cycles.

Figure 4-48. Double-Word Instruction Fetch Without Blockfetch Transaction – \overline{DTACK} Only



Note: The same protocol diagram applies for retry and bus arbitration except that the address bus, data bus, and control signals are not 3-stated during the time RETRY is active and RRRACK is not issued.

Figure 4-49. Bus Arbitration During Relinquish and Retry

BUS OPERATION
Supplementary Protocol Diagrams

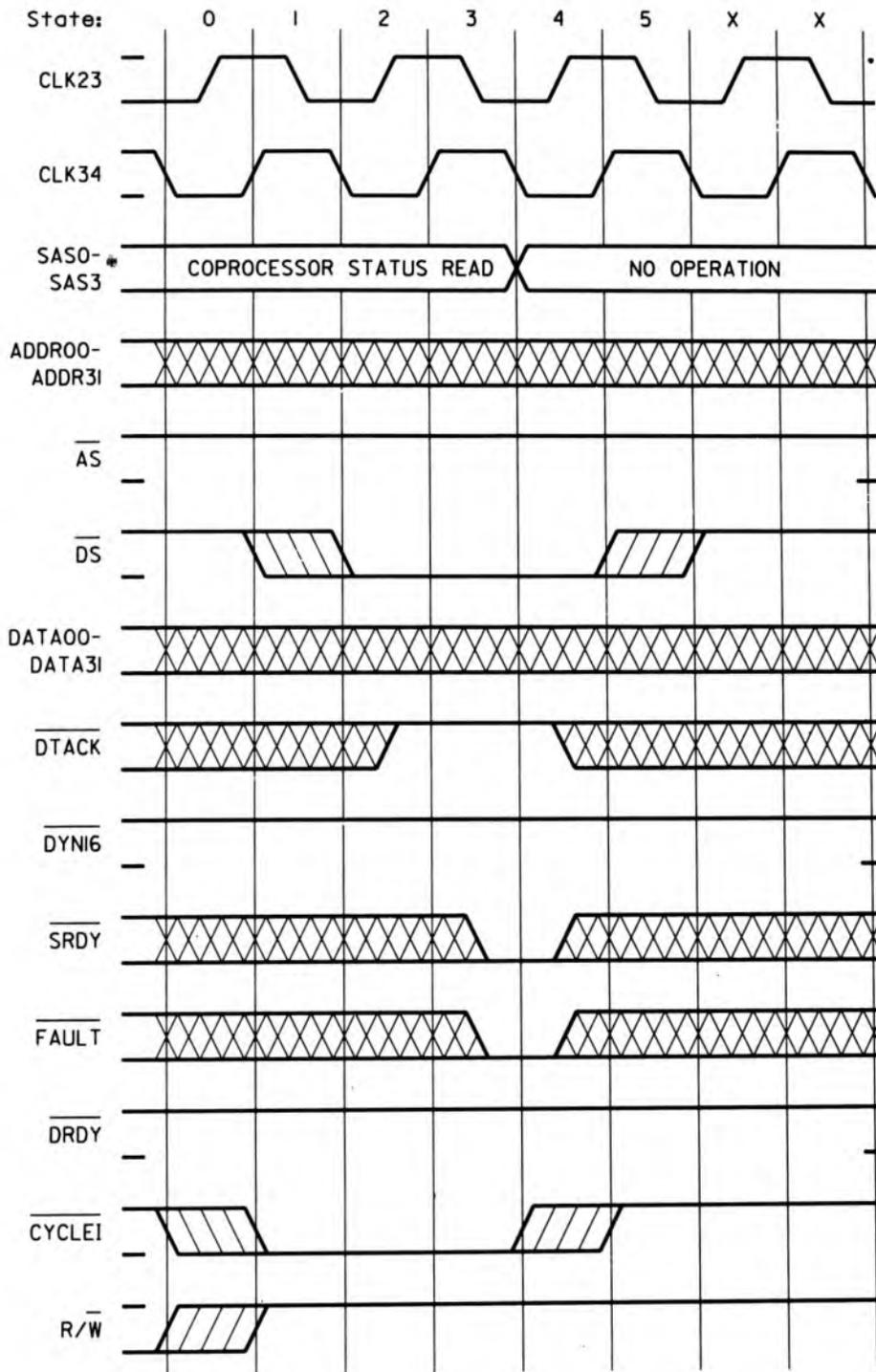
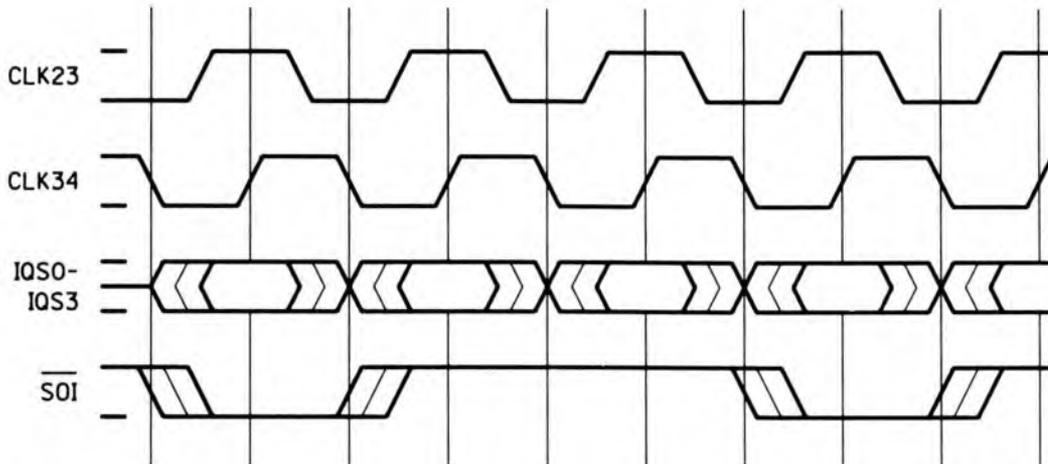


Figure 4-50. Faulted Coprocessor Status Read

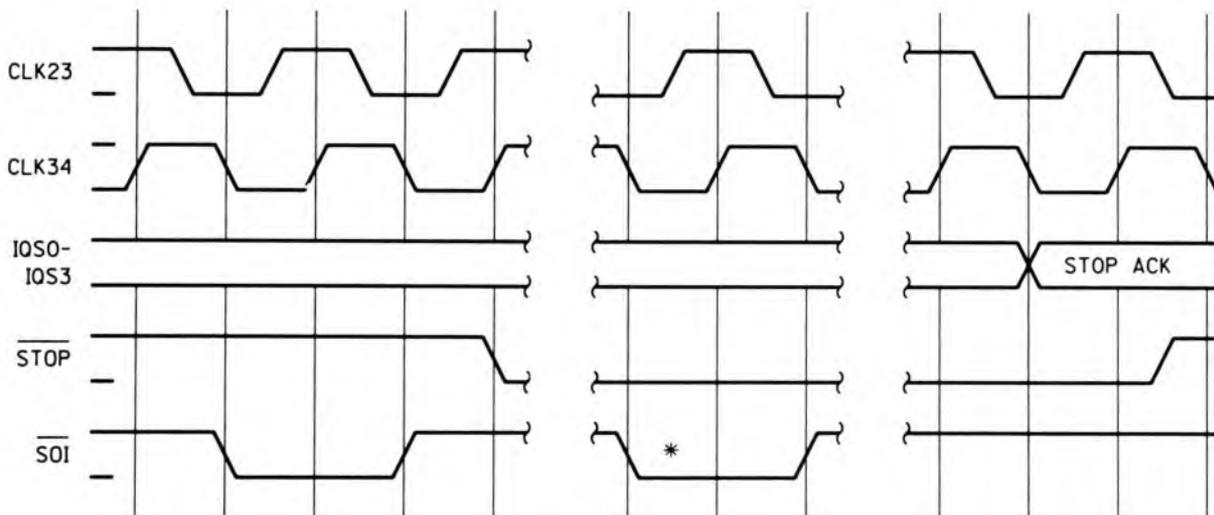


Notes:

IQS signals are valid every clock cycle and are guaranteed to be stable and valid at mid-cycle.

$\overline{\text{SOI}}$ is valid for only one cycle and is guaranteed to be stable and valid at mid-cycle.

Figure 4-51. Timing of IQS and $\overline{\text{SOI}}$ Signals

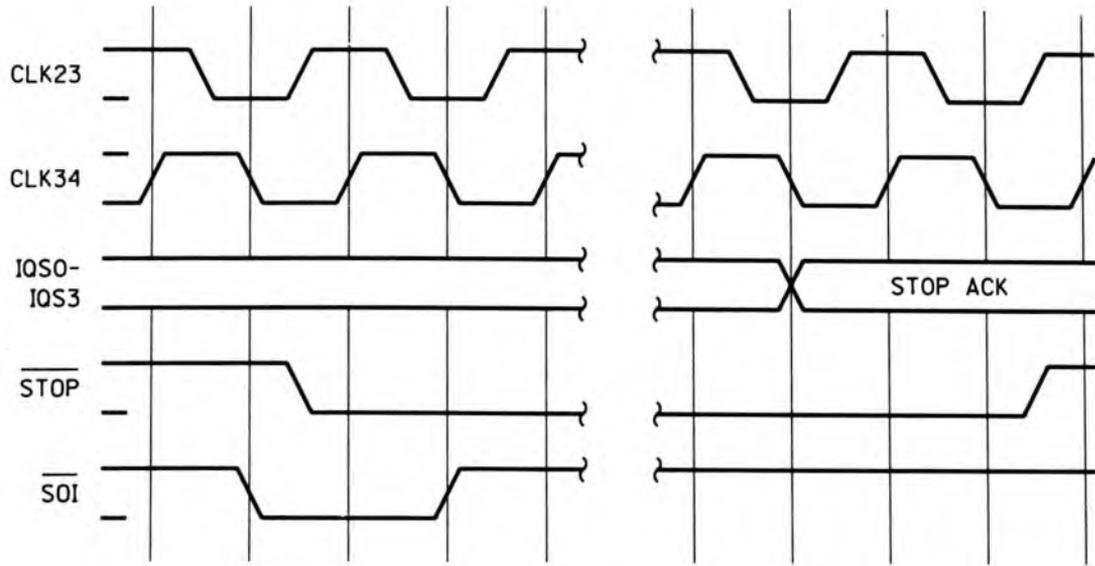


- * At most, 1 full assertion of $\overline{\text{SOI}}$ may appear before $\overline{\text{STOP}}$ is acknowledged. Both $\overline{\text{BARB}} = 0$ and $\overline{\text{BRACK}} = 1$ in order to see $\overline{\text{STOP}}$ acknowledge.

Figure 4-52. Assertion of $\overline{\text{STOP}}$ After $\overline{\text{SOI}}$

BUS OPERATION

Supplementary Protocol Diagrams



Note: Both $\overline{\text{BARB}} = 0$ and $\overline{\text{BRACK}} = 1$ in order to see $\overline{\text{STOP}}$ acknowledge.

Figure 4-53. Assertion of $\overline{\text{STOP}}$ with $\overline{\text{SOI}}$

Chapter 5
Instruction Set
and
Addressing Modes

CHAPTER 5. INSTRUCTION SET AND ADDRESSING MODES

CONTENTS

5. INSTRUCTION SET AND ADDRESSING MODES	5-1
5.1 REGISTERS	5-1
5.2 ADDRESSING MODES	5-3
5.2.1 Format 1 Addressing Modes	5-5
Register Mode	5-9
Register Deferred Mode	5-11
Displacement Mode	5-12
Deferred Displacement Mode	5-14
Immediate Mode	5-15
Absolute Mode	5-16
Absolute Deferred Mode	5-16
Expanded Operand Mode	5-18
5.2.2 Format 2 Addressing Modes	5-20
Auto Pre/Post Increment/Decrement	5-20
Indexed Register Modes	5-21
Format 1 Addressing Mode for Registers 16-31	5-25
5.3 INSTRUCTION SET FUNCTIONAL GROUPS	5-27
5.3.1 Data Transfer Instructions	5-28
5.3.2 Arithmetic Instructions	5-29
5.3.3 BCD Instructions	5-31
5.3.4 Logical Instructions	5-32
5.3.5 Program Control Instructions	5-34
5.3.6 Coprocessor Instructions	5-40
5.3.7 Stack and Miscellaneous Instructions	5-41
5.4 INSTRUCTION SET LISTINGS	5-42
5.4.1 Notation	5-42
5.4.2 Instruction Set Summary by Mnemonic	5-44
5.4.3 Instruction Set Summary by Opcode	5-49
5.4.4 Instruction Set Descriptions	5-54
Add (ADDB2, ADDH2, ADDW2)	5-55
Add, 3 Address (ADDB3, ADDH3, ADDW3)	5-56
Add Packed Decimal (ADDPB2)	5-57
Add Packed Decimal, 3 Address (ADDPB3)	5-58
Arithmetic Left Shift (ALSW3)	5-59
And (ANDB2, ANDH2, ANDW2)	5-60
And, 3 Address (ANDB3, ANDH3, ANDW3)	5-61
Arithmetic Right Shift (ARSB3, ARSH3, ARSW3)	5-62
Branch on Carry Clear (BCCB, BCCH)	5-63
Branch on Carry Set (BCSB, BCSH)	5-64
Branch on Equal (BEB, BEH)	5-65
Branch on Greater Than (Signed) (BGB, BGH)	5-66
Branch on Greater Than or Equal (Signed) (BGEB, BGEH)	5-67
Branch on Greater Than or Equal (Unsigned) (BGEUB, BGEUH)	5-68
Branch on Greater Than (Unsigned) (BGUB, BGUH)	5-69

Bit Test (BITB, BITH, BITW).....	5-70
Branch on Less Than (Signed) (BLB, BLH).....	5-71
Branch on Less Than or Equal (Signed) (BLEB, BLEH).....	5-72
Branch on Less Than or Equal (Unsigned) (BLEUB, BLEUH).....	5-73
Branch on Less Than (Unsigned) (BLUB, BLUH).....	5-74
Branch on Not Equal (BNEB, BNEH).....	5-75
Breakpoint Trap (BPT).....	5-76
Branch (BRB, BRH).....	5-77
Branch to Subroutine (BSBB, BSBH).....	5-78
Branch on Overflow Clear (BVCB, BVCH).....	5-79
Branch on Overflow Set (BVSB, BVSH).....	5-80
Call Procedure (CALL).....	5-81
Compare and Swap Word Interlock (CASWI).....	5-82
Cache Flush (CFLUSH).....	5-83
Clear (CLRB, CLRH, CLRW).....	5-84
Clear X Bit (CLRX).....	5-85
Compare (CMPB, CMPH, CMPW).....	5-86
Decrement (DECB, DECH, DECW).....	5-87
Divide (DIVB2, DIVH2, DIVW2).....	5-88
Divide, 3 Address (DIVB3, DIVH3, DIVW3).....	5-89
Decrement and Test (DTB, DTH).....	5-90
Extract Field (EXTFB, EXTFH, EXTFW).....	5-91
Extended Opcode (EXTOP).....	5-92
Increment (INCB, INCH, INCW).....	5-93
Insert Field (INSFB, INSFH, INSFW).....	5-94
Jump (JMP).....	5-96
Jump to Subroutine (JSB).....	5-97
Logical Left Shift (LLSB3, LLSH3, LLSW3).....	5-98
Logical Right Shift (LRSW3).....	5-99
Move Complemented (MCOMB, MCOMH, MCOMW).....	5-100
Move Negated (MNEGB, MNEGH, MNEGW).....	5-101
Modulo (MODB2, MODH2, MODW2).....	5-102
Modulo, 3 Address (MODB3, MODH3, MODW3).....	5-103
Move Address (Word) (MOVAW).....	5-104
Move (MOVB, MOVH, MOVW).....	5-105
Move Block (MOVBLW).....	5-107
Multiply (MULB2, MULH2, MULW2).....	5-109
Multiply, 3 Address (MULB3, MULH3, MULW3).....	5-110
Move Version Number (MVERNO).....	5-111
No Operation (NOP, NOP2, NOP3).....	5-112
OR (ORB2, ORH2, ORW2).....	5-113
OR, 3 Address (ORB3, ORH3, ORW3).....	5-114
Pack BCD Halfword.....	5-115
Pop (Word) (POPW).....	5-116
Push Address (Word) (PUSHAW).....	5-117
Push (Word) (PUSHW).....	5-118
Return on Carry Clear (RCC).....	5-119
Return on Carry Set (RCS).....	5-119

Return on Equal (REQL, REQLU).....	5-120
Restore Registers (RESTORE).....	5-121
Return from Procedure (RET).....	5-122
Return on Greater Than or Equal (Signed) (RGEQ).....	5-123
Return on Greater Than or Equal (Unsigned) (RGEQU).....	5-123
Return on Greater Than (Signed) (RGTR).....	5-124
Return on Greater Than (Unsigned) (RGTRU).....	5-124
Return on Less Than or Equal (Signed) (RLEQ).....	5-125
Return on Less Than or Equal (Unsigned) (RLEQU).....	5-125
Return on Less Than (Signed) (RLSS).....	5-126
Return on Less Than (Unsigned) (RLSSU).....	5-126
Return on Not Equal (RNEQ, RNEQU).....	5-127
Rotate (ROTW).....	5-128
Return from Subroutine (RSB).....	5-129
Return on Overflow Clear (RVC).....	5-129
Return on Overflow Set (RVS).....	5-130
Save Registers (SAVE).....	5-131
Set X Bit (SETX).....	5-132
Coprocessor Operation (no operands) (SPOP).....	5-133
Coprocessor Operation Read (SPOPRS, SPOPRD, SPOPRT).....	5-134
Coprocessor Operation, 2 Address (SPOPS2, SPOPD2, SPOPT2).....	5-135
Coprocessor Operation Write (SPOPWS, SPOPWD, SPOPWT).....	5-137
String Copy (STRCPY).....	5-138
String End (STREND).....	5-140
Subtract (SUBB2, SUBH2, SUBW2).....	5-141
Subtract, 3 Address (SUBB3, SUBH3, SUBW3).....	5-142
Subtract Packed Decimal (SUBPB2).....	5-143
Subtract Packed Decimal, 3 Address (SUBPB3).....	5-144
Swap (Interlocked) (SWAPBI, SWAPHI, SWAPWI).....	5-145
Test Equal, Decrement, and Test (TEDTB, TEDTH).....	5-146
Test Greater than, Decrement, and Test (TGDTB, TGDTH).....	5-147
Test Greater than or Equal, Decrement, and Test (TGEDTB, TGEDTH).....	5-148
Test not Equal, Decrement, and Test (TNEDTB, TNEDTH).....	5-149
Test (TSTB, TSTH, TSTW).....	5-150
Unpack BCD Byte (UNPACKB).....	5-151
Exclusive Or (XORB2, XORH2, XORW2).....	5-152
Exclusive Or, 3 Address (XORB3, XORH3, XORW3).....	5-153

5. INSTRUCTION SET AND ADDRESSING MODES

The *WE 32200* Microprocessor has a powerful instruction set that includes the standard data transfer, arithmetic, and logical operations for microprocessors, and some unique operating system operations. Its program control instructions (branch, jump, return) provide flexibility for altering the sequence in which instructions are executed. Some of these instructions check the setting of the processor's condition flags before execution. For operating systems, the processor has instructions to establish an environment that permits other processes to take control of the CPU. The special instructions dedicated to operating system use are not discussed in this chapter, but are reserved for Chapter 6.

The microprocessor instructions are mnemonic-based assembly language statements. A mnemonic defines the operation an instruction performs. For most arithmetic or logical operations, the mnemonic also defines one of the data types:

- byte – 8-bit data
- halfword – 16-bit data
- word – 32-bit data

Some instructions perform operations on a bit field, a sequence of 1 to 32 bits contained in a word, or on a block (or string) of data locations. Data types are discussed in section 2.5.

5.1 REGISTERS

A processor register may contain the operand for an instruction or may be used when computing the address of an operand. Therefore, most addressing modes, other than absolute, immediate, or literal, reference a processor register. In general, any of the 32 processor registers, except the processor status word (r11), process control block pointer (r13), interrupt stack pointer (r14), and upper eight (r24 through r31) registers may be used as an operand in all of the addressing modes. Registers r11, r13, r14, r15, and r24 through r31 are privileged registers and can be written to only in kernel mode. They can be used as operands in all addressing modes. Table 5-1 lists the registers and assigned functions.

The general-purpose registers, r0 through r8 and r16 through r31, may be used for accumulation, addressing, or temporary data storage. The remaining processor registers are special-purpose and are usually referenced with different names. Three of these registers are used to store pointers to data on the execution stack:

Register	Name
r9	Frame pointer (FP)
r10	Argument pointer (AP)
r12	Stack pointer (SP)

INSTRUCTION SET AND ADDRESSING MODES

Registers

Function calls and returns affect the AP, FP, and SP implicitly. The FP identifies the starting location of local variables for the function, while the AP identifies the beginning of the set of arguments passed to the function. The SP always points to the next available word location on the stack. Note that the stack grows upward to higher memory addresses.

Register	Name	Assembler Syntax	Assigned Function
0	r0	%r0	General-purpose
1	r1	%r1	General-purpose
2	r2	%r2	General-purpose
3	r3	%r3	General-purpose
4	r4	%r4	General-purpose
5	r5	%r5	General-purpose
6	r6	%r6	General-purpose
7	r7	%r7	General-purpose
8	r8	%r8	General-purpose
9	FP	%fp or %r9	Frame pointer
10	AP	%ap or %r10	Argument pointer
11	PSW	%psw or %r11	Processor status word
12	SP	%sp or %r12	Stack pointer
13	PCBP	%pcbp or %r13	Processor control block pointer
14	ISP	%isp or %r14	Interrupt stack pointer
15	PC	%pc or %r15	Program counter
16	r16	%r16	General-purpose
17	r17	%r17	General-purpose
18	r18	%r18	General-purpose
19	r19	%r19	General-purpose
20	r20	%r20	General-purpose
21	r21	%r21	General-purpose
22	r22	%r22	General-purpose

Table 5-1. Register Set (Continued)			
Register	Name	Assembler Syntax	Assigned Function
23	r23	%r23	General-purpose
24	r24	%r24	General-purpose
25	r25	%r25	General-purpose
26	r26	%r26	General-purpose
27	r27	%r27	General-purpose
28	r28	%r28	General-purpose
29	r29	%r29	General-purpose
30	r30	%r30	General-purpose
31	r31	%r31	General-purpose

Block or string instructions may use registers r0—r2 as implied arguments for indexing or addressing. Operating system instructions also use these registers. The PSW, PCBP, ISP, and registers r24—r31 are privileged registers. Writing to these registers when the CPU is not in kernel execution level causes a privileged-register exception (see section 6.2.1). The PSW is unusual in that some of its fields are read-only, and the flag bits are handled differently when the PSW is a source or destination of an instruction. The PSW can only be addressed in register addressing mode. The PC can only be addressed in word, halfword, byte displacement, and displacement deferred addressing modes. In addition, it is referenced implicitly in all program-control instructions and for all function calls and returns.

Some of the registers have restrictions on usage in instructions. Because registers 11, 13, 14, and 24 through 31 are privileged, these may be written only when kernel execution level is in effect. Register 11, the processor status word (PSW), contains status information about the current instruction and process. Register 13, the process control block pointer (PCBP), identifies a block of status information and pointers for a process. Register 14, the interrupt stack pointer (ISP), functions as a stack pointer for the interrupt stack.

5.2 ADDRESSING MODES

The addressing mode of an operand in an instruction refers to the way in which the location of the operand is determined. A memory-based operand requires an effective address to be determined according to the operand descriptor, and the operand descriptor for register operands specifies the register. Unless specified by the instruction, all operands are addressed by a descriptor.

The *WE 32200* Microprocessor has addressing modes that have been extensively enhanced beyond those of the *WE 32100* Microprocessor, and thus provides two encoding formats to the first byte of the descriptor. If the byte is equal to 0x5B, 0xAB, 0xBB, 0xCB, or 0xDB, then the address mode is a format 2 type, unique to the *WE 32200* Microprocessor. Otherwise, the address mode is a format 1 type, the same as the *WE 32100* Microprocessor addressing modes.

INSTRUCTION SET AND ADDRESSING MODES

Addressing Modes

An assembly language instruction for the WE 32200 Microprocessor consists of a mnemonic, such as ADDW, MOVH, INCB, followed by up to four operand descriptors. Each operand is physically located in: one of the microprocessor's registers, a memory location, an input-output port, or directly within the instruction. The operand specified by the assembly language instruction must provide sufficient information for the actual operand to be located by the microprocessor. The specification by the assembly language instruction of an operand's address is called addressing mode information.

An assembly language instruction is stored in memory as a one- or two-byte opcode followed by up to four operands. Figure 5-1 illustrates the memory storage format of an assembly instruction, previously described in Chapter 2. Recall that each operand shown on Figure 5-1 consists of a descriptor byte, followed by up to four bytes of data, as illustrated on Figure 5-2.

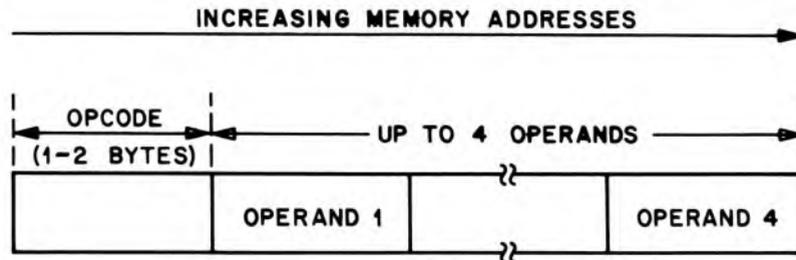


Figure 5-1. Instruction Format

The descriptor byte defines an operand's addressing mode and register field. Bytes that follow the descriptor byte contain any data required by the address mode. Figure 5-3 illustrates the format of the descriptor byte, which consists of two 4-bit fields.

The register field, denoted as rrrr, consists of bits 0 through 3 of the descriptor byte, and contains the number of a register, 0 through 15. The mode field, denoted as mmmm, consists of the four higher-order bits of the descriptor byte, bits 4 through 7. This field contains an addressing mode number, 0 through 15.

INSTRUCTION SET AND ADDRESSING MODES

Format 1 Addressing Modes

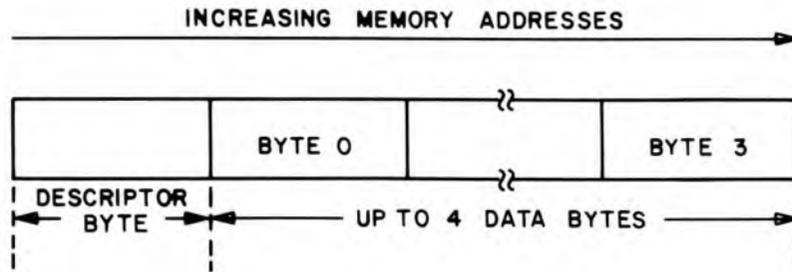


Figure 5-2. Descriptor Byte Format

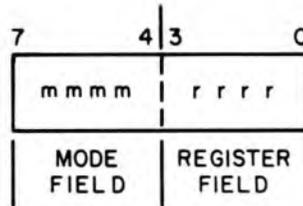


Figure 5-3. Operand Format

5.2.1 Format 1 Addressing Modes

Table 5-2 lists all format 1 addressing modes and their corresponding mode-field values (0—15).

INSTRUCTION SET AND ADDRESSING MODES

Format 1 Addressing Modes

Mode Field Value	Addressing Mode	Description
0—3	Positive literal	The register field bits are concatenated with the two low-order mode field bits to form an unsigned 6-bit immediate data.
4	Register	The operand is contained in one of the 16 registers. If register 15 is specified in the register field, this mode becomes the word immediate mode.
5	Register deferred	The register specified in the register field contains the operand address. If register 15 is specified in the register field, this mode becomes the halfword immediate mode.
6	FP short offset	The FP (register 9) is implicitly referred to by this mode. Register field bits are used as an offset and are added to the FP to form the operand address. This addressing mode is an optimized case of the register deferred mode, produced by the assembler. If register 15 is specified in the register field, this mode becomes the byte immediate mode.
7	AP short offset	The AP (register 10) is implicitly referred to by this mode. Register field bits are used as an offset and are added to the AP to form the operand address. This addressing mode is an optimized case of the register deferred mode, produced by the assembler. If register 15 is specified by the register field, this mode becomes the absolute mode, and the four bytes following the descriptor byte contain the operand address.
8	Word displacement	The four bytes following the descriptor byte are added to the contents of the register specified in the register field. The sum forms the address of the operand.
9	Word displacement deferred	The four bytes following the descriptor byte are added to the contents of the register specified in the register field. The sum forms the address of a pointer, which contains the operand address.
A	Halfword displacement	The two bytes following the descriptor byte are added to the contents of the register specified in the register field to form the operand address.

INSTRUCTION SET AND ADDRESSING MODES
Format 1 Addressing Modes

Table 5-2. Addressing Modes (Continued)		
Mode Field Value	Addressing Mode	Description
B	Halfword displacement deferred	The two bytes following the descriptor byte are added to the contents of the register specified in the register field. The sum forms the address of a pointer, which contains the operand address.
C	Byte displacement	The byte following the descriptor byte is added to the contents of the register specified in the register field to form the operand address.
D	Byte displacement deferred	The byte following the descriptor byte is added to the contents of the register specified in the register field. The sum forms the address of a pointer, which contains the operand address.
E	Expanded operand	This mode is used to modify the data type of an operand. If register 15 is specified in the register field, this mode becomes the absolute deferred mode.
F	Negative literal	The register field bits are concatenated with the mode field bits to form a negative literal in the range -1 to -16 .

Table 5-3 lists the address modes by address type (absolute, displacement, immediate, register, or special mode) and gives the syntax for each. The descriptions and the table use the following notation:

- 0xnnn** Hexadecimal number nnn, where n is a hexadecimal digit 0 to 9 or a to f (or A to F); may also be written 0Xnnn.
- ap** Argument pointer (AP); contains the starting location on the stack of a list of arguments for a function.
- expr** User-supplied expression that yields a byte, halfword, or word.
- fp** Frame pointer (FP); contains the starting location on the stack of local variables for a function.
- imm8** Signed integer in the range -128 to $+127$; i.e., -2^7 to $(+2^7-1)$.
- imm16** Signed integer in the range -32768 to $+32767$; i.e., -2^{15} to $(+2^{15}-1)$.
- imm32** Signed integer in the range -2^{31} to $(+2^{31}-1)$.

INSTRUCTION SET AND ADDRESSING MODES

Format 1 Addressing Modes

- lit* Signed integer in the range -16 to +63.
- opnd* An operand that uses a mode other than the expanded-operand type.
- %rn* References a processor register; use the syntax shown in Table 5-1 for the desired register.
- so* Short offset; an integer in the range 0 to 14.
- type* Data type: byte or sbyte (for signed byte), ubyte (for unsigned byte), half or shalf (for signed halfword), uhalf (for unsigned halfword), word or sword (for signed word), uword (for unsigned word). See Expanded-Operand Type later in this section for more details.

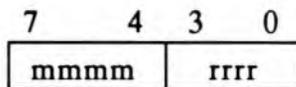
Table 5-3. Addressing Modes by Type – Format 1					
Mode	Syntax	Mode Field	Register Field	Total Bytes	Notes
Absolute					
Absolute	<i>\$expr</i>	7	15	5	—
Absolute deferred	<i>*\$expr</i>	14	15	5	—
Displacement (from a register)					
Byte displacement	<i>expr(%rn)</i>	12	0—10,12—15	2	5
Byte displacement deferred	<i>*expr(%rn)</i>	13	0—10,12—15	2	5
Halfword displacement	<i>expr(%rn)</i>	10	0—10,12—15	3	5
Halfword displacement deferred	<i>*expr(%rn)</i>	11	0—10,12—15	3	5
Word displacement	<i>expr(%rn)</i>	8	0—10,12—15	5	6
Word displacement deferred	<i>*expr(%rn)</i>	9	0—10,12—15	5	6
AP short offset	<i>so(%ap)</i>	7	0—14	1	1
FP short offset	<i>so(%fp)</i>	6	0—14	1	1
Immediate					
Byte immediate	<i>&imm8</i>	6	15	2	2,3
Halfword immediate	<i>&imm16</i>	5	15	3	2,3
Word immediate	<i>&imm32</i>	4	15	5	2,3
Positive literal	<i>&lit</i>	0—3	0—15	1	2,3
Negative literal	<i>&lit</i>	15	0—15	1	2,3
Register					
Register	<i>%rn</i>	4	0—14	1	1,3
Register deferred	<i>(%rn)</i>	5	0—10,12—14	1	1,5
Special Mode					
Expanded operand type	<i>{type}opnd</i>	14	0—14	2—6	4

Notes

- ¹The mode field has a special meaning if register field is 15; see absolute or immediate mode.
- ²This mode may not be used for a destination operand.
- ³This mode may not be used if the instruction takes the effective address of the operand.
- ⁴*type* overrides instruction type; *type* determines the operand type, except that it does not determine the length for immediates or literals or whether literals are signed or unsigned. *opnd* determines the actual address mode. For total bytes, add 1 to byte count for address mode determined by *opnd*.
- ⁵In this mode, r11 refers to format 2 addressing mode.
- ⁶In this mode, r11 causes an invalid descriptor exception.

Examples follow for each of the basic addressing modes and their required descriptor bytes.

As described before, the descriptor byte, which defines the address mode, has two 4-bit fields:



The register field, rrrr, bits 0 through 3, contains the number of a register, 0 through 15. The mode field mmmm, bits 4 through 7, contains an address-mode number, 0 through 15. Table 5-3 lists the value in the mode field and the possible values in the register field for each address mode. If the register field contains 15, the mode field may be interpreted differently. It should be noted that certain branch instructions and all coprocessor words do not require addressing mode information and, therefore, do not use a descriptor byte.

For assembly language programming, values follow the C language conventions:

- Leading 0x or 0X denotes a hexadecimal value
- Leading 0 followed by the digits 0 through 7 is octal
- Digits 0 through 9, but no leading zero is decimal.

The byte boxes illustrating the instruction stream in the following examples contain hexadecimal values.

Register Mode

Any operand directly located in one of the microprocessor's registers is accessed using the register address mode. This mode is indicated in assembly language with the percent symbol (%).

INSTRUCTION SET AND ADDRESSING MODES

Register Mode

For example, the instruction `INCW %r2` causes the 32-bit contents of register `r2` to be incremented by one.

The general syntax, mode, and register fields used to signify the register addressing mode are:

syntax: `%rn` where *n* is a register number

mmm: 4

rrr: 0 to 14

Thus, the instruction `INCW %r2` is stored in memory as illustrated on Figure 5-4.

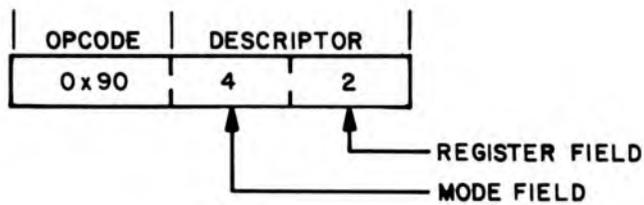


Figure 5-4. Register Mode Example

Register Deferred Mode

Deferred addressing mode involves indirect addressing using pointers. A pointer is either a register or memory location containing an address. Figure 5-5 illustrates the relationship between the address contained in a pointer and the operand ultimately obtained. The term *deferred* is used to describe this procedure because the operand finally obtained is deferred, or delayed, by first going to the pointer for an address. The address contained in the pointer is then used to access the desired operand.

When deferred addressing is used and the pointer is one of the microprocessor's registers, the addressing mode is referred to as a register deferred mode. This addressing mode is designated in assembly language by using parentheses around the pointer register.

For example, the instruction `MOVW (%r2),%r3` causes the CPU to regard the data in register r2 as an address. The contents of the memory location having this address is copied into register r3. Notice that this instruction uses two operands, and each operand has its own addressing mode. Although a register deferred mode was used for the source operand and a register mode was used for the destination operand, any other valid addressing modes could have been used.



Figure 5-5. Deferred Addressing Using a Pointer

INSTRUCTION SET AND ADDRESSING MODES

Displacement Mode

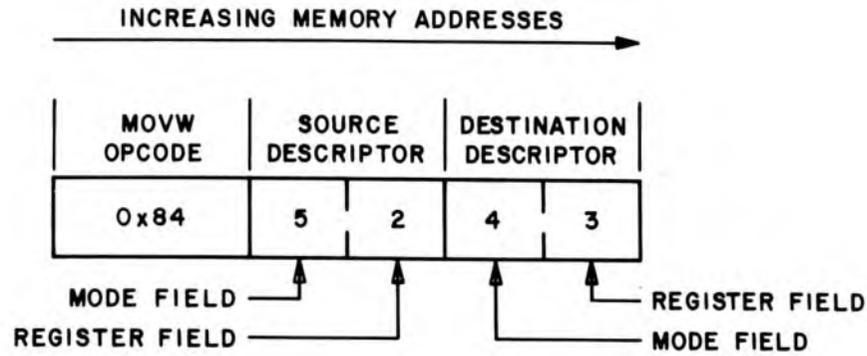


Figure 5-6. Register Deferred Mode Example

The general syntax, mode, and register fields for a register deferred mode operand are:

syntax: (%rn) where *n* is a register number
 mmmm:5
 rrrr: 0 to 10, 12 to 14

Using this information, the instruction MOVW (%r2),%r3 is stored in memory as shown on Figure 5-6.

Displacement Mode

The displacement mode forms an operand's address by adding an offset to the contents of a WE 32200 Microprocessor register. For example, the instruction MOVW 0x30(%r2),%r3 copies the contents of a memory location into register r3. The source operand's memory address is calculated as the contents of register r2 plus an offset of 0x30. Figure 5-7 illustrates the result of this MOVW instruction.

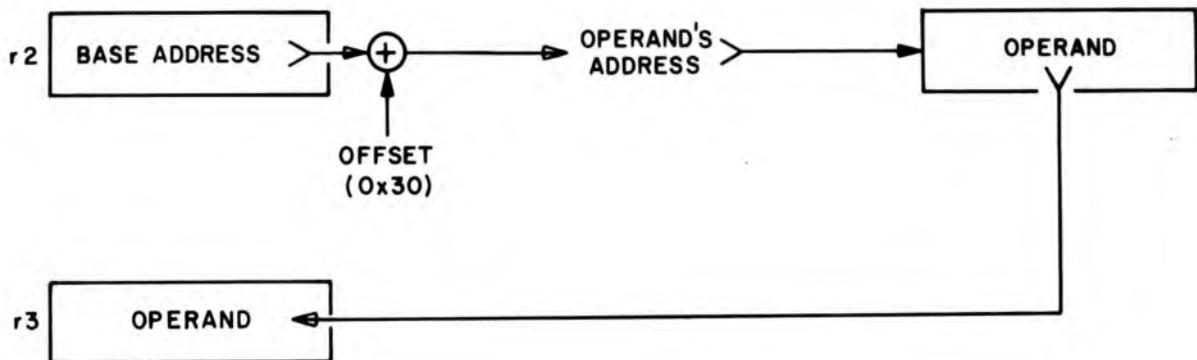


Figure 5-7. Example of MOVW 0x30(%r2),%r3

INSTRUCTION SET AND ADDRESSING MODES

Displacement Mode

The general syntax, and valid register fields for a displacement mode operand are:

syntax: offset(%rn) where *n* is a register number
 mmmm: 8, 10, or 12 (word, halfword, or byte offset)
 rrrr: 0 to 10, 12 to 15

Using the appropriate mode and register fields, the instruction `MOVB 0x30(%r2),%r3` is stored in memory as shown on Figure 5-8.

The offset used in the displacement mode is a two's complement value (positive or negative) that is a byte (8-bits), halfword (16-bits), word (32-bits), or an expression yielding such a value. Negative byte and halfword offsets are sign-extended to 32 bits before being used to obtain the operand's final address. This sign extension converts a negative byte or halfword into its equivalent 32-bit counterpart.

When the displacement mode is used with registers FP and AP, only a short offset between 0 and 14 may be used. This facilitates storage of a shortened instruction format in memory. The mode fields, when the frame and argument registers are used in the displacement mode, are 6 and 7, respectively. The short offset (0—14) is stored in the register field and extra bytes for an offset are not included in the stored instruction.

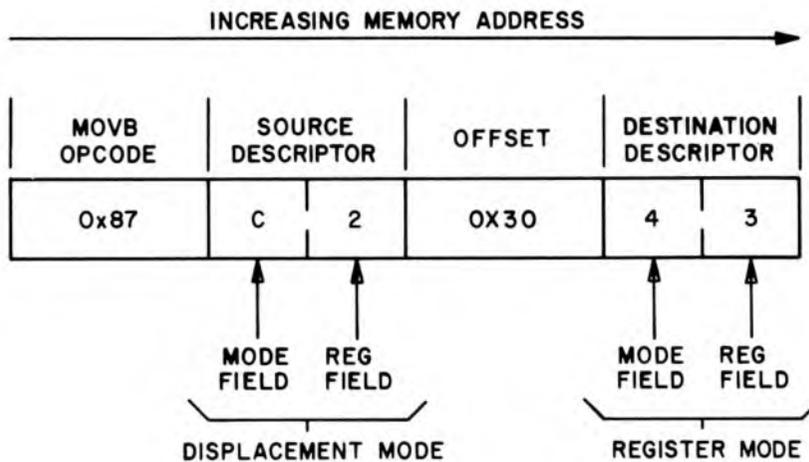


Figure 5-8. A Displacement Mode Source Operand

INSTRUCTION SET AND ADDRESSING MODES

Deferred Displacement Mode

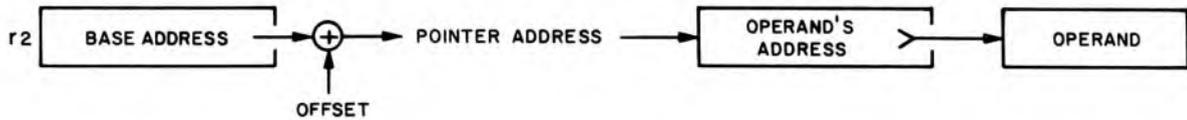


Figure 5-9. Deferred Displacement Addressing

Deferred Displacement Mode

The deferred displacement mode uses the contents of the address calculated in the displacement mode as a pointer that contains the address of the desired operand. Consider the example shown on Figure 5-9. For a typical displacement mode, the operand would be located in the first memory address calculated. In deferred displacement mode, the contents of this location are used as the address of the desired operand.

The deferred displacement mode is indicated to the assembler by the use of an asterisk before the offset.

For example, the instruction `INCW *0x30(%r2)` adds one to the contents of a memory location whose address is contained within a pointer. The address of the pointer is the contents of register `r2` plus `0x30`.

The general syntax, mode field, and register field for a deferred displacement mode operand is:

syntax: **expr*(%*rn*)
mmmm: 9, 11, or 13 (word, halfword, or byte offset)
rrrr: 0—10, or 12—15

Using this information, the instruction `MOVB *0x30(%r2),%r3` is stored in memory as illustrated on Figure 5-10.

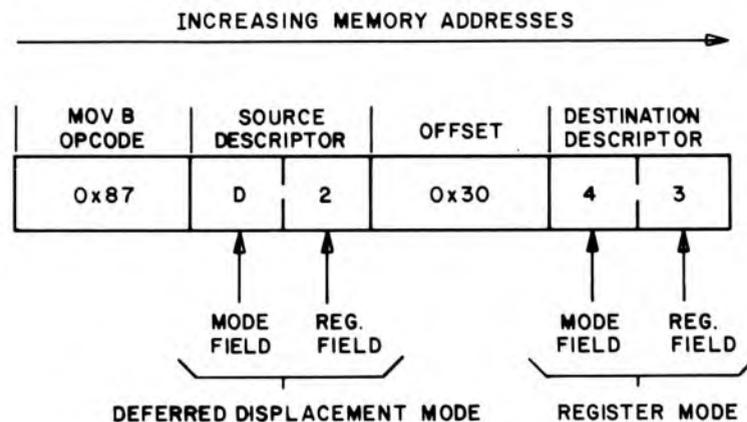


Figure 5-10. A Deferred Displacement Mode Source Operand

Immediate Mode

In the immediate addressing mode, the operand is contained within the instruction. The ampersand (&) is used to indicate this addressing mode to the assembler.

For example, the instruction `MOVB &0x50,%r6` copies the immediate data, 0x50, into register r6. The & signifies that the data immediately following is to be treated as immediate data. The percent (%) indicates that the register mode is being used for the destination operand.

The general syntax, valid mode, and register fields for the immediate addressing mode are:

```

syntax: &data    (data = 8-, 16-, or 32-bits)
mmmm: 4, 5, or 6
rrrr: 15
    
```

A mode field of 4 indicates that the immediate data is 32-bits long, while mode fields of 5 and 6 are used for 16-bit and 8-bit immediate data, respectively. Figure 5-11 illustrates the storage of the instruction `MOVW &0x12345678,%r2` in memory. This instruction causes the immediate data, 0x12345678, to be placed into register r2.

Notice on Figure 5-11 that the immediate data is stored in memory with lower order bytes stored at lower order addresses. This is true for all immediate data; for example, the 16-bit immediate data 0xABCD would be stored as CDAB, with the byte containing CD stored at the immediately lower address than the byte containing AB.

The immediate mode also has a short storage form for positive immediate data between 0 and 63, and negative data between -1 and -16. In these two cases, the immediate data is stored directly within the descriptor byte.

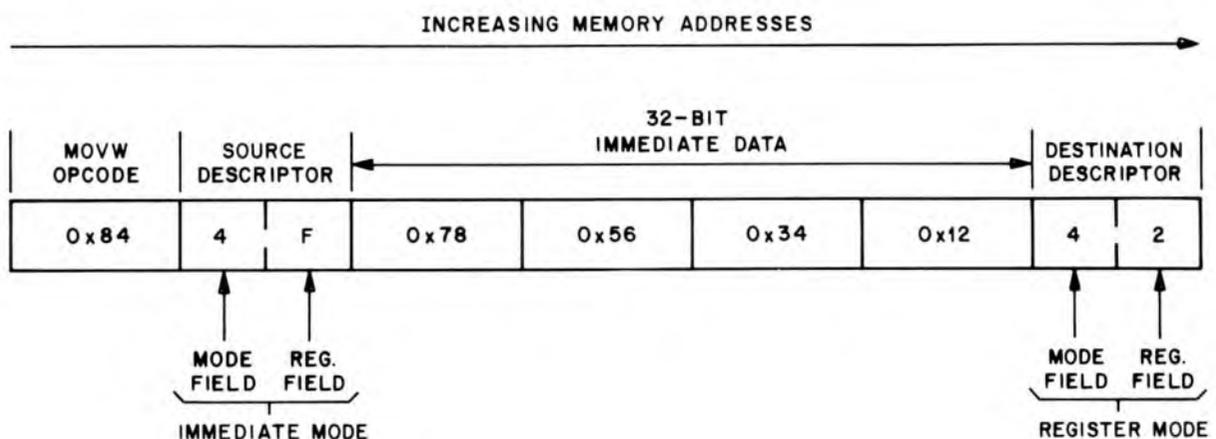


Figure 5-11. A 32-bit Immediate Source Operand

INSTRUCTION SET AND ADDRESSING MODES

Absolute Mode

Absolute Mode

In this mode, the address of the desired operand is contained directly within the instruction. The dollar symbol is used to indicate this addressing mode to the assembler.

For example, the instruction `MOVB $0x2E04,%r0` moves the byte starting at location `0x2E04` into register `r0`. The general syntax, mode, and register fields for the absolute address mode are:

```
syntax: $exp (exp must evaluate to a byte, halfword, or word)
m m m m: 7
r r r r: 15
```

Thus, the instruction `MOVB $0x2E04,%r0` is stored in memory as shown on Figure 5-12.

As illustrated on Figure 5-12, the memory address is stored as a 32-bit address, with lower order bytes stored in lower order memory addresses.

Absolute Deferred Mode

In the absolute deferred mode, the address contained within the instruction is used as a pointer to a word containing the address of the operand. As in all deferred modes, an asterisk is used to indicate deferred addressing to the assembler.

For example, the instruction `MOVB *$0x2E04,%r0` uses the data contained within memory location `0x2E04` as the address of the source operand. The general syntax, mode, and register fields for this deferred mode are:

```
syntax: *$exp (exp must evaluate to a byte, halfword, or word)
m m m m: 14
r r r r: 15
```

Thus, the instruction `MOVB *$0x2E04,%r0` is stored in memory as illustrated on Figure 5-13.

INSTRUCTION SET AND ADDRESSING MODES

Expanded-Operand Mode

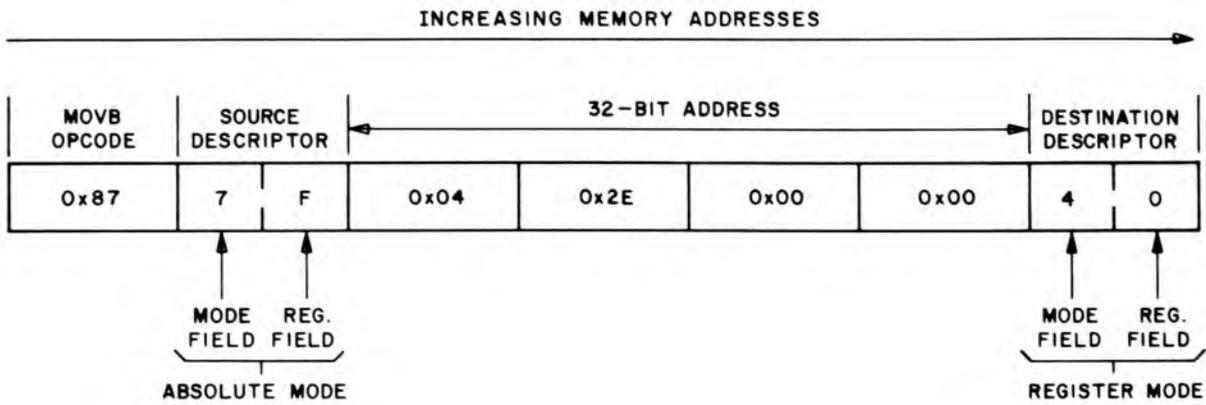


Figure 5-12. An Absolute Mode Source Operand

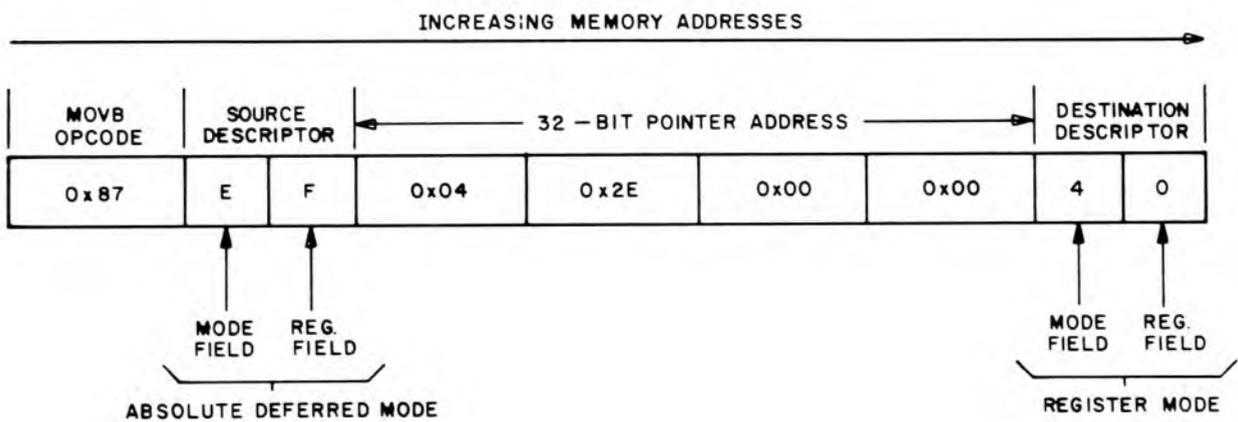


Figure 5-13. An Absolute Deferred Mode Source Operand

INSTRUCTION SET AND ADDRESSING MODES

Expanded-Operand Mode

Expanded-Operand Mode

The expanded-operand mode changes the type of an operand. For example, using this mode, a signed byte located in a register could be converted to an unsigned halfword stored into memory.

The expanded-operand mode does not affect the length of immediate operands, but does affect whether they are treated as signed or unsigned. The expanded-operand mode does not affect the treatment of literals.

In assembly language, the syntax of this mode is

{type}operand

where *operand* is an operand having any address mode except an expanded-operand mode. When the expanded-operand mode is used, *type* overrides the operand's normal data type, except as noted above. The new type remains in effect for the operands that follow in the instruction unless another expanded-operand mode overrides it. Table 5-4 lists the syntax for *type*.

The expanded-operand mode requires two descriptor bytes, as shown on Figure 5-14. The first byte identifies the expanded-operand mode and the new type, while the second is the descriptor byte for the address mode. The type field contains the value of the new type (see Table 5-4). The second byte contains the mode field (mmmm) and the register field (rrrr) for the address mode. This byte is the descriptor byte for the new address mode. For example, the following instruction converts a signed byte into an unsigned halfword:

```
MOVB {sbyte}%r0,{uhalf}4(%r1)
```

0xE	TYPE FIELD	MODE FIELD	REG. FIELD
-----	---------------	---------------	---------------

Figure 5-14. Expanded-Operand Mode Descriptor Bytes

INSTRUCTION SET AND ADDRESSING MODES

Expanded-Operand Mode

The first operand's real mode is register; the second operand's is byte displacement. The instruction reads bits 0 through 7 from register 0, extends the sign bit (7) through 32 bits, and writes an unsigned halfword. The bytes are stored in memory as illustrated on Figure 5-15.

The expanded-operand mode is illegal with coprocessor instructions, CALL, SAVE, RESTORE, SWAP INTERLOCKED, PUSHW, PUSHAW, POPW, JSB, PACKB, UNPACKB, BCD arithmetic instructions, and loop control instructions.

Table 5-4. Options for <i>type</i> in Expanded-Operand Mode		
Type	Syntax	Type Field*
Signed byte	byte or sbyte	E7
Signed halfword	half or shalf	E6
Signed word	word or sword	E4
Unsigned byte	ubyte	E3
Unsigned halfword	uhalf	E2
Unsigned word	uword	E0

* Type fields E1, E5, E8—E14 are reserved data types. Type field EF is an absolute deferred data type.

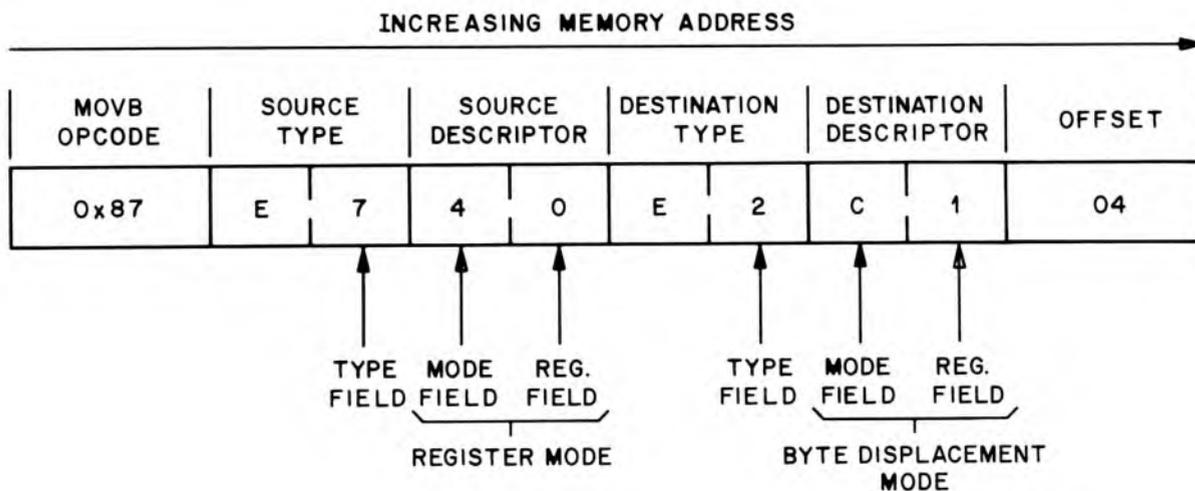


Figure 5-15. Expanded-Operand Mode Example

INSTRUCTION SET AND ADDRESSING MODES

Auto Pre/Post Increment/Decrement

5.2.2 Format 2 Addressing Modes

The format 2 addressing modes are unique to the *WE 32200* Microprocessor and are not supported by the *WE 32100* Microprocessor. This format is specified by the first byte of the descriptor, which must be 0x5B, 0xAB, 0xBB, 0xCB, or 0xDB.

Auto Pre/Post Increment/Decrement

The auto pre/post increment/decrement modes are convenient when performing arithmetic or sort operations on contiguously addressed blocks of memory. When using the auto pre increment/decrement modes, the specified register is first incremented or decremented; then, the value of that register is used as a pointer to the address of the operand. With post increment/decrement modes, the value of the specified register is used as a pointer to the address of the operand, and then is incremented or decremented. The second byte following the operand mode descriptor fetched from the instruction stream specifies whether the mode is auto increment or auto decrement and the corresponding register. The register (r0—r31) is represented by bits 0 through 4. Bits 5 through 7 represent the auto pre/post increment/decrement modes. The modes are shown in Table 5-5 and are described below.

Auto Pre-Decrement. The register specified is first decremented by the size of the operand: 1 for byte, 2 for halfword, or 4 for word. The decremented register contains the address of the operand.

Auto Post-Decrement. The register specified is used as a pointer to the address of the operand. The register is then decremented by the size of the operand: 1 for byte, 2 for halfword, or 4 for word.

Auto Pre-Increment. The specified register is first incremented by the size of the operand: 1 for byte, 2 for halfword, or 4 for word. The incremented register contains the address of the operand.

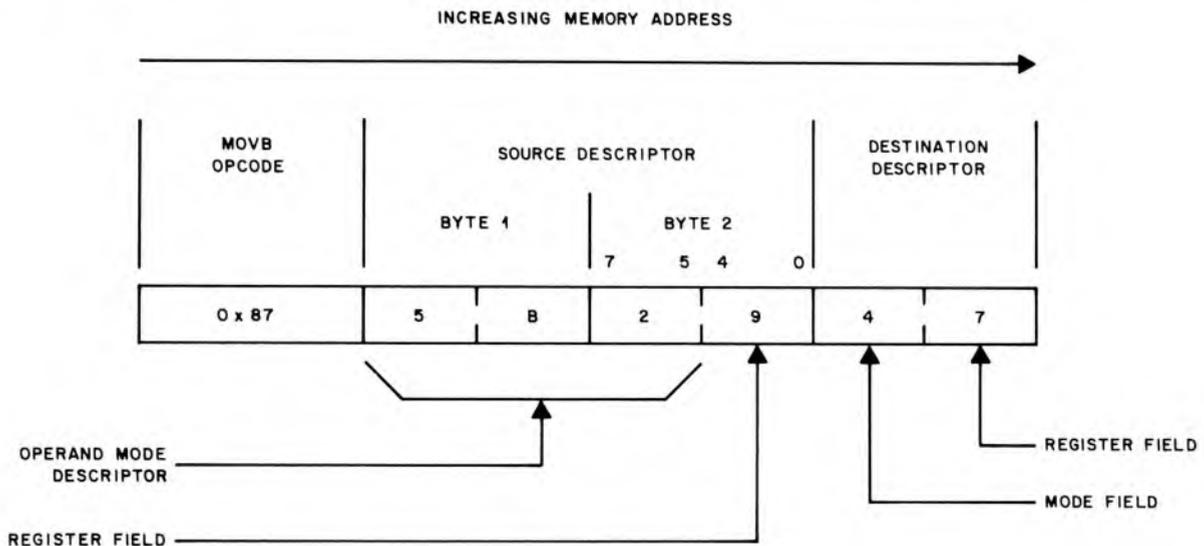
Auto Post-Increment. The specified register is used as a pointer to the address of the operand. The register is then incremented by the size of the operand: 1 for byte, 2 for halfword, or 4 for word.

Operand Mode Descriptor		Name	Syntax	dest?	EA?	PC?
1st Byte	Bits 5—7 2nd Byte					
0x5B	0	auto pre-decrement	-(%rx)	yes	yes	ill
0x5B	2	auto post-decrement	(%rx)-	yes	yes	ill
0x5B	4	auto pre-increment	+(%rx)	yes	yes	ill
0x5B	6	auto post-increment	(%rx)+	yes	yes	ill

Legend:

- EA – effective address access valid
- PC – PC (r15) in register field
- dest – write access (destination) valid
- ill – illegal
- x – register number in range 0 to 31

Figure 5-16 shows an example of auto post-decrement addressing. The first byte is the opcode for the MOV_B instruction (0x87). The next two bytes make up the source descriptor. Byte one and bits 5 through 7 of byte two of the source descriptor specify the auto post-decrement mode. Bits 0 through 4 of byte two define the source register, r9. The last byte is the destination descriptor. The upper nibble specifies the format 1 addressing mode. In this case, it is mode 4, register mode. The lower nibble specifies the destination register, r7.



MOV_B (%r9)–, %r7

Figure 5-16. Auto Post-Decrement Mode Example

The register is incremented or decremented only once. If bits 5 through 7 of the byte following the operand descriptor is equal to 1, 3, 5, or 7, an invalid descriptor exception occurs. This exception also occurs if auto mode is used with PSW, PC, coprocessor instructions, or the CASWI instruction.

Indexed Register Modes

Indexed register addressing modes are useful for the efficient manipulation of arrays.

The indexed register modes with byte or halfword displacements add the sum of the two specified registers to the byte or halfword offset. This total is then used as the pointer to the address of the operand.

INSTRUCTION SET AND ADDRESSING MODES

Indexed Register Modes

Indexed register mode with scaling multiplies the contents of the specified register (r0 through r15) by 1, 2, or 4, depending on the operand size (byte, halfword, or word, respectively). Then, the contents of the specified register (r16 through r31) are added to the above product, and are used as a pointer to the address of the operand.

The indexed register modes are summarized in Table 5-6, and their operations are discussed below.

Operand Mode Descriptor	Name	Syntax	dest?	EA?	PC?
0xAB	Indexed with byte displacement	expr(%rn, %rm)	yes	yes	yes
0xBB	Indexed with halfword displacement	expr(%rn, %rm)	yes	yes	yes
0xDB	Indexed with scaling	%rm[%rn]	yes	yes	ill

Legend:

EA – effective address access valid

PC – PC (r15) in register field

dest – write access (destination) valid

expr – user supplied expression that yields a byte, halfword, or word

n – register number in range 0 to 15

m – register number in range 16 to 31

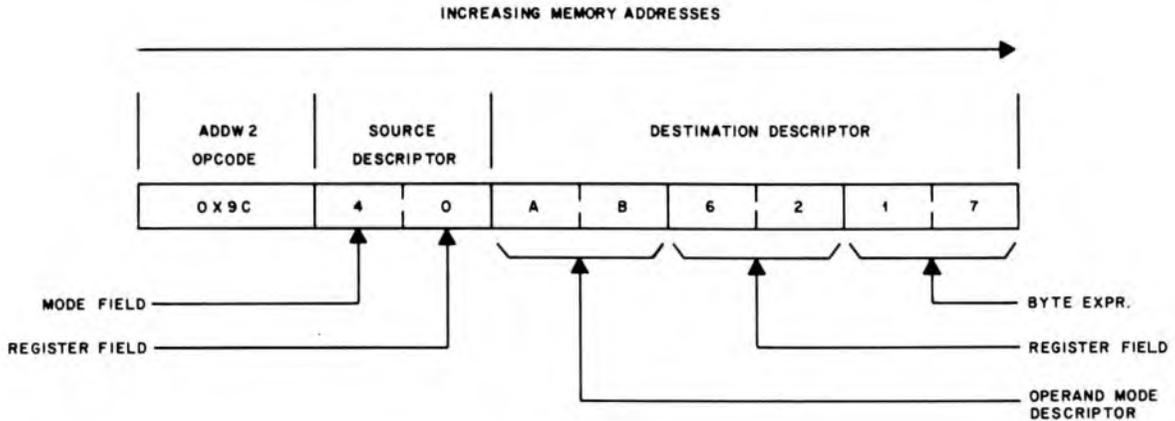
ill – illegal

Indexed Register with Byte Displacement. The CPU, in this mode, fetches the two bytes following the operand descriptor byte from the instruction stream to identify the operand. The least significant nibble of the first byte specifies register %r16 through %r31. The most significant nibble of the same byte specifies register %r0 through %r15. Assuming the least significant nibble to be y, the register specified is register y+16. The second byte is sign-extended to 32 bits and added to the sum of the contents of the two registers. The total is the address of the operand to be used in the instruction. Use of the PSW or PC causes an invalid descriptor exception.

Figure 5-17 shows an example of indexed register with byte displacement addressing. The first byte is the ADDW2 opcode (0x9C). The second byte, the source descriptor, specifies register mode, using r0 as the source. The next three bytes make up the destination descriptor. The first byte of the destination descriptor specifies indexed register with byte displacement addressing. The most significant nibble of the second byte specifies r6. The least significant nibble specifies r18 (Y+16, where Y = 2). The last byte is the expression that is sign-extended to thirty-two bits. This expression is added to the sum of the contents of r6 and r18. The total is the address of the operand in the instruction.

INSTRUCTION SET AND ADDRESSING MOD

Indexed Register with Halfword Displacement



ADDW2 %r0, 0x17(%r18, %r2)

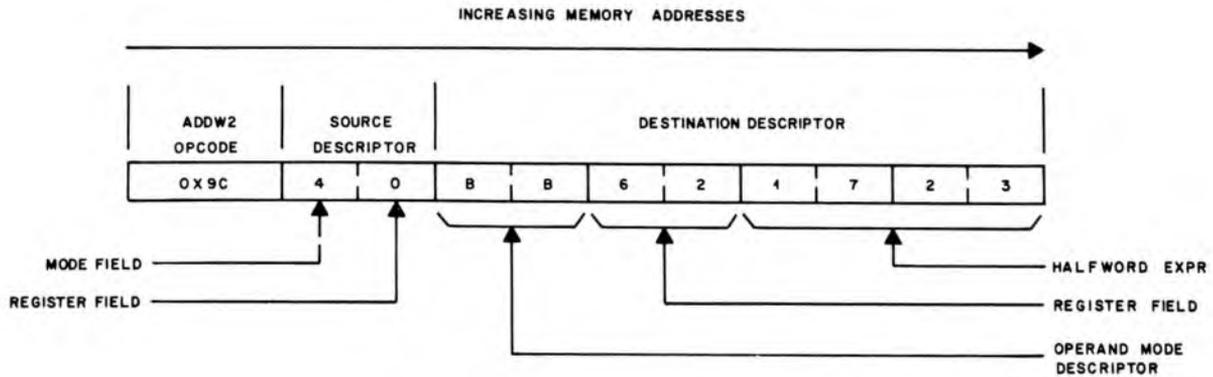
Figure 5-17. Indexed Register (Byte Displacement) Mode Example

Indexed Register with Halfword Displacement. In this mode, the CPU fetches the three bytes following the operand mode descriptor byte from the instruction stream to identify the operand. The most significant nibble of the first byte specifies register %r0 through %r15. The least significant nibble of the same byte specifies register %r16 through %r31. Assuming the least significant nibble is y, the register specified is register y+16. The remaining halfword (second and third bytes) is sign-extended to 32 bits and added to the sum of the contents of the two registers. The total is the address of the operand to be used in the instruction. Use of the PSW or PC causes an invalid descriptor exception.

Figure 5-18 shows an example of indexed register with halfword displacement addressing mode. The first byte is the ADDW2 opcode (0x9C). The second byte, the source descriptor, specifies register mode, using r0 as the source. The next four bytes make up the destination descriptor. The first byte of the destination descriptor specifies indexed register with halfword displacement addressing. The most significant nibble of the second byte specifies r6. The least significant nibble specifies r18 (Y+16, where Y = 2). The last two bytes are the halfword expression that is sign-extended to 32 bits. This expression is added to the sum of the contents of r6 and r18. The total is the address of the operand in the instruction.

INSTRUCTION SET AND ADDRESSING MODES

Indexed Register with Scaling



ADDW2 %r0, 0x1723(%r18, %r6)

Figure 5-18. Indexed Register (Halfword Displacement) Mode Example

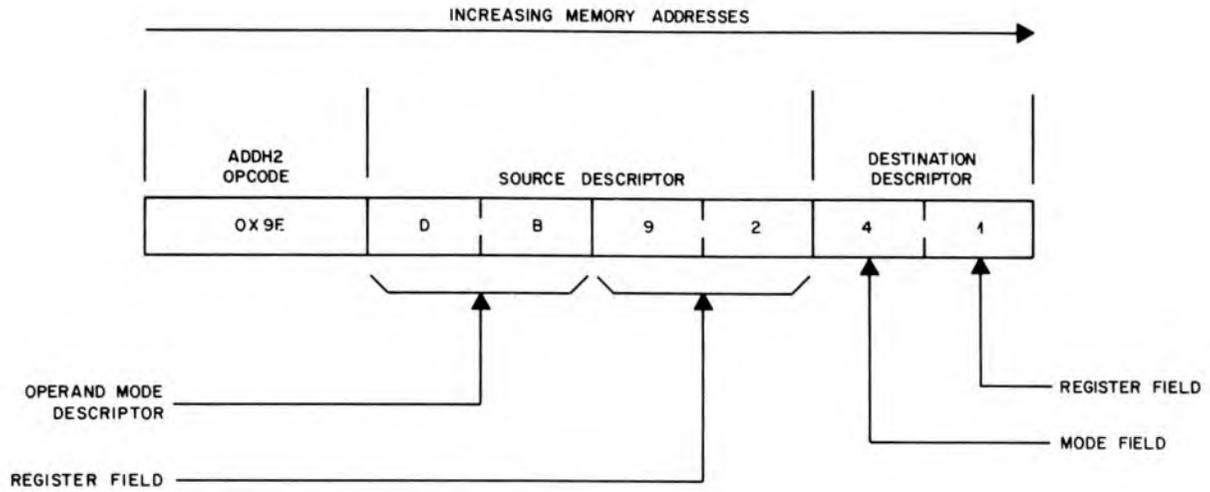
Indexed Register with Scaling. The CPU, in this mode, fetches the byte following the operand mode descriptor byte from the instruction stream, to identify the operand. The most significant nibble of the byte specifies the index register (%r n , where $n = 0$ through 15). The least significant nibble of the byte specifies the base register %r16 through %r31. Assuming the least significant nibble is y , the register specified is register $y+16$. Depending on the operand size (byte, halfword, or word), the contents of the register (%r0 through %r15) specified by the most significant nibble are first multiplied by 1, 2, or 4 respectively. This product is then added to the contents of the register (%r16 through %r31) specified by the least significant nibble of the byte. Finally, the sum is the address of the operand to be used in the instructions. Use of the PSW or PC causes an invalid descriptor exception, as does the use of this mode with coprocessor instructions.

Figure 5-19 shows an example of indexed register with scaling addressing mode. The first byte is the ADDH2 opcode (0x9E). The next byte, the first byte of the source descriptor, specifies indexed register with scaling addressing. The following byte describes the source operand index and base registers. Bits 4 through 7 represent r9 (index register), while bits 0 through 3 represent r18 (base register). The destination descriptor specifies r1, using register addressing mode (mode 4).

In this example, the value in r9 is multiplied by 2 (halfword operation) and is added to the value in r18. This sum is used as the address of the source operand for this instruction. The value stored at that address is added to the contents of r1, and the sum is stored in r1.

INSTRUCTION SET AND ADDRESSING MODES

Format 1 Addressing Modes for Registers 16—31



ADDH2 %r18[%r9], %r1

Figure 5-19. Example of Indexed Register Mode with Scaling

Format 1 Addressing Modes for Registers 16—31

Table 5-7 summarizes the format 1 addressing modes for registers 16—31. They are essentially the same as those discussed under section 5.2.2 Format 1 Addressing Modes, but are applicable only to registers 16 through 31.

INSTRUCTION SET AND ADDRESSING MODES

Indexed Register with Byte Displacement

Operand Mode Descriptor		Name	Syntax	dest?	EA?	PC?
1st Byte	Bits 4—7 2nd Byte					
0xCB	4	Register mode	%rm	yes	no	NA
0xCB	5	Register deferred	(%rm)	yes	yes	NA
0xCB	8	Word displacement	expr(%rm)	yes	yes	NA
0xCB	9	Word displacement deferred	*expr(%rm)	yes	yes	NA
0xCB	10	Halfword displacement	expr(%rm)	yes	yes	NA
0xCB	11	Halfword displacement deferred	*expr(%rm)	yes	yes	NA
0xCB	12	Byte displacement	expr(%rm)	yes	yes	NA
0xCB	13	Byte displacement deferred	*expr(%rm)	yes	yes	NA

Legend:

EA – effective address access valid

PC – PC (r15) in register field

dest – write access (destination) valid

expr – user supplied expression that yields a byte, halfword, or word

m – register number in range 16 to 31

N.A. – not applicable

The second byte following the operand mode descriptor is fetched from the instruction stream and is interpreted as the first byte of a format 1 operand descriptor. In other words, the byte is decoded the same as format 1. This byte cannot contain the first byte of a format two operand. Bits 0 through 3 of the byte represent a register, 16 through 31. Assuming bits 0 through 3 equal y , the register number is $y+16$. Bits 4 through 7 are to be decoded as shown in section 5.2.1; however, only some of the addressing modes in format 1 are legal. These modes are:

- Register mode
- Register deferred mode
- Word displacement mode
- Halfword displacement mode
- Byte displacement mode
- Word displacement deferred mode
- Halfword displacement deferred mode
- Byte displacement deferred mode.

The expanded operand and literal or short-offset modes cause an invalid descriptor exception. Operand descriptors for format 2 expand modes should start with expanded mode (format 1) followed by any format 2 addressing modes.

INSTRUCTION SET AND ADDRESSING MODES

Instruction Set Functional Groups

5.3 INSTRUCTION SET FUNCTIONAL GROUPS

The WE 32200 Microprocessor instruction set may be separated into seven functional groups: data transfer, arithmetic, logical, program control, coprocessor, operating system (see Chapter 6), and stack and miscellaneous instructions. This sections contains a description and listing of each group. The conditions column in the instruction listing refers to the condition flag code assignment cases listed in Table 5-8.

Table 5-8. Condition Flag Code Assignments					
Case	Condition Flags				Special Conditions*
	N(Negative)	Z(Zero)	C(Carry)	V(Overflow)	
1	MSB of <i>dst</i>	1 if <i>dst</i> = 0	0	0	V flag is set when expanded operand mode is used, and the result is truncated when represented in destination.
2	1 if result < 0	1 if result = 0	1 on carry or borrow	1 on integer overflow	—
3	1 if <i>dst</i> < 0	1 if <i>dst</i> = 0	0	1 on integer overflow	—
4	1 if <i>dst</i> < 0	1 if <i>dst</i> = 0	0	1 on integer overflow	V flag may not set when <i>dst</i> is signed word, bit 31 of absolute value of the result is 1, and while bits 32—63 of the absolute value of the result are 0s.
5	1 if <i>dst</i> < 0	1 if <i>dst</i> = 0	0	0	V flag is set if expanded-operand mode changes the type of <i>dst</i> and integer overflow occurs.
6	1 if <i>src</i> < 0	1 if <i>src</i> = 0	0	0	N flag is affected if <i>src</i> is signed integer.

INSTRUCTION SET AND ADDRESSING MODES

Data Transfer Instructions

Case	Condition Flags				Special Conditions*
	N(Negative)	Z(Zero)	C(Carry)	V(Overflow)	
7	MSB of word returned	1 if word returned = 0	0	0	—
8	—	—	—	—	All flags determined by new PSW.
9	—	—	—	—	All flags determined by restored PSW.
10	—	—	—	—	When coprocessor status word is accepted, bits 18—21 of the word read are put into bits 18—21 of the PSW, respectively.

Legend:

MSB — Most Significant Bit

dst — destination

src — source

- * For cases 1 through 6, when the PSW is used as a source, the condition flags are unaffected; when the PSW is used as a destination, the condition flags assume the value of bits 18—21 of the result of the operation performed.

5.3.1 Data Transfer Instructions

The instructions listed in Table 5-9 transfer data between registers and memory. Most of them have three types, indicated by the last character of the mnemonic: byte (B), halfword (H), and word (W). A mnemonic's type determines the type of each operand in the instruction, unless the expanded-operand mode changes an operand's type. The destination operand (*dst*) result determines how the condition flags are set.

INSTRUCTION SET AND ADDRESSING MODES

Arithmetic Instructions

Table 5-9. Data Transfer Instruction Group			
Instruction	Mnemonic	Opcode	Conditions*
Move: Move byte Move halfword Move word	MOVB MOVH MOVW	0x87 0x86 0x84	Case 1
Move address (word)	MOVAW	0x04	
Move complemented byte Move complemented halfword Move complemented word	MCOMB MCOMH MCOMW	0x8B 0x8A 0x88	
Move negated byte Move negated halfword Move negated word	MNEGB MNEGH MNEGW	0x8F 0x8E 0x8C	Case 2
Move version number	MVERNO	0x3009	Unchanged
Swap (Interlocked): Swap byte interlocked Swap halfword interlocked Swap word interlocked	SWAPBI SWAPHI SWAPWI	0x1F 0x1E 0x1C	Case 1
Compare and Swap: Compare and swap word interlocked	CASWI	0x09	Case 2
Block Operations: Move block of words	MOVBLW	0x3019	Unchanged
Field Operations: Extract field byte Extract field halfword Extract field word	EXTFB EXTFH EXTFW	0xCF 0xCE 0xCC	Case 1
Insert field byte Insert field halfword Insert field word	INSFB INSFH INSFW	0xCB 0xCA 0xC8	
String Operations: String copy String end	STRCPY STREND	0x3035 0x301F	

* Refer to Table 5-8 for condition flag code assignments.

5.3.2 Arithmetic Instructions

The arithmetic instructions listed in Table 5-10 perform arithmetic operations on data in registers and memory. Most of these instructions have three options, specified by the last alphabetic character of the mnemonic: byte (B), halfword (H), and word (W). This type specification applies to each operand in the instruction, unless the

INSTRUCTION SET AND ADDRESSING MODES

Arithmetic Instructions

expanded-operand mode changes an operand's type. The destination operand (*dst*) result determines how the condition flags are set.

Many arithmetic operations are available as two- or three-address instructions. A two-address instruction has a source operand (*src*) and a destination operand. Three-address instructions have two source operands (*src1*, *src2*) and a destination operand. A few instructions also have a count operand (*count*).

If the result of an arithmetic operation is too large to be represented in 32 bits, the high-order bits are truncated and the processor issues an integer-overflow exception.

Table 5-10. Arithmetic Instruction Group			
Instruction	Mnemonic	Opcode	Conditions*
Add:			Case 2
Add byte	ADDB2	0x9F	
Add halfword	ADDH2	0x9E	
Add word	ADDW2	0x9C	
Add byte, 3-address	ADDB3	0xDF	
Add halfword, 3-address	ADDH3	0xDE	
Add word, 3-address	ADDW3	0xDC	
Subtract:			
Subtract byte	SUBB2	0xBF	
Subtract halfword	SUBH2	0xBE	
Subtract word	SUBW2	0xBC	
Subtract byte, 3-address	SUBB3	0xFF	
Subtract halfword, 3-address	SUBH3	0xFE	
Subtract word, 3-address	SUBW3	0xFC	
Increment:			
Increment byte	INCB	0x93	
Increment halfword	INCH	0x92	
Increment word	INCW	0x90	
Decrement:			
Decrement byte	DECB	0x97	
Decrement halfword	DECH	0x96	
Decrement word	DECW	0x94	
Multiply:			Case 3
Multiply byte	MULB2	0xAB	
Multiply halfword	MULH2	0xAA	
Multiply word	MULW2	0xA8	Case 4
Multiply byte, 3-address	MULB3	0xEB	
Multiply halfword, 3-address	MULH3	0xEA	
Multiply word, 3-address	MULW3	0xE8	

* Refer to Table 5-8 for condition flag code assignments.

INSTRUCTION SET AND ADDRESSING MODES
Arithmetic Instructions

Table 5-10. Arithmetic Instruction Group (Continued)			
Instruction	Mnemonic	Opcode	Conditions*
Divide: Divide byte Divide halfword Divide word	DIVB2 DIVH2 DIVW2	0xAF 0xAE 0xAC	Case 3
Divide byte, 3-address Divide halfword, 3-address Divide word, 3-address	DIVB3 DIVH3 DIVW3	0xEF 0xEE 0xEC	Case 4
Modulo: Modulo byte Modulo halfword Modulo word	MODB2 MODH2 MODW2	0xA7 0xA6 0xA4	Case 3
Modulo byte, 3-address Modulo halfword, 3-address Modulo word, 3-address	MODB3 MODH3 MODW3	0xE7 0xE6 0xE4	Case 4
Arithmetic Shift: Arithmetic left shift word	ALSW3	0xC0	Case 5
Arithmetic right shift byte Arithmetic right shift halfword Arithmetic right shift word	ARSB3 ARSH3 ARSW3	0xC7 0xC6 0xC4	Case 3

* Refer to Table 5-8 for condition flag code assignments.

5.3.3 BCD Instructions

There are three types of BCD instructions: additive BCD arithmetic instructions, BCD data conversion instructions, and flag setting and clearing instructions. These instructions perform BCD arithmetic, data conversion, and logical operations, respectively, on data in registers and memory. All BCD arithmetic instructions read the X flag as an input. The BCD instructions are listed in Table 5-11.

INSTRUCTION SET AND ADDRESSING MODES

Registers

Table 5-11. BCD Instruction Group			
Instruction	Mnemonic	Opcode	Conditions*
Add: Add packed byte Add packed byte, 3-address	ADDPB2 ADDPB3	0xA3 0xE3	Case 7
Subtract: Subtract packed byte Subtract packed byte, 3-address	SUBPB2 SUBPB3	0x9B 0xDB	
Data Conversion: Pack BCD halfword Unpack BCD byte	PACKB UNPACKB	0x0E 0x0F	Unchanged
Flags: Clear X bit in PSW Set X bit in PSW	CLR X SET X	0x0B 0x0A	Only X bit in PSW affected

* Refer to Table 5-8 for condition flag code assignments.

5.3.4 Logical Instructions

The logical instructions listed in Table 5-12 perform logical operations on data in registers in memory. Most of these instructions have three options, specified by the last character of the mnemonic: byte (B), halfword (H), and word (W). A mnemonic's type determines the type of each operand in the instruction, unless the expanded-operand mode changes an operand's type. The destination operand (*dst*) result determines how the condition flags are set.

Many logical operations are available as two- or three-address instructions. A two-address instruction has a source operand (*src*) and a destination operand (*dst*). A few instructions have a read-only count operand (*count*).

INSTRUCTION SET AND ADDRESSING MODES
Registers

Table 5-12. Logical Instruction Group				
Instruction	Mnemonic	Opcode	Conditions*	
AND: AND byte AND halfword AND word	ANDB2 ANDH2 ANDW2	0xBB 0xBA 0xB8	Case 1	
AND byte, 3-address AND halfword, 3-address AND word, 3-address	ANDB3 ANDH3 ANDW3	0xFB 0xFA 0xF8		
Exclusive OR (XOR): Exclusive OR byte Exclusive OR halfword Exclusive OR word	XORB2 XORH2 XORW2	0xB7 0xB6 0xB4		
Exclusive OR byte, 3-address Exclusive OR halfword, 3-address Exclusive OR word, 3-address	XORB3 XORH3 XORW3	0xF7 0xF6 0xF4		
OR: OR byte OR halfword OR word	ORB2 ORH2 ORW2	0xB3 0xB2 0xB0		
OR byte, 3-address OR halfword, 3-address OR word, 3-address	ORB3 ORH3 ORW3	0xF3 0xF2 0xF0		
Compare or Test: Compare byte Compare halfword Compare word	CMPB CMPH CMPW	0x3F 0x3E 0x3C		Case 2
Test byte Test halfword Test word	TSTB TSTH TSTW	0x2B 0x2A 0x28		Case 6
Bit test byte Bit test halfword Bit test word	BITB BITH BITW	0x3B 0x3A 0x38		Case 1
Clear: Clear byte Clear halfword Clear word	CLRB CLRH CLRW	0x83 0x82 0x80		Case 2
Rotate or Logical Shift: Rotate word	ROTW	0xD8		Case 2

* Refer to Table 5-8 for condition flag code assignments.

INSTRUCTION SET AND ADDRESSING MODES

Registers

Instruction	Mnemonic	Opcode	Conditions*
Logical left shift byte	LLSB3	0xD3	Case 1
Logical left shift halfword	LLSH3	0xD2	
Logical left shift word	LLSW3	0xD0	
Logical right shift word	LRSW3	0xD4	

* Refer to Table 5-8 for condition flag code assignments.

5.3.5 Program Control Instructions

The program control instructions listed in Table 5-13 change the program sequence, but generally do not alter the condition flags.

Conditional/Unconditional Transfer. Branch instructions have two types, specified by the last character of the mnemonic: byte displacement (B) and halfword displacement (H). A mnemonic's type determines if an 8- or 16-bit displacement is embedded in the instruction. This displacement (*disp8*, *disp16*) is read, its sign extended through 32 bits, and the result added to the program counter (PC) to compute the target address. Jump instructions have a read-only, 32-bit destination (*dst*) operand that replaces the contents of the PC.

Jump instructions are always unconditional, but both conditional and unconditional branch and return instructions are provided. Unconditional transfers change the contents of the PC to the value specified. Conditional transfers first examine the status of the processor's condition flags to determine if the the transfer should be executed.

Loop Control. Loop control instructions are similar to conditional transfers, but with the added feature that the contents of an operand are also examined to determine if a specified transfer should be executed. Subroutine and procedure-call (function) transfer instructions save or restore registers so execution can transfer to the subroutine or function and then return to the original program sequence.

Subroutine Transfer. A subroutine is different from a normal transfer. Before transferring to a subroutine, the address of the next instruction is saved.

Call and return instructions for subroutines always implicitly affect the stack pointer (SP). For subroutines, a call saves the address of the next instruction on the stack at the location identified by the SP, increments the SP by 4, and then alters the PC. On return from a subroutine, the SP is decremented by 4, and the saved address is retrieved from the stack, and stored in the PC.

Procedure Transfer. For procedure transfers it is necessary to save other registers. These instructions establish the environment for a function in a high-level language. Call and save instructions automatically save the calling function's pointers, set up pointers to the new function's environment, call the function, and save registers for local variables. Restore and return instructions remove that environment and return to the calling function.

INSTRUCTION SET AND ADDRESSING MODES

Program Control Instructions

A stack frame provides reserved space, including a register-save area, for each function. The register-save area stores the calling function's FP, AP, return PC, and registers 3 through 8 (r3 — r8), if requested. Saving r3 through r8 gives the new function space for up to six register variables. The SP is not saved because its value is always implicit.

All function calls have a fixed-size register-save area, even though some of it may not be used. Save and restore control the number of user registers r3 through r8 that will be saved and restored. A return from a function retrieves the saved pointers and registers to restore the original function's environment.

Table 5-13. Program Control Instruction Group			
Instruction	Mnemonic	Opcode	Conditions*
Unconditional Transfer:			Unchanged
Branch with byte displacement	BRB	0x7B	
Branch with halfword displacement	BRH	0x7A	
Jump	JMP	0x24	
Conditional Transfers:			
Branch on carry clear byte	BCCB	0x53**	
Branch on carry clear halfword	BCCH	0x52**	
Branch on carry set byte	BCSB	0x5B**	
Branch on carry set halfword	BCSH	0x5A**	
Branch on overflow clear, byte displacement	BVCB	0x63	
Branch on overflow clear, halfword displacement	BVCH	0x62	
Branch on overflow set, byte displacement	BVSB	0x6B	
Branch on overflow set, halfword displacement	BVSH	0x6A	
Branch on equal byte (duplicate)	BEB	0x6F	
Branch on equal byte	BEB	0x7F	
Branch on equal halfword (duplicate)	BEH	0x6E	
Branch on equal halfword	BEH	0x7E	

* Refer to Table 5-8 for condition flag code assignments.

** Indicates that another mnemonic has the same opcode.

INSTRUCTION SET AND ADDRESSING MODES
Program Control Instructions

Table 5-13. Program Control Instruction Group (Continued)			
Instruction	Mnemonic	Opcode	Conditions*
Branch on not equal byte (duplicate)	BNEB	0x67	Unchanged
Branch on not equal byte	BNEB	0x77	
Branch on not equal halfword (duplicate)	BNEH	0x66	
Branch on not equal halfword	BNEH	0x76	
Branch on less than byte (signed)	BLB	0x4B	
Branch on less than halfword (signed)	BLH	0x4A	
Branch on less than byte (unsigned)	BLUB	0x5B**	
Branch on less than halfword (unsigned)	BLUH	0x5A**	
Branch on less than or equal byte (signed)	BLEB	0x4F	
Branch on less than or equal halfword (signed)	BLEH	0x4E	
Branch on less than or equal byte (unsigned)	BLEUB	0x5F	
Branch on less than or equal halfword (unsigned)	BLEUH	0x5E	
Branch on greater than byte (signed)	BGB	0x47	
Branch on greater than halfword (signed)	BGH	0x46	
Branch on greater than byte (unsigned)	BGUB	0x57	
Branch on greater than (unsigned)	BGUH	0x56	
Branch on greater than or equal byte (signed)	BGEB	0x43	
Branch on greater than or equal halfword (signed)	BGEH	0x42	
Branch on greater than or equal byte (unsigned)	BGEUB	0x53**	
Branch on greater than or equal halfword (unsigned)	BGEUH	0x52**	
Return on carry clear	RCC	0x50**	
Return on carry set	RCS	0x58**	

* Refer to Table 5-8 for condition code flag assignments.

** Indicates that another mnemonic has the same opcode.

INSTRUCTION SET AND ADDRESSING MODES
Program Control Instructions

Table 5-13. Program Control Instruction Group (Continued)			
Instruction	Mnemonic	Opcode	Conditions*
Return on overflow clear	RVC	0x60	Unchanged
Return on overflow set	RCS	0x68	
Return on equal (unsigned)	REQLU	0x6C†	
Return on equal	REQL	0x7C†	
Return on not equal (unsigned)	RNEQU	0x64††	
Return on not equal	RNEQ	0x74††	
Return on less than (signed)	RLSS	0x48	
Return on less than (unsigned)	RLSSU	0x58**	
Return on less than or equal (signed)	RLEQ	0x4C	
Return on less than or equal (unsigned)	RLEQU	0x5C	
Return on greater than (signed)	RGTR	0x44	Unchanged
Return on greater than (unsigned)	RGTRU	0x54	
Return on greater than or equal (signed)	RGEQ	0x40	
Return on greater than or equal (unsigned)	RGEQU	0x50**	
Loop Control:			Unchanged
Decrement, test, and branch byte (signed)	DTB	0x29	
Decrement, test, and branch halfword (signed)	DTH	0x19	
Test equal, decrement, and test byte	TEDTB	0x4D	
Test equal, decrement, and test halfword	TEDTH	0x0D	
Test greater, decrement, and test byte	TGDTB	0x6D	
Test greater, decrement, and test halfword	TGDTH	0x2D	
Test greater or equal, decrement, and test byte	TGEDTB	0x5D	
Test greater or equal, decrement, and test halfword	TGEDTH	0x1D	
Test not equal, decrement, and test byte	TNEDTB	0x7D	
Test not equal, decrement, and test halfword	TNEDTH	0x3D	

* Refer to Table 5-8 for condition code flag assignments.

** Indicates that another mnemonic has the same opcode.

† REQLU and REQL perform the same operation.

†† RNEQU and RNEQ perform the same operation.

INSTRUCTION SET AND ADDRESSING MODES

Program Control Instructions

Table 5-13. Program Control Instruction Group (Continued)				
Subroutine Transfer:				
Branch to subroutine, byte displacement	BSBB	0x37	Unchanged	
Branch to subroutine, halfword displacement	BSBH	0x36		
Jump to subroutine	JSB	0x34		
Return from subroutine	RSB	0x78		
Procedure Transfer:				
Save registers	SAVE	0x10		
Restore registers	RESTORE	0x18		
Call procedure	CALL	0x2C		
Return from procedure	RET	0x08		

Procedure-call instructions explicitly manipulate four registers:

- **PC.** The call instruction saves the old program counter (PC) as the return address (RA) and sets the PC to the first executable instruction of the function being called. The return instruction restores PC to the RA (the next executable instruction of the calling function).
- **SP.** The call instruction adjusts the stack pointer (SP) to point to the top of the stack whenever they store or retrieve items.
- **FP.** The save instruction sets the frame pointer (FP) to the address just above the saved registers. The FP accesses a region on the stack that stores temporary (or automatic) variables for the function.
- **AP.** The call instruction adjusts the argument pointer (AP) to the beginning of a list of arguments for the function.

On a function call, the calling function contains a call (CALL) instruction; the save (SAVE) instruction should be the first statement of the called function. For a return, a restore (RESTORE) and a return (RET) instruction appear in the function being exited.

Figure 5-20 shows the stack after the CALL-SAVE sequence:

```

PUSHW arg1          /*push three arguments */
PUSHW arg2
PUSHW arg3
CALL -(3*4)(%sp),func1 /*call function*/
    .
    .
    .
func1: SAVE %r3      /*save r3 through r8*/
    
```

INSTRUCTION SET AND ADDRESSING MODES

Coprocessor Instructions

First, three arguments are pushed onto the stack; each push increments the SP. Then, the CALL instruction automatically saves the old pointers. It uses its first operand to set the AP to the beginning of the three arguments, and its second operand to call the function. Next, the SAVE instruction (the first statement in the function) is executed, automatically saving registers r3 through r8 by pushing them on the stack. It also adjusts the SP and FP for each push.

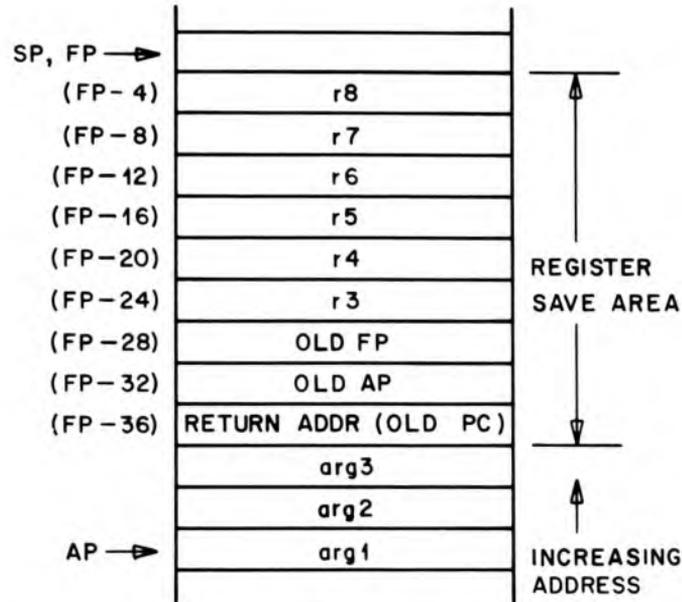


Figure 5-20. Stack After CALL-SAVE Sequence

To return to the original sequence, the function, func1, contains the following instructions:

```

func1:  SAVE %r3                /*save r3 through r8*/
        .
        .
        .
        RESTORE %r3            /*restore re through r8*/
        RET                    /*return to main function*/
    
```

The RESTORE instruction retrieves registers r3 through r8 from the stack. The RESTORE must have the same operand as the original SAVE; otherwise, the return (RET) cannot restore the correct AP and PC. Both instructions decrement the SP as they pop the register contents from the stack.

INSTRUCTION SET AND ADDRESSING MODES

Stack and Miscellaneous Instructions

5.3.6 Coprocessor Instructions

The instructions listed in Table 5-14 implement the interface with coprocessors. Most programmers will find it convenient to access the math acceleration unit (MAU) by using its own instruction set. All coprocessor instructions have an 8-bit opcode followed by one word. This word is transmitted on the data bus and interpreted by the coprocessor. The word is not used by the CPU. If no coprocessor responds to the transmitted word, an external memory fault occurs.

If $\overline{\text{DYN16}}$ is asserted during the coprocessor broadcast cycle, the transaction is repeated with the lower 16-bit part of the coprocessor ID replicated on the lower and upper parts of the data bus. The transaction (SAS code = "coprocessor broadcast") is qualified with a DSIZE of halfword.

After the word following the opcode is transmitted, the source operands, if any, are fetched from memory. The CPU then waits until the coprocessor done ($\overline{\text{DONE}}$) signal is asserted, after which the CPU attempts to read a word. If this access is faulted, an external memory fault occurs. If this access is not faulted, bits 18 through 21 of the word are copied into bits 18 through 21 (condition flags) of the PSW. The resulting operand, if any, is then written to memory.

If $\overline{\text{DYN16}}$ is asserted during the coprocessor status fetch cycle, the upper half of the data bus is used, and the transaction is repeated with a DSIZE of halfword in order to fetch the lower half of the status word.

The CPU acknowledges interrupts from the start of the instruction until the coprocessor status fetch access is started.

If an interrupt occurs while a coprocessor instruction is being traced, the instruction is terminated and control is handed to the trace handler. Since the instruction has not completed when the interrupt occurs, the PC points to the beginning of the coprocessor instruction upon return from the trace handler. If no interrupt occurs and the instruction completes normally, it traces just like any other instruction.

Coprocessor instructions can have from zero to two operands. The operands may be of three data types, specified by the last character of the mnemonic: single-word (S), double-word (D), and triple-word (T). All operands must start on an address evenly divisible by four (a word boundary).

INSTRUCTION SET AND ADDRESSING MODES

Stack and Miscellaneous Instructions

Table 5-14. Coprocessor Instruction Group			
Instruction	Mnemonic	Opcode	Conditions*
Coprocessor operation	SPOP	0x32	Case 11
Coprocessor operation read single	SOPRS	0x22	
Coprocessor operation read double	SOPRD	0x02	
Coprocessor operation read triple	SOPRT	0x06	
Coprocessor operation single 2-address	SOPPS2	0x23	
Coprocessor operation double 2-address	SOPD2	0x03	
Coprocessor operation triple 2-address	SOPT2	0x07	
Coprocessor operation write single	SPOPWS	0x33	
Coprocessor operation write double	SOPWD	0x13	
Coprocessor operation write triple	SPOPWT	0x17	

* Refer to Table 5-8 for condition flag code assignments.

5.3.7 Stack and Miscellaneous Instructions

The stack instructions listed in Table 5-15 are used to manipulate the stack. The push and pop instructions always process a word and alter the SP. They have a source operand (*src*) or a destination operand (*dst*).

Miscellaneous instructions include those that alter the machine state or have an effect on the cache memory. The breakpoint instruction causes a breakpoint-trap exception. Control transfers to the operating system for the appropriate exception handler. The NOP instructions come in three lengths: 1, 2, or 3 bytes. If an instruction, other than a conditional transfer, reads the PSW, the assembler inserts a NOP before that instruction. This allows time for the PSW codes to settle before the new instruction tries to access them. Cache flush makes the instruction cache invalid.

Table 5-15. Stack and Miscellaneous Instruction Group			
Instruction	Mnemonic	Opcode	Conditions*
Stack Operations:			Case 1
Push address word	PUSHAW	0xE0	
Push word	PUSHW	0xA0	
Pop word	POPW	0x20	
Miscellaneous:			Unchanged
No operation, 1 byte	NOP	0x70	
No operation, 2 byte	NOP2	0x73	
No operation, 3 byte	NOP3	0x72	
Breakpoint trap	BPT	0x2E	
Cache flush	CFLUSH	0x27	
Extended opcode	EXTOP	0x14	

* Refer to Table 5-8 for condition flag code assignments.

INSTRUCTION SET AND ADDRESSING MODES

Instruction Set Listings

5.4 INSTRUCTION SET LISTINGS

Section 5.4.4 presents descriptions of each member of the instruction set for the *WE 32200* Microprocessor. The descriptions are in alphabetical order, and any instructions that operate on more than one type of operand, byte, halfword, or word are listed on the same page. For quick reference to the instructions by function, mnemonic, or opcode see sections 5.3 (Tables 5-9 through 5-15), section 5.4.2, and section 5.4.3. The operating system instructions are listed in Chapter 6.

5.4.1 Notation

Each instruction description contains several parts: assembler syntax, opcode operation, address modes, condition flags, exceptions, examples, and notes (optional).

Assembler Syntax. Presents the assembly language syntax for the instruction, including any required spacing and punctuation. The user-specified elements appear in italics. All operands must appear in the order shown. If an instruction has byte, halfword, and word forms, all three forms are presented.

The syntax uses the following symbols to denote operands that may be written in the address modes shown in Table 5-3: *count*, *dst*, *offset*, *src*, *width*. Program control instructions use *disp8* or *disp16* as a displacement operand. The operand does not use an address mode, but is written as an 8- or 16-bit literal.

Opcodes. Lists each opcode with the appropriate mnemonic and function.

Operation. Describes the operation performed. The description generally uses C language syntax and the operators and symbols shown in Table 5-16.

Address Modes. Identifies the valid address modes for each operand. Refer to Table 5-3 for address mode syntax and to Table 5-1 for the syntax for referencing registers.

Condition Flags. Identifies the effect of the instruction on each of the condition flags.

Exceptions. Identifies any error conditions that may result in illegal operands, opcodes, or operations.

Examples. Presents examples of the instruction written in assembly language. In some cases, the contents of the registers, before and after execution, are given. Register bytes are read from right to left and their contents are given as hexadecimal values.

Notes (Optional). Explains other parts of the description when necessary.

Table 5-16. Assembly Language Operators and Symbols	
Symbol	Description
*x	Indirection; value pointed to by x
&x	Address of x
!x	Not x
++x	Increment x
--x	Decrement x
x	Complement x; form one's complement of x
-x	Negate x; form two's complement of x
x+y	Add y to x
x-y	Subtract y from x
x*y	Multiply x by y
x/y	Divide y into x
x%y	Modulo x and y (remainder of x/y)
x&y	Bitwise AND x and y
xly	Bitwise inclusive OR x and y
xy	Bitwise exclusive OR (XOR) x and y
x<<y	Shift x to the left y bits
x>>y	Shift x to the right y bits
x<y	x less than y
x>y	x greater than y
x==y	Equality; x equal to y
x!=y	x not equal to y
x=y	Assigns the value in location y to location x
AP	Argument pointer; register 10 (r10)
count	Count operand
dst	Destination operand
FP	Frame pointer; register 9 (r9)
PC	Program counter; register 15 (r15)
PSW	Processor status word; register 11 (r11)
SEXT(x)	Function that returns x, sign-extended through 32 bits
SP	Stack pointer; register 12 (r12)
*(--SP)	A pop from the stack; decrement SP by 4 before removing data () from the stack
*(SP++)	A push onto the stack; store data and increment SP by 4
src	Source operand
0xn	Hexadecimal value where n is the digits 0 through 9 and a through f (or A through F); may also be written 0Xn
/*comment*/	A comment, not an operation
{operation}	An operation other than an instruction

INSTRUCTION SET AND ADDRESSING MODES
Instruction Set by Mnemonic

5.4.2 Instruction Set Summary by Mnemonic

A mnemonic listing of instructions is given in Table 5-17.

Table 5-17. Instruction Set Summary by Mnemonic		
Mnemonic	Opcode	Instruction
ADDB2	0x9F	Add byte
ADDB3	0xDF	Add byte, 3-address
ADDH2	0x9E	Add halfword
ADDH3	0xDE	Add halfword, 3-address
ADDPB2	0xA3	Add packed BCD
ADDPB3	0xE3	Add packed BCD, 3-address
ADDW2	0x9C	Add word
ADDW3	0xDC	Add word, 3-address
ALSW3	0xC0	Arithmetic left shift word
ANDB2	0xBB	AND byte
ANDB3	0xFB	AND byte, 3-address
ANDH2	0xBA	AND halfword
ANDH3	0xFA	AND halfword, 3-address
ANDW2	0xB8	AND word
ANDW3	0xF8	AND word, 3-address
ARSB3	0xC7	Arithmetic right shift byte
ARSH3	0xC6	Arithmetic right shift halfword
ARSW3	0xC4	Arithmetic right shift word
BCCB	0x53*	Branch on carry clear byte
BCCH	0x52*	Branch on carry clear halfword
BCSB	0x5B*	Branch on carry set byte
BCSH	0x5A*	Branch on carry set halfword
BEB	0x6F	Branch on equal byte
BEB	0x7F	Branch on equal byte (duplicate)
BEH	0x6E	Branch on equal halfword
BEH	0x7E	Branch on equal halfword (duplicate)
BGB	0x47	Branch on greater than byte (signed)
BGEB	0x43	Branch on greater than or equal byte (signed)
BGEH	0x42	Branch on greater than or equal halfword (signed)
BGEUB	0x53*	Branch on greater than or equal byte (unsigned)
BGEUH	0x52*	Branch on greater than or equal halfword (unsigned)
BGH	0x46	Branch on greater than halfword (signed)
BGUB	0x57	Branch on greater than byte (unsigned)
BGUH	0x56	Branch on greater than halfword (unsigned)

* Indicates that another mnemonic has the same opcode.

INSTRUCTION SET AND ADDRESSING MODES
Instruction Set by Mnemonic

Table 5-17. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	Opcode	Instruction
BITB	0x3B	Bit test byte
BITH	0x3A	Bit test halfword
BITW	0x38	Bit test word
BLB	0x4B	Branch on less than byte (signed)
BLEB	0x4F	Branch on less than or equal byte (signed)
BLEH	0x4E	Branch on less than or equal halfword (signed)
BLEUB	0x5F	Branch on less than or equal byte (unsigned)
BLEUH	0x5E	Branch on less than or equal halfword (unsigned)
BLH	0x4A	Branch on less than halfword (signed)
BLUB	0x5B*	Branch on less than byte (unsigned)
BLUH	0x5A*	Branch on less than halfword (unsigned)
BNEB	0x67	Branch on not equal byte (duplicate)
BNEB	0x77	Branch on not equal byte
BNEH	0x66	Branch on not equal halfword (duplicate)
BNEH	0x76	Branch on not equal halfword
BPT	0x2E	Breakpoint trap
BRB	0x7B	Branch with byte displacement
BRH	0x7A	Branch with halfword displacement
BSBB	0x37	Branch to subroutine, byte displacement
BSBH	0x36	Branch to subroutine, halfword displacement
BVCB	0x63	Branch on overflow clear, byte displacement
BVCH	0x62	Branch on overflow clear, halfword displacement
BVSB	0x6B	Branch on overflow set, byte displacement
BVSH	0x6A	Branch on overflow set, halfword displacement
CALL	0x2C	Call procedure
CALLPS**	0x30AC	Call process
CASWI	0x09	Compare and swap word interlock
CFLUSH	0x27	Cache flush
CLRB	0x83	Clear byte
CLRH	0x82	Clear halfword
CLRW	0x80	Clear word
CLRX	0x0B	Clear X bit in PSW
CMPB	0x3F	Compare byte
CMPH	0x3E	Compare halfword
CMPW	0x3C	Compare word
DECB	0x97	Decrement byte
DECH	0x96	Decrement halfword
DECW	0x94	Decrement word
DISVJMP**	0x3013	Disable virtual pin and jump

* Indicates that another mnemonic has the same opcode.

** Indicates operating system instruction.

INSTRUCTION SET AND ADDRESSING MODES

Instruction Set by Mnemonic

Table 5-17. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	Opcode	Instruction
DIVB2	0xAF	Divide byte
DIVB3	0xEF	Divide byte, 3-address
DIVH2	0xAE	Divide halfword
DIVH3	0xEE	Divide halfword, 3-address
DIVW2	0xAC	Divide word
DIVW3	0xEC	Divide word, 3-address
DTB	0x29	Decrement, test, and branch byte
DTH	0x19	Decrement, test, and branch halfword
ENBVJMP**	0x300D	Enable virtual pin and jump
EXTFB	0xCF	Extract field byte
EXTFH	0xCE	Extract field halfword
EXTFW	0xCC	Extract field word
EXTOP	0x14	Extended opcode
GATE**	0x3061	Gate
INCB	0x93	Increment byte
INCH	0x92	Increment halfword
INCW	0x90	Increment word
INSFB	0xCB	Insert field byte
INSFH	0xCA	Insert field halfword
INSFW	0xC8	Insert field word
INTACK**	0x302F	Interrupt acknowledge
JMP	0x24	Jump
JSB	0x34	Jump to subroutine
LLSB3	0xD3	Logical left shift byte
LLSH3	0xD2	Logical left shift halfword
LLSW3	0xD0	Logical left shift word
LRSW3	0xD4	Logical right shift word
MCOMB	0x8B	Move complemented byte
MCOMH	0x8A	Move complemented halfword
MCOMW	0x88	Move complemented word
MNEGB	0x8F	Move negated byte
MNEGH	0x8E	Move negated halfword
MNEGW	0x8C	Move negated word
MODB2	0xA7	Modulo byte
MODB3	0xE7	Modulo byte, 3-address
MODH2	0xA6	Modulo halfword
MODH3	0xE6	Modulo halfword, 3-address
MODW2	0xA4	Modulo word
MODW3	0xE4	Modulo word, 3-address

* Indicates that another mnemonic has the same opcode.

** Indicates operating system instruction.

INSTRUCTION SET AND ADDRESSING MODES
Instruction Set by Mnemonic

Table 5-17. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	Opcode	Instruction
MOVAW	0x04	Move address (word)
MOVB	0x87	Move byte
MOVBLW	0x3019	Move block of words
MOVH	0x86	Move halfword
MOVTRW**	0x0C	Move translated word
MOVW	0x84	Move word
MULB2	0xAB	Multiply byte
MULB3	0xEB	Multiply byte, 3-address
MULH2	0xAA	Multiply halfword
MULH3	0xEA	Multiply halfword, 3-address
MULW2	0xA8	Multiply word
MULW3	0xE8	Multiply word, 3-address
MVERNO	0x3009	Move version number
NOP	0x70	No operation, 1 byte
NOP2	0x73	No operation, 2 byte
NOP3	0x72	No operation, 3 byte
ORB2	0xB3	OR byte
ORB3	0xF3	OR byte, 3-address
ORH2	0xB2	OR halfword
ORH3	0xF2	OR halfword, 3-address
ORW2	0xB0	OR word
ORW3	0xF0	OR word, 3-address
PACKB	0x0E	Pack BCD
POPW	0x20	Pop word
PUSHAW	0xE0	Push address word
PUSHW	0xA0	Push word
RCC	0x50*	Return on carry clear
RCS	0x58*	Return on carry set
REQLU	0x6C†	Return on equal (unsigned)
REQL	0x7C†	Return on equal (signed)
RESTORE	0x18	Restore registers
RET	0x08	Return from procedure
RETG**	0x3045	Return from gate
RETPS**	0x30C8	Return to process
RETQINT**	0x98	Return from quick interrupt
RGEQ	0x40	Return on greater than or equal (signed)
RGEQU	0x50*	Return on greater than or equal (unsigned)
RGTR	0x44	Return on greater than (signed)
RGTRU	0x54	Return on greater than (unsigned)

* Indicates that another mnemonic has the same opcode.

** Indicates operating system instruction.

† REQLU and REQL perform the same operation.

INSTRUCTION SET AND ADDRESSING MODES

Instruction Set by Opcode

Table 5-17. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	Opcode	Instruction
RLEQ	0x4C	Return on less than or equal (signed)
RLEQU	0x5C	Return on less than or equal (unsigned)
RLSS	0x48	Return on less than (signed)
RLSSU	0x58*	Return on less than (unsigned)
RNEQU	0x64†	Return on not equal (unsigned)
RNEQ	0x74†	Return on not equal (signed)
ROTW	0xD8	Rotate word
RSB	0x78	Return from subroutine
RVC	0x60	Return on overflow clear
RVS	0x68	Return on overflow set
SAVE	0x10	Save registers
SETX	0x0A	Set X bit in PSW
SPOP	0x32	Coprocessor operation
SPOPD2	0x03	Coprocessor operation double, 2-address
SPOPRD	0x02	Coprocessor operation read double
SPOPRS	0x22	Coprocessor operation read single
SPOPRT	0x06	Coprocessor operation read triple
SPOPS2	0x23	Coprocessor operation single, 2-address
SPOPT2	0x07	Coprocessor operation triple, 2-address
SPOPWD	0x13	Coprocessor operation write double
SPOPWS	0x33	Coprocessor operation write single
SPOPWT	0x17	Coprocessor operation write triple
STRCPY	0x3035	String copy
STREND	0x301F	String end
SUBB2	0xBF	Subtract byte
SUBB3	0xFF	Subtract byte, 3-address
SUBH2	0xBE	Subtract halfword
SUBH3	0xFE	Subtract halfword, 3-address
SUBPB2	0x9B	Subtract packed BCD
SUBPB3	0xDB	Subtract packed BCD, 3-address
SUBW2	0xBC	Subtract word
SUBW3	0xFC	Subtract word, 3-address
SWAPBI	0x1F	Swap-byte interlocked
SWAPHI	0x1E	Swap-halfword interlocked
SWAPWI	0x1C	Swap-word interlocked

* Indicates that another mnemonic has the same opcode.

** Indicates operating system instruction.

† RNEQU and RNEQ perform the same operation.

Table 5-17. Instruction Set Summary by Mnemonic (Continued)		
Mnemonic	Opcode	Instruction
TEDTB	0x4D	Test equal, decrement and test byte
TGEDTB	0x5D	Test greater or equal, decrement and test byte
TGDTB	0x6D	Test greater, decrement and test byte
TNEDTB	0x7D	Test not equal, decrement and test byte
TEDTH	0x0D	Test equal, decrement and test halfword
TGEDTH	0x1D	Test greater or equal, decrement and test halfword
TGDTH	0x2D	Test greater, decrement and test halfword
TNEDTH	0x3D	Test not equal, decrement and test halfword
TSTB	0x2B	Test byte
TSTH	0x2A	Test halfword
TSTW	0x28	Test word
UCALLPS**	0x30C0	User call process
UNPACKB	0x0F	Unpack BCD
WAIT**	0x2F	Wait for interrupt
XORB2	0xB7	Exclusive OR byte
XORB3	0xF7	Exclusive OR byte, 3-address
XORH2	0xB6	Exclusive OR halfword
XORH3	0xF6	Exclusive OR halfword, 3-address
XORW2	0xB4	Exclusive OR word
XORW3	0xF4	Exclusive OR word, 3-address

* Indicates that another mnemonic has the same opcode.

** Indicates operating system instruction.

5.4.3 Instruction Set Summary by Opcode

The instruction sets are listed by opcode in Table 5-18.

Table 5-18. Instruction Set Summary by Opcode		
Opcode	Mnemonic	Instruction
0x02	SPOPRD	Coprocessor operation read double
0x03	SPOPD2	Coprocessor operation double, 2-address
0x04	MOVAV	Move address (word)
0x06	SPOPRT	Coprocessor operation read triple
0x07	SPOPT2	Coprocessor operation triple, 2-address
0x08	RET	Return from procedure
0x09	CASWI	Compare and swap-word interlock
0x0A	SETX	Set X bit in PSW
0x0B	CLR X	Clear X bit in PSW
0x0C	MOVTRW*	Move translated word
0x0D	TEDTH	Test equal, decrement and test halfword
0x0E	PACKB	Pack BCD
0x0F	UNPACKB	Unpack BCD

* Indicates operating system instruction.

INSTRUCTION SET AND ADDRESSING MODES

Instruction Set by Opcode

Table 5-18. Instruction Set Summary by Opcode (Continued)		
Opcode	Mnemonic	Instruction
0x10	SAVE	Save registers
0x13	SPOPWD	Coprocessor operation write double
0x14	EXTOP	Extended opcode
0x17	SPOPWT	Coprocessor operation write triple
0x18	RESTORE	Restore registers
0x19	DTH	Decrement, test, and branch halfword
0x1C	SWAPWI	Swap-word interlocked
0x1D	TGEDTH	Test greater or equal, decrement and test halfword
0x1E	SWAPHI	Swap-halfword interlocked
0x1F	SWAPBI	Swap-byte interlocked
0x20	POPW	Pop word
0x22	SPOPRS	Coprocessor operation read single
0x23	SPOPS2	Coprocessor operation single, 2-address
0x24	JMP	Jump
0x27	CFLUSH	Cache flush
0x28	TSTW	Test word
0x29	DTP	Decrement, test, and branch byte
0x2A	TSTH	Test halfword
0x2B	TSTB	Test byte
0x2C	CALL	Call procedure
0x2D	TGDTH	Test greater, decrement and test halfword
0x2E	BPT	Breakpoint trap
0x2F	WAIT*	Wait for interrupt
0x3009	MVERNO	Move version number
0x300D	ENBVJMP*	Enable virtual pin and jump
0x3013	DISVJMP*	Disable virtual pin and jump
0x3019	MOVBLW	Move block of words
0x301F	STREND	String end
0x302F	INTACK*	Interrupt acknowledge
0x3035	STRCPY	String copy
0x3045	RETG*	Return from gate
0x3061	GATE*	Gate
0x30AC	CALLPS*	Call process
0x30C0	UCALLPS*	User call process
0x30C8	RETPS*	Return to process
0x32	SPOP	Coprocessor Operation
0x33	SPOPWS	Coprocessor operation write single
0x34	JSB	Jump to subroutine
0x36	BSBH	Branch to subroutine, halfword displacement

* Indicates operating system instruction.

INSTRUCTION SET AND ADDRESSING MODES
Instruction Set by Opcode

Table 5-18. Instruction Set Summary by Opcode (Continued)		
Opcode	Mnemonic	Instruction
0x37	BSBB	Branch to subroutine, byte displacement
0x38	BITW	Bit test word
0x3A	BITH	Bit test halfword
0x3B	BITB	Bit test byte
0x3C	CMPW	Compare word
0x3D	TNEDTH	Test not equal, decrement and test halfword
0x3E	CMPH	Compare halfword
0x3F	CMPB	Compare byte
0x40	RGEQ	Return on greater than or equal (signed)
0x42	BGEH	Branch on greater than or equal halfword (signed)
0x43	BGEB	Branch on greater than or equal byte (signed)
0x44	RGTR	Return on greater than (signed)
0x46	BGH	Branch on greater than halfword (signed)
0x47	BGB	Branch on greater than byte (signed)
0x48	RLSS	Return on less than (signed)
0x4A	BLH	Branch on less than halfword (signed)
0x4B	BLB	Branch on less than byte (signed)
0x4C	RLEQ	Return on less than or equal (signed)
0x4D	TEDTB	Test equal, decrement and test byte
0x4E	BLEH	Branch on less than or equal halfword (signed)
0x4F	BLEB	Branch on less than or equal byte (signed)
0x50**	RCC	Return on carry clear
0x50**	RGEQU	Return on greater than or equal (unsigned)
0x52**	BCCH	Branch on carry clear halfword
0x52**	BGEUH	Branch on greater than or equal halfword (unsigned)
0x53**	BCCB	Branch on carry clear byte
0x53**	BGEUB	Branch on greater than or equal byte (unsigned)
0x54	RGTRU	Return on greater than (unsigned)
0x56	BGUH	Branch on greater than halfword (unsigned)
0x57	BGUB	Branch on greater than byte (unsigned)
0x58**	RCS	Return on carry set
0x58**	RLSSU	Return on less than (unsigned)
0x5A**	BCSH	Branch on carry set halfword
0x5A**	BLUH	Branch on less than halfword (unsigned)
0x5B**	BCSB	Branch on carry set byte
0x5B**	BLUB	Branch on less than byte (unsigned)
0x5C	RLEQU	Return on less than or equal (unsigned)
0x5D	TGEDTB	Test greater or equal, decrement and test byte
0x5E	BLEUH	Branch on less than or equal halfword (unsigned)
0x5F	BLEUB	Branch on less than or equal byte (unsigned)

** Indicates that another mnemonic has the same opcode.

INSTRUCTION SET AND ADDRESSING MODES

Instruction Set by Opcode

Table 5-18. Instruction Set Summary by Opcode (Continued)		
Opcode	Mnemonic	Instruction
0x60	RVC	Return on overflow clear
0x62	BVCH	Branch on overflow clear, halfword displacement
0x63	BVCB	Branch on overflow clear, byte displacement
0x64†	RNEQU	Return on not equal (unsigned)
0x66	BNEH	Branch on not equal halfword (duplicate)
0x67	BNEB	Branch on not equal byte (duplicate)
0x68	RVS	Return on overflow set
0x6A	BVSH	Branch on overflow set, halfword displacement
0x6B	BVSB	Branch on overflow set, byte displacement
0x6C††	REQU	Return on equal (unsigned)
0x6D	TGDTB	Test greater, decrement and test byte
0x6E	BEH	Branch on equal halfword (duplicate)
0x6F	BEB	Branch on equal byte (duplicate)
0x70	NOP	No operation, 1 byte
0x72	NOP3	No operation, 3 byte
0x73	NOP2	No operation, 2 byte
0x74†	RNEQ	Return on not equal (signed)
0x76	BNEH	Branch on not equal halfword
0x77	BNEB	Branch on not equal byte
0x78	RSB	Return from subroutine
0x7A	BRH	Branch with halfword displacement
0x7B	BRB	Branch with byte displacement
0x7C††	REQL	Return on equal (signed)
0x7D	TNEDTB	Test not equal, decrement and test byte
0x7E	BEH	Branch on equal halfword
0x7F	BEB	Branch on equal byte
0x80	CLRW	Clear word
0x82	CLRH	Clear halfword
0x83	CLRB	Clear byte
0x84	MOVW	Move word
0x86	MOVH	Move halfword
0x87	MOVB	Move byte
0x88	MCOMW	Move complemented word
0x8A	MCOMH	Move complemented halfword
0x8B	MCOMB	Move complemented byte
0x8C	MNEGW	Move negated word
0x8E	MNEGH	Move negated halfword
0x8F	MNEGB	Move negated byte
0x90	INCW	Increment word
0x92	INCH	Increment halfword
0x93	INCB	Increment byte

† RNEQU and RNEQ perform the same operation.

†† REQU and REQL perform the same operation.

INSTRUCTION SET AND ADDRESSING MODES
Instruction Set by Opcode

Table 5-18. Instruction Set Summary by Opcode (Continued)		
Opcode	Mnemonic	Instruction
0x94	DECW	Decrement word
0x96	DECH	Decrement halfword
0x97	DECB	Decrement byte
0x98	RETQINT*	Return from quick interrupt
0x9B	SUBPB2	Subtract packed BCD, 2-address
0x9C	ADDW2	Add word
0x9E	ADDH2	Add halfword
0x9F	ADDB2	Add byte
0xA0	PUSHW	Push word
0xA3	ADDPB2	Add packed BCD, 2-address
0xA4	MODW2	Modulo word
0xA6	MODH2	Modulo halfword
0xA7	MODB2	Modulo byte
0xA8	MULW2	Multiply word
0xAA	MULH2	Multiply halfword
0xAB	MULB2	Multiply byte
0xAC	DIVW2	Divide word
0xAE	DIVH2	Divide halfword
0xAF	DIVB2	Divide byte
0xB0	ORW2	OR word
0xB2	ORH2	OR halfword
0xB3	ORB2	OR byte
0xB4	XORW2	Exclusive OR word
0xB6	XORH2	Exclusive OR halfword
0xB7	XORB2	Exclusive OR byte
0xB8	ANDW2	AND word
0xBA	ANDH2	AND halfword
0xBB	ANDB2	AND byte
0xBC	SUBW2	Subtract word
0xBE	SUBH2	Subtract halfword
0xBF	SUBB2	Subtract byte
0xC0	ALSW3	Arithmetic left shift word
0xC4	ARSW3	Arithmetic right shift word
0xC6	ARSH3	Arithmetic right shift halfword
0xC7	ARSB3	Arithmetic right shift byte
0xC8	INSFW	Arithmetic left shift word
0xCA	INSFH	Insert field halfword
0xCB	INSFB	Insert field byte
0xCC	EXTFW	Extract field word
0xCE	EXTFH	Extract field halfword
0xCF	EXTFB	Extract field byte

* Indicates operating system instruction.

INSTRUCTION SET AND ADDRESSING MODES

Instruction Set Descriptions

Table 5-18. Instruction Set Summary by Opcode (Continued)		
Opcode	Mnemonic	Instruction
0xD0	LLSW3	Logical left shift word
0xD2	LLSH3	Logical left shift halfword
0xD3	LLSB3	Logical left shift byte
0xD4	LRSW3	Logical right shift word
0xD8	ROTW	Rotate word
0xDB	SUBPB3	Subtract packed BCD, 3-address
0xDC	ADDW3	Add word, 3-address
0xDE	ADDH3	Add halfword, 3-address
0xDF	ADDB3	Add byte, 3-address
0xE0	PUSHAW	Push address word
0xE3	ADDPB3	Add packed BCD, 3-address
0xE4	MODW3	Modulo word, 3-address
0xE6	MODH3	Modulo halfword, 3-address
0xE7	MODB3	Modulo byte, 3-address
0xE8	MULW3	Multiply word, 3-address
0xEA	MULH3	Multiply halfword, 3-address
0xEB	MULB3	Multiply byte, 3-address
0xEC	DIVW3	Divide word, 3-address
0xEE	DIVH3	Divide halfword, 3-address
0xEF	DIVB3	Divide byte, 3-address
0xF0	ORW3	OR word, 3-address
0xF2	ORH3	OR halfword, 3-address
0xF3	ORB3	OR byte, 3-address
0xF4	XORW3	Exclusive OR word, 3-address
0xF6	XORH3	Exclusive OR halfword, 3-address
0xF7	XORB3	Exclusive OR byte, 3-address
0xF8	ANDW3	AND word, 3-address
0xFA	ANDH3	AND halfword, 3-address
0xFB	ANDB3	AND byte, 3-address
0xFC	SUBW3	Subtract word, 3-address
0xFE	SUBH3	Subtract halfword, 3-address
0xFF	SUBB3	Subtract byte, 3-address

5.4.4 Instruction Set Descriptions

The instruction set is described in detail on the following pages. For a summary of the notations used in these descriptions, see section 5.4.1.

ADDB2
ADDH2
ADDW2

ADDB2
ADDH2
ADDW2

ADD

**Assembler
Syntax**

ADDB2 *src,dst* Add byte
ADDH2 *src,dst* Add halfword
ADDW2 *src,dst* Add word

Opcodes

0x9F ADDB2
0x9E ADDH2
0x9C ADDW2

Operation

$dst = dst + src$

Address

src all modes

Modes

dst all modes except literal or immediate

**Condition
Flags**

N = 1, if $(dst + src) < 0$
Z = 1, if $(dst + src) == 0$
C = 1, if carry out of sign bit of *dst*
V = 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate modes are used for *dst*.
Integer overflow exception occurs if PSW<OE> = 1, and there is truncation.

Examples

ADDB2 \$0x100,%r0
ADDH2 %r0,%r3
ADDW2 4(%r3),*\$0x110

ADDB3
ADDH3
ADDW3

ADDB3
ADDH3
ADDW3

ADD, 3 ADDRESS

Assembler Syntax	ADDB3 <i>src1,src2,dst</i>	Add byte, 3 address
	ADDH3 <i>src1,src2,dst</i>	Add halfword, 3 address
	ADDW3 <i>src1,src2,dst</i>	Add word, 3 address
Opcodes	0xDF ADDB3 0xDE ADDH3 0xDC ADDW3	
Operation	$dst = src1 + src2$	
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = 1, if $(src1 + src2) < 0$ Z = 1, if $(src1 + src2) == 0$ C = 1, if carry out of sign bit of <i>dst</i> V = 1, if overflow	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer overflow exception occurs if PSW<OE> = 1, and there is truncation.	
Examples	ADDB3 %r0,%r3,%r5 ADDH3 4(%r2),*\$0x110,%r3 ADDW3 *\$0x1F0,4(%r1),%r0	

ADDPB2

ADDPB2

ADD PACKED DECIMAL

Assembler Syntax	ADDPB2 <i>src, dst</i> Add packed byte, 2-address
Opcodes	0xA3 ADDPB2
Operation	$dst = src + dst + X$
Address Modes	<i>src</i> all modes except expanded-operand <i>dst</i> all modes except expanded-operand
Condition Flags	N = 0 Z = 1 if $(src + dst + X) == 0$ C = carry for BCD addition, or borrow for BCD subtraction X = carry for BCD addition, or borrow for BCD subtraction V = 0
Exceptions	Invalid descriptor exception occurs if expanded-operand descriptor is used.
Examples	ADDPB2 %r1, \$0x15000 ADDPB2 %r1, %r2
Notes	<i>src</i> is read-only. <i>dst</i> is write-only.

ADDPB3

ADDPB3

ADD PACKED DECIMAL, 3 ADDRESS

Assembler Syntax	ADDPB3 <i>src1,src2,dst</i> Add packed byte, 3-address
Opcodes	0xE3 ADDPB3
Operation	$dst = src1 + src2 + X$
Address Modes	<i>src1</i> all modes except expanded-operand <i>src2</i> all modes except expanded-operand <i>dst</i> all modes except expanded-operand
Condition Flags	N = 0 Z = 1 if ($src1 + src2 + X$) == 0 C = carry for BCD addition, or borrow for BCD subtraction X = carry for BCD addition, or borrow for BCD subtraction V = 0
Exceptions	Invalid descriptor exception occurs if expanded-operand descriptor is used.
Examples	ADDPB3 \$0x12000,\$0x13000,\$0x15000 ADDPB3 %r1,%r2,%r3
Notes	<i>src1</i> and <i>src2</i> are read-only. <i>dst</i> is write-only.

ARITHMETIC LEFT SHIFT

Assembler Syntax	ALSW3 <i>count,src,dst</i> Arithmetic left shift word
Opcode	0xC0 ALSW3
Operation	$dst = src \ll (\text{count} \& 0x1F)$ bits
Address Modes	<i>count</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate
Condition Flags	N = 1, if <i>dst</i> < 0 Z = 1, if <i>dst</i> == 0 C = 0 V = 0 (see Notes)
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .

Example Before: r0

8F	0F	DF	FD
----	----	----	----

← Increasing Bits

ALSW3 &2,%r0,%r0

After: r0

3C	3F	7F	F4
----	----	----	----

Notes All operands are of word type. However, only the five low-order bits of *count* are used; the upper bits are ignored. No bits are shifted past the sign bit, so integer overflow cannot occur. However, the V bit can be set if an expanded-operand mode changes the type of *dst*. Zeros replace bits that are shifted out. The sign bit is not changed.

ANDB2
ANDH2
ANDW2

ANDB2
ANDH2
ANDW2

AND

Assembler Syntax	<i>ANDB2 src,dst</i> AND byte <i>ANDH2 src,dst</i> AND halfword <i>ANDW2 src,dst</i> AND word
Opcodes	0xBB <i>ANDB2</i> 0xBA <i>ANDH2</i> 0xB8 <i>ANDW2</i>
Operation	<i>dst = dst & src</i>
Address Modes	<i>src</i> all modes <i>dst</i> all modes except literal or immediate
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if result must be truncated to fit <i>dst</i> size
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .
Examples	<i>ANDB2 &7,6(%r1)</i> <i>ANDH2 %r0,*\$result</i> <i>ANDW2 (%r1),%r4</i>

ANDB3
ANDH3
ANDW3

ANDB3
ANDH3
ANDW3

AND, 3 ADDRESS

Assembler Syntax	<i>ANDB3 src1,src2,dst</i> <i>ANDH3 src1,src2,dst</i> <i>ANDW3 src1,src2,dst</i>	AND byte, 3 address AND halfword, 3 address AND word, 3 address
Opcodes	0xFB <i>ANDB3</i> 0xFA <i>ANDH3</i> 0xF8 <i>ANDW3</i>	
Operation	<i>dst = src2 & src1</i>	
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if result must be truncated to fit <i>dst</i> size	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .	
Examples	<i>ANDB3 &0x27,*\$0x300,%r6</i> <i>ANDH3 0x31(%r5),%r0,%r1</i> <i>ANDW3 %r2,%r1,%r0</i>	

ARSB3
ARSH3
ARSW3

ARSB3
ARSH3
ARSW3

ARITHMETIC RIGHT SHIFT

Assembler Syntax	<i>ARSB3 count,src,dst</i> Arithmetic right shift byte <i>ARSH3 count,src,dst</i> Arithmetic right shift halfword <i>ARSW3 count,src,dst</i> Arithmetic right shift word								
Opcodes	0xC7 ARSB3 0xC6 ARSH3 0xC4 ARSW3								
Operation	$dst = src \gg (count \& 0x1f)$ bits								
Address Modes	<i>count</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate								
Condition Flags	N = 1, if <i>dst</i> < 0 Z = 1, if <i>dst</i> == 0 C = 0 V = 0								
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .								
Example	Before: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="padding: 2px 5px;">0F</td><td style="padding: 2px 5px;">0F</td><td style="padding: 2px 5px;">77</td><td style="padding: 2px 5px;">AF</td></tr></table> <p style="text-align: center;">← Increasing Bits</p> <i>ARSH3 &2,%r0,%r0</i> After: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="padding: 2px 5px;">00</td><td style="padding: 2px 5px;">00</td><td style="padding: 2px 5px;">1D</td><td style="padding: 2px 5px;">EB</td></tr></table>	0F	0F	77	AF	00	00	1D	EB
0F	0F	77	AF						
00	00	1D	EB						
Notes	All operands are of word type. However, only the five low-order bits of <i>count</i> are used; the upper bits are ignored. The sign bit (MSB) of <i>src</i> is copied as bits are shifted out. The type of <i>src</i> does not affect sign extension.								

BCCB
BCCH

BCCB
BCCH

BRANCH ON CARRY CLEAR

Assembler Syntax	BCCB <i>disp8</i> Branch on carry clear, byte displacement BCCH <i>disp16</i> Branch on carry clear, halfword displacement
Opcodes	0x53 BCCB 0x52 BCCH
Operation	if (C == 0) PC = PC + SEXT(<i>disp</i>)
Address Modes	None valid
Condition Flags	Unchanged
Exceptions	None
Examples	BCCB 0x9 BCCH 0xFF23
Notes	<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.

BCSB
BCSH

BCSB
BCSH

BRANCH ON CARRY SET

Assembler Syntax **BCSB** *disp8* Branch on carry set, byte displacement
 BCSH *disp16* Branch on carry set, halfword displacement

Opcodes 0x5B **BCSB**
 0x5A **BCSH**

Operation if (C ==1)
 PC = PC + SEXT(*disp*)

Address Modes None valid

Condition Flags Unchanged

Exceptions None

Examples **BCSB** 0xFF
 BCSH 0x1234

Notes *disp8* is a signed 8-bit value.
 disp16 is a signed 16-bit value.

BEB
BEH

BEB
BEH

BRANCH ON EQUAL

Assembler Syntax **BEB** *disp8* Branch on equal, byte displacement
 BEH *disp16* Branch on equal, byte displacement

Opcodes 0x7F **BEB**
 0x6F **BEB**
 0x7E **BEH**
 0x6E **BEH**

Operation if (Z == 1)
 PC = PC + SEXT(*disp*)

Address Modes None valid

Condition Flags Unchanged

Exceptions None

Examples **BEB** 0xF1
 BEH 0x4221

Notes *disp8* is a signed 8-bit value.
 disp16 is a signed 16-bit value.

BGB
BGH

BGB
BGH

BRANCH ON GREATER THAN (SIGNED)

Assembler Syntax **BGB *disp8*** Branch on greater than, byte displacement (signed)
 BGH *disp16* Branch on greater than, halfword displacement (signed)

Opcodes 0x47 BGB
 0x46 BGH

Operation if ((N & Z) == 0)
 PC = PC + SEXT(*disp*)

Address Modes None valid
Condition Flags Unchanged

Exceptions None

Examples BGB more
 BGH less

Notes *disp8* is a signed 8-bit value.
 disp16 is a signed 16-bit value.

BGEB
BGEH

BGEB
BGEH

BRANCH ON GREATER THAN OR EQUAL (SIGNED)

Assembler Syntax	BGEB <i>disp8</i> Branch on greater than or equal, byte displacement (signed) BGEH <i>disp16</i> Branch on greater than or equal, halfword displacement (signed)
Opcodes	0x43 BGEB 0x42 BGEH
Operation	if ((N == 0) (Z == 1)) PC = PC + SEXT(<i>disp</i>)
Address Modes	None valid
Condition Flags	Unchanged
Exceptions	None
Examples	BGEB again BGEH 0xF102
Notes	<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.

BGEUB
BGEUH

BGEUB
BGEUH

BRANCH ON GREATER THAN OR EQUAL (UNSIGNED)

Assembler Syntax	BGEUB <i>disp8</i> BGEUH <i>disp16</i>	Branch on greater than or equal, byte displacement (unsigned) Branch on greater than or equal, halfword displacement (unsigned)
Opcodes	0x53 BGEUB 0x52 BGEUH	
Operation	if (C == 0) PC = PC + SEXT(<i>disp</i>)	
Address Modes	None valid	
Condition Flags	Unchanged	
Exceptions	None	
Examples	BGEUB 0xA1 BGEUH ahead	
Notes	<i>disp8</i> is an unsigned 8-bit value. <i>disp16</i> is an unsigned 16-bit value.	

BGUB
BGUH

BGUB
BGUH

BRANCH ON GREATER THAN (UNSIGNED)

Assembler Syntax *BGUB disp8* Branch on greater than, byte displacement (unsigned)
 BGUH disp16 Branch on greater than, halfword displacement (unsigned)

Opcodes 0x57 *BGUB*
 0x56 *BGUH*

Operation if ((C & Z) == 0)
 PC = PC + SEXT(*disp*)

Address Modes None valid

Condition Flags Unchanged

Exceptions None

Examples *BGUB* 0xDE
 BGUH 0xF123

Notes *disp8* is an unsigned 8-bit value.
 disp16 is an unsigned 16-bit value.

BITB
BITH
BITW

BITB
BITH
BITW

BIT TEST

Assembler Syntax *BITB src1,src2* Bit test byte
 BITH src1,src2 Bit test halfword
 BITW src1,src2 Bit test word

Opcodes 0x3B *BITB*
 0x3A *BITH*
 0x38 *BITW*

Operation $temp = src2 \& src1$

Address Modes *src1* all modes
 src2 all modes

Condition Flags *N* = MSB of temp
 Z = 1, if temp == 0
 C = 0
 V = 0

Exceptions None

Examples *BITB %r0,{uhalf}%r1*
 *BITH *\$0xFF,%r3*
 BITW bit (%r3),(%r0)

Notes The final value of temp, a temporary register, determines the setting of the condition codes. Temp is discarded upon completion of the instruction.

BLB

BLB

BLH

BLH

BRANCH ON LESS THAN (SIGNED)

Assembler Syntax	BLB <i>disp8</i> BLH <i>disp16</i>	Branch on less than, byte displacement (signed) Branch on less than, halfword displacement (signed)
Opcodes	0x4B BLB 0x4A BLH	
Operation	if ((N == 1) & (Z == 0)) PC = PC + SEXT(<i>disp</i>)	
Address Modes	None valid	
Condition Flags	Unchanged	
Exceptions	None	
Examples	BLB 0x1F BLH back	
Notes	<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.	

BLEB
BLEH

BLEB
BLEH

BRANCH ON LESS THAN OR EQUAL (SIGNED)

Assembler Syntax	BLEB <i>disp8</i>	Branch on less than or equal, byte displacement (signed)
	BLEH <i>disp16</i>	Branch on less than or equal, halfword displacement (signed)
Opcodes	0x4F BLEB 0x4E BLEH	
Operation	if ((N Z) == 1) PC = PC + SEXT(<i>disp</i>)	
Address Modes	None valid	
Condition Flags	Unchanged	
Exceptions	None	
Examples	BLEB 0x6 BLEH 0xFFFF	
Notes	<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.	

BLEUB
BLEUH

BLEUB
BLEUH

BRANCH ON LESS THAN OR EQUAL (UNSIGNED)

Assembler Syntax	BLEUB <i>disp8</i> BLEUH <i>disp16</i>	Branch on less than or equal, byte displacement (unsigned) Branch on less than or equal, halfword displacement (unsigned)
Opcodes	0x5F BLEUB 0x5E BLEUH	
Operation	if ((C Z) == 1) PC = PC + SEXT(<i>disp</i>)	
Address Modes	None valid	
Condition Flags	Unchanged	
Exceptions	None	
Examples	BLEUB 0x14 BLEUH back	
Notes	<i>disp8</i> is an unsigned 8-bit value. <i>disp16</i> is an unsigned 16-bit value.	

BLUB
BLUH

BLUB
BLUH

BRANCH ON LESS THAN (UNSIGNED)

Assembler Syntax	BLUB <i>disp8</i> BLUH <i>disp16</i>	Branch on less than byte displacement (unsigned) Branch on less than halfword displacement (unsigned)
Opcodes	0x5B BLUB 0x5A BLUH	
Operation	if (C == 1) PC = PC + SEXT(<i>disp</i>)	
Address Modes	None valid	
Condition Flags	Unchanged	
Exceptions	None	
Examples	BLUB 0x12 BLUH 0xFF12	
Notes	<i>disp8</i> is an unsigned 8-bit value. <i>disp16</i> is an unsigned 16-bit value.	

BNEB
BNEH

BNEB
BNEH

BRANCH ON NOT EQUAL

Assembler Syntax **BNEB** *disp8* Branch on less than, byte displacement
 BNEH *disp16* Branch on less than, halfword displacement

Opcodes 0x77 **BNEB**
 0x67 **BNEB**
 0x76 **BNEH**
 0x66 **BNEH**

Operation if ($Z == 0$)
 $PC = PC + \text{SEXT}(disp)$

Address Modes None valid

Condition Flags Unchanged

Exceptions None

Examples **BNEB** 0xFE
 BNEH 0xFF13

Notes *disp8* is a signed 8-bit value.
 disp16 is a signed 16-bit value.

BPT

BPT

BREAKPOINT TRAP

Assembler Syntax	BPT Breakpoint trap
Opcodes	0x2E BPT
Operation	/*BPT executes the following processor operation*/ {breakpoint trap}
Address Modes	None
Condition Flags	Unchanged
Exceptions	Generates breakpoint trap exception.
Example	BPT

BRB
BRH

BRB
BRH

BRANCH

Assembler Syntax BRB *disp8* Branch with byte displacement
 BRH *disp16* Branch with halfword displacement

Opcodes 0x7B BRB
 0x7A BRH

Operation PC = PC + SEXT(*disp*)

Address Modes None valid

Condition Flags Unchanged

Exceptions None

Examples BRB 0xA
 BRH 0xFAA

Notes *disp8* is a signed 8-bit value.
 disp16 is a signed 16-bit value.

BSBB
BSBH

BSBB
BSBH

BRANCH TO SUBROUTINE

Assembler Syntax **BSBB** *disp8* Branch to subroutine, byte displacement
 BSBH *disp16* Branch to subroutine, halfword displacement

Opcodes 0x37 **BSBB**
 0x36 **BSBH**

Operation $*(SP++) = \text{address of next instruction}$
 $PC = PC + \text{SEXT}(disp)$

Address Modes None valid

Condition Flags Unchanged

Exceptions None

Examples **BSBB** sub2
 BSBH sub1

Notes *disp8* is a signed 8-bit value.
 disp16 is a signed 16-bit value.

BVCB
BVCH

BVCB
BVCH

BRANCH ON OVERFLOW CLEAR

Assembler Syntax **BVCB** *disp8* Branch to subroutine, byte displacement
 BVCH *disp16* Branch to subroutine, halfword displacement

Opcodes 0x63 **BVCB**
 0x62 **BVCH**

Operation if (V == 0)
 PC = PC + SEXT(*disp*)

Address Modes None valid

Condition Flags Unchanged

Exceptions None

Examples **BVCB** 0x7E
 BVCH 0x8F21

Notes *disp8* is a signed 8-bit value.
 disp16 is a signed 16-bit value.

BVSB
BVSH

BVSB
BVSH

BRANCH ON OVERFLOW SET

Assembler Syntax	BVSB <i>disp8</i> Branch on overflow set, byte displacement BVSH <i>disp16</i> Branch on overflow set, halfword displacement
Opcodes	0x6B BVSB 0x6A BVSH
Operation	if (V == 1) PC = PC + SEXT(<i>disp</i>)
Address Modes	None valid
Condition Flags	Unchanged
Exceptions	None
Examples	BVS 0xF1 BVSB 0xFF77
Notes	<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.

CALL

CALL

CALL PROCEDURE

Assembler Syntax	CALL <i>src,dst</i> Call procedure
Opcode	0x2C CALL
Operation	tempa = & <i>src</i> tempb = & <i>dst</i> *(SP+4) = AP *SP = address of next instruction SP = SP+8 PC = tempb AP = tempa
Address Modes	<i>src</i> all modes except literal, register, or immediate <i>dst</i> all modes except literal, register, or immediate
Condition Flags	Unchanged
Exceptions	Illegal operand exception occurs if literal, register, expanded-operand, or immediate modes are used for <i>src</i> or <i>dst</i> .
Example	CALL -(3*4)(%sp),func1 (see Figure 5-20)
Notes	Both operands are effective addresses. Temp is a temporary register. CALL sets up the protocol for a C language function call. (Also see return from procedure.) CALL sets AP to the first of the word arguments that the calling function pushed on the stack before executing the call.

COMPARE AND SWAP WORD INTERLOCK

Assembler Syntax	CASWI <i>src1,src2,dst</i> Compare and swap word interlock
Opcodes	0x09 CASWI
Operation	If ($dst - src2 == 0$) then ($dst = src1$) else ($src2 = dst$)
Address Modes	<i>src1</i> register mode <i>src2</i> register mode <i>dst</i> all modes except literal, immediate, register, or auto
Condition Flags	The flags of the PSW are set according to the result of $dst - src2$ N = 1, if $(dst - src2) < 0$ Z = 1, if $(dst - src2) == 0$ C = 1, if carry out of sign bit of $(dst - src2)$ V = 1, if overflow
Exceptions	Invalid descriptor exception occurs if <i>src1</i> and <i>src2</i> are not register operands, or: <ul style="list-style-type: none"> • if <i>dst</i> is not a memory operand • if expand mode is used for any of the operands • if auto increment/decrement mode is used for any of the operands.
Example	CASWI %r1,%r2,\$0x15000
Note:	When the CPU issues the read for <i>dst</i> , it issues a status indicator that the read should be interlocked. The interlocked status indicator remains unchanged until the <i>dst</i> or <i>src1</i> is updated.

CFLUSH

CFLUSH

CACHE FLUSH

Assembler Syntax	CFLUSH Cache flush
Opcode	0x27 CFLUSH
Operation	/*CFLUSH executes the following processor operation*/ {all entries in instruction cache are marked invalid}
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	CFLUSH
Notes	CFLUSH is a nonprivileged instruction. This instruction operates identically if the instruction cache is enabled (PSW<CD>==0) or disabled (PSW<CD>==1).

CLRB
CLRH
CLRW

CLRB
CLRH
CLRW

CLEAR

Assembler Syntax	CLRB <i>dst</i> Clear byte CLRH <i>dst</i> Clear halfword CLRW <i>dst</i> Clear word
Opcodes	0x83 CLRB 0x82 CLRH 0x80 CLRW
Operation	<i>dst</i> = 0
Address Modes	<i>dst</i> all modes except literal or immediate
Condition Flags	N = 0 Z = 1 C = 0 V = 0
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .
Examples	CLRB *&0x300 CLRH %r1 CLRW (%r0)

CLR X

CLR X

CLEAR X BIT

Assembler Syntax CLR X Clear X bit of PSW

Opcodes 0x0B CLR X

Operation PSW<X> = 0

Address Modes None valid

Condition Flags X flag cleared, others unchanged

Exceptions None

Example CLR X

Notes This instruction clears the X bit, which is the carry/borrow bit for BCD add/subtract operations.

CMPB
CMPH
CMPW

CMPB
CMPH
CMPW

COMPARE

Assembler Syntax	CMPB <i>src1,src2</i> Compare byte CMPH <i>src1,src2</i> Compare halfword CMPW <i>src1,src2</i> Compare word
Opcodes	0x3F CMPB 0x3E CMPH 0x3C CMPW
Operation	$temp = src2 - src1$
Address Modes	<i>src1</i> all modes <i>src2</i> all modes
Condition Flags	N = 1, if <i>src2</i> < <i>src1</i> (signed) Z = 1, if <i>src2</i> == <i>src1</i> C = 1, if <i>src2</i> < <i>src1</i> (unsigned) V = 0
Exceptions	None
Examples	CMPB &10,%r0 CMPH (%r0),(%r1) CMPW *\$0x12F7,%r2
Notes	This instruction sets the condition flags N, Z, and C as if a subtract had been executed. Neither operand is altered. (Also see Test.)

DECB
DECH
DECW

DECB
DECH
DECW

DECREMENT

Assembler Syntax	DECB <i>dst</i> Decrement byte DECH <i>dst</i> Decrement halfword DECW <i>dst</i> Decrement word
Opcodes	0x97 DECB 0x96 DECH 0x94 DECW
Operation	$dst = dst - 1$
Address Modes	<i>dst</i> all modes except literal or immediate
Condition Flags	N = 1, if $(dst - 1) < 0$ Z = 1, if $(dst - 1) == 0$ C = 1, if borrow into sign bit of <i>dst</i> V = 1, if overflow
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.
Examples	DECB 4(%fp) DECH \$result DECW *\$last

DIVB2
DIVH2
DIVW2

DIVB2
DIVH2
DIVW2

DIVIDE

Assembler Syntax	DIVB2 <i>src,dst</i> Divide byte DIVH2 <i>src,dst</i> Divide halfword DIVW2 <i>src,dst</i> Divide word
Opcodes	0xAF DIVB2 0xAE DIVH2 0xAC DIVW2
Operation	$dst = dst / src$
Address Modes	<i>src</i> all modes <i>dst</i> all modes except literal or immediate
Condition Flags	N = 1, if $(dst / src) < 0$ Z = 1, if $(dst / src) == 0$ C = 0 V = 1, if overflow
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer zero-divide exception occurs if <i>src</i> is equal to 0. Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.
Examples	DIVB2 &40,%r6 DIVH2 4(%r3),(%r4) DIVW2 \$first,\$last

DIVB3
DIVH3
DIVW3

DIVB3
DIVH3
DIVW3

DIVIDE, 3 ADDRESS

Assembler Syntax	DIVB3 <i>src1,src2,dst</i> DIVH3 <i>src1,src2,dst</i> DIVW3 <i>src1,src2,dst</i>	Divide byte, 3 address Divide halfword, 3 address Divide word, 3 address
Opcodes	0xEF DIVB3 0xEE DIVH3 0xEC DIVW3	
Operation	$dst = src2 / src1$	
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = 1, if $(src2 / src1) < 0$ Z = 1, if $(src2 / src1) == 0$ C = 0 V = 1, if overflow	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer zero-divide exception occurs if <i>src1</i> is equal to 0. Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.	
Examples	DIVB3 &0x30,%r3,12(%ap) DIVH3 &0x3030,(%r2),5(%r2) DIVW3 &0x304050,(%r1),4(%r1)	

DTB
DTH

DTB
DTH

DECREMENT AND TEST

Assembler Syntax	DTB <i>dst,disp8</i> DTH <i>dst,disp16</i>	Decrement, test, and branch byte Decrement, test, and halfword
Opcodes	0x29 DTB 0x19 DTH	
Operation	$dst = dst - 1$ IF ($dst > -1$) { PC = PC + SEXT(<i>disp</i>) }	
Address Modes	<i>dst</i>	all modes except expanded-operand
Condition Flags	Unchanged	
Exceptions		Invalid operand descriptor exception occurs if expanded-operand mode is used.
Examples	DTB %r1, 0x12 DTH %r1,0x1234	
Notes		<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.

EXTFB
EXTFH
EXTFW

EXTFB
EXTFH
EXTFW

EXTRACT FIELD

Assembler Syntax	EXTFB <i>width,offset,src,dst</i> EXTFH <i>width,offset,src,dst</i> EXTFW <i>width,offset,src,dst</i>	Extract field from byte Extract field from halfword Extract field from word				
Opcodes	0xCF EXTFB 0xCE EXTFH 0xCC EXTFW					
Operation	$dst = \text{FIELD}(offset,width,src)$					
Address Modes	<i>width</i> all modes <i>offset</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate					
Condition Flags	N = high-order bit of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 0 (see Notes)					
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .					
Example	Before: Location L1 = 0x01234567 EXTFW &10,&4,L1,%r0 After: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td style="padding: 2px 10px;">00</td><td style="padding: 2px 10px;">00</td><td style="padding: 2px 10px;">04</td><td style="padding: 2px 10px;">56</td></tr></table> ← Increasing Bits		00	00	04	56
00	00	04	56			
Notes	Only the low-order five bits of <i>width</i> and <i>offset</i> are examined. If the sum <i>width</i> plus <i>offset</i> is greater than 32 (bits), the field wraps around through bit 0 of the base word. The field specified by <i>width</i> , <i>offset</i> , and <i>src</i> is stored, right-adjusted, in <i>dst</i> . The remaining bits of <i>dst</i> are set to 0. If the field is too large for the size of <i>dst</i> , the excess high-order bits are discarded and the V flag is set.					

EXTOP

EXTOP

EXTENDED OPCODE

Assembler Syntax	EXTOP <i>byte</i> Extended opcode
Opcode	0x14 EXTOP
Operation	/*EXTOP executes the following processor operation*/ {reserved-opcode exception}
Address Modes	None valid
Condition Flags	Unchanged
Exceptions	Generates reserved opcode exception. See Notes.
Example	EXTOP 0x2F
Notes	The EXTOP opcode is an escape in order to form additional instructions. The processor does not access <i>byte</i> when executing this instruction. Instead, it generates a reserved-opcode exception after decoding the opcode. The operating system's exception handler should access <i>byte</i> . <i>byte</i> is an 8-bit value.

INCB
INCH
INCW

INCB
INCH
INCW

INCREMENT

Assembler Syntax	INCB <i>dst</i> Increment byte INCH <i>dst</i> Increment halfword INCW <i>dst</i> Increment word
Opcodes	0x93 INCB 0x92 INCH 0x90 INCW
Operation	$dst = dst + 1$
Address Modes	<i>dst</i> all modes except literal or immediate
Condition Flags	N = 1, if $(dst + 1) < 0$ Z = 1, if $(dst + 1) == 0$ C = 1, if carry into sign bit of <i>dst</i> V = 1, if overflow
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer overflow exception occurs if PSW<OE> = 1 and a truncation takes place.
Examples	INCB 4(%r2) INCH %r0 INCW (%r1)

INSFB
INSFH
INSFW

INSFB
INSFH
INSFW

INSERT FIELD

Assembler Syntax	INSFB <i>width,offset,src,dst</i>	Insert field from byte
	INSFH <i>width,offset,src,dst</i>	Insert field from halfword
	INSFW <i>width,offset,src,dst</i>	Insert field from word
Opcodes	0xCB INSFB	
	0xCA INSFH	
	0xC8 INSFW	
Operation	FIELD(<i>offset,width,dst</i>) = <i>src</i>	
Address Modes	<i>width</i> all modes	
	<i>offset</i> all modes	
	<i>src</i> all modes	
	<i>dst</i> all modes except literal or immediate	
Condition Flags	N = bit 31 of <i>dst</i>	
	Z = 1, if <i>dst</i> == 0	
	C = 0	
	V = 0 (see Notes)	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .	

Example

Before: r0

AB	CD	EF	01
----	----	----	----

r1

00	00	05	67
----	----	----	----

← Increasing Bits

INSFW &11,&8,%r1,%r0

After: r0

AB	C5	67	01
----	----	----	----

The field insertion starts at bit 8 of r0, skipping bits 0 through 7, and extends through bit 19. Therefore, bits 8 through 19 of r0 now contain the same value as bits 0 through 11 of r1.

INSFB
INSFH
INSFW

INSFB
INSFH
INSFW

INSERT FIELD

Notes Only the low-order five bits of *width* and *offset* are examined. If the sum *width* plus *offset* is greater than 32 (bits), the field wraps around to bit 0 of the destination. Starting with bit 0 of *src*, (*width*+1) bits are placed into *dst*, beginning at the bit designated by *offset*. If *dst* is a byte or halfword and (*width*+*offset*) specifies a field that extends beyond *dst*, no bits beyond *dst* are altered but the V flag is set.

JMP

JMP

JUMP

Assembler Syntax	JMP <i>dst</i> Jump
Opcode	0x24 JMP
Operation	PC = & <i>dst</i>
Address Modes	<i>dst</i> all modes except literal, register, or immediate
Condition Flags	Unchanged
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .
Example	JMP label
Notes	The operand <i>dst</i> is an effective address; i.e., the 32-bit address of <i>dst</i> is used as the destination rather than the word stored at that address.

JUMP TO SUBROUTINE

Assembler Syntax	JSB <i>dst</i> Jump to subroutine
Opcode	0x34 JSB
Operation	*(SP++) = address of next instruction PC = & <i>dst</i>
Address Modes	<i>dst</i> all modes except literal, register, or immediate
Condition Flags	Unchanged
Exceptions	Illegal operand exception occurs if literal, expanded-operand, or immediate modes are used for <i>dst</i> .
Example	JSB error
Notes	The operand <i>dst</i> is an effective address; i.e., the 32-bit address of <i>dst</i> is used as the destination rather than the word at that address.

LLSB3
LLSH3
LLSW3

LLSB3
LLSH3
LLSW3

LOGICAL LEFT SHIFT

Assembler Syntax LLSB3 *count,src,dst* Logical left shift byte
 LLSH3 *count,src,dst* Logical left shift halfword
 LLSW3 *count,src,dst* Logical left shift word

Opcodes 0xD3 LLSB3
 0xD2 LLSH3
 0xD0 LLSW3

Operation $dst = src \ll (count \& 0x1F)$ bits

Address Modes *count* all modes
 src all modes
 dst all modes except literal or immediate

Condition Flags N = MSB of *dst*
 Z = 1, if *dst* == 0
 C = 0
 V = 1, if result must be truncated to fit *dst* size

Exceptions Illegal operand exception occurs if literal or immediate modes are used for *dst*.

Example Before: r0

0F	0F	DF	FD
----	----	----	----

← Increasing Bits

LLSH3 &2,%r0,%r0

After: r0

FF	FF	7F	F4
----	----	----	----

Notes Only the five low-order bits of *count* are used; the high-order bits are ignored. Zeros replace the bits shifted out of the low-order bit position (bit 0).

LOGICAL RIGHT SHIFT

Assembler Syntax	LRSW3 <i>count,src,dst</i> Logical right shift word								
Opcode	0xD4 LRSW3								
Operation	$dst = src \gg (count \& 0x1F)$ bits								
Address Modes	<i>count</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate								
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if result must be truncated to fit <i>dst</i> size								
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .								
Example	<p>Before: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>C3</td><td>C0</td><td>00</td><td>00</td></tr></table></p> <p style="text-align: center;">← Increasing Bits</p> <p>LRSW3 &0x11,%r0,%r0</p> <p>After: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>00</td><td>00</td><td>61</td><td>E0</td></tr></table></p>	C3	C0	00	00	00	00	61	E0
C3	C0	00	00						
00	00	61	E0						
Notes	All operands are word type. However, only the five low-order bits of <i>count</i> are used; the high-order bits are ignored. Zeros replace the bits shifted out of the high-order bit position (bit 31).								

MCOMB
MCOMH
MCOMW

MCOMB
MCOMH
MCOMW

MOVE COMPLEMENTED

Assembler Syntax	MCOMB <i>src,dst</i> MCOMH <i>src,dst</i> MCOMW <i>src,dst</i>	Move complemented byte Move complemented halfword Move complemented word								
Opcodes	0x8B MCOMB 0x8A MCOMH 0x88 MCOMW									
Operation	$dst = \sim src$									
Address Modes	<i>src</i> all modes <i>dst</i> all modes except literal or immediate									
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if result must be truncated to fit <i>dst</i> size									
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .									
Example	Before: r0 <table border="1"><tr><td>12</td><td>34</td><td>56</td><td>78</td></tr></table> ← Increasing Bits MCOMW %r0,%r1 After: r1 <table border="1"><tr><td>ED</td><td>CB</td><td>A9</td><td>87</td></tr></table>	12	34	56	78	ED	CB	A9	87	
12	34	56	78							
ED	CB	A9	87							
Notes	<i>dst</i> is the one's complement of <i>src</i> .									

MNEGB
MNEGH
MNEGW

MNEGB
MNEGH
MNEGW

MOVE NEGATED

Assembler Syntax	MNEGB <i>src, dst</i> MNEGH <i>src, dst</i> MNEGW <i>src, dst</i>	Move negated byte Move negated halfword Move negated word				
Opcodes	0x8F MNEGB 0x8E MNEGH 0x8C MNEGW					
Operation	$dst = -src$					
Address Modes	<i>src</i> all modes <i>dst</i> all modes except literal or immediate					
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if integer overflow					
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .					
Example	Before: r0	<table border="1"><tr><td>01</td><td>23</td><td>45</td><td>67</td></tr></table> ← Increasing Bits MNEGB %r0,%r1 After: r1	01	23	45	67
01	23	45	67			
Notes	<i>dst</i> is the two's complement of <i>src</i> .					

MODB2
MODH2
MODW2

MODB2
MODH2
MODW2

MODULO

Assembler Syntax

MODB2 *src,dst* Modulo byte
MODH2 *src,dst* Modulo halfword
MODW2 *src,dst* Modulo word

Opcodes

0xA7 MODB2
0xA6 MODH2
0xA4 MODW2

Operation

$dst = dst \% src$

Address Modes

src all modes
dst all modes except literal or immediate

Condition Flags

N = 1, if $(dst \% src) < 0$
Z = 1, if $(dst \% src) == 0$
C = 0
V = 1, if overflow

Exceptions

Illegal operand exception occurs if literal or immediate modes are used for *dst*.
Integer zero-divide exception occurs if *src* is equal to 0.
Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.

Examples

MODB2 &40,%r3
MODH2 4(%r3),%r3
MODW2 %r0,*\$result

MODB3
MODH3
MODW3

MODB3
MODH3
MODW3

MODULO, 3 ADDRESS

Assembler Syntax	MODB3 <i>src1,src2,dst</i> MODH3 <i>src1,src2,dst</i> MODW3 <i>src1,src2,dst</i>	Modulo byte, 3 address Modulo halfword, 3 address Modulo word, 3 address
Opcodes	0xE7 MODB3 0xE6 MODH3 0xE4 MODW3	
Operation	$dst = src2 \% src1$	
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = 1, if $(src2 \% src1) < 0$ Z = 1, if $(src2 \% src1) == 0$ C = 0 V = 1, if overflow	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer zero-divide exception occurs if <i>src1</i> is equal to 0. Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.	
Examples	MODB3 &40,%r3,0x1101(%r2) MODH3 %r3,\$real,%r3 MODW3 4(%r2),*\$0x34,%r0	

MOVAW

MOVAW

MOVE ADDRESS (WORD)

Assembler Syntax	<code>MOVAW <i>src, dst</i></code> Move address (word)												
Opcode	0x04 MOVAW												
Operation	$dst = \&src$												
Address Modes	<i>src</i> all modes except literal, register, or immediate <i>dst</i> all modes except literal or immediate												
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 0												
Exceptions	Illegal operand exception occurs if literal, register, or immediate modes are used for <i>src</i> , or if literal or immediate modes are used for <i>dst</i> .												
Example	Before: r0 <table border="1"><tr><td>00</td><td>00</td><td>10</td><td>10</td></tr></table> r1 <table border="1"><tr><td>AB</td><td>AB</td><td>AB</td><td>AB</td></tr></table> ← Increasing Bits MOVAW 4(%r0),%r1 After: r1 <table border="1"><tr><td>00</td><td>00</td><td>10</td><td>14</td></tr></table>	00	00	10	10	AB	AB	AB	AB	00	00	10	14
00	00	10	10										
AB	AB	AB	AB										
00	00	10	14										
Notes	Source operand type is effective address.												

MOVB
MOVH
MOVW

MOVB
MOVH
MOVW

MOVE

**Assembler
 Syntax**

MOVB *src,dst* Move byte
 MOVH *src,dst* Move halfword
 MOVW *src,dst* Move word

Opcodes

0x87 MOVB
 0x86 MOVH
 0x84 MOVW

Operation

dst = *src*

**Address
 Modes**

src all modes
dst all modes except literal or immediate

**Condition
 Flags**

N = MSB of *dst*
 Z = 1, if *dst* == 0
 C = 0
 V = 1, if result must be truncated to fit *dst* size
 See Notes

Exceptions

Illegal operand exception occurs if literal or immediate modes are used for *dst*.

Example

Before: r0

01	23	45	67
----	----	----	----

 r1

AB	AB	AB	AB
----	----	----	----

 ← Increasing Bits

MOVW %r0,%r1

After: r0

01	23	45	67
----	----	----	----

 r1

01	23	45	67
----	----	----	----

 NZCV = 0000

MOVB
MOVH
MOVW

MOVB
MOVH
MOVW

Notes

If the expanded-type mode is used for *dst* or for both operands, this instruction can convert data from one type to another. The *src* operand determines the type of extension performed: if *src* is signed byte or halfword, sign extension occurs; if *src* is byte or unsigned halfword, zero extension occurs.

Use the following instructions for conversions if the destination is not a register:

Instruction	Conversion
<code>MOVB {sbyte}src,{shalf}dst</code>	Signed byte to signed halfword
<code>MOVB {sbyte}src,{sword}dst</code>	Signed byte to signed word
<code>MOVH src,{sword}dst</code>	Byte to signed word
<code>MOVB src,{shalf}dst</code>	Byte to signed halfword
<code>MOVB src,{sword}dst</code>	Byte to signed word
<code>MOVH {uhalf}src,{sword}dst</code>	Unsigned halfword to signed word
<code>MOVH src,{sbyte}dst</code>	Halfword to signed byte
<code>MOVW src,{sbyte}dst</code>	Word to signed byte
<code>MOVW src,{shalf}dst</code>	Word to signed halfword

If the destination is a register, use the following instructions for conversions:

Instruction	Conversion
<code>ANDH3 &0xff,src,{byte}dst</code>	Halfword to byte
<code>ANDW3 &0xff,src,{byte}dst</code>	Word to byte
<code>MOVW src,dst; MOVH dst,dst</code>	Word to halfword

The instructions "MOVW —,%psw" and "MOVW %psw,—" do not change the condition flags.

MOVBLW

MOVBLW

MOVE BLOCK

Assembler Syntax MOVBLW Move block of words

Opcode 0x3019 MOVBLW

Operation while (R2 > 0) {
 *R1 = *R0;
 {disable interrupts}
 --R2;
 R0=R0+4;
 R1=R1+4;
 {enable interrupts}
 }

Address Modes None

Condition Flags Unchanged

Exceptions External memory fault may occur in the middle of an iteration.

Example Before: r0

00	00	01	00
----	----	----	----

 r1

00	00	02	00
----	----	----	----

 r2

00	00	00	03
----	----	----	----

← Increasing Bits

Assume three word locations starting at 0x100 contain the word values 0x5, 0x10 and 0x20, respectively.

MOVBLW

MOVBLW

MOVBLW

After: r0

00	00	01	0C
----	----	----	----

r1

00	00	02	0C
----	----	----	----

r2

00	00	00	00
----	----	----	----

Three word locations starting at 0x200 now also contain 0x5, 0x10 and 0x20, respectively.

Notes

Opcode occupies 16 bits. All operands are implicitly defined in the registers (r0, r1, and r2) and are 32-bit words. These registers must be preset with the following information before executing MOVBLW:

- r0 Address of source
- r1 Address of destination
- r2 Number of words to be moved

The instruction may be interrupted *only* at the end of an iteration. A memory fault may occur in the middle of an iteration. To restart the instruction after a fault, execute MOVBLW again; the registers are updated after the only memory access that could cause the fault. At each iteration, r0 and r1 are incremented by 4, and r2 is decremented by 1. Execution of MOVBLW is finished when r2 is 0.

MULB2
MULH2
MULW2

MULB2
MULH2
MULW2

MULTIPLY

Assembler Syntax **MULB2** *src, dst* Multiply byte
 MULH2 *src, dst* Multiply halfword
 MULW2 *src, dst* Multiply word

Opcodes 0xAB **MULB2**
 0xAA **MULH2**
 0xA8 **MULW2**

Operation $dst = dst * src$

Address Modes *src* all modes
 dst all modes except literal or immediate

Condition Flags $N = 1$, if $(dst * src) < 0$
 $Z = 1$, if $(dst * src) == 0$
 $C = 0$
 $V = 1$, if overflow

Exceptions Illegal operand exception occurs if literal or immediate modes are used for *dst*.

Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.

Example **MULB2** %r2, {sbyte}4(%r6)

MULB3
MULH3
MULW3

MULB3
MULH3
MULW3

MULTIPLY, 3 ADDRESS

Assembler Syntax	MULB3 <i>src1,src2,dst</i> MULH3 <i>src1,src2,dst</i> MULW3 <i>src1,src2,dst</i>	Multiply byte, 3 address Multiply halfword, 3 address Multiply word, 3 address
Opcodes	0xEB MULB3 0xEA MULH3 0xE8 MULW3	
Operation	$dst = src1 * src2$	
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = 1, if (<i>src1</i> * <i>src2</i>) < 0 Z = 1, if (<i>src1</i> * <i>src2</i>) == 0 C = 0 V = 1, if overflow	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.	
Example	MULH3 %r3,\$0x1004,%r4	

MVERNO

MVERNO

MOVE VERSION NUMBER

Assembler Syntax	MVERNO Move processor version number
Opcode	0x3009 MVERNO
Operation	r0 = processor version number {instruction fetch after PC discontinuity at the address formed by PC + 2}
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	MVERNO
Notes	Opcode occupies 16 bits. Version number is the version of the processor and may range from -128 to +127.

NOP
NOP2
NOP3

NOP
NOP2
NOP3

NO OPERATION

Assembler Syntax	NOP	No operation, 1 byte
	NOP2	No operation, 2 bytes
	NOP3	No operation, 3 bytes

Opcodes	0x70	NOP
	0x73	NOP2
	0x72	NOP3

Operation	None
------------------	------

Address Modes	None
----------------------	------

Condition Flags	Unchanged
------------------------	-----------

Exceptions	None
-------------------	------

Examples	NOP
	NOP2
	NOP3

Notes	The assembler inserts a NOP before instructions (other than branch) that read the PSW. This NOP allows the condition bits to stabilize. The bytes following NOP2 and NOP3 are generated by the assembler and are ignored by the processor. They may be of any value.
--------------	--

ORB2
ORH2
ORW2

ORB2
ORH2
ORW2

OR

Assembler Syntax ORB2 *src,dst* OR byte
 ORH2 *src,dst* OR halfword
 ORW2 *src,dst* OR word

Opcodes 0xB3 ORB2
 0xB2 ORH2
 0xB0 ORW2

Operation $dst = dst | src$

Address Modes *src* all modes
 dst all modes except literal or immediate

Condition Flags N = MSB of *dst*
 Z = 1, if *dst* == 0
 C = 0
 V = 1, if result must be truncated to fit *dst* size

Exceptions Illegal operand exception occurs if literal or immediate modes are used for *dst*.

Examples ORB2 &12,4(%fp)
 ORH2 %r0,4(%r0)
 ORW2 %r3,\$result

ORB3
ORH3
ORW3

ORB3
ORH3
ORW3

OR, 3 ADDRESS

Assembler Syntax	ORB3 <i>src1,src2,dst</i> OR byte, 3 address ORH3 <i>src1,src2,dst</i> OR halfword, 3 address ORW3 <i>src1,src2,dst</i> OR word, 3 address
Opcodes	0xF3 ORB3 0xF2 ORH3 0xF0 ORW3
Operation	$dst = src2 src1$
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if result must be truncated to fit <i>dst</i> size
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .
Examples	ORB3 &16,*\$0x304,%r0 ORH3 %r1,4(%r1),%r1 ORW3 %r2,%r3,%r1

PACKB

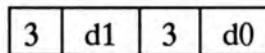
PACKB

PACK BCD HALFWORD

Assembler Syntax	PACKB <i>src, dst</i> Halfword → packed byte
Opcodes	0x0E PACKB
Operation	(packed) <i>dst</i> = (unpacked) <i>src</i>
Address Modes	<i>src</i> all modes except expanded-operand <i>dst</i> all modes except expanded-operand, literal, and immediate
Condition Flags	Unchanged
Exceptions	Invalid descriptor exception occurs if an expanded-operand descriptor is used.
Example	The lower nibble of each byte is packed and stored in <i>dst</i> . A halfword is packed into a byte. Consider the following example in ASCII, where 3 is used in the upper nibble:

Before

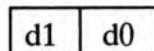
The halfword at location *src* is:



PACKB *src, dst*

After

The result, stored in location *dst* is:



POPW

POPW

POP (WORD)

Assembler Syntax	POPW <i>dst</i> Pop (word)
Opcode	0x20 POPW
Operation	$dst = *(--SP)$
Address Modes	<i>dst</i> all modes except literal or immediate (see Notes)
Condition Flags	N = MSB of <i>dst</i> Z = 1, if $dst == 0$ C = 0 V = 0
Exceptions	Illegal operand exception occurs if literal, expanded-operand or immediate modes are used for <i>dst</i> .
Example	POPW (%r2)
Notes	If <i>dst</i> is the stack pointer (%sp), the results are indeterminate.

PUSHAW

PUSHAW

PUSH ADDRESS (WORD)

Assembler Syntax `PUSHAW src` Push address (word)

Opcode `0xE0 PUSHAW`

Operation `*(SP++) = &src`

Address Modes *src* all modes except literal, register, or immediate

Condition Flags `N = MSB of address of src`
`Z = 1, if src == 0`
`C = 0`
`V = 0`

Exceptions Illegal operand exception occurs if literal, register, expanded-operand, or immediate modes are used for *src*.

Example `PUSHAW 0x14(%r6)`

Notes Source operand is effective address. This instruction is the same as a move address (MOVAW) instruction, except that the destination for PUSHAW is an implied stack push.

PUSHW

PUSHW

PUSH (WORD)

Assembler Syntax **PUSHW** *src* Push (word)

Opcode 0xA0 **PUSHW**

Operation $*(SP++) = src$

Address Modes *src* all modes

Condition Flags N = MSB of *src*
 Z = 1, if *src* == 0
 C = 0
 V = 0

Exceptions Illegal operand exception occurs if expanded-operand addressing mode is used.

Example **PUSHW** (%r2)

RCC

RCC

RETURN ON CARRY CLEAR

Assembler Syntax	RCC	Return on carry clear
Opcode	0x50	RCC
Operation	if (C==0) PC = *(--SP)	
Address Modes	None	
Condition Flags	Unchanged	
Exceptions	None	
Example	RCC	

RCS

RCS

RETURN ON CARRY SET

Assembler Syntax	RCS	Return on carry set
Opcode	0x58	RCS
Operation	if (C==1) PC = *(--SP)	
Address Modes	None	
Condition Flags	Unchanged	
Exceptions	None	
Example	RCS	

REQL
REQLU

REQL
REQLU

RETURN ON EQUAL

Assembler Syntax	REQL Return on equal (signed) REQLU Return on equal (unsigned)
Opcodes	0x7C REQL 0x6C REQLU
Operation	if (Z==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	REQL

RESTORE

RESTORE

RESTORE REGISTERS

Assembler Syntax `RESTORE %rn` Restore registers

Opcode `0x18 RESTORE`

Operation `tempa = FP - 28;`
`tempb = *(FP - 28);`
`tempc = FP - 24;`
`while (n != FP){`
 `register[n] = (tempc);`
 `n+=1;`
`}`
`FP = tempb;`
`SP = tempa`

Address Modes Register mode, where *n* ranges from 3 through 9

Condition Flags Unchanged

Exceptions See Notes.

Example `RESTORE %r3`

Notes If the operand is not register mode or *n* is not in the range 3 through 9, the results are indeterminate.

RESTORE is the inverse of SAVE and should precede a return from procedure (RET). (Also see SAVE and CALL.) The operand `%rn` should be the same as in the corresponding SAVE, where *n* specifies the number of registers (9 — *n*) to be restored for the original function.

RESTORE implements a stack frame for use in the C language function-calling sequence. The instruction can restore up to six registers (from register 8 through register 3) for use by the function. While restoring these registers, it also adjusts SP and FP.

Illegal operand exception occurs if expanded-operand address mode is used.

RET

RET

RETURN FROM PROCEDURE

Assembler Syntax RET Return from procedure

Opcode 0x18 RET

Operation tempa = AP;
 tempb = *(SP-4);
 tempc = *(SP-8);
 AP = tempb;
 PC = tempc;
 SP = tempa;

Address Modes None

Condition Flags Unchanged

Exceptions None

Example RET

Note The return from procedure (RET) is the inverse of the call (CALL) instruction. A restore should precede a RET inside the function being exited. RESTORE sets up the protocol for a C language return from function. RET restores AP, PC, and SP to the values saved on the stack with the corresponding CALL.

RGEQ

RGEQ

RETURN ON GREATER THAN OR EQUAL (SIGNED)

Assembler Syntax	RGEQ Return on greater than or equal (signed)
Opcode	0x40 RGEQ
Operation	if ((N==0) (Z==1)) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RGEQ

RGEQU

RGEQU

RETURN ON GREATER THAN OR EQUAL (UNSIGNED)

Assembler Syntax	RGEQU Return on greater than or equal (unsigned)
Opcode	0x50 REGEQU
Operation	if (C==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RGEQU

RGTR

RGTR

RETURN ON GREATER THAN (SIGNED)

Assembler Syntax	RGTR Return on greater than (signed)
Opcode	0x44 RGTR
Operation	if ((N & Z)==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RGTR

RGTRU

RGTRU

RETURN ON GREATER THAN (UNSIGNED)

Assembler Syntax	RGTRU Return on greater than
Opcode	0x54 RGTRU
Operation	if ((C & Z)==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RGTRU

RLEQ

RLEQ

RETURN ON LESS THAN OR EQUAL (SIGNED)

Assembler Syntax	RLEQ Return on less than or equal
Opcode	0x4C RLEQ
Operation	if ((N Z)==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RLEQ

RLEQU

RLEQU

RETURN ON LESS THAN OR EQUAL (UNSIGNED)

Assembler Syntax	RLEQU Return on less than or equal (unsigned)
Opcode	0x5C RLEQU
Operation	if ((C Z)==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RLEQU

RLSS

RLSS

RETURN ON LESS THAN (SIGNED)

Assembler Syntax	RLSS Return on less than (signed)
Opcode	0x48 RLSS
Operation	if ((N==1)&(Z==0)) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RLSS

RLSSU

RLSSU

RETURN ON LESS THAN (UNSIGNED)

Assembler Syntax	RLSSU Return on less than (unsigned)
Opcode	0x58 RLSSU
Operation	if (C==1) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RLSSU

RNEQ
RNEQU

RNEQ
RNEQU

RETURN ON NOT EQUAL

Assembler Syntax	RNEQ Return on not equal (signed) RNEQU Return on not equal (unsigned)
Opcode	0x74 RNEQ 0x64 RNEQU
Operation	if (Z==0) PC = *(--SP)
Address Modes	None
Condition Flags	Unchanged
Exceptions	None
Example	RNEQ

ROTW

ROTW

ROTATE

Assembler Syntax	ROTW <i>count,src,dst</i> Rotate word								
Opcode	0xD8 ROTW								
Operation	$dst = src$ rotated right (<i>count</i> & 0x1F) bits								
Address Modes	<i>count</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate								
Condition Flags	N = MSB of <i>dst</i> Z = 1, if $dst == 0$ C = 0 V = 0								
Exceptions	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .								
Example	Before: r0 <table border="1"><tr><td>0F</td><td>00</td><td>00</td><td>7E</td></tr></table> ← Increasing Bits ROTW &0x404,%r0,%r0 After: r0 <table border="1"><tr><td>E0</td><td>F0</td><td>00</td><td>07</td></tr></table>	0F	00	00	7E	E0	F0	00	07
0F	00	00	7E						
E0	F0	00	07						
Notes	All operands are word type. However, only the five low-order bits of <i>count</i> are used; the high-order bits are ignored.								

RSB

RSB

RETURN FROM SUBROUTINE

Assembler Syntax	RSB	Return from subroutine (unconditional)
Opcode	0x78	RSB
Operation	PC = *(--SP)	
Address Modes	None	
Condition Flags	Unchanged	
Exceptions	None	
Example	RSB	

RVC

RVC

RETURN ON OVERFLOW CLEAR

Assembler Syntax	RVC	Return on overflow clear
Opcode	0x60	RVC
Operation	if (V==0) PC = *(--SP)	
Address Modes	None	
Condition Flags	Unchanged	
Exceptions	None	
Example	RVC	

RVS

RVS

RETURN ON OVERFLOW SET

Assembler Syntax	RVS	Return on overflow set
Opcode	0x68	RVS
Operation	if (V==1) PC = *(--SP)	
Address Modes	None	
Condition Flags	Unchanged	
Exceptions	None	
Example	RVS	

SAVE REGISTERS

Assembler Syntax `SAVE %rn` Save registers

Opcode `0x10 SAVE`

Operation

```
temp = SP
*(SP++) = FP
while (n != FP){
    *(SP++) = register[n]
    n+=1;
}
SP =temp + 28;
FP = SP;
```

Address Modes Register mode, where *n* ranges from 3 through 9

Condition Flags Unchanged

Exceptions See Notes.

Example `SAVE %r3` (see Figure 5-20)

Notes If the operand is not register mode or *n* is not in the range 3 to 9, the results are indeterminate.

Temp is a temporary register, and *n* specifies the number of registers (9 — *n*) to be saved for the calling function.

SAVE implements a stack frame for use in the C language function-calling sequence. It should be the first statement in the called function. (Also see RESTORE and RET instructions.) SAVE can save up to six registers, from register 8 (r8) through register 3 (r3), freeing them for the new function. After saving these registers, SAVE adjusts SP and FP to point beyond the end of a fixed-size register-save area. Figure 5-20 shows the stack after executing `SAVE %r3`.

Illegal operand exception occurs if expanded-operand addressing mode is used.

SETX

SETX

SET X BIT

Assembler Syntax	SETX Set X bit of PSW
Opcode	0x0B SETX
Operation	PSW<X> = 1
Address Modes	none valid
Condition Flags	X flag set, others unchanged
Example	SETX
Notes	This instruction sets the X bit, which is the carry/borrow bit for BCD add/subtract operations.

SPOP

SPOP

COPROCESSOR OPERATION (no operands)

Assembler Syntax	SPOP <i>word</i> Coprocessor operation
Opcode	0x32 SPOP
Operation	<pre>/* coprocessor operation executes the following CPU operations */ { "word" is written out with an access status of "coprocessor broadcast" } { wait for "coprocessor done" } { a word is written into PSW with an access status of "coprocessor status fetch" }</pre>
Address Modes	None valid, word = 32-bit value
Condition Flags	Determined by the coprocessor status.
Exceptions	External memory fault may occur.
Example	SPOP 0XFFFFFFFF

SPOPRS
SPOPRD
SPOPRT

SPOPRS
SPOPRD
SPOPRT

COPROCESSOR OPERATION READ

Assembler Syntax	SPOPRS <i>word,src</i> Coprocessor operation read single
	SPOPRD <i>word,src</i> Coprocessor operation read double
	SPOPPT <i>word,src</i> Coprocessor operation read triple

Opcode	0x22 SPOPRS
	0x02 SPOPRD
	0x06 SPOPRT

Operation	<pre>/* coprocessor operation read executes the following CPU operations */ { "word" is written out with an access status of "coprocessor broadcast" } { "src" is read with an access status of "coprocessor data fetch" } { wait for "coprocessor done" } { a word is written into PSW with an access status of "coprocessor status fetch" }</pre>
------------------	--

Address Modes	<i>word</i> none valid
	<i>src</i> all modes except register, literal, immediate, auto, or index with scaling

Condition Flags	Determined by the coprocessor status.
------------------------	---------------------------------------

Exceptions	External memory fault may occur.
-------------------	----------------------------------

Examples	SPOPRS 0xF379FFFF,*\$0xFF37
	SPOPRD 0xFFFFFFFF,%r3
	SPOPRT 0x00000000,(%r4)

Notes	<i>word</i> is a 32-bit value.
--------------	--------------------------------

SPOPS2
SPOPD2
SPOPT2

SPOPS2
SPOPD2
SPOPT2

COPROCESSOR OPERATION, 2 ADDRESS

Assembler Syntax	SPOPS2 <i>word,src,dst</i>	Coprocessor operation single, 2-address
	SPOPD2 <i>word,src,dst</i>	Coprocessor operation double, 2-address
	SPOPT2 <i>word,src,dst</i>	Coprocessor operation triple, 2-address
Opcode	0x23 SPOPS2 0x03 SPOPD2 0x07 SPOPT2	
Operation	/* coprocessor operation executes the following CPU operations */ { " <i>word</i> " is written out with an access status of "coprocessor broadcast" } { " <i>src</i> " is read with an access status of "coprocessor data fetch" } { wait for "coprocessor done" } { a word is written into PSW with an access status of "coprocessor status fetch" } { " <i>dst</i> " is written with an access status of "coprocessor data write" }	
Address Modes	<i>word</i> none valid <i>src</i> all modes except register, literal, immediate, auto or index with scaling <i>dst</i> all modes except register, literal, immediate, auto or index with scaling	
Condition Flags	Determined by the coprocessor status.	
Exceptions	External memory fault may occur.	

SPOPS2
SPOPD2
SPOPT2

SPOPS2
SPOPD2
SPOPT2

Examples SPOPS2 0xFF,4(%r0)
 SPOPD2 0xFFF,%r3
 SPOPT2 0xFE,(%r0)

Notes *word* is a 32-bit value.

SPOPWS
SPOPWD
SPOPWT

SPOPWS
SPOPWD
SPOPWT

COPROCESSOR OPERATION WRITE

Assembler Syntax SPOPWS word,dst Coprocessor operation write single
 SPOPWD word,dst Coprocessor operation write double
 SPOPWT word,dst Coprocessor operation write triple

Opcode 0x33 SPOPWS
 0x13 SPOPWD
 0x17 SPOPWT

Operation /* coprocessor operation write executes the following
 CPU operations */
 { "word" is written out with an access status of
 "coprocessor broadcast" }
 { wait for "coprocessor done" }
 { a word is written into PSW with an access status of
 "coprocessor status fetch" }
 { "dst" is written with an access status of
 "coprocessor data write" }

Address Modes *word* none valid
 dst all modes except register, literal, immediate,
 auto, or index with scaling

Condition Flags Determined by the coprocessor status.

Exceptions External memory fault may occur.

Examples SPOPWS 0x00,%r0
 SPOPWD 0x0F,(%r1)
 SPOPWT 0x1000,4(%r2)

Notes *word* is a 32-bit value.

STRCPY

STRCPY

STRING COPY

Assembler Syntax STRCPY String copy

Opcode 0x3035 STRCPY

Operation while ((*r1 = *r0)!=0){
 {disable interrupts}
 r0++;
 r1++;
 {enable interrupts}
 }
 {instruction fetch after PC discontinuity at the address formed by
 PC +2}

Address Modes None

Condition Flags Unchanged

Exceptions External memory fault may occur in the middle of an iteration.

Example Before: r0

00	00	01	00
----	----	----	----

 r1

00	00	40	00
----	----	----	----

 ← Increasing Bits

The byte locations starting at 0x100 contain the values 0x01, 0x24, 0xE6, 0x7F, 0x11, and 0x00 (location 0x105).

STRCPY

After: r0

00	00	01	05
----	----	----	----

 r1

00	00	40	05
----	----	----	----

STRCPY

STRCPY

The byte locations from 0x4000 through 0x4005 now contain the same values as locations 0x100 through 0x105.

Notes

Opcode occupies 16 bits. All operands are defined implicitly in the registers, r0 and r1, that function as byte pointers. These registers must be preset with the following information before executing STRCPY:

r0 Address of source string
r1 Address of destination string

STRCPY implements the string-copy function commonly used in C language. The instruction may be interrupted *only* at the end of an iteration. A memory fault may occur in the middle of an iteration. To restart the instruction after a fault, execute STRCPY again; the registers are updated after the only memory access that could cause the fault. The assignment is a byte move, and both R0 and R1 are incremented by 1 at each iteration. Execution of STRCPY is finished when a null (zero) byte is reached. The null byte is always copied.

STREND

STREND

STRING END

Assembler Syntax	STREND String end				
Opcode	0x301F STREND				
Operation	<pre>while (*r0 != 0){ r0++; } {instruction fetch after PC discontinuity at the address followed by PC + 2}</pre>				
Address Modes	None				
Condition Flags	Unchanged				
Exceptions	External memory fault may occur in the middle of an iteration.				
Example	Before: r0 <table border="1"><tr><td>00</td><td>00</td><td>04</td><td>00</td></tr></table>	00	00	04	00
00	00	04	00		

← Increasing Bits

The byte locations 0x400 through 0x404 contain the values 0x44, 0x55, 0x01, 0x22, 0x00, respectively.

STREND

After: r0

00	00	04	04
----	----	----	----

Notes	Opcode occupies 16 bits. The operand is defined implicitly in the register r0, a byte pointer that must be preset with the starting address of the source C language string. STREND moves the pointer to the end of the string and could be used as part of a string-length or string-concatenation function. The instruction may be interrupted at any time. A memory fault may occur in the middle of an iteration. To restart the instruction after a fault, execute STREND again; the register is updated after the only instruction that could cause the fault. Each iteration tests a byte and increments the pointer r0 by 1. Execution of STREND terminates when a null (zero) byte is found. Register r0 is left with the address of the null byte.
--------------	--

SUBB2
SUBH2
SUBW2

SUBB2
SUBH2
SUBW2

SUBTRACT

Assembler Syntax **SUBB2** *src, dst* Subtract byte
 SUBH2 *src, dst* Subtract halfword
 SUBW2 *src, dst* Subtract word

Opcodes 0xBF **SUBB2**
 0xBE **SUBH2**
 0xBC **SUBW2**

Operation $dst = dst - src$

Address Modes *src* all modes
 dst all modes except literal or immediate

Condition Flags **N** = 1, if $(dst - src) < 0$
 Z = 1, if $(dst - src) == 0$
 C = 1, if borrow from sign bit of *dst*
 V = 1, if overflow

Exceptions Illegal operand exception occurs if literal or immediate modes are used for *dst*.
 Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.

Examples **SUBB2** %r6,*\$0x30(%r2)
 SUBH2 %r0,\$resulth
 SUBW2 %r3,\$resultw

SUBB3
SUBH3
SUBW3

SUBB3
SUBH3
SUBW3

SUBTRACT, 3 ADDRESS

Assembler Syntax	SUBB3 <i>src1,src2,dst</i> SUBH3 <i>src1,src2,dst</i> SUBW3 <i>src1,src2,dst</i>	Subtract byte, 3 address Subtract halfword, 3 address Subtract word, 3 address
Opcodes	0xFF SUBB3 0xFE SUBH3 0xFC SUBW3	
Operation	$dst = src2 - src1$	
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = 1, if $(src2 - src1) < 0$ Z = 1, if $(src2 - src1) == 0$ C = 1, if carry out of sign bit of <i>dst</i> V = 1, if overflow	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> . Integer overflow exception occurs if PSW<OE> = 1 and there is truncation.	
Examples	SUBB3 %r3,*\$0x1005,%r2 SUBH3 %r1,%r3,%r0 SUBW3 \$N1,\$N2,\$result	

SUBPB2

SUBPB2

SUBTRACT PACKED DECIMAL

Assembler Syntax	SUBPB2 <i>src,dst</i> Subtract packed byte, 2-address
Opcodes	0x9B SUBPB2
Operation	$dst = dst - src - X$
Address Modes	<i>src</i> all modes except expanded-operand <i>dst</i> all modes except literal, immediate, or expanded-operand
Condition Flags	N = 0 Z = 1 if $(dst - src - X) == 0$ C = borrow X = borrow V = 0
Exceptions	Invalid descriptor exception occurs if expanded-operand descriptor is used.
Examples	SUBPB2 %r1,\$0x15000 SUBPB2 %r1,%r2

SUBPB3

SUBPB3

SUBTRACT PACKED DECIMAL, 3 ADDRESS

Assembler Syntax	SUBPB3 <i>src1,src2,dst</i> Subtract packed byte, 3-address
Opcodes	0x9B SUBPB3
Operation	$dst = src2 - src1 - X$
Address Modes	<i>src1</i> all modes except expanded-operand <i>src2</i> all modes except expanded-operand <i>dst</i> all modes except literal, immediate, or expanded-operand mode
Condition Flags	N = 1 Z = 1 if $(src2 - src1 - X) == 0$ C = borrow X = borrow V = 0
Exceptions	Invalid descriptor exception occurs if expanded-operand descriptor is used.
Examples	SUBPB3 %r3,%r2,%r1 SUBPB3 %r1,%r2,\$0x15000
Notes	<i>src1</i> and <i>src2</i> are read-only. <i>dst</i> is write-only.

SWAPBI
SWAPHI
SWAPWI

SWAPBI
SWAPHI
SWAPWI

SWAP (INTERLOCKED)

Assembler Syntax	SWAPBI <i>dst</i> SWAPHI <i>dst</i> SWAPWI <i>dst</i>	Swap byte (interlocked) Swap halfword (interlocked) Swap word (interlocked)
Opcodes	0x1F SWAPBI 0x1E SWAPHI 0x1C SWAPWI	
Operation	{set interlock} tempa = <i>dst</i> <i>dst</i> = r0 r0 = tempa	
Address Modes	<i>dst</i> all modes except register, literal, or immediate	
Condition Flags	N = MSB of r0 Z = 1, if r0 == 0 C = 0 V = 0	
Exceptions	Illegal operand exception occurs if register, literal, expanded-operand, or immediate modes are used for <i>dst</i> .	
Example	The swap instruction can manipulate interlocks for multiprocessors. Suppose location A is the interlock for a critical section of code, and a nonzero means the lock is busy. Then, the following instructions provide a busy-waiting loop: MOVW &1,%r0 L1: SWAPWI A BNEB L1	
Notes	Final value of r0 sets the condition codes. The SAS code is read interlocked (7) for both the read and write bus transactions. If auto increment/decrement with %r0 is used for the destination, the r0 is incremented/decremented before the destination is overwritten.	

TEDTB
TEDTH

TEDTB
TEDTH

TEST EQUAL, DECREMENT, AND TEST

Assembler Syntax	TEDTB <i>dst,disp8</i> Test equal, decrement and test byte TEDTH <i>dst,disp16</i> Test equal, decrement and test halfword
Opcodes	0x4D TEDTB 0x0D TEDTH
Operation	IF (not equal) { IF (<i>dst</i> -1 > -1) { <i>dst</i> = <i>dst</i> - 1 PC = PC + SEXT(<i>disp</i>) } ELSE { <i>dst</i> = <i>dst</i> - 1 } }
Address Modes	<i>dst</i> all modes except literal, immediate, or expanded-operand
Condition Flags	Unchanged
Exceptions	Invalid descriptor exception occurs if literal, immediate, or expanded-operand descriptors are used.
Examples	TEDTB %r1,&0x12 TEDTH \$0x15000,&0x1234
Notes	<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.

TGDTB
TGDTH

TGDTB
TGDTH

TEST GREATER THAN, DECREMENT, AND TEST

Assembler Syntax TGDTB *dst,disp8* Test greater than, decrement and test byte
 TGDTH *dst,disp16* Test greater than, decrement and test halfword

Opcodes 0x6D TGDTB
 0x2D TGDTH

Operation IF (less than or equal) {
 IF (*dst*-1 > -1) {
 dst = *dst* - 1
 PC = PC + SEXT(*disp*)
 }
 ELSE {
 dst = *dst* - 1
 }
 }

Address Modes *dst* all modes except literal, immediate, or expanded-operand

Condition Flags Unchanged

Exceptions Invalid descriptor exception occurs if literal, immediate, or expanded-operand descriptors are used.

Examples TGDTB %r1, 0x21
 TGDTH \$0x15000, 0x3036

Notes *disp8* is a signed 8-bit value. *disp16* is a signed 16-bit value.

TGEDTB
TGEDTH

TGEDTB
TGEDTH

TEST GREATER THAN OR EQUAL, DECREMENT, AND TEST

Assembler Syntax	TGEDTB <i>dst,disp8</i> Test greater than or equal, decrement and test byte TGEDTH <i>dst,disp16</i> Test greater than or equal, decrement and test halfword
Opcodes	0x5D TGEDTB 0x1D TGEDTH
Operation	IF (less than) { IF (<i>dst</i> -1 > -1) { <i>dst</i> = <i>dst</i> - 1 PC = PC + SEXT(<i>disp</i>) } ELSE { <i>dst</i> = <i>dst</i> - 1 } }
Address Modes	<i>dst</i> all modes except literal, immediate, or expanded-operand
Condition Flags	Unchanged
Exceptions	Invalid descriptor exception occurs if literal, immediate, or expanded-operand descriptors are used.
Examples	TGEDTB %r1,&0x12 TGEDTH \$0x15000,&0x1234
Notes	<i>disp8</i> is a signed 8-bit value. <i>disp16</i> is a signed 16-bit value.

TNEDTB
TNEDTH

TNEDTB
TNEDTH

TEST NOT EQUAL, DECREMENT, AND TEST

Assembler Syntax **TNEDTB** *dst,disp8* Test not equal, decrement and test byte
 TNEDTH *dst,disp16* Test not equal, decrement and test halfword

Opcodes 0x7D **TNEDTB**
 0x3D **TNEDTH**

Operation IF (equal) {
 IF (*dst*-1 > -1) {
 dst = *dst* - 1
 PC = PC + SEXT(*disp*)
 }
 }
 ELSE {
 dst = *dst* - 1
 }
 }
 }

Address Modes *dst* all modes except literal, immediate, or expanded-operand

Condition Flags Unchanged

Exceptions Invalid descriptor exception occurs if literal, immediate, or expanded-operand descriptors are used.

Examples **TNEDTB** %r1, &0x12
 TNEDTH \$0x15000, &0x1234

Notes *disp8* is a signed 8-bit value. *disp16* is a signed 16-bit value.

TSTB
TSTH
TSTW

TSTB
TSTH
TSTW

TEST

Assembler Syntax *TSTB src* Test byte
 TSTH src Test halfword
 TSTW src Test word

Opcodes 0x2B *TSTB*
 0x2A *TSTH*
 0x28 *TSTW*

Operation *temp = src - 0*

Address Modes *src* all modes

Condition Flags *N = 1*, if *src < 0* (signed)
 Z = 1, if *src == 0*
 C = 0
 V = 0

Exceptions None

Example *TSTH 14(%r2)*

Notes This instruction sets only condition codes. Its action is the same as a compare instruction, where the first operand is zero, such as in

CMPB &0,src2

However, test is faster because it is one byte shorter.

UNPACKB

UNPACKB

UNPACK BCD BYTE

Assembler Syntax	UNPACKB <i>src1,src2,dst</i> Packed byte → halfword
Opcodes	0x0F UNPACKB
Operation	(packed) <i>dst</i> = (packed) <i>src</i> "OP" <i>src1</i> where "OP" is a specified operation described below
Address Modes	<i>src1</i> all modes except expanded-operand <i>src2</i> all modes except expanded-operand <i>dst</i> all modes except literal, immediate, or expanded-operand
Condition Flags	Unchanged
Exceptions	Invalid descriptor exception occurs if expanded-operand descriptor is used.
Example	The "OP" denotes a special operation as described below: <ul style="list-style-type: none">• The most significant nibbles of <i>src1</i> and <i>src2</i> are put into bits 8-11 and bits 12-15 of the result, respectively.• The least significant nibbles of <i>src1</i> and <i>src2</i> are put into bits 0-3 and bits 4-7 of the result, respectively.

Before:

src1 and *src2* are bytes.

src1

d1	d0
----	----

src2

d3	d2
----	----

UNPACKB *src1,src2,\$0x15000*

After:

The result, stored in \$0x15000 is:

d3	d1	d2	d0
----	----	----	----

XORB2
XORH2
XORW2

XORB2
XORH2
XORW2

EXCLUSIVE OR

Assembler Syntax	XORB2 <i>src, dst</i> XORH2 <i>src, dst</i> XORW2 <i>src, dst</i>	Exclusive OR byte Exclusive OR halfword Exclusive OR word
Opcodes	0xB7 XORB2 0xB6 XORH2 0xB4 XORW2	
Operation	$dst = dst \wedge src$	
Address Modes	<i>src</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if result must be truncated to fit <i>dst</i> size	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .	
Examples	XORB2 &40,4(%r4) XORH2 %r1,\$result XORW2 4(%r1),\$result	

XORB3
XORH3
XORW3

XORB3
XORH3
XORW3

EXCLUSIVE OR, 3 ADDRESS

Assembler Syntax	<i>XORB3 src1,src2,dst</i> <i>XORH3 src1,src2,dst</i> <i>XORW3 src1,src2,dst</i>	Exclusive OR byte, 3 address Exclusive OR halfword, 3 address Exclusive OR word, 3 address
Opcodes	0xF7 <i>XORB3</i> 0xF6 <i>XORH3</i> 0xF4 <i>XORW3</i>	
Operation	$dst = src2 \wedge src1$	
Address Modes	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
Condition Flags	N = MSB of <i>dst</i> Z = 1, if <i>dst</i> == 0 C = 0 V = 1, if result must be truncated to fit <i>dst</i> size	
Exceptions	Illegal operand exception occurs if literal or immediate modes are used for <i>dst</i> .	
Examples	<i>XORB3 &4,*12(%r3),*\$0x400</i> <i>XORH3 %r1,4(%r1),%r0</i> <i>XORW3 %r0,%r1,%r3</i>	

Chapter 6

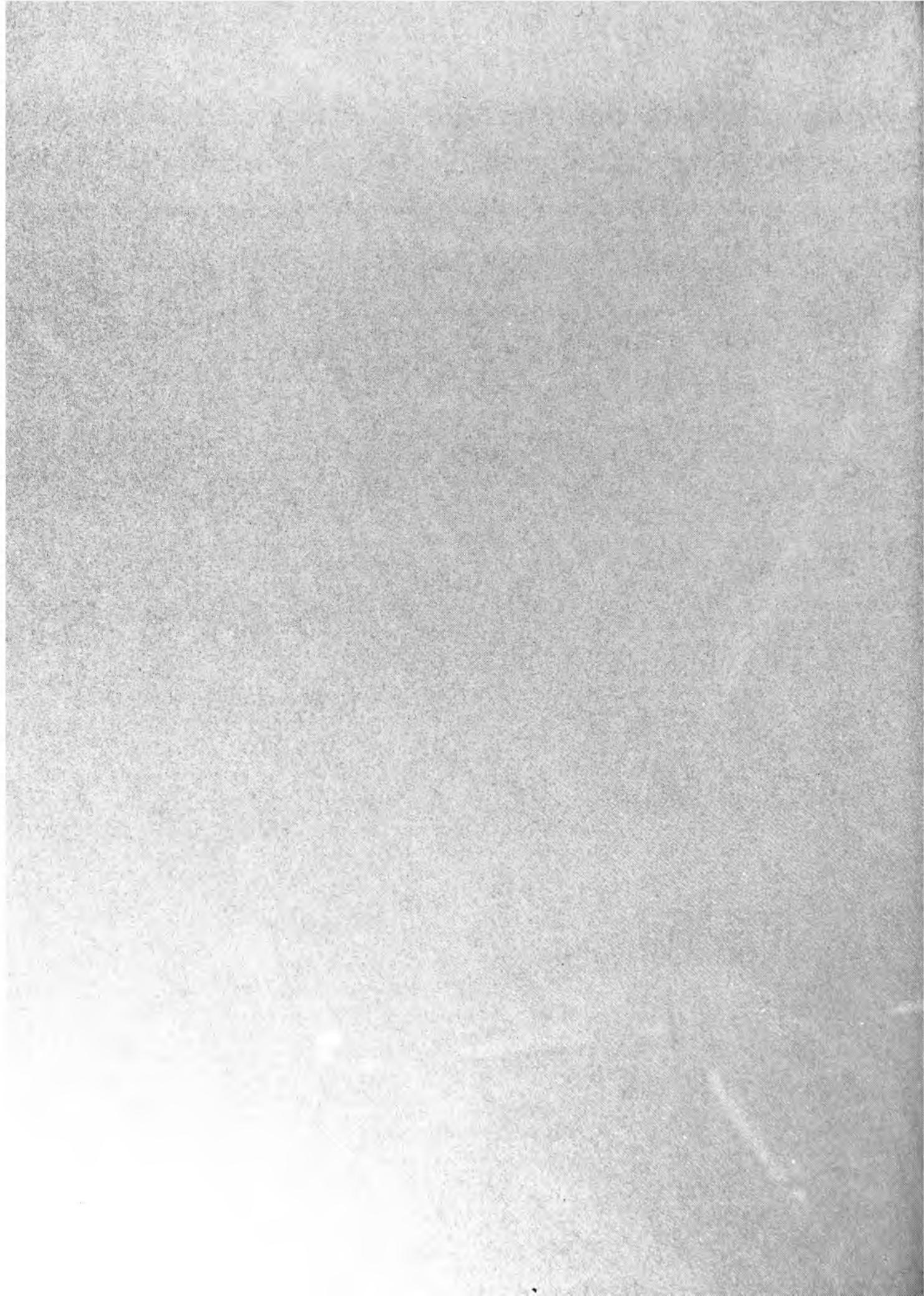
**Operating
System
Considerations**

CHAPTER 6. OPERATING SYSTEM CONSIDERATIONS

CONTENTS

6. OPERATING SYSTEM CONSIDERATIONS	6-1
6.1 FEATURES OF THE OPERATING SYSTEM	6-1
6.1.1 Memory Management Considerations for Virtual Memory Systems	6-4
6.2 STRUCTURE OF A PROCESS	6-5
6.2.1 Execution Privilege	6-5
6.2.2 Execution Stack	6-6
6.2.3 Process Control Block	6-7
Initial Context for a Process	6-11
Saved Context for a Process	6-12
Memory Specifications	6-12
6.2.4 Processor Status Word	6-13
6.3 SYSTEM CALL	6-13
6.3.1 Gate Mechanism	6-17
Pointer Table	6-17
Handling-Routine Tables	6-17
6.3.2 GATE Instruction	6-18
First Entry Point GATE Instruction Entry	6-18
Second Entry Point—The Gate Mechanism	6-19
6.3.3 Return-from-Gate (RETG) Instruction	6-21
6.4 PROCESS SWITCHING	6-21
6.4.1 Context-Switching Strategy	6-22
R-Bit	6-22
AR-Bit	6-22
I-Bit	6-23
6.4.2 Call Process (CALLPS) Instruction	6-27
6.4.3 Return-to-Process (RETPS) Instruction	6-28
6.4.4 User Call Process (UCALLPS) Instruction	6-30
6.5 INTERRUPTS	6-30
6.5.1 Interrupt-Handler Model	6-30
6.5.2 Interrupt Mechanism	6-31
Full-Interrupt Handler's PCB	6-33
Interrupt Stack and ISP	6-33
Interrupt-Vector Table	6-35
6.5.3 On-Interrupt Microsequence	6-35
6.5.4 Returning From an Interrupt	6-36
Full-interrupts	6-36
Quick-Interrupts	6-37
6.6 EXCEPTIONS	6-37
6.6.1 Levels of Exception Severity	6-37
6.6.2 Exception Handler	6-37
6.6.3 Exception Microsequences	6-38
Normal Exceptions	6-39
Stack Exceptions	6-42
Process Exceptions	6-43

Reset Exceptions	6-44
6.7 MEMORY MANAGEMENT FOR VIRTUAL MEMORY SYSTEMS	6-44
6.7.1 Virtual Address Fields	6-45
6.7.2 Initializing the Memory Management Unit	6-48
Defining Virtual Memory	6-48
Peripheral Mode	6-48
6.7.3 MMU Interactions	6-48
MMU Exceptions	6-49
Flushing	6-49
Translation Probe	6-49
6.7.4 Efficient Mapping Strategies	6-49
6.7.5 Indirect Segment Descriptors	6-50
6.7.6 Using the Cacheable Bit	6-50
6.7.7 Physical Data Cache	6-50
Selective Caching	6-51
Flush Data Cache Register (FDCR)	6-51
Reset	6-51
Disabling the Cache	6-51
6.7.8 Using the Page-Write Fault	6-51
6.7.9 Access Protection	6-52
6.7.10 Using the Software Bits	6-52
6.8 OPERATING SYSTEM INSTRUCTIONS	6-52
6.8.1 Notation	6-52
6.8.2 Privileged Instructions	6-53
Call Process (CALLPS)	6-55
Disable Virtual Pin and Jump (DISVJMP)	6-57
Enable Virtual Pin and Jump (ENBVJMP)	6-58
Interrupt Acknowledge (INTACK)	6-59
Return to Process (RETPS)	6-60
Return From Quick-Interrupt (RETQINT)	6-63
Wait (WAIT)	6-65
6.8.3 Nonprivileged Instructions	6-66
Gate (GATE)	6-67
Move Translated Word (MOVTRW)	6-71
Return from Gate (RETG)	6-73
User Call Process (UCALLPS)	6-75
6.8.4 Microsequences	6-78
On-Normal Exception	6-79
On-Stack Exception	6-83
On-Process Exception	6-85
On-Reset Exception	6-87
On-Interrupt	6-89
XSWITCH Microsequence	6-94



6. OPERATING SYSTEM CONSIDERATIONS

The *WE 32200* Microprocessor allows cost-effective design of operating systems by providing the system designer with special-purpose operating system instructions and an architecture that supports process-oriented operating system design. In general, a process is a separately scheduled, independently executed unit of activity. It generally consists of routines (functions) that perform a major task (such as a program manager, a file manager, or a memory manager). This chapter presents the operating system considerations that allow the system designer to make full use of the power of the *WE 32200* Microprocessor as an execution vehicle for today's efficient process-oriented operating systems.

The typical operating system for the *WE 32200* Microprocessor schedules and initiates all processes, handles error conditions (exceptions to normal processing), provides system security, and resets the microprocessor when appropriate. Processes are scheduled through common scheduling algorithms and are initiated through a process switch. A process switch is a change in the process controlling the microprocessor invoked by either an implicit or explicit request. Execution of the call-process (*CALLPS*), user call process (*UCALLPS*), or return-to-process (*RETPS*) operating system instructions will cause an explicit process switch. An implicit process switch occurs as a result of a reset request, a full-interrupt request, or certain exception conditions. In theory, the microprocessor can handle an unlimited number of processes, but real limits are imposed by the operating system design (e.g., limiting the size of the interrupt stack). System security is enforced by the microprocessor and by the *WE 32201* Memory Management Unit or the *WE 32101* Memory Management Unit, an essential part of a virtual memory-based operating system using the *WE 32200* Microprocessor. The microprocessor is reset by the operating system through a reset exception handler process. This handler should initialize the system hardware and reload the operating system.

6.1 FEATURES OF THE OPERATING SYSTEM

As part of its architecture, the microprocessor provides four execution or access levels for processes. This allows each process to have functions that operate at different levels in order to provide the proper levels of system protection. These levels range from the most privileged (level 0) to the least privileged (level 3). Through built-in microprocessor safeguards, the privilege level serves as a protection level. One of the functions of the MMU is to ensure that code and data in any particular level are accessed only by code or processes that have the right permissions. The four execution levels are defined as:

- Kernel (level 0) – The most privileged level; it contains the operating system's most privileged services (e.g., device drivers and interrupt handlers).
- Executive (level 1) – This level is provided for greater flexibility in the operating system design.
- Supervisor (level 2) – Common library routines can operate at this level and be safe from corruption by level 3 activities.

OPERATING SYSTEM CONSIDERATIONS

Features of the Operating System

- User (level 3) – The least privileged level; most user programs can run in this level.

Table 6-1 lists the powerful WE 32200 Microprocessor instructions provided for operating systems. These instructions have a two-level hierarchy: privileged and nonprivileged. Privileged instructions may be executed only if the processor is in kernel level, and they are used to perform process switches, to enable or disable the MMU, or to suspend fetching of instructions. Nonprivileged instructions do not depend on the execution level (i.e., they can be executed at any level) and are used to switch between execution levels (in ways restricted by the operating system) or to convert a virtual address to a physical address.

The processor automatically executes the appropriate microsequence (a built-in sequence of actions) when an interrupt is requested or an exception occurs. These microsequences and many operating system instructions can call functions (also microsequences) that do the context switching (changing the hardware context for the new process to be executed). This feature takes the requirements of context switching out of the operating system, allowing for quicker and more efficient operating system design and execution. The operating system instructions and microsequences are described in section 6.8.

Privileged Instructions			
Instruction	Assembly Syntax	Hex Opcode	Description
Enable virtual pin and jump	ENBVJMP	300D	Asserts the $\overline{\text{VAD}}$ pin on the microprocessor, thus enabling the MMU to translate virtual addresses into physical addresses. The virtual address of the first instruction to be executed after the MMU is enabled must be stored in register r0 prior to execution of this instruction.
Disable virtual pin and jump	DISVJMP	3013	Negates the $\overline{\text{VAD}}$ pin on the microprocessor, thus disabling the MMU from translating virtual addresses into physical addresses. The physical address of the first instruction to be executed after the MMU is disabled must be stored in register r0 prior to execution of this instruction.
Call process	CALLPS	30AC	Used by the operating system to force an explicit process switch. The value of the new process control block pointer (PCBP) must be stored in r0 before this instruction is executed.

OPERATING SYSTEM CONSIDERATIONS
Features of the Operating System

Table 6-1. Operating System Instructions (Continued)			
Privileged Instructions			
Instruction	Assembly Syntax	Hex Opcode	Description
Return to process	RETPTS	30C8	Terminates the current process (its context is not saved) and returns to the interrupted process whose PCBP was saved on top of the interrupt stack.
Wait for interrupt	WAIT	2F	Causes the system to idle until an interrupt or reset occurs.
Interrupt acknowledge	INTACK	302F	Stores interrupt id in r0.
Return from quick interrupt	RETQINT	98	Returns control to the function that called the quick-interrupt. The PC and PSW returned to are popped from the interrupt stack and put in their respective registers.
Nonprivileged Instructions			
Gate	GATE	3061	Mechanism used to transfer control between different execution levels.
Return to gate	RETG	3045	Returns control to the function that called the gate. Linear ordering of execution levels is enforced by RETG (i.e., new execution level may not be more privileged than the current level).
Move translated word	MOVTRW	0C	The MMU converts the virtual address specified by <i>src</i> to a physical address. The result is stored in <i>dst</i> . Can be used to obtain a physical address to send to an I/O device.
User call process	UCALLPS	30C0	Performs a process switch by saving the current process, pushing its PCBP on the the interrupt stack, and entering the new process.

OPERATING SYSTEM CONSIDERATIONS

Memory Management Considerations for Virtual Memory Systems

Other features of the microprocessor architecture provided for operating system design are:

- The microprocessor supports different levels of execution privilege and enforces linear ordering of these levels only on a return-from-gate (RETG) instruction, as discussed in section 6.3.3.
- The microprocessor provides flexibility in transferring execution control between privilege levels. Control is transferred through the gate mechanism, as discussed in section 6.3.
- A scheduler may explicitly switch processes (using CALLPS or RETPS instructions), but part of the interrupt structure and certain exception conditions involve implicit switching of processes and therefore possess the advantages of a process switch.
- The CPU supports a layered exception-handling structure that uses different mechanisms (process switching or gate mechanism), depending on the severity of the exception.
- The CPU supports full- and quick-interrupt handlers that use different mechanisms (process switching or gate mechanism). A full-interrupt is handled as an implicit process switch, while a quick-interrupt is handled as an implicit gate.
- The address space of each process may include the space that contains the operating systems; i.e., the user may pass and address arguments across system calls efficiently but need not switch memory map information across such calls.
- The CPU supports memory management, permitting the user to think that there is virtually limitless memory space. However, the operating system must provide the information required by a memory management unit to translate virtual addresses (i.e., memory descriptors) or to disable the MMU for physical addressing. Systems without an MMU use only physical addressing.

6.1.1 Memory Management Considerations for Virtual Memory Systems

A memory management unit is required for virtual memory systems. The primary function of an MMU is to translate virtual addresses into physical addresses and implement the protection of data for each process. A virtual memory operating system is supported by a variety of features:

- Contiguous and paged segments. Segments, or blocks of memory, are defined by memory descriptors. The WE 32201 Memory Management Unit uses segment descriptors to define contiguous segments (i.e., a block of memory defined to be up to 128 Kbytes in length) and segment and page descriptors to define paged segments (i.e., a block of memory defined to contain up to sixty-four 2 Kbyte pages, thirty-two 4 Kbyte pages, or sixteen 8 Kbyte pages).
- Present bits to indicate whether or not a segment is currently in main memory.
- Referenced and modified bits to aid implementation of the least recently used algorithm in the operating system.

- An indirection feature that allows segments to be given different access permissions (i.e., read or write), yet still be shared by different routines running at the same execution level (see section 6.7.5).
- Access fields contained in segment descriptors used to provide protection so that segments are accessed in the appropriate way by the appropriate execution level. An access exception is generated if access is disallowed.
- Segment marking as cacheable or not cacheable. This can be an aid in the use of an external data cache in the system main memory (see section 6.7.6).

Note: The *WE 32201* MMU has an internal data cache.

- A unique exception (page-write) that can be issued on any attempt to write a given page (see section 7.7).

6.2 STRUCTURE OF A PROCESS

Each process executing in the *WE 32200* Microprocessor consists of the following elements:

- A processor status word (PSW). This privileged CPU register contains status information on both the instruction just executed and the current process.
- A process control block (PCB). This process data structure in external memory contains the hardware context of a process when the process is not executing.
 - If $PSW\langle R \rangle == 1$ and $PSW\langle AR \rangle == 0$, this context consists of the initial and current contents of control registers: PSW, program counter (PC), and stack pointer (SP); the last contents of the general-purpose registers r0 through r8, frame pointer (FP), and argument pointer (AP), boundaries for an execution stack; and block-move specifications for the process.
 - If $PSW\langle AR \rangle == 1$ and $PSW\langle R \rangle == 1$, the context consists of the initial and current contents of control registers: PSW, PC, and SP; the last contents of the general-purpose registers r0 through r8 and r16 through r23, FP, and AP, and boundaries for an execution stack.
- A process control block pointer (PCBP). This privileged CPU register identifies the starting location of the PCB for the process currently executing.
- Memory address space (the areas in memory allocated for the process). This space can be defined by memory management specifications in the PCB block-move area.
- Segment and page descriptors and MMU SRAM's register contents, if the system uses an MMU. This information can be defined in the PCB block-move area for automatic transfer to the MMU during a process switch.

6.2.1 Execution Privilege

As stated previously, the CPU recognizes four execution levels: kernel (most privileged), executive, supervisor, and user (least privileged level). Controlled entry to an execution level does not require a particular order of levels to be followed.

OPERATING SYSTEM CONSIDERATIONS

Execution Stack

However, a controlled return requires a transfer to a less privileged execution level. See section 6.3 for a description of controlled transfers across privilege levels. The operating system design may use the four execution levels to manage layers of control. However, further protection for memory access must be built into a memory management system.

To protect against an unwanted process switch, privileged operating system instructions may be executed only in kernel mode. The other operating system instructions and the instruction set may be executed in any of the four modes. Thus, only a two-level privilege hierarchy exists for instruction execution.

Information associated with a process is protected by the restriction that the CPU be in kernel mode when writing the following registers:

- Processor status word (PSW) – provides information about the current process. The microprocessor implicitly alters the condition flags after most instructions. In addition, some PSW fields change their contents to identify the type and severity of an exception and help the operating system select the appropriate exception handler.
- Process control block pointer (PCBP) – contains the starting address of the PCB for the current process. Because the PCB for a process is assigned to a fixed starting location, the PCBP content changes only during a process switch.
- Interrupt stack pointer (ISP) – points to a stack used to store the PCBP for interrupted processes and restores the PCBP when a process returns from its interrupted state. Generally, the ISP is altered only on a process switch.

If the CPU is not in kernel mode, it generates a normal exception (privileged register) when an instruction tries to write to the PSW, PCBP, or ISP. The use of privileged registers is discussed later.

6.2.2 Execution Stack

During the execution of a process, the CPU SP register identifies the address of the next available location on an execution stack. Conventionally, such a stack could be used for linking functions and passing arguments between:

- Functions that execute at the same level.
- A privileged function and its less privileged caller.
- An exception handler and the function that caused the exception.
- An interrupt handler and the interrupted function.

An execution stack also provides temporary storage for local variables.

Unlike other architectures that require at least two stacks, the WE 32200 Microprocessor uses only one execution stack per process. Other CPUs generally use a stack for each privilege level. A privileged stack in other architectures is protected from errors in less privileged levels that could destroy its contents.

In the *WE 32200* Microprocessor architecture, a process uses one stack in all execution levels. Each process stack is protected through maintenance of its upper- and lower-bounds in the process control block (the data area that stores the hardware context) for the process and checking of the bounds during a gate operation. Thus, each execution level is protected from stack errors by other execution levels. In addition, using only one stack reduces the overhead for stack allocation and simplifies the management of process stacks.

Before executing a transfer to a more privileged level through a system *CALL* or *GATE*, the CPU checks the current *SP* against the stack bounds. The transfer occurs if the *SP* falls within bounds. Otherwise, a stack exception (stack bound) is generated.

Using the execution stack for the process, the CPU handles gate-like normal exceptions within the process in which they occurred. Before transferring to the appropriate exception handler, it checks the current *SP* against the stack bounds.

Because an interrupt other than a quick-interrupt causes a process switch, the CPU interrupt structure uses a different execution stack for each interrupt handler. Therefore, the sanity of the interrupted process execution stack does not have to be checked. In addition, the CPU stores the *PCBP* of each interrupted process on one system-wide interrupt stack and retrieves it from that stack when the process resumes execution. Quick interrupts save the *PC* and *PSW* context on the execution stack of the active process and are handled in the same manner as a normal exception.

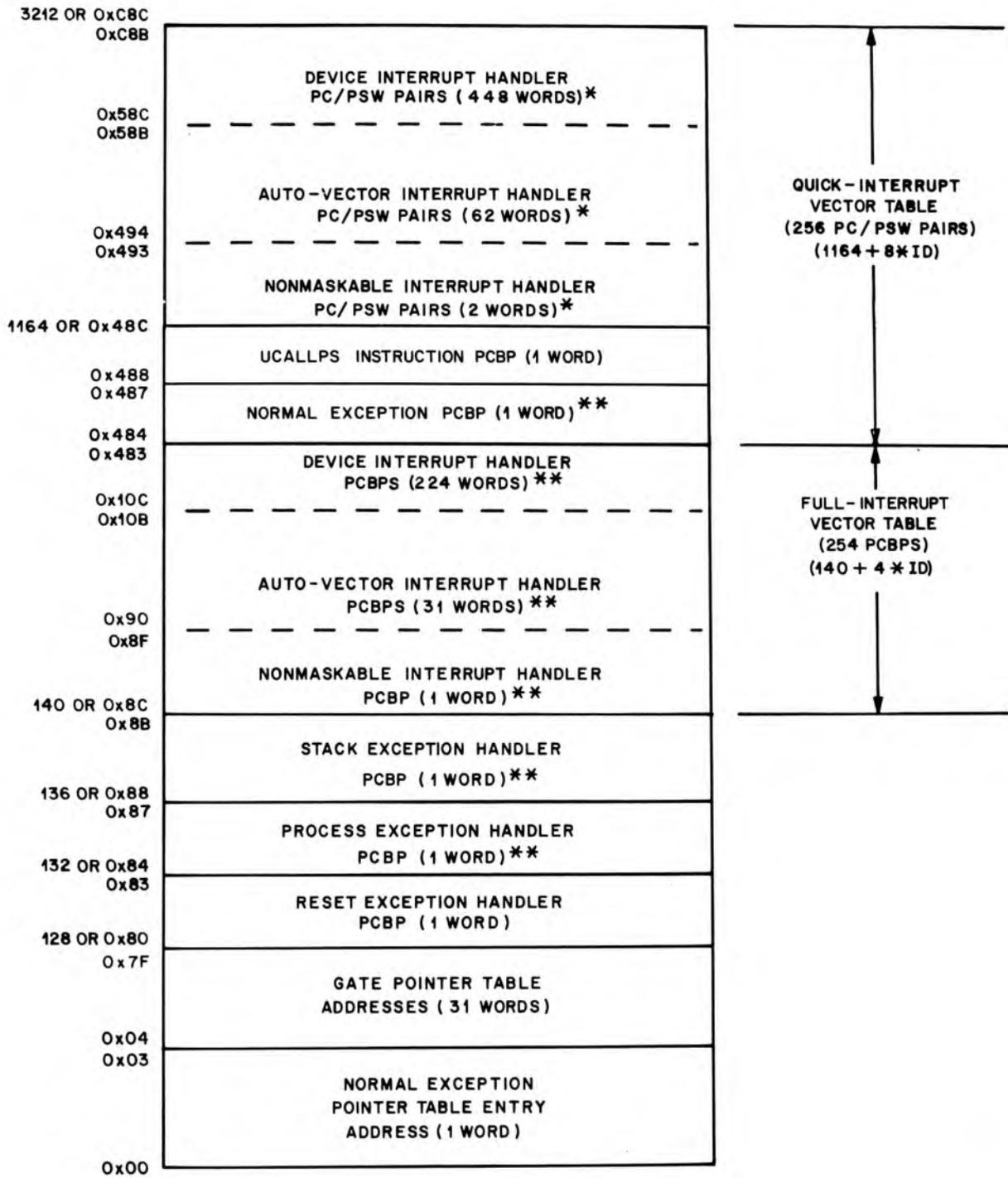
6.2.3 Process Control Block

Each process has a process control block (*PCB*). Elements in the *PCB* are accessed through the process control block pointer (*PCBP*). This privileged register contains the starting address in memory of the *PCB* for the process that is currently executing. Although *PCBs* can be stored anywhere in memory, Figure 6-1 identifies where the *PCBP* must be stored for various processes.

On execution of the *CALLPS* instruction, the *PCBP* is taken from register *r0*; it must be stored in *r0* prior to the execution of *CALLPS*. On execution of the *UCALLPS* instruction, the *PCBP* is taken from memory location *0x488*, the last entry in the full-interrupt vector table. The memory location containing the *PCBP* should be write-protected from users. The *PCBP* is taken from the top of the interrupt stack when *RETPS* is executed.

OPERATING SYSTEM CONSIDERATIONS

Process Control Block



* Implicit Gate

** Implicit Process Switch

Figure 6-1. Memory Map

Because only one process executes at a time in a multiprogramming system, the PCB of a process retains its hardware context when that process is not executing. Figure 6-2 shows a PCB when the R bit of the PSW is set and the AR bit of the PSW is cleared. It contains:

- **Initial context.** The three control registers (PC, PSW, and SP) are loaded with initial values when a process starts executing for the first time. First time execution is indicated by the I bit in the PSW being set.
- **Control register save area.** When an executing process is either interrupted or a process switch occurs during its execution, the current contents of its control registers are saved here. These values are loaded when the interrupted process resumes execution and the I bit in the PSW is cleared.

Note: If the I bit in the PSW of a process is initially set, execution starts from its initial-context values. If the bit is clear, execution resumes from an intermediate context. See section 6.4.1 for more information on the I bit.

- **Stack bounds.** The upper and lower stack bounds define the area allocated to the execution stack for this process.
- **General register save area.** This area is reserved for saving the contents of register r0 through r10. Registers r9 and r10 are the frame pointer (FP) and argument pointer (AP), respectively. These are used to specify the location of variables or arguments. The FP locates local variables for a function, while the AP locates arguments passed to the function.
- **One or more block-move areas.** If a process does not require any block moves (usually used to perform a change in memory management specifications), only the null block is required in the PCB. Otherwise, it contains a block-move area for each move to be performed.

If both the AR and the R bit of the PSW are set, the PCB has a slightly different form, as shown on Figure 6-3. It contains:

- The initial context, control register save area, and the stack bounds, as described above.
- **General register save area.** As above, the contents of registers r0 through r10 are saved. Additionally, the contents of registers r16 through r23 are also saved.

Note that, unlike the PCB illustrated on Figure 6-2, block-move areas are not saved in this PCB.

Note: The R bit of the PSW must be set if the general registers are to be saved for the old process or loaded for the new process and if the block moves are to be executed for the new process. See section 6.4.1 for more information on the R bit.

In general, the PC and SP values and block addresses stored in a PCB may be physical or virtual addresses. If they are virtual addresses, the MMU must be enabled to translate them into physical addresses. Two operating system instructions, enable virtual pin and jump (ENBVJMP) and disable virtual pin and jump

OPERATING SYSTEM CONSIDERATIONS

Process Control Block

(DISVJMP), enable or disable the CPU's virtual address pin (\overline{VAD}) to tell the MMU it is generating virtual (enable) or physical (disable) addresses. Before the ENBVJMP or DISVJMP instruction is executed, the virtual or physical address, respectively, of the next instruction to be executed must be stored in r0.

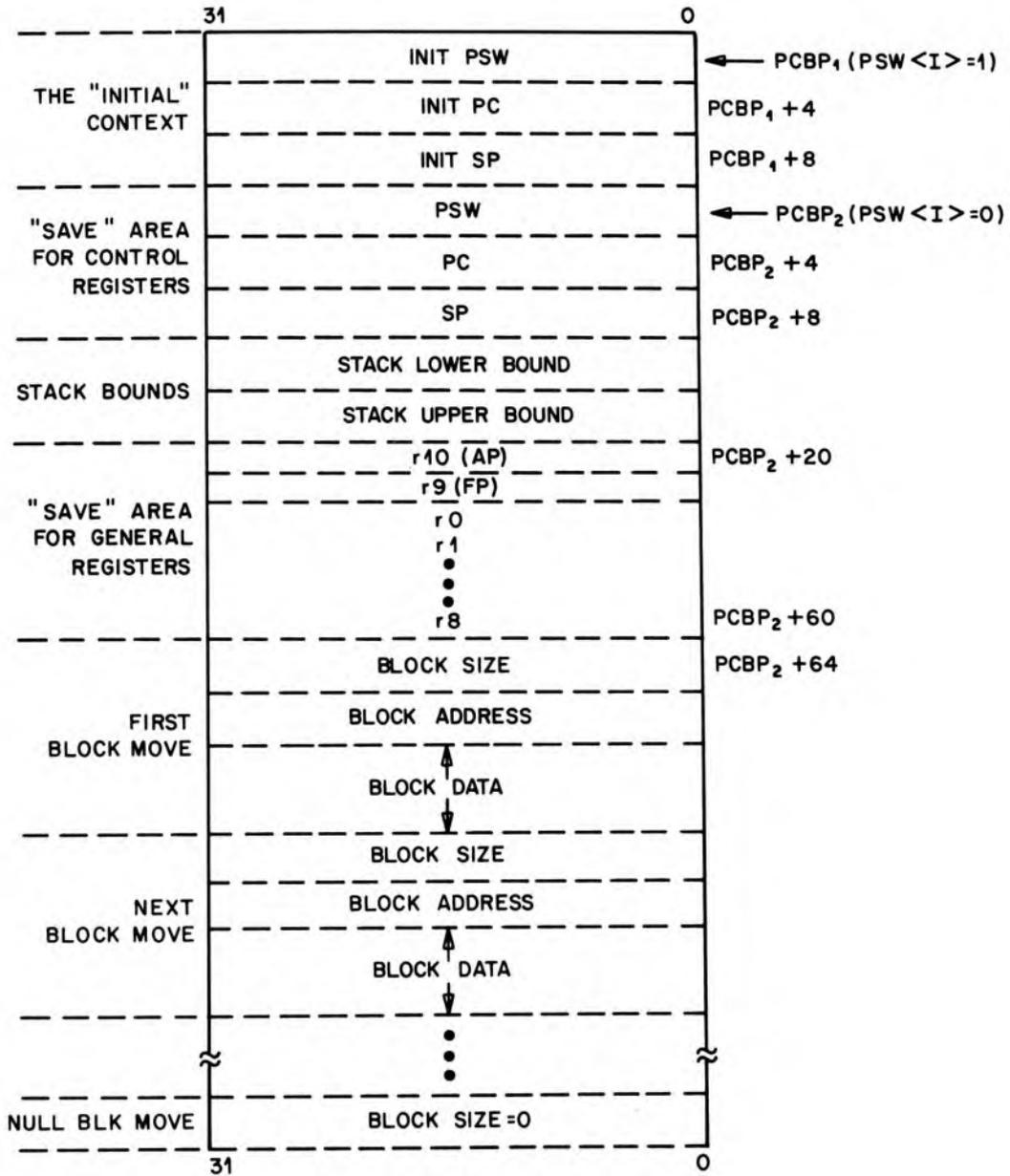


Figure 6-2. Typical Process Control Block

(PSW<R> == 1 and PSW<AR> == 0)

OPERATING SYSTEM CONSIDERATIONS

Initial Context for a Process

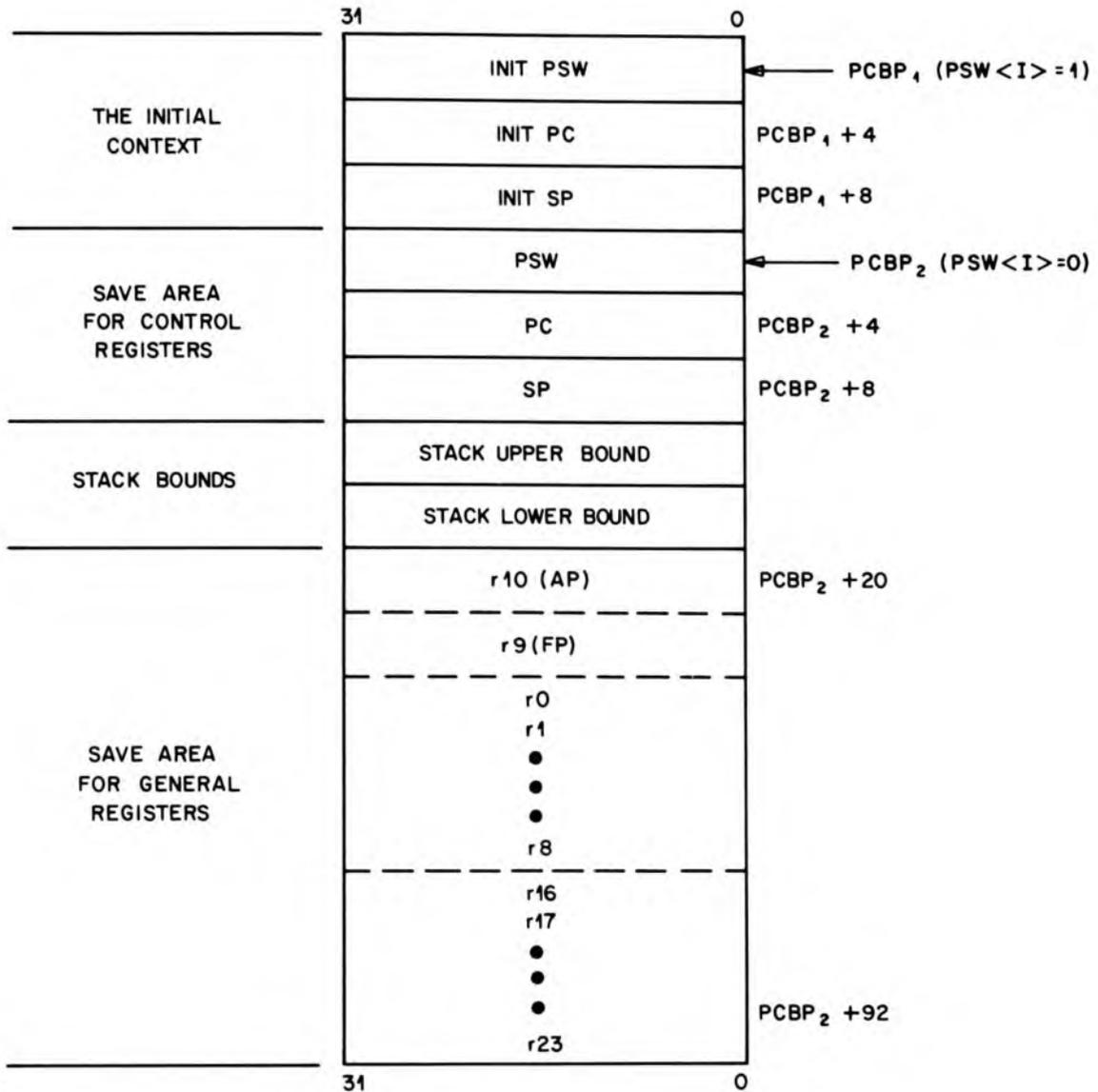


Figure 6-3. Typical Process Control Block
(PSW<R> == 1 and PSW<AR> == 1)

Initial Context for a Process

The initial context of the executing process is set up as follows:

1. The PCBP, stored in memory, points to the initial-context area of its PCB.
2. The initial PSW occupies the first PCB location, and its I bit should be set to identify that the process starts executing from its initial context. The R bit should be set if this process will use general registers. When the process switch occurs, if

OPERATING SYSTEM CONSIDERATIONS

Saved Context for a Process

the I bit is set, the PCBP is incremented by 0x0C (12) to point to the saved context area. The I bit is then cleared in the PSW register of the initial context area, but the I bit in the saved context area is unaffected. See section 6.2.4 and section 6.4.1 for more details about the R and I bits.

3. The second PCB word, the initial PC, is the address of the first instruction that process executes.
4. The third PCB word contains the initial SP (the address of the first location on the execution stack).
5. The seventh and eighth PCB words define the upper and lower limits of the execution stack.

The values in the initial-context area and the stack bounds never change during normal execution.

Saved Context for a Process

When a process switch occurs, the CPU uses the current PCBP to save the context of process A (the executing process) in the current PCB. Using offsets from the PCBP to access the correct PCB location for process A, the CPU stores PC, PSW, SP and if the PSW R bit is set, the general registers (additional registers are saved if the PSW AR bit is set). It then reads in a new PCBP value for process B (the incoming process) and loads the process B context from its PCB.

Memory Specifications

On each process switch, if the R bit in the PSW is set and the AR bit in the PSW is cleared, the CPU, using information in the process PCB, performs a series of block moves. The PCB provides three elements for each block move (see Figure 6-2):

- Block size – this word value specifies the length of the block (number of words to be moved) and implicitly identifies the starting location of the next block-move area.
- Block address – this word value is the destination address at which the CPU starts writing the block data.
- Block data – this series of words represents the data to be moved. If the system has an MMU, it could be the information written to MMU registers (or tables) to set up the memory context for the new process.

The CPU executes a move block (MOVBLW) instruction for each block until a zero-length block (block size = 0) is reached.

A memory management scheme does not alter the way the CPU performs the block moves or how many block moves occur. However, memory management may affect block addresses. Systems with an MMU should use a virtual address for each block when the MMU is enabled and physical addresses when the MMU is disabled. For a system without an MMU, a block address must be a physical address.

6.2.4 Processor Status Word

The CPU maintains a 32-bit processor status word (PSW) register which defines the state of a currently running process. Table 6-2 identifies its contents.

The read-only fields of the PSW cannot be altered by software regardless of the execution mode. An exception or process switch always directly affects the ET, ISC, and TM fields. The ET and ISC fields, which identify the type and cause of an exception, are part of the exception mechanism described in section 6.6. The TE and TM fields are part of the trace-trap mechanism.

An instruction may read the PSW at any time but may write it explicitly only when the process is in kernel mode. However, the CPU implicitly alters some fields during normal execution at other levels. In particular, most instructions change the condition flags. Note that the PSW bits CD, QIE, and EA can be changed only by microsequence instructions or quick-interrupts. The microsequence instructions that may alter these bits are GATE, CALLPS, RETPS, and the reset sequence.

6.3 SYSTEM CALL

The system CALL (gate mechanism) provides a means of controlled entry into a function by installing new values in the PSW and PC. If the new PSW has a different privilege level than the current PSW, a transition to a different execution level occurs.

On simpler CPUs, a trap or supervisor call instruction picks up a new PC and PSW from a fixed location. Then the software has to perform further indirection based on the trap number. The gate mechanism, embodied in its gate (GATE) instruction, automatically performs this second level of indirection for the user. The gate mechanism is described in section 6.3.1.

Table 6-2. Processor Status Word Fields													
Bit(s)	Field	Contents	Description										
0—1	ET	Exception type	This read-only field indicates the type of exception generated during operations and is interpreted as: <table style="margin-left: 2em; border: none;"> <thead> <tr> <th style="text-align: left;">Code</th> <th style="text-align: left;">Description</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>On Reset Exception</td> </tr> <tr> <td>01</td> <td>On Process Exception</td> </tr> <tr> <td>10</td> <td>On Stack Exception</td> </tr> <tr> <td>11</td> <td>On Normal Exception</td> </tr> </tbody> </table>	Code	Description	00	On Reset Exception	01	On Process Exception	10	On Stack Exception	11	On Normal Exception
Code	Description												
00	On Reset Exception												
01	On Process Exception												
10	On Stack Exception												
11	On Normal Exception												

OPERATING SYSTEM CONSIDERATIONS

System Call

Table 6-2. Processor Status Word Fields (Continued)

Bit(s)	Field	Contents	Description										
2	TM	Trace mask	The read-only TM field enables masking of a trace trap. This bit masks the trace enable (TE) bit for the duration of one instruction to avoid a trace trap. The TM bit is set at the beginning of every instruction and cleared as part of every microsequence that performs a context switch or a return from gate.										
3—6	ISC	Internal state code	This 4-bit code distinguishes between exceptions of the same exception type. The ISC is a read-only field.										
7—8	RI	Register-initial context	These bits control the context switching strategy. The I bit (bit 7) determines if a process executes from initial or intermediate context. The R bit (bit 8, read only) determines if the registers of a process should be saved. It also controls block moves to change map information.										
9—10	PM	Previous execution level	This field defines the previous execution level. The code is interpreted as: <table border="0" style="margin-left: 20px;"> <tr> <td>Code</td> <td>Description</td> </tr> <tr> <td>00</td> <td>Kernel level</td> </tr> <tr> <td>01</td> <td>Executive level</td> </tr> <tr> <td>10</td> <td>Supervisor level</td> </tr> <tr> <td>11</td> <td>User level</td> </tr> </table>	Code	Description	00	Kernel level	01	Executive level	10	Supervisor level	11	User level
Code	Description												
00	Kernel level												
01	Executive level												
10	Supervisor level												
11	User level												
11—12	CM	Current execution level	This field defines the current execution level. The CM code is interpreted the same way as the PM code. Changes to the CM field via instructions with the PSW as an explicit destination may cause the XMD pins to change in the middle of a memory access, which could cause a spurious exception or system problem. Therefore, only microsequence instructions should be used to change the CM field state.										
13—16	IPL	Interrupt priority level	The IPL field represents the current interrupt priority level. Fifteen levels of interrupts are available. An interrupt, unless it is a nonmaskable interrupt, must have a higher priority level than the current IPL in order to be acknowledged. Therefore, level 0000 indicates that any of the fifteen interrupt priority levels (0001 through 1111) can interrupt the CPU; level 1111, the highest interrupt priority level, indicates that no interrupts (except a nonmaskable interrupt) can interrupt the CPU.										

Table 6-2. Processor Status Word Fields (Continued)			
Bit(s)	Field	Contents	Description
17	TE	Trace enable	This bit enables the trace function. When TE is set, it causes a trace trap to occur after execution of the next instruction. Debugging and analysis software use this facility for single-stepping a program. Changes to the state of the TE bit via instructions with the PSW as an explicit destination may cause unpredictable trace behavior. Therefore, only microsequence instructions should be used to change the TE bit state.
18—21	NZVC	Condition codes	The condition codes reflect the resulting status of the most recent instruction execution that affects them. These codes are tested by using the conditional branch instructions and indicate the following when set: N – Negative (bit 21) V – Overflow (bit 19) Z – Zero (bit 20) C – Carry (bit 18)
22	OE	Enable Overflow trap	This bit enables overflow traps. It is cleared whenever an overflow trap is detected and handled.
23	CD	Cache disable	This bit enables and disables the instruction cache. When the CD bit is set, the cache is not used. Only microsequences (CALLPS, GATE, RETPS, UCALLPS, and On-Reset) or quick-interrupts can be used to change the state of the CD bit.
24	QIE	Quick-interrupt enable	The QIE enables and disables the quick-interrupt facility. If QIE is set, an interrupt is handled via the quick-interrupt sequence. This bit can be changed only by microsequence instructions or quick interrupts.
25	CFD	Cache Flush disable	When this bit is set, it disables cache flushing (emptying of the instruction cache contents) during the XSWITCH_TWO microsequence.
26	X	Extend carry/borrow	This bit represents the condition code for the BCD operations. The extended carry/borrow bit can be set only if there is a carry or borrow from a BCD arithmetic operation on two 2-BCD digits. When there is no carry or borrow from such operations, the bit is cleared.

OPERATING SYSTEM CONSIDERATIONS

System Call

Table 6-2. Processor Status Word Fields (Continued)

Bit(s)	Field	Contents	Description
27	AR	Additional register save	This bit, when set, enables the additional 8 registers to be saved during a process switch with no block moves. When cleared, the CPU returns to <i>WE 32100</i> CPU compatible mode and the process switch mechanism is the same as the <i>WE 32100</i> CPU's. On external reset, this bit is cleared.
28	EX/UC	Normal exception and user call process option	This bit, when set, causes normal exception procedures to be process switch-like, where the sequence of operations is identical to a process switch except that the PCBP is located at a different specified address. Additionally, the UCALLPS instruction is also enabled, which performs a process switch. The RETPS instruction can be used to terminate both processes. When cleared, the option reverts to <i>WE 32100</i> CPU mode, where the normal exception is gate-like. Additionally, the UCALLPS instruction is disabled. If the UCALLPS instruction is used, an illegal opcode exception is generated. On external reset, this bit is cleared.
29	EA	Arbitrary byte alignment enable	This bit enables and disables the arbitrary byte alignment feature in the <i>WE 32200</i> CPU. Arbitrary byte alignment is enabled when the EA bit is set, allowing the CPU to read or write word and halfword data from any byte boundary. If EA is cleared, arbitrary byte alignment is disabled and the alignment fault detection is enabled (i.e., compatible with the <i>WE 32100</i> CPU). On external reset, arbitrary byte alignment is also disabled. This bit can be changed only by microsequence instructions or quick interrupts.
30—31	—	Unused	These bits are always cleared, and should not be used.

6.3.1 Gate Mechanism

The CPU contains a microsequence program that locates the handling routine for the gate mechanism. To use this mechanism, the operating system must provide the following gate mechanism tables:

- **Pointer table** – contains the 32-bit starting addresses for a set of handling-routine tables. The CPU assumes address 0 as the beginning of the table. The table contains thirty-two 4-byte (word) addresses, one for each handling-routine table.

Note: Use of kernel level is forced whenever this table is accessed during execution of the GATE instruction.

- **Handling-routine tables** – each table in the set contains the entry points (PSW and PC values) for a group of functions. A table is limited to 4096 two-word entries; one a new PSW and the other a new PC (in that order) for a controlled transfer.

Two indexes, obtained from a GATE instruction's implied operands, locate the appropriate PC and PSW pair for the controlled transfer.

Pointer Table

This table contains thirty-two entries and starts at location 0. It must be contained in secure memory (write permission for kernel level only) to prevent unwarranted access. See Figure 6-1 for the location of the gate pointer table in memory. The first entry is reserved for normal-exception handling. Therefore, address 0 must locate the handling-routine table (entry point set) for the normal-exception handlers.

The rest of the addresses in the pointer table may define sets of entry points for controlled transfers. For example, one entry can be used to locate the handling-routine table for kernel level entries, one entry for executive level entries, one for supervisor level entries, and one for user level entries.

All thirty-two entries in the pointer table must be defined. A typical use for the remaining entries is to define all unused pointer table entries to point to a dummy handling-routine table. The dummy table is typically used to prevent an exception from occurring should an offset into the pointer table result in locating an undefined handling-routine table.

Handling-Routine Tables

A handling-routine table stores a maximum of 4096 entry points (PSW and PC pairs) and may be placed anywhere in memory (virtual memory if the system has an MMU that is enabled; physical memory if it does not). However, each must start at an address that is a multiple of eight. In a typical system, the handling-routine tables for entry into kernel level reside in a section of memory that is shared by all processes.

Note: Sections of memory do not imply execution level. The GATE instruction forces kernel level before it accesses any handling-routine tables. To preserve table security, these tables should be protected so only the kernel level can write to them.

OPERATING SYSTEM CONSIDERATIONS

GATE Instruction

6.3.2 Gate (GATE) Instruction

The GATE instruction is modeled after the jump to subroutine (JSB) instruction rather than the call procedure (CALL) instruction, which calls a function. In the typical system environment (e.g., *UNIX* System, C compiler), the compiler generates a call to an assembly-language function which then executes the GATE instruction. GATE needs to execute only a simple jump since the 'call frame' already exists.

Although GATE may be executed at any privilege level, the CPU forces and releases kernel level for memory access. The GATE instruction has two entry points. GATE starts execution at the first entry point, while the on-normal exception microsequence enters at the second (see section 6.6). The second entry point is also the start of the GATE mechanism.

Before a GATE instruction is executed, two registers must be loaded:

- Register 0 (r0) must be loaded with the offset for constructing index1 (the index into the pointer table). Index1 identifies the starting address of the appropriate handling-routine table. This value is a byte offset and should be a multiple of 4.
- Register 1 (r1) must be loaded with the offset for constructing index2 (the index into the handling-routine table). Index2 locates the new PSW and PC pair. This value is a byte offset and should be a multiple of 8.

An example of indexing for the GATE instruction is shown on Figure 6-4.

The on-normal exception microsequence is modeled after a GATE. On a normal exception, the CPU supplies all the information needed to execute a gate-like sequence.

First Entry Point – GATE Instruction Entry

The GATE instruction executes the following tasks in sequence.

1. GATE forces kernel level on memory accesses and checks the current SP against the upper- and lower-stack bounds in the currently executing process PCB. A memory exception on accessing either of the stack bounds from the PCB causes a process exception (gate-pcb). If the SP is outside either boundary, a stack exception (stack bound) is generated. GATE then releases kernel level for memory accesses.
2. GATE writes 1, 0, 2 to the ISC, TM, and ET fields, respectively, of PSW. Then it saves the address of the next instruction (PC + 2) and the current PSW on the execution stack. If a memory exception occurs on the stack accesses, the CPU generates a stack exception (stack bound).
3. GATE computes index1 for the pointer table by masking the contents of r0 with 0x7C and places the result in tempa. It then masks the contents of r1 with 0x7FF8 for index2 and stores the result in tempb. (Special registers tempa and tempb are used in later steps for accessing the handling-routine tables.)

Second Entry Point – The Gate Mechanism

(On-normal Exception Entry)

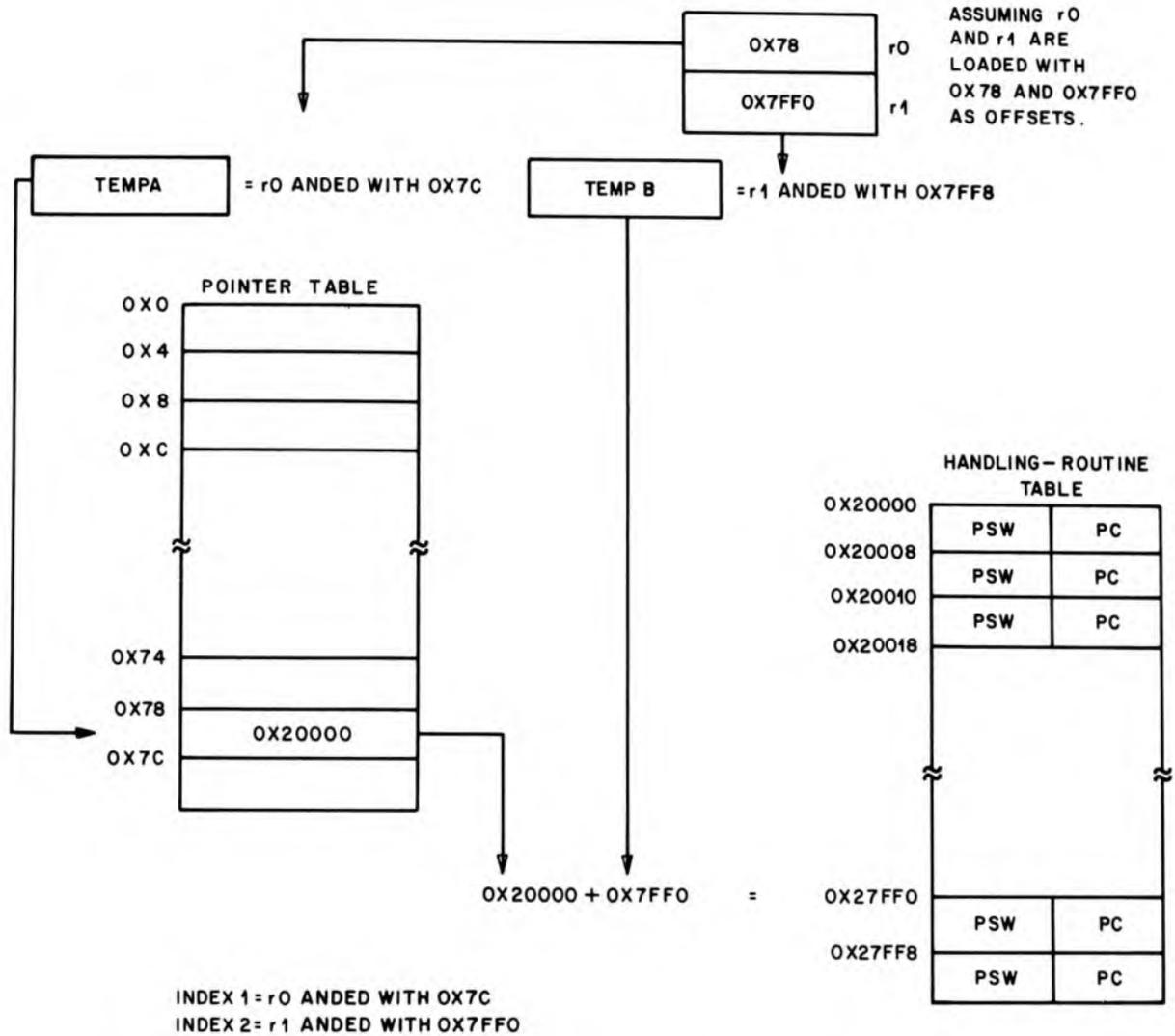
1. GATE again forces kernel execution level for memory accesses.
2. GATE uses tempa as a pointer to read the starting address of a handling-routine table from the pointer table and writes it to tempa. It then adds tempa and tempb (the offset into the handling-routine table) and stores the result, index2, in tempa. This is the address of the new PSW and entry point PC for the GATE jump.
3. GATE uses index2 to get new values for PSW fields IPL, R, and AR from the old PSW values. It then sets PSW fields ISC, TM, and ET to 7, 1, and 3, respectively.
4. The new PSW<PM> field is set to the value from the the old PSW<CM> field.
5. GATE uses index2 to locate and load the new PC.
6. GATE adjusts SP to a location above the saved PC and PSW (thus completing a push of the PC and PSW onto the stack) and releases kernel level for memory accesses.

The CPU then begins executing the handling routine. When the routine finishes, a return-from-gate (RETG) instruction returns to the function that issued the system call.

Note: If the GATE instruction is invoked directly, a memory exception that occurs during the remaining steps causes a normal exception (gate vector). A normal-exception microsequence entering here will already have kernel level in effect and values in tempa and tempb. Entering at this point from a normal-exception microsequence means that a memory exception for any step generates a reset exception (gate vector).

OPERATING SYSTEM CONSIDERATIONS

Second Entry Point – The Gate Mechanism



Notes:
 Masking $r0$ with $0x7c$ forces entry at a word boundary into the pointer table.
 Masking $r1$ with $0x7ff8$ forces entry at a double-word boundary into the handling-routine table.

Figure 6-4. Pointer and Handling Table Indexing

6.3.3 Return-From-Gate (RETG) Instruction

The RETG instruction is modeled after a return-from-subroutine (RSB) instruction rather than after a return-from-procedure (RET) instruction. Unlike the GATE instruction, RETG enforces linear ordering of execution levels, which means the new execution level may not be more privileged than the current level. During an RETG, the microsequence forces and releases kernel level as required for memory access.

The return-from-gate instruction performs the following sequential actions to return to the calling function.

1. Retrieves the old PSW and next-instruction address (stored on the execution stack by the corresponding GATE) and places these in tempa and tempb, respectively.
2. Compares the CM field in the current PSW to the CM field of the old PSW (in tempa) to verify that the new execution level is less than or equal to the current level. If this test fails, the CPU issues a normal exception (illegal level change).
3. Writes the PSW fields except for IPL, CFD, QIE, CD, R, AR, EX, and EA, using the values in tempa (the old PSW).
4. Loads PC from tempb.
5. Adjusts SP to the location below the saved PSW and PC (thus completing a pop of the PSW and PC from the stack).
6. Writes 7, 0, and 3 to PSW fields ISC, TM, and ET, respectively.

The function that called the GATE then starts executing its next instruction.

Note: If a memory exception occurs on a stack access during these steps, a stack exception is issued.

6.4 PROCESS SWITCHING

Using a PCB, the WE 32200 Microprocessor invokes a process switch by automatically saving or restoring a process's context. However, a PCB defines only hardware-context (as described in section 6.2.3). not software-maintained information (i.e., variables and arguments pointed to by the argument pointer and frame pointer) for the process. The PCBP register always contains the address of the PCB of the current process.

To avoid destroying the PCB content on a process switch, a call process (CALLPS or UCALLPS) instruction performs both the save of the previous context and the load of the new process context. The CPU does not accept interrupts until the CALLPS or UCALLPS instruction is completed. This prevents an undefined state between a save and a load. In this state, a PCBP would still point to the PCB for the old (exiting) process. If the system completes a save just as an interrupt occurs, the interrupt-handling scheme causes the saved PCB context to be overwritten. This cannot happen with the WE 32200 Microprocessor.

OPERATING SYSTEM CONSIDERATIONS

Context Switching Strategy

6.4.1 Context-Switching Strategy

The process-switch mechanism uses three PSW parameter bits, R, AR, and I, to control the context-switching strategy:

- The R bit determines if the CPU general registers used by a process should be saved. It also controls block moves.
- The AR bit is used in conjunction with the R bit and if the AR bit is set, the additional 8 registers (r16—23) are saved.
- The I bit determines if a process executes from an initial context or intermediate context. It also affects the setting of the PCBP register.

To save or load the appropriate information on a process switch, the CPU uses the R, AR, and I bits in the PSW of the new or incoming process. The use of the R and I bits is explained next.

R Bit

The use of the R bit is understood by considering two processes: process A as the current or old process and process B as an incoming process. If process B's PSW R bit is set, process B wants to use the general registers, and thus the CPU's general registers are saved in process A's PCB save area for general registers when the process switch occurs. Later, on return to process A, the general registers will be restored for process A. If process B requires block moves, the R bit must be set. On a process switch, where a call process (CALLPS) instruction or simulated CALLPS is performed, the CPU saves the general registers for process A and performs block moves contained in process B if the R bit of process B's PSW is set. When a process switch occurs as a result of the RETPS instruction, the general registers are restored if process A's PSW R bit is set. (This value was copied from process B's PSW when CALLPS occurred).

To generalize, set the R bit in the initial-context PSW of any process that uses the general registers. The R bit setting never changes, even though a process may switch in and out many times.

AR Bit

The AR bit is used with the R bit when switching processes. As in the example above, if process B's PSW AR bit is set, process B wants to use the additional 8 registers (r16 through r23), and the CPU's upper eight general registers are saved in process A's PCB general register save area when the process switch occurs. On return to process A, registers r16 through r23 will be restored.

If process B requires block moves, the AR bit of its PSW must be cleared. On a process switch, where a call process (CALLPS) instruction or simulated CALLPS is performed, the CPU saves only the lower general registers (r0 through r10) for process A (if the R bit is set) and performs block moves contained in process B. It should be noted that process switches requiring block moves cannot use registers r16 through r23.

I Bit

The I bit function identifies whether a process is to start from an initial or intermediate context. It also affects the PCBP register.

The function of the I bit is understood by considering two processes: process A (the current or old process) and process B (the incoming or new process).

- On leaving process A, the CPU always writes the PC, SP, and PSW values, starting at the location pointed to by process A's PCBP and then, stores process A's PCBP on the interrupt stack. On entry to process B, the CPU always reads the PSW, PC, and SP values starting from the location pointed to by the process B's PCBP. These operations are the same for the CALLPS instruction, the UCALLPS instruction, full interrupts, and exceptions that perform a process switch.
- If the I bit is set in process B's PSW, process B's PCBP is incremented by twelve bytes (three words) after the PSW, PC, and SP are loaded, and the I bit is cleared. Incrementing the PCBP guarantees that the initial context loaded in the first step will not be overwritten if process B is interrupted or executes a CALLPS or UCALLPS instruction. Clearing the I bit ensures that the adjustment of the PCBP is done only once. (If this was not done and the I bit was to remain set, and if process B was repeatedly interrupted and resumed, process B's PCBP would be incremented by twelve on each RETPS instruction).
- When process B executes a RETPS instruction, process A's PC, SP, and PSW context is loaded from the locations pointed to by PCBP popped off the interrupt stack.

The main idea is that the effect of the I bit of a given process is not seen until that process is itself interrupted and then returned to by another process.

If the I bit of a process is set when it is entered initially, the process' initial context will be preserved if it is interrupted or if it calls another process. The saved context will be written to and retrieved from the twelve bytes following the initial context. Otherwise, if the I bit is zero initially, the initial context (if writable) will be overwritten in the course of servicing the interrupt or CALLPS or UCALLPS instructions.

Another way to look at the I bit is that if the PSW I bit feature did not exist and the user wanted to modify the PCBP via software to save the initial process context, it could not be guaranteed that the PCBP would be adjusted before another interrupt was taken. Since the I bit adjustment is done in a CPU microsequence, it guarantees that the PCBP adjustment is made while the CPU is immune to interrupts.

The following describes the effects on the PCBP and the initial- and saved-context areas during process switches.

When process A is called initially by the CALLPS or UCALLPS instruction (an explicit process switch), the CPU loads the PCBP register with the starting address (address A) of process A's PCB (see part A of Figure 6-5). The CPU then loads the PSW, the program counter, and the stack pointer with their initial context. Next, if

OPERATING SYSTEM CONSIDERATIONS

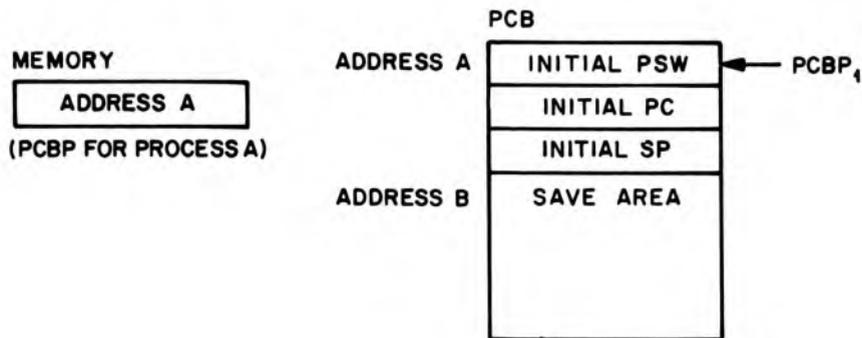
I Bit

the I bit in the PSW is set, the CPU clears the I bit and increments the PCBP register by twelve bytes to the saved-context area (address B) of process A's PCB (see part B of Figure 6-5). This causes any later process switch to save the PSW, PC, and SP values in the intermediate context area instead of overwriting the initial-context values. Process A's initial-context area and its PCBP stored in memory are not affected on this process switch.

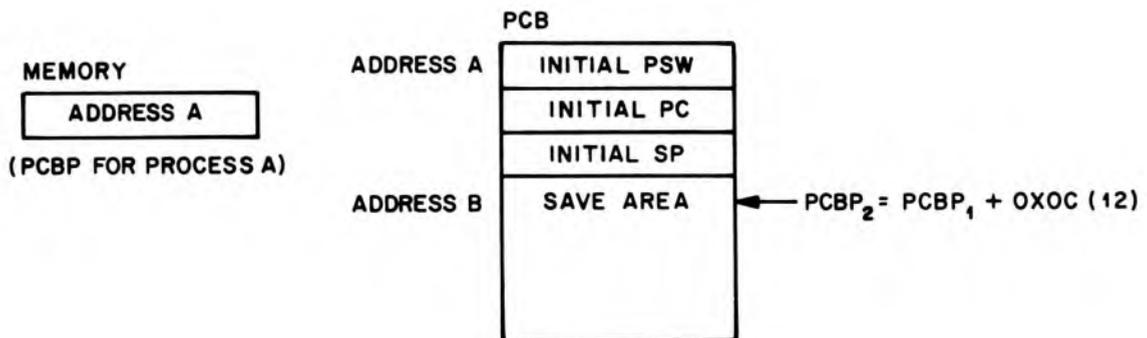
Part A of Figure 6-6 shows the effect on the PCBP and process A's PCB if a process switch occurs before process A is finished. Here, the CPU uses the adjusted PCBP (assuming the I bit was set when process A was initiated) to save the intermediate context of the control registers and store the PCBP on the interrupt stack. This time, the PSW I bit is cleared and the PC points to the next process A instruction.

When the CPU restores process A (see part B of Figure 6-6), the CPU retrieves the PCBP from the interrupt stack. Remember that the PCBP points to the saved-context area (if the initial I bit value was zero, the saved-context area overwrote the initial-context area) and the I bit of the PSW is clear. The CPU then loads the control registers with their intermediate context and process A resumes execution with its next instruction. If the initial value of the I bit for process A was clear, the initial-context area becomes the save area since the PCBP was never adjusted to point to the saved-context area. That is, address B in Figures 6-5 and 6-6 is the same as address A, and the initial-context area no longer exists.

The initial context of a process never changes, provided the initial I bit setting is one. Additionally, the PCBP stored in memory always points to the initial context. This enables an interrupt-handler process to get its PCBP from memory without going through a scheduler. A suspended process restarts from an intermediate context on a return from a full-interrupt handler, certain exception handlers, or the return-to-process (RETPTS) instruction. Also, a process that had an initial I bit value of zero is restarted from an intermediate context on any subsequent CALLPS or UCALLPS instruction after it was first switched to. A process starts from its initial context (initial I bit value is set) whenever a CALLPS or UCALLPS instruction is executed.



A. Context at Start of Switch to Process A

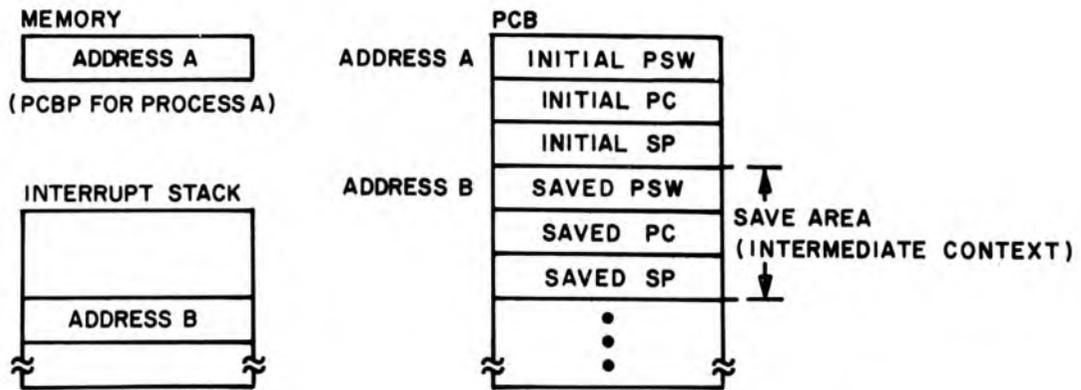


B. Context After Switch to Process A

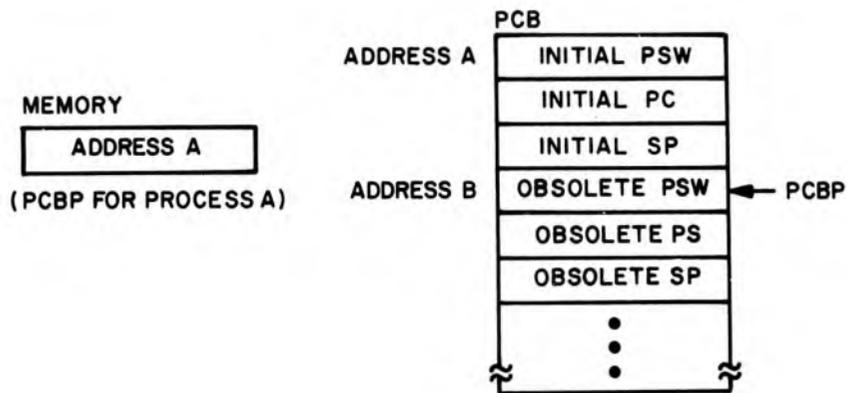
Figure 6-5. A PCB on an Initial Process Switch to a Process

OPERATING SYSTEM CONSIDERATIONS

I Bit



A. Context After Switch to Some Other Process



B. Context After Process A Is Switched Back to and Restored

Figure 6-6. A PCB on a Process Switch During Execution of a Process

6.4.2 Call Process (CALLPS) Instruction

The CALLPS instruction is the process analog of the call procedure (CALL) and save registers (SAVE) instructions that carry out a function call. To execute CALLPS, the CPU must be in kernel mode. In addition, r0 must be preloaded with the new PCBP (address of the PCB for the new process).

The call process instruction performs an explicit process switch. Using process A as the current (old) process and process B as the incoming (new) process, CALLPS performs the following sequential steps:

1. Places the content of r0 (process B PCBP) into register tempa and forces kernel execution level on memory accesses.
2. Saves process A PCBP on the interrupt stack (see Interrupt Stack and ISP under section 6.5.2). If a memory exception occurs when accessing this stack, the CPU issues a reset exception (interrupt stack).
3. Adjusts PC to the address of the instruction that process A would have executed next (PC + 2).
4. Calls the function XSWITCH_ONE() to save process A's context. (All writes are made to the saved context area of process PCB because the I bit of an executing process PSW is always cleared.) If a memory exception occurs on a PCB access, the CPU issues a process exception (old PCB).

XSWITCH_ONE does the following:

- a. Saves the PC (address of the next instruction) in the process A PCB.
 - b. Using tempa as a pointer to the process B PCB, it copies the R and AR bits from the new PSW into the R and AR bits of the current PSW (these bits are used later).
 - c. Stores the current PSW in the process A PCB and writes 0, 0, 1 to the ISC, TM, and ET fields, respectively, of the saved PSW. The SP is also stored in the process A PCB.
 - d. Writes r0 through r10 to the general register save area of the process A PCB if the R bit of the process B PSW is set. Otherwise, these registers are not saved.
 - e. Writes r16 through r23 to the general register save area of the process A PCB if the AR bit of the process B PSW is set. Otherwise, these registers are not saved.
 - f. Returns control to CALLPS.
5. Calls the function XSWITCH_TWO() to load the process B context. If a memory exception occurs when accessing its PCB, the CPU issues a process exception (new PCB).

OPERATING SYSTEM CONSIDERATIONS

Return-to-Process (RETPTS) Instruction

XSWITCH_TWO does the following:

- a. Loads the PCBP from tempa (which contains the process B PCBP value).
 - b. Reads in the new PSW and clears TM bit to 0. Next, it loads the new PC and SP. The PC now contains the address of the first instruction for process B.
 - c. Tests the PSW I bit. If the I bit is set, the I bit is cleared, and the PCBP is adjusted to the saved context area of the process B PCB.
 - d. Flushes the instruction cache if the CFD (cache flushing disabled) bit of the PSW is cleared (0).
 - e. Returns control to CALLPS.
6. Sets the ISC, TM, and ET bits of the PSW to 7, 0, and 3, respectively.
 7. Calls the function XSWITCH_THREE() for block moves.

XSWITCH_THREE does the following:

- a. Tests the R and AR bits in the PSW.
 - If the R bit is set and the AR bit is cleared, the block move information is loaded from the block move areas of the process B PCB. For each block to be moved, it preloads r0 with the starting address of the block move area, r2 with the size of the block (number of words to be moved), and r1 with the destination of the move. Then it executes a move block (MOVBLW) instruction. After each MOVBLW instruction is executed, the next destination address is placed in r1, and r2 is decremented by the number of words moved by that instruction. This continues until the contents of r2 is zero.
 - If the R bit is cleared and/or the AR bit is set, no block moves are performed.
 - b. Increments r0 by 4.
 - c. Returns control to CALLPS.
8. Releases kernel execution level on memory accesses and Process B begins executing.

6.4.3 Return-to-Process (RETPTS) Instruction

The RETPTS instruction restores a process from its interrupted state and may be executed only when the CPU is in kernel mode. An attempt to execute the RETPTS instruction while not in kernel mode generates a normal exception (privileged instruction). RETPTS is the process analog of a function return that uses the restore registers (RESTORE) and return-from-procedure (RET) instructions. Again, the R, AR, and I bits in the PSW determine the context-switching strategy.

The CALLPS and RETPTS instructions act similarly, except the RETPTS does not save the context of the exiting process. For this discussion, process A is the returned-to-process. RETPTS performs the following sequential steps:

OPERATING SYSTEM CONSIDERATIONS

Return-to-Process (RETPS) Instruction

1. Forces kernel execution level on memory access and moves the Process A PCBP from the interrupt stack into register tempa. If a memory exception occurs on the stack access, the CPU issues a reset exception (interrupt stack fault).
2. Loads the PSW R and AR bits with the R and AR bits from tempa.
3. Calls XSWITCH_TWO() to restore the Process A context. If a memory exception occurs when accessing its PCB, a process exception (new PCB) is issued. (The PCBP for process A is still at the top of the interrupt stack.)

XSWITCH_TWO does the following:

- a. Loads the PCBP from tempa.
 - b. Loads the PSW from the PCB, writes a 0 to the TM bit, and then loads the PC and SP. Because this is a return process, the I bit is cleared and all control registers are loaded from the saved context area of its PCB.
 - c. Returns control to RETPS.
4. Sets the ISC, TM, and ET bits of the PSW to 7, 0, and 3, respectively.
 5. If the R bit is set and the AR bit is cleared, calls XSWITCH_THREE() to perform any block moves.

XSWITCH_THREE does the following:

- a. Tests the R and AR bits in the PSW.
 - If the R bit is set and the AR bit is cleared, the block move information is loaded from the block move areas of process B's PCB. For each block to be moved, r0 is preloaded with the starting address of the block move area, r2 with the size of the block (number of words to be moved) and r1 with the destination of the move. Then, executes a move block (MOVBLW) instruction. After each MOVBLW instruction is executed, the next destination address is placed in r1, and r2 is decremented by the number of words moved by that instruction. This continues until the contents of r2 is zero.
 - If the R bit is cleared and/or the AR bit is set, no block moves are performed.
 - b. Incremented r0 by 4.
 - c. Returns control to RETPS.
6. If the R bit is set, RETPS loads r0—r10 from the general register save area of process A PCB.
 7. If the AR bit is set, RETPS loads r16—23 from the general register save area of process A PCB.
 8. Releases kernel execution level on memory accesses and process A resumes execution.

6.4.4 User Call Process (UCALLPS) Instruction

The UCALLPS instruction, when enabled (PSW<EX> == 1), performs a process

OPERATING SYSTEM CONSIDERATIONS

User Call Process (UCALLPS) Instruction

switch, saving the current process, pushing its PCBP onto the interrupt stack, and entering a new process. The PCBP of the new process is read from memory address 0x488, which is the last location in the full-interrupt PCBP table. Using process A as the current (old) process and process B as the incoming (new) process, UCALLPS performs the following sequential steps:

1. Places the contents of address 0x488 (process B PCBP) into register tempa and forces kernel execution level on memory accesses.
2. Saves process A PCBP on the interrupt stack. If a memory exception occurs when accessing this stack, the CPU issues a reset exception (interrupt stack fault).
3. Adjusts PC to the address of the instruction that process A would have executed next (PC + 2).
4. Calls the function XSWITCH_ONE() to save the process A's context. If a memory exception occurs on a PCB access, the CPU issues a process exception (old PCB).
5. Calls the function XSWITCH_TWO() to load process B's context. If a memory exception occurs when accessing its PCB, the CPU issues a process exception (new PCB).
6. Sets the ISC, TM, and ET bits of the process B's PSW to 7, 0, and 3, respectively.
7. Releases kernel execution level on memory accesses and process B begins executing.

The functions XSWITCH_ONE(), XSWITCH_TWO(), and XSWITCH_THREE() operate in the same manner as described above for the CALLPS instruction. When the UCALLPS instruction is disabled (PSW<EX>==0) and executed, an illegal opcode exception is generated.

6.5 INTERRUPTS

When an external device requests an interrupt, a CPU temporarily stops its current execution and jumps to code that services the interrupt. On completion of the interrupt handler code, execution resumes at the point where the interrupt occurred. An interrupt mechanism performs the execution switch.

6.5.1 Interrupt-Handler Model

An interrupt handler may be a process switch or a gate-like sequence. In most existing architectures, an interrupt handler is a function that is invoked on an interrupt. The function executes as part of the interrupted process context or as part

of a system-wide context. Although easy to implement, the function call does not isolate interrupt handlers, execute them at any level, or return from them to a different process.

The *WE 32200* Microprocessor uses either the process switch or a gate-like sequence for its interrupt mechanism. In the process switch model, an interrupt (called a full-interrupt in this case) causes an implicit process switch to a new process. In the gate-like sequence model, an interrupt (called a quick-interrupt in this case) causes a save of the PC/PSW pair onto the interrupt stack (whereas a gate instruction saves the PC/PSW pair on the execution stack). When full-interrupts are used, the CPU interrupt mechanism meets the isolation and execution-level requirements because each interrupt handler is a separate process with its own execution stack. The CPU tracks full-interrupt nesting in such a way that a full-interrupt handler at any priority level may preempt the original process, thus meeting the return requirement. With the quick-interrupt feature, interrupts can be handled as described above for most existing architectures.

For efficient operation, the implicit process switch on a full-interrupt does the following:

- Minimizes the loading and saving of an interrupt handler's context.
- Allocates only one stack to each interrupt handler.

6.5.2 Interrupt Mechanism

There are three functions of the interrupt mechanism:

- Determining if there is an interrupt.
- Determining how an interrupt request is acknowledged and what the interrupt-ID value is.
- Saving the old context and bringing in a new context.

The first function involves checking the $\overline{\text{NMINT}}$ and IPL0—IPL3 signals and the IPL field of the PSW. The second function involves the $\overline{\text{NMINT}}$, $\overline{\text{AVEC}}$, IPL0—IPL3 , and $\overline{\text{INTOPT}}$ signals, and an interrupt acknowledge or autovector interrupt acknowledge bus cycle. The final function involves the QIE field of the PSW and a quick-interrupt or a full-interrupt.

The following algorithm describes the interrupt behavior. The notation used is:

- $\text{INT} = 1$ if there is to be an interrupt
- ID is the value of the interrupt-ID in the on-interrupt microsequence
- NMI, $\overline{\text{INTOPT}}$, and $\overline{\text{AVEC}}$ represent the complements of the values of the nonmaskable interrupt ($\overline{\text{NMINT}}$), interrupt option ($\overline{\text{INTOPT}}$), and autovector ($\overline{\text{AVEC}}$) signals, respectively.

OPERATING SYSTEM CONSIDERATIONS

Interrupt Mechanism

```
I = 0;
if(NMINT==0) {
    INT = 1;
    ID = 0;
}
else if((requested_interrupt_level)>(PSW<IPL>)) {
    INT = 1;
    if(AVEC==0)
        ID = (INTOPT concatenated with interrupt request level);
    else ID = (value fetched in interrupt acknowledge cycle);
}
if(INT==1) {
    call on-interrupt sequence;
}
else {
    no interrupt;
}
```

An interrupt occurs if the priority level requested is greater than the priority level in the IPL field of the PSW. Thus, if $PSW<IPL>=15$, no interrupts will be acknowledged (except the nonmaskable interrupt).

After acknowledging an interrupt, the CPU performs its on-interrupt microsequence which is similar to a CALLPS instruction for a full-interrupt. For a quick-interrupt, the old PC/PSW pair are saved and the new PC/PSW pair are fetched.

When a full-interrupt activates an interrupt-handler process, the interrupt handler starts from its initial state. However, unlike ordinary processes, this initial context consists of only the three registers and the stack bounds; general registers are not loaded for any process starting from an initial context.

A higher priority interrupt may interrupt the current interrupt-handler process. When this happens, its intermediate context is stored in the save area of the PCB, rather than the initial-context area. Thus, the interrupted handler can resume execution from that point later.

The I bit in the process PSW controls which starting point and context to use (see section 6.4.1).

To return from a full-interrupt, an interrupt-handler process executes a RETPS instruction. This process switch does not save the state of the exiting interrupt-handler process (see section 6.4.3).

When a quick-interrupt activates an interrupt handler, the current PC and PSW values are stored on the interrupt stack. The PC and PSW registers are then loaded with initial information for the interrupt handler.

To return from a quick-interrupt, an interrupt handler should execute a RETQINT instruction.

Full-Interrupt Handler's PCB

Before an interrupt handler is activated, its PCBP points to the initial-context area of its PCB, which contains initial values for the PSW, PC, and SP. The IPL field in this PSW is usually set at least as high as the priority level of the device associated with the interrupt

handler (Interrupt priority levels range from 0, the lowest, to 15, the highest, which indicates "no interrupts.") In addition, the I bit in this PSW must be set. If the interrupt handler wants to use the general registers, the PSW R bit must be set. If the interrupt handler wants to use r16 through r23, the PSW AR bit must be set.

If the new PSW has its I bit set when an interrupt handler is activated, the I bit in the PSW register is cleared and the PCBP register is adjusted to the saved-context area of the handler's PCB. The save area is used to store the handler's control registers if another interrupt occurs.

If the PSW's I bit is set, an interrupt-handler process always starts from the same initial state whenever it is initially activated because its initial-context values never change. However, after being interrupted, the saved-context area always reflects its state at the time of the interrupt. Thus, the restored interrupt handler starts from the appropriate intermediate state.

An interrupt handler's MMU map specification, if maintained in the PCB block-move areas, is used when loading an initial context or restoring an intermediate context. Therefore, the user must ensure that the operating system restores the map data to its initial state before a return-from-interrupt. This can be done by maintaining appropriate R bit and AR bit values in the PCBs involved.

Interrupt Stack and ISP

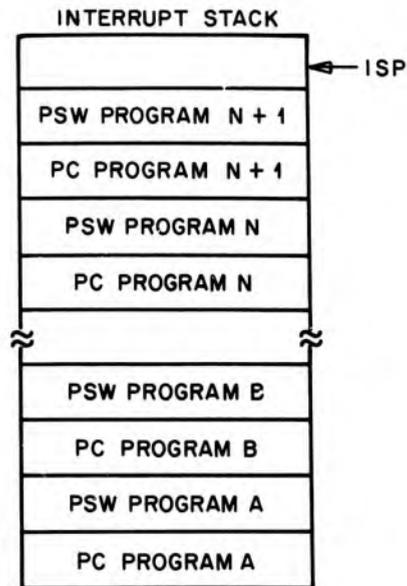
The user must design the operating system to allocate memory space for one interrupt stack. This system data structure enables the CPU to track the nesting of interrupt handlers and active processes and is never used as an execution stack.

The CPU uses its interrupt stack pointer (ISP) register to access the interrupt stack. This privileged register always contains the address of the top of the stack. When it saves the current PCBP, a CALLPS or on-interrupt microsequence automatically increments ISP by four. A RETPS decrements ISP by four when it restores the PCBP. For quick-interrupts, the ISP is automatically incremented by eight. A RETQINT instruction decrements ISP by eight. An attempt to write this register other than in kernel level causes a normal exception (privileged register).

At any level of interrupt handling, the interrupt stack contains the PCBPs or PC/PSW pair for all lower priority interrupt handlers that were interrupted while executing. The entry at the bottom of the stack is the PCBP or PC/PSW pair for the process that was interrupted by the first interrupt handler (see Figure 6-7).

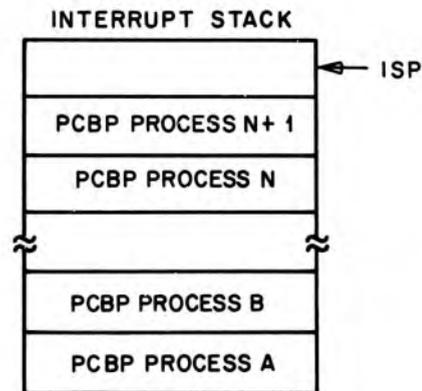
OPERATING SYSTEM CONSIDERATIONS

Interrupt Stack and ISP



Program B interrupted program A.
Program N+1 is last program interrupted.

A). Quick-Interrupt



Process B interrupted process A.
Process N+1 is last process interrupted.

B). Full-Interrupt

Figure 6-7. Interrupt Stacks (Quick- and Full-Interrupts)

Interrupt Vector Table

The user must provide interrupt vector tables for full- and quick-interrupts, depending

on how interrupts are to be handled (PCBP's and/or PC/PSW pairs). Figure 6-1 shows the memory locations where interrupt PCBP's and PC/PSW pairs must be stored. If the nonmaskable and autovector interrupts are not used, those locations can be used to store the PCBP's for device-interrupt handlers. The full-interrupt vector table starts at location 140 (8C hex) to store the PCBP (up to 256 PCBP's) for each interrupt handler and the quick-interrupt vector table starts at location 1164 (48C hex) to store PC/PSW pairs (up to 256 pairs) for each interrupt handler. Commonly, each device that requests an interrupt may require a different handling routine. The CPU locates the appropriate interrupt handler by using an 8-bit code (interrupt-ID) as an offset into the vector tables. The code is used to form the address ($140 + 4 * \text{interrupt-ID}$) to obtain the PCBP for a full-interrupt handler or the address ($1164 + 8 * \text{interrupt-ID}$) to obtain the PC/PSW pair for a quick-interrupt handler.

Note: The use of the last two entries of the full-interrupt table are for normal exceptions and the UCALLPS instruction.

6.5.3 On-Interrupt Microsequence

The on-interrupt microsequence is a sequence of actions built into the WE 32200 Microprocessor that responds to interrupts. The on-interrupt microsequence handles both full- and quick-interrupts. For full-interrupts, the CPU performs an implicit process switch. For quick-interrupts, the CPU performs a PC/PSW pair save and loading sequence. Here, process A is the interrupted process and process B is the interrupt handler. (See section 6.4.2 for a description of the XSWITCH functions.)

The microsequence performs the following sequential steps:

1. Writes the interrupt-ID to register tempa. If a memory exception occurs, the CPU generates a stack exception (interrupt-ID fetch).
2. Forces kernel level on memory accesses.
3. Skips to step 12 if it is a quick-interrupt (the PSW's QIE field is set).
4. Performs steps 5 through 11 for a full-interrupt.
5. Forms an index, $140 + 4 * \text{tempa}$, which is written to tempa. This index is used to locate the PCBP of the appropriate interrupt handler.
6. Stores the process A PCBP on the interrupt stack. If a memory exception occurs on this stack operation, the CPU generates a reset exception (interrupt stack).
7. Calls XSWITCH_ONE() to store the process A context in the saved-context area of its PCB and then writes 0, 0, 1 to the ISC, TM, and ET fields, respectively, of the saved PSW. If any of these operations causes a memory exception, the CPU generates a process exception (old PCB).
8. Calls XSWITCH_TWO() to load process B's PCBP and new PC, PSW, and SP values from the initial-context area of its PCB. A memory exception on any XSWITCH_TWO operation causes a process exception (new PCB). If it is set, the PSW I bit is cleared and the PCBP adjusted to the saved-context area of process B PCB.

OPERATING SYSTEM CONSIDERATIONS

Returning From an Interrupt

9. Sets the ISC, TM, and ET bits of the PSW to 7, 0, and 3, respectively.
10. Calls XSWITCH_THREE() to make any necessary block moves. A memory exception here causes a process exception (new PCB).
11. Releases kernel level on memory accesses. For full-interrupts, this is the last step of the on-interrupt microsequence.
12. Resumes quick-interrupt here.
13. Forms an interrupt-ID value via an interrupt acknowledge bus cycle and stores it in tempa ($\text{tempa} = \text{interrupt-ID} * 8$).
14. Forms an index, $1164 + \text{tempa}$, which is written to tempa. This index is used to locate the PSW and PC of the appropriate interrupt handler.
15. Writes the values 2, 0, and 0 to the ISC, TM, and ET bit fields, respectively.
16. Forces kernel level on memory accesses, and saves the old PC and PSW on the interrupt stack.
17. Writes the values 1, 0, and 0 to the ISC, TM, and ET bit fields, respectively, and fetches the new PC and PSW values from the interrupt table. If a memory exception occurs from here until the end of the sequence, a reset (system data) is generated.
18. Adjusts the previous execution level in the new PSW with the value of the old PSW's current execution level. The new PSW field bits (EA, EX, X, CFD, QIE, CD, OE, NZVC, TE, IPL, and CM) are adjusted by the values indexed by tempa.
19. Places the new PC values into tempa.
20. Finishes the push of the old PC and PSW by incrementing ISP by 8. The values 7, 0, and 3 are written to the ISC, ET, and TM bit fields, respectively. The value of tempa is written to PC.
21. Releases kernel level on memory accesses.

Process B (the interrupt handler) takes its priority level from the PSW that was just loaded and starts executing. Execution may be interrupted only by a higher priority interrupt (higher than the IPL value of the PSW).

6.5.4 Returning from an Interrupt

Full-Interrupts

A full-interrupt handler may restore the interrupted process or may return to another process after servicing the interrupting device. To accomplish either process switch, the full-interrupt handler must contain a return-to-process (RETPTS) instruction. Unlike the call process, RETPTS does not save the exiting process (interrupt handler) context.

Note: If a full-interrupt handler is not to return to the process interrupted, the interrupt-handler routine must alter the interrupt stack before a RETPTS instruction. The PCBP for the process returned to must replace the PCBP that was saved for the

interrupted process.

The PCBP of the process to which the return-from-interrupt occurs is removed from the interrupt stack. The full context of the returning process is restored from its PCB, and any required map changes are made (block moves are performed).

Quick-Interrupts

A quick-interrupt handler returns to the function that was interrupted (i.e., restores the PC and PSW registers with the values popped off the interrupt stack). To return from a quick-interrupt handler, the handler must execute a return-from-quick-interrupt (RETQINT) instruction.

6.6 EXCEPTIONS

An exception is an error condition, other than an interrupt, that requires special processing for recovery. That is, an exception mechanism is needed to correct the error condition so that normal processing can continue. Exceptions are caused by three types of events:

- **Internal faults** – error conditions detected by the CPU during instruction execution. The fault handler for such events may restart the instruction that caused the fault.
- **External faults** – error conditions detected outside the CPU and conveyed to it over its fault input. The CPU recognizes the fault during instruction execution and the appropriate fault handler may then restart the execution.
- **Traps** – internal error conditions detected by the CPU at the end of an instruction. After the trap is handled, execution may resume with the next instruction.

The exception mechanism for the *WE 32200* Microprocessor is implemented through microsequences. Depending on the level of exception severity, the microprocessor responds with the appropriate microsequence to facilitate correction of the condition.

6.6.1 Levels of Exception Severity

The CPU recognizes four levels of exception severity, with zero as the highest level. It uses the ET (exception type) and ISC (internal state code) fields of the PSW to identify the severity and type of exception, respectively. Because all exception microsequences preserve the ET and ISC values in the current PSW, the incoming exception handler may use them. The ET value gives the class of exception and corresponds to its severity level, while ISC distinguishes among error conditions of the same class. During normal program execution, the ET field is 3 and the ISC field is 7. Table 6-3 identifies the severity levels for exceptions. The meaning of the ISC values for each exception severity level is identified later.

6.6.2 Exception Handler

On-stack, on-process, on-reset, and the process-switch-like on-normal exception microsequences do not use the ET and ISC values, but preserve them for an incoming exception handler. The gate-like on-normal exception microsequence uses

OPERATING SYSTEM CONSIDERATIONS

Exception Handler

them to locate the appropriate handling routine, as well as preserving them.

ET	Level	Processor Response
0	Reset	Executes on-reset microsequence; highest severity level.
1	Process	Executes on-process exception microsequence.
2	Stack	Executes on-stack exception microsequence.
3	Normal	Executes on-normal exception microsequence; lowest severity level.

The ET and ISC values help identify the task an exception handler must perform. What an exception handler should do with the ET and ISC values or how it should handle the error depends on the needs of the system. In general, if computation can continue, resumption of the process may be chosen. However, if an error is too serious for the original process to continue its computations, the exception handler should ask the scheduler to terminate the bad process.

The operating system designer must provide an exception vector table. Figure 6-1 shows the addresses where the vector tables reside. All locations must be filled with either PCBP's or the address of the handling-routine table (for normal exceptions).

6.6.3 Exception Microsequences

The CPU's microsequences enable it to execute an appropriate sequence of actions when it detects an exception. By design, an exception that occurs during one of these microsequences has a higher severity level than if it occurred at another time. Such an exception, therefore, stops the current microsequence, and the CPU starts performing a higher level microsequence. Thus, the CPU can increase the level of exception severity.

Any exception during an on-reset sequence (the severest exception level) causes the CPU to restart the on-reset sequence. Trying to recover from the exception, the CPU goes into an infinite loop and consequently can recover from transient faults.

The sections that follow describe the error conditions for each class of exception and the response of the microsequence. When describing this response, process A is the process that caused the exception and process B is the exception handler. In general, a normal exception results in a simulated gate instruction, but a stack, process, or reset exception causes an implicit process switch. Descriptions of microsequences follow the operating system instructions at the end of this chapter.

Normal Exceptions

This group of exceptions includes most of those that occur in other microprocessor architectures. Table 6-4 identifies the ISC and the cause of each normal exception.

If the EX bit in the PSW is cleared, the on-normal exception sequence is identical to that of GATE, except that 0 is used (instead of r0) as the offset into the first-level

table (offset1), and the ISC value (instead of r1) is used as the offset into the second-level table (offset2). A RETG instruction can be used to return from this gate-like normal exception.

If the EX bit is set (1), the on-normal exception sequence is identical to that of a process switch, except that the PCBP is located at the second to last entry of the full-interrupt table, address 0x484 (see Figure 6-1). A RETPS instruction can be used to return from this process-switch-like normal exception.

When a normal exception occurs, the CPU executes the on-normal exception microsequence. After some set-up operations, the microsequence enters the gate instruction at its second entry point (see section 6.3.2). Using the ISC code, this simulated GATE finds the appropriate exception-handler function and transfers control to it. Both the microsequence and the exception handler execute within the process that caused the error condition.

To locate the exception handler, GATE requires two implied operands that serve as indexes into the pointer table and the correct handling-routine table. (See section 6.3.1 for a description of these tables). For GATE index1, the microsequence supplies the value of 0. For GATE index2, it uses the ISC in the saved PSW, shifting three bits toward the most significant bit (MSB). As shown in Figure 6-8, this shifted ISC value forms an index into the handling-routine table. Thus, a normal exception results in a controlled transfer to the corresponding exception handler. On completion of the on-normal exception microsequence, the ISC, TM, and ET fields of the PSW presented to the exception handler contains 7, 1, 3, respectively.

OPERATING SYSTEM CONSIDERATIONS

Normal Exceptions

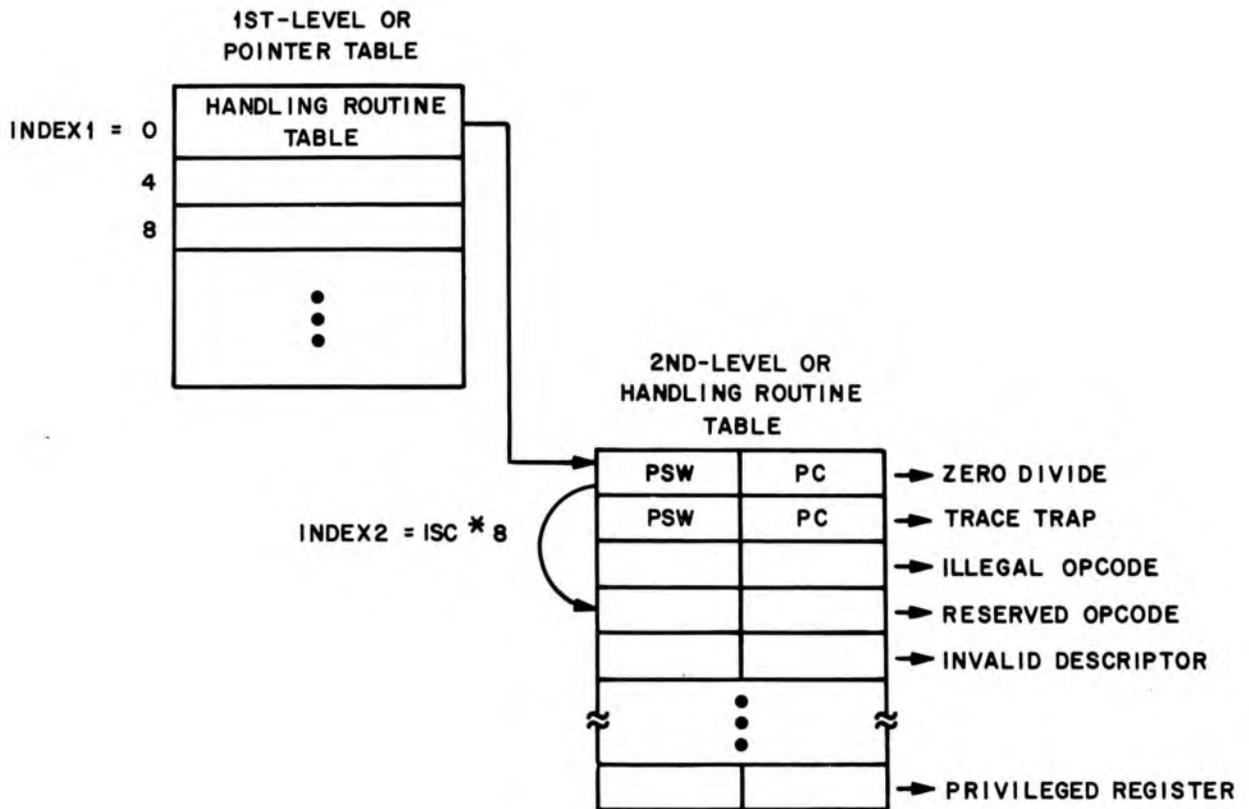


Figure 6-8. On-Normal (Gate-like) Exception Indexing

Because a normal-exception handler executes as part of process A, it uses the same execution stack. After handling the error condition, a normal-exception handler must execute an RETG to restore control to process A.

OPERATING SYSTEM CONSIDERATIONS
Normal Exceptions

Table 6-4. Normal Exception (ET=3)		
ISC	Exception	Cause
0	Integer zero divide (internal fault)	An attempt to divide by zero. This exception is always enabled.*
1	Trace (trap)	Normal response to the end of an instruction if the TE bit is set in the PSW.
2	Illegal opcode (internal fault)	Use of an undefined opcode.
3	Reserved opcode (internal fault)	Use of an opcode reserved for future implementation. This is also the normal response to the extended opcode (EXTOP) instruction.
4	Invalid descriptor (internal fault)	Use of literal or immediate address mode for a destination operand; instruction's opcode requests the effective address of a literal, immediate, or register operand.*
5	External memory (external fault)	An exception when accessing external memory. Also can occur internally for misaligned words/halfwords when arbitrary byte alignment is disabled.
6	Gate vector (external fault)	A memory exception when accessing the gate tables as part of a GATE.
7	Illegal level change (internal fault)	An attempt to increase the current execution privilege level on a RETG.
8	Reserved data type (internal fault)	Use of an operand type that is not defined for the expanded-operand type address mode.*
9	Integer overflow (internal fault)	An attempt to write data into a destination that is too small. This exception is enabled when the OE bit is set in the PSW.**
10	Privileged opcode (internal fault)	An attempt to execute an opcode defined for kernel level at a different execution level.
11—13	Unused	—
14	Breakpoint (trap)	Normal response to a breakpoint trap (BPT) instruction.
15	Privileged register (internal fault)	An attempt to write a privileged register when not in kernel level.*

*This exception sets the condition flags as if the instruction was successfully completed.

** Before the overflow trap occurs, the CPU may execute the next instruction after the one that caused the overflow.

OPERATING SYSTEM CONSIDERATIONS

Stack Exceptions

Stack Exceptions

Table 6-5 lists the ISC and the cause of each stack exception.

On a stack fault, the memory exception occurs when SP is used as an operand. Thus, the CPU first detects a normal exception and then detects the stack exception while executing the implicit GATE (system call). In effect, the CPU automatically increases the severity level from a normal exception to a stack exception.

A stack exception occurs because process A (the process at fault) cannot use its execution stack. As a result, a stack exception cannot be handled as part of process A (unlike normal exceptions). Instead, the CPU performs the on-stack exception microsequence, which performs a process switch and thus provides the exception handler with a new execution stack.

The interrupt-ID fetch exception does not involve the stack, but it is treated as a stack exception since it is system wide. Thus, no context information is lost.

The on-stack exception microsequence saves the process A's PCBP on the interrupt stack, stores the control registers in its PCB, and loads a new PCBP (for process B) from location 136 (0x88). Then it carries out an implicit process switch to the stack-exception handler, process B. Although the microsequence does not use the ISC value, it preserves this value across the process switch. On completion of the microsequence, the ISC field in the PSW saved for process A still contains the code for the stack exception, and the TM and ET fields contain 0 and 3, respectively. When process B starts executing, the PSW's ISC, TM, and ET fields contain 7, 0, 3, respectively.

Because a stack-exception handler is implemented as a process, the user may want to prevent interrupts from entering the handler. Entry prevention is accomplished by raising the interrupt priority level (the IPL field of its PSW) to 15, thus disabling all interrupts except a nonmaskable interrupt. Such a stack-exception handler should execute only a few instructions.

A stack-exception handler can correct a stack-bound or stack-fault problem by:

- Increasing the size of the stack for the process.
- Bringing in a missing page of the stack (in demand-paging systems).

Table 6-5. Stack Exceptions (ET=2)		
ISC	Exception	Cause
0	Stack bound (internal fault)	An SP value outside the upper or lower stack bound on a system call (a GATE or on-normal exception microsequence).
1	Stack (external fault)	A memory exception when storing the PC or PSW on the execution stack during a system call.
3	Interrupt ID fetch (external fault)	A memory fault when accessing the interrupt vector tables during the on-interrupt microsequence.

Process Exceptions

A process exception is generated if the process receives a memory exception signal on a PCB access. The exception is local to process A (the process that caused it) and implies a severe error condition. The ISC field of process A's PSW is presented to the exception handler (process B) and identifies the condition that caused the exception. Table 6-6 lists the ISC and the cause for each process exception.

When a process exception occurs, the CPU executes its on-process exception microsequence, an implicit process switch. Because the error condition signifies that process A's PCB cannot be accessed, its context cannot be saved. The microsequence stores process A's PCBP on the interrupt stack and loads process B's PCBP from location 132 (0x84). The ISC field from process A's PSW is copied before the process B's context is loaded. When process B begins executing, its PSW contains the code for the exception condition, and the TM and ET fields contain 0 and 3, respectively.

Because the CPU could not save the process A's hardware context, process B normally kills process A. However, the CPU can identify an old (good) process from its PCBP on the interrupt stack. If the exception is a new PCB exception, process A's PCBP is at the top of the interrupt stack. If it is an old PCB exception and a process switch from a third process, C, had been made, then process C's PCBP is the second element from the top of the stack. In either case, process B could restart the last good process because its context was not lost.

Table 6-6. Process Exceptions (ET=1)		
ISC	Exception	Cause
0	Old PCB (external fault)	A memory exception when accessing the PCB for the exiting process on a process switch.
1	Gate PCB (external fault)	A memory exception when accessing the PCB for a stack bounds check during a GATE.
4	New PCB (external fault)	A memory exception when accessing the PCB for the new process during a process switch.

OPERATING SYSTEM CONSIDERATIONS

Reset Exceptions

Reset Exceptions

A reset exception implies an error condition in accessing critical system data and requires restarting the system. On a reset exception, the CPU acts as if an external reset occurred. The ISC field in the PSW of the current process identifies the condition as an internal error or external request for a system reset. Table 6-7 lists the ISC and cause of the reset exceptions.

On a reset exception, the CPU performs an implicit process switch. It executes the on-reset microsequence after first disabling the memory management unit. The microsequence picks up a new PCBP from physical address location 128 (0x80) and loads the reset-handler process (process B). When process B begins executing, its PSW contains the code corresponding to the condition that caused the reset exception, and its TM and ET fields contain 0 and 3, respectively.

Process B should restart the system (i.e., reinitialize the system), possibly after checking the validity of system data.

Table 6-7. Reset Exceptions (ET=0)

ISC	Exception	Cause
0	Old PCB (external fault)	A memory exception when accessing the PCB of a process-exception handler.
1	System data (external fault)	A memory exception when accessing an interrupt vector or while processing an exception.
2	Interrupt stack (external fault)	A memory exception when accessing the interrupt stack while processing an exception.
3	External reset (external fault)	Normal response to an external (system) reset signal.
4	New PCB (external fault)	A memory exception when accessing the PCB of an exception-handler process.
6	Gate vector (external fault)	A memory exception when accessing a gate table while processing a normal exception. (Here, the PSW's ET field contains 0. If ET is 3, a gate vector exception is treated as a normal exception because it occurred during a GATE instruction, rather than as part of the on-normal exception microsequence.)

6.7 MEMORY MANAGEMENT FOR VIRTUAL MEMORY SYSTEMS

When a virtual memory system is used for a WE 32200 Microprocessor based system, a memory management unit (MMU) is required. The main function of an MMU is to translate virtual addresses into physical addresses. The MMU has the additional responsibility of protecting the system memory space.

The WE 32201 Memory Management Unit was developed to complement the

WE 32200 Microprocessor for creation of a virtual memory system. Memory performance is greatly improved by an on-chip 4-Kbyte, 2-way, set-associative instruction/data cache, which returns data with zero wait states on virtual and physical memory accesses. This section describes the features of the MMU and data cache that are important for system design. A complete technical summary of the MMU is provided in the *WE[®] 32201 Memory Management Unit Information Manual*.

The *WE 32201* Memory Management Unit divides the virtual address space into four 1-Gbyte sections. Each section may consist of up to 8K segments, where each segment is a maximum of 128 Kbytes long. Segments consist of pages that may each be 2, 4, or 8 Kbytes long. The MMU can support one page size at a time, set in the page size (PS) field of the configuration register. Segments are a multiple of pages and, therefore, always start on a page boundary.

6.7.1 Virtual Address Fields

The *WE 32201* MMU divides virtual addresses into four fields: a section identifier (SID) field, which specifies the section of virtual address space; a segment select (SSL) field, which specifies the segment within the section; a page select (PSL) field, which specifies the page in the segment; and a page offset (POT) field, which specifies the byte in the page. The formats of a virtual address in a paged segment are shown on Figures 6-9 through 6-11.

Bit	31	30	29	17	16	11	10	0
Field	SID		SSL		PSL		POT	

Figure 6-9. Virtual Address Fields for 2K Page Size

Bit	31	30	29	17	16	12	11	0
Field	SID		SSL		PSL		POT	

Figure 6-10. Virtual Address Fields for 4K Page Size

Bit	31	30	29	17	16	13	12	0
Field	SID		SSL		PSL		POT	

Figure 6-11. Virtual Address Fields for 8K Page Size

The MMU performs address translation by using descriptors that contain the information necessary for segment and page mapping. The MMU has two types of

OPERATING SYSTEM CONSIDERATIONS

Virtual Address Fields

descriptors: segment descriptors (SD) for mapping segments and page descriptors (PD) for mapping pages. These descriptors are stored in physical memory in descriptor tables (segment descriptor tables (SDT) for SDs and page descriptor tables (PDT) for PDs. There is one SDT for each section and one PDT for each paged segment. A paged SD contains an address that points to the base address of the associated page descriptor table (PDT). A PD contains a page-base address that is concatenated with the page offset (POT from the virtual address) to form the physical address.

Other fields contained in SDs and PDs provide functions other than address translation. For example, the access fields in the SDs are used by the MMU to enforce protection of system memory. This field and other fields are described later in this section.

Figure 6-12 is a model showing showing how a virtual address is translated to a physical address. The SID field is used to find the base address of the required SDT (the base address of the SDT for each section is stored in the MMU). This address and the SSL field are combined to index an SD within the SDT. The address in the SD is used as the base address of the PDT. This address is combined with the PSL field to index a PD. This PD contains the starting address of the paged segment that is concatenated with the POT field to form the required physical address.

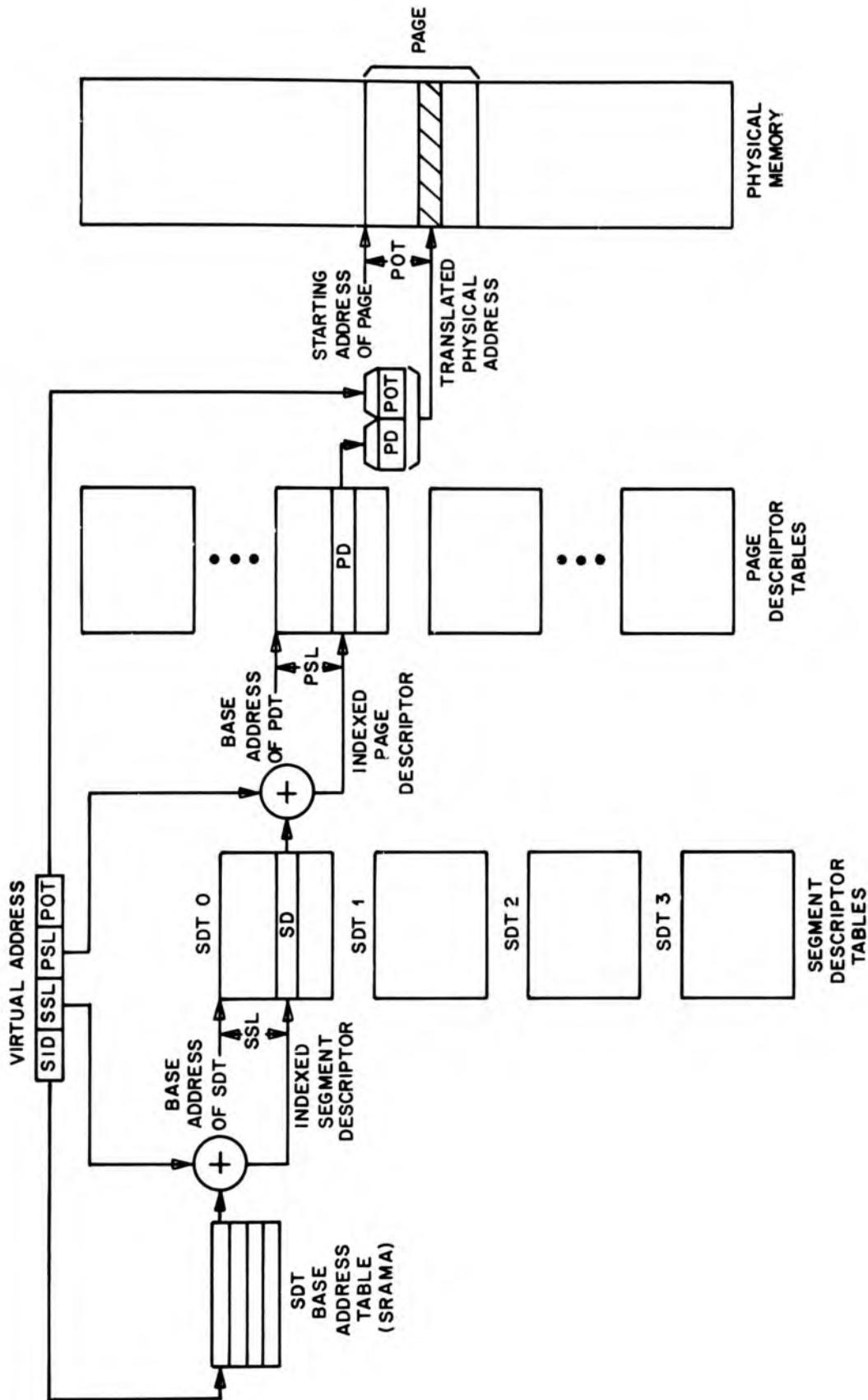


Figure 6-12. Virtual to Physical Address Translation

OPERATING SYSTEM CONSIDERATIONS

Initializing the Memory Management Unit

6.7.2 Initializing the Memory Management Unit

The operating system is required to initialize the MMU. Typical MMU initialization consists of:

- Defining physical memory with segment descriptor tables and page descriptor tables for each process
- Writing SDT addresses and length into MMU section RAMs (SRAMs)
- Writing the configuration register (CR) to define pages as 2, 4, or 8 Kbytes.

The operating system should also set up the block-move area of the PCB for each process in the system. Block moves can be used to set the MMU section RAMs, if desired, when process switches occur. Setting the section RAMs causes the MMU to flush its caches.

Defining Virtual Memory

The operating system must define the way virtual memory is to be configured. In systems using an MMU, this requires that segment and page descriptor tables be set up in physical memory and that the CR be configured for 2, 4, or 8 Kbyte pages. The way these tables are set up determines where segments and pages reside in physical memory.

Peripheral Mode

The peripheral mode of the MMU is used by the operating system in several ways. One use is to initialize the section RAMs and the configuration register (CR), internal elements of the MMU. Section RAMs are loaded with the SDT's base addresses and length. The descriptor caches may be preloaded to avoid miss-processing (for a real-time process or other special case).

Other uses of the peripheral mode by the operating system include:

- Setting or clearing the configuration register referenced and/or modified bits
- Reading the fault code register (FLTCR) and fault address register (FLTAR) in order to handle MMU-generated exceptions
- Reading the cache contents in the case of serious exceptions

6.7.3 MMU Interactions

The MMU interacts with the operating system through address translation, miss-processing, exception detection, and other events. Once the MMU is initialized, it translates virtual addresses by using the SDs and PDs. It caches descriptors from the SDTs and PDTs to minimize translation time. The MMU handles the transfer of descriptors between its caches and physical memory during miss-processing without operating system intervention. The MMU also checks for violations (e.g., address or access) without operating system action. If violations occur, exceptions are issued and the operating system's exception handler can respond accordingly.

MMU Exceptions

Operating system action is required when the MMU signals to the CPU that an exception (external fault) has occurred. The MMU detects several exceptions that relate to errors (such as memory exceptions when the MMU does not correctly read an SDT or PDT) and places the corresponding code in the FLTCR and the FLTAR.

Other exceptions signal that data is not present in physical memory. In these cases, the MMU tells the CPU that a required page or segment is not in physical memory and must be brought into physical memory. The operating system is responsible for these activities; it must do any I/O operations that are necessary and adjust the appropriate SDT and/or PDT values.

The MMU provides hardware support for operating system page- or segment-replacement algorithms by setting the R and M bits in the segment and page descriptors whenever a segment or page is referenced or modified. If the operating system periodically clears all of the R bits, for example, it can use the R bits to implement a variation of the least recently used replacement algorithm. It could choose to replace segments or pages that still have their R bits clear when an exception occurs, reasoning that those segments or pages have been referenced less recently than the ones with the R bits set.

Flushing

The operating system occasionally alters the contents of the descriptor tables in memory. For example, it must do this to set and clear bits that indicate whether a page or segment is present whenever they are swapped in and out of physical memory. Any alteration of the table contents must be followed by some type of flushing of the MMU caches to prevent the chaos that would result if tables and caches contained conflicting information. If the operating system alters a table entry for one page or segment, it must flush any existing cache entry for that page or segment. If the operating system alters or deletes many entries in a table, it may be more efficient to flush an entire section than to flush several cache entries one at a time.

Translation Probe

The WE 32201 MMU provides a translation probe that allows the operating system to quickly simulate a translation. The probe, which uses the MOVTR word instruction, can be used to improve performance by looking ahead for possible exceptions. In certain instances, fault handling of these exceptions could be eliminated. For example, if a page in memory is accessed by a user who does not have access permissions, the exception that would be generated if a GATE or process switch takes place could be eliminated by using the translation probe to bypass the GATE or process switch.

6.7.4 Efficient Mapping Strategies

The memory mapping defined by the operating system may have an enormous effect

OPERATING SYSTEM CONSIDERATIONS

Indirect Segment Descriptors

on the performance of the system. There are some basic rules for efficient mapping strategies. Large blocks that will remain in physical memory for long periods could be defined as 8-Kbyte paged segments so that few entries are needed in the descriptor tables and descriptor caches. If physical memory is scarce, however, use of several large pages could result in long waits to move the pages in and out, thus wasting the physical memory where another large page cannot fit.

If only part of a segment need be in memory at any one time, smaller paged segments make more efficient use of memory.

6.7.5 Indirect Segment Descriptors

Indirect segment descriptors provide a mechanism to create shared segments that may be easily swapped out. The only segment descriptor that has to be modified by the operating system when the shared segment is swapped or moved is the last one (i.e., the descriptor that directly references the segment data).

Indirect segment descriptors are useful for shared segments when different processes running at the same execution level are given different access permissions to the segment. The access permissions in the last descriptor are superseded by the access permissions in the first descriptor used in the reference.

Indirect segment descriptors can also be used to provide chains of descriptors so that the path to the last segment descriptor can be passed from one process to another. This is similar to the passing of pointers in a programming language, except that here each process that owns a descriptor that others are linked to can rewrite that descriptor, thus breaking or redirecting the chain.

6.7.6 Using the Cacheable Bit

Cached segment and page descriptors each contain one cacheable bit (represented by \$ for the MMU). Whenever a descriptor is used for translation, the MMU reflects the value of the \$ bit in the cached descriptor through the cacheable (CABLE) output.

The \$ bit in the segment descriptor is copied into the cached page descriptor during miss-processing so that (from the operating system designer's point of view) the \$ bit values are associated with segments, not individual pages.

The MMU does not manipulate the \$ bits and the CABLE output signal in any other way, so this facility can be used in any way desired by the system designer. One possible use (from which the name cacheable is derived) is to provide an interface to a cache memory other than the MMU's own descriptor caches, to indicate the contents of the associated segments that are not cacheable.

6.7.7 Physical Data Cache

The WE 32201 MMU has an on-chip 4-Kbyte, 2-way, set-associative physical cache that stores instructions and data. Its purpose is to increase system performance by

reducing the average system access time. The MMU provides three means for the operating system to control the data cache: selective caching, the capability of flushing the cache, and reset.

Selective Caching

Operating systems containing information regarding devices that should not be cached (either for security reasons or to prevent cache interference with the devices) are supported by a bit in the segment descriptor and the cacheable in ($\overline{\text{CACHEIN}}$) signal. When operating in virtual mode, the cacheable bit (\$) may be set in any segment descriptor to identify noncacheable segments. The $\overline{\text{CABLEIN}}$ signal may be used to protect noncacheable memory in both virtual and physical mode.

The MMU does not cache any data from a segment that has its \$ bit cleared (0). On a hit, the data cache entry is flushed, and on a miss, no update is performed. This case is also indicated by the $\overline{\text{CABLE}}$ signal in order to support external caches. The $\overline{\text{CABLEIN}}$ signal is always sampled during a memory access if the data cache is enabled. If $\overline{\text{CABLEIN}}$ is negated, the access is not cached.

Flush Data Cache Register (FDCR)

The operating system can flush the data cache by writing all zeros to the flush data cache register (FDCR).

Reset

When the MMU reset ($\overline{\text{MRESET}}$) signal is asserted, the cache is flushed. This flushing is independent of the data cache enable (DCE) bit.

Disabling the Cache

The operating system may disable the data cache by clearing (0) the DCE bit in the configuration register. When this occurs, the cache is completely inactive in that it does not return data on hits, does not update on misses, and does not monitor the address bus.

6.7.8 Using the Page-Write Fault

The fault on write (W) bit in the MMU's page descriptors is checked during address translation after all other checks have been done. If the W bit is set and the access type is a write, a page-write fault occurs. This feature can lead to increased efficiency in the implementation of a *UNIX* System fork. The W bit could be set when the fork is invoked, and then both the parent and child processes could continue to use those pages without having the MMU and operating system physically copy the shared pages until one of those pages is written. A write operation would cause a page-write fault, and the pages would be copied and the write bits reset. In this way the system copies pages only as necessary.

OPERATING SYSTEM CONSIDERATIONS

Access Protection

6.7.9 Access Protection

Access bits contained within segment descriptors specify the access permission (no access, execute-only, read/execute, and read/write) for each execution level (kernel, executive, supervisor, and user). These bits ensure that segments are accessed on the appropriate level. If an access permission is disallowed, an access exception occurs.

6.7.10 Using the Software Bits

Three software bits are contained in each segment and page descriptor. The MMU does not alter the value of these bits at any time. This allows the operating system designer to use these bits in any manner. For example, a software bit can be used to avoid allocating any stack space until a process actually needs it, by assigning this software bit to signify that a page does not exist. Normally, a process start-up would create a (sometimes large) stack of zeros; the software bit could be used to avoid creating the stack until the user program references that page. Only then would the page-not-present fault cause the operating system to allocate the stack space. If the user program never references that page, the software bit saves memory for other processes.

6.8 OPERATING SYSTEM INSTRUCTIONS

The remainder of this chapter describes the operating system instructions (listed in Table 6-1) and the microsequences. Each description includes the assembler syntax, operation performed, effect of address modes on condition flags, exceptions generated, and an example.

Some operating system instructions and all microsequences call at least one XSWITCH function to do parts of the context switch. These functions, XSWITCH_ONE(), XSWITCH_TWO(), and XSWITCH_THREE(), are included among the microsequences.

6.8.1 Notation

Operations are described in C language where possible. In particular, the following notation is used where a C language operator or symbol does not exist:

- *x Word of register *x* contains the address of
 (a pointer to) the operand.
- *x++ Use word or register *x* as a pointer to the
 operand; then increment *x* by 1, 2, or 4 for a
 byte, halfword, or word operation, respectively.

*--x	Decrement word or register <i>x</i> by 1, 2, or 4 for a byte, halfword, or word operation, respectively; then use <i>x</i> as a pointer to the operand.
interrupt-ID	An 8-bit value, generated on the interrupt acknowledge access cycle, identifies the interrupt vector to the process.
dst	Replace with destination operand.
src	Replace with source operand.
{operation}	Text between braces describes an operation in general terms.
R<a> = <x>	Replace field (or bits) <i>a</i> of word R with the value <i>x</i> .

Table 6-2 lists the symbols used to define the bits fields being altered in the PSW. See Tables 6-3 through 6-5 for the ISC values.

The following symbols are used to identify CPU registers:

AP	Argument pointer, r10 (assembler syntax %ap)
FP	Frame pointer, r9 (assembler syntax %fp)
ISP	Interrupt stack pointer, r13 (assembler syntax %isp)
PC	Program counter, r15 (assembler syntax %pc)
PCBP	Program control block pointer, r14 (assembler syntax %pcbp)
PSW	Processor status word, r11 (assembler syntax %psw)
Rn	Register <i>n</i> , <i>rn</i> , where <i>n</i> = 0 to 8 (assembler syntax %rn)
SP	Stack pointer, r12 (assembler syntax %sp)

6.8.2 Privileged Instructions

These instructions are executed only when the process is in the kernel execution mode. Attempting to invoke them at a lower level causes a normal exception (privileged opcode).

OPERATING SYSTEM CONSIDERATIONS

Privileged Instructions

Instruction	Mnemonic
Call process	CALLPS
Disable virtual pin and jump	DISVJMP
Enable virtual pin and jump	ENBVJMP
Interrupt acknowledge	INTACK
Return-to-process	RETPS
Return-from-quick-interrupt	RETOINT
Wait	WAIT

The DISVJMP and ENBVJMP instructions disable or enable the CPU's virtual address pin and then jump to an address. ENBVJMP enables an MMU, signaling that the CPU is now supplying virtual addresses for translation. DISVJMP disables the MMU and only physical addresses are supplied. With an ENBVJMP instruction, a new (virtual) address is loaded into the PC; hence, the jump. For DISVJMP, a physical address is loaded into the PC. The use of CALLPS and RETPS was previously discussed in section 6.4.2 and section 6.4.3, respectively. RETOINT is used to return from a quick interrupt. WAIT provides a processor-level execution halt that remains in effect until an interrupt occurs.

The following descriptions provide more detail about the instructions.

CALLPS

CALLPS

Call Process

Assembler Syntax **CALLPS**

Opcode **0x30AC**

Description This instruction performs a process switch, saving the current process, pushing its PCBP onto the interrupt stack, and entering a new process. The instruction performs the following:

1. Saves the context (register contents) of the current process in the current PCB if the PSW's R bit of the new process is set.
2. Saves the additional registers (r16—r23) of the current process in the current PCB if the PSW's AR bit of the new process is set.
3. Pushes the current PCBP value onto the interrupt stack.
4. Puts the new PCBP value (from register r0) into the PCBP register.
5. Sets the PSW, PC, and SP registers from the new PCB.
6. Performs block moves (if any) for the new process (if R bit of PSW is set and AR bit of PSW is cleared).
7. Exits, going to the new process.

Operands r0 is an implicit source operand (it should contain the PCBP of the new process).

Operation if (!kernel-level)
 normal-exception (privileged-opcode)

```
/* Put new PCBP into tempa. */  
tempa = r0  
  
/* Push old PCBP onto interrupt stack. */  
{force kernel level on memory accesses}  
*ISP++ = PCBP  
if(memory-exception)  
    reset-exception(interrupt-stack)
```

OPERATING SYSTEM CONSIDERATIONS

Privileged Instructions

CALLPS

CALLPS

```
/* Any memory exception in the XSWITCH_ONE subroutine will
cause a process exception (old PCB). The address of the next
instruction is always PC + 2. */
```

```
PC = address of next instruction
```

```
/* Set old PSW ISC/TM/ET to 0/0/1, respectively. */
```

```
PSW<ISC> = 0
```

```
PSW<TM> = 0
```

```
PSW<ET> = 1
```

```
XSWITCH_ONE()
```

```
/* Any memory exception in the XSWITCH_TWO subroutine will
cause a process exception (new PCB). */
```

```
XSWITCH_TWO()
```

```
/* Set new PSW ISC/TM/ET to 7/0/3, respectively. */
```

```
PSW<ISC> = 7
```

```
PSW<TM> = 0 /* Avoid CALLPS trace trap. */
```

```
PSW<ET> = 3
```

```
/* Any memory exception in the XSWITCH_THREE subroutine
will cause a process exception (new PCB).*/
```

```
XSWITCH_THREE()
```

```
{unforce kernel level on memory accesses}
```

```
{end of operation}
```

Address

None

Modes

Condition

Set by new PSW.

Flags

Exceptions

Normal exception (privileged opcode)
Process exception (old PCB or new PCB)
Reset exception (interrupt stack)

Example

```
{load new PCBP into r0}  
CALLPS
```

Notes

Opcode occupies 16 bits. The ISC/TM/ET fields of the PSW saved contain 0/0/1, respectively. These fields in the new process PSW contain 7/0/3, respectively.

DISVJMP

DISVJMP

Disable Virtual Pin and Jump

Assembler Syntax DISVJMP

Opcode 0x3013

Description This instruction changes the CPU to physical addressing mode (disables the MMU) and puts a new value in the PC (switching addressing modes usually makes the old PC value incorrect).

Operands r0 is an implicit source operand (it should contain the new physical PC value).

Operation if(!kernel-level)
 normal-exception (privileged-opcode)
 {Reset virtual address pin (\overline{VAD}) to 1}
 PC = r0
 {flush instruction cache}

Address Modes None

Condition Flags Unchanged

Exceptions Normal exception (privileged opcode)

Example {load physical address of next instruction into r0}
 DISVJMP

Notes Opcode occupies 16 bits.

OPERATING SYSTEM CONSIDERATIONS

Privileged Instructions

ENBVJMP

ENBVJMP

Enable Virtual Pin and Jump

Assembler Syntax	ENBVJMP
Opcode	0x300D
Description	This instruction changes the CPU to virtual addressing mode (enables the MMU) and puts a new value in the PC (switching addressing modes usually makes the old PC value incorrect).
Operands	r0 is an implicit source operand (it should contain the new virtual PC value).
Operation	if(!kernel-level) normal-exception (privileged-opcode) {Set virtual address pin (\overline{VAD}) to 0} PC = r0 {flush instruction cache}
Address Modes	None
Condition Flags	Unchanged
Exceptions	normal exception (privileged opcode)
Example	{load virtual address of next instruction into r0} ENBVJMP
Notes	Opcode occupies 16 bits.

INTACK

INTACK

Interrupt Acknowledge

Assembler Syntax	INTACK interrupt acknowledge
Opcode	0x302F INTACK
Operation	under "interrupt acknowledge" status $r0 = (\text{Interrupt-ID}) \ll 2$
Address Modes	None
Condition Flags	Unchanged
Exceptions	Privileged-opcode exception
Examples	INTACK
Notes	This instruction is privileged.

If $\overline{\text{NMINT}}=0$ and $\overline{\text{AVEC}}=0$, an interrupt acknowledge access is performed, fetching an 8-bit interrupt-ID. This value is zero-extended to a word, shifted left by two bit positions, and stored in r0. If $\overline{\text{NMI}}=0$, an auto-vector-interrupt acknowledge access is performed (with all 1s on the address bus) and 0 is stored in r0. If $\overline{\text{NMI}}=1$ and $\overline{\text{AVEC}}=0$, an auto-vector-interrupt acknowledge access is performed, and the requesting level (inverted and put on address bus and returned as interrupt-ID) is indeterminate.

OPERATING SYSTEM CONSIDERATIONS

Privileged Instructions

RETPS

RETPS

Return to Process

Assembler RETPS
Syntax

Opcode 0x30C8

Description This instruction terminates the current process (its context is not saved) and returns to the process whose PCBP is on the top of the interrupt stack. The instruction performs the following:

1. Pops the saved (old) PCBP value from the interrupt stack.
2. Puts the old PCBP value into the PCBP register.
3. Sets the PSW, PC, and SP registers from the saved values in the old PCB.
4. Performs block moves (if any) for the old process (if the R bit of the PSW is set and the AR bit of the PSW is cleared).
5. Puts the saved register values from the old PCB into the CPU registers (if the R bit in PSW is set; additional register values if the AR bit in the PSW is set).
6. Exits, going to the old process.

Operands None

Operation if (!kernel-level)
 normal-exception (privileged-opcode)

```
/* Pop new PCBP from interrupt stack. */  
{force kernel level on memory accesses}  
tempa = *--ISP  
if(memory_exception)  
    reset-exception(interrupt stack)
```

/* Any memory exception in the following operation will cause a process exception (old PCB).

Transfer R and AR bits from new PSW to current PSW so block moves and register restores will occur if needed. */

RETPS

RETPS

```

PSW<R> = *tempa<R>
PSW<AR> = *tempa<AR>

/* Any memory exception in the following microsequences will
cause a process exception (new PCB).*/

/* Put new PCBP in PCBP register and get new PC, PSW, and SP.
*/
XSWITCH_TWO()

/* Set new PSW ISC/TM/ET to 7/0/3, respectively. */
PSW<ISC> = 7
PSW<TM> = 0 /* Prevent RETPS trace trap. */
PSW<ET> = 3

/* Do block moves if PSW<R> == 1 and PSW<AR> == 0. */

XSWITCH_THREE()

/* If R bit is set (1), move saved register values (r0—r10) from new
PCB into CPU registers. */
if(PSW<R>==1) {
    AP = *(PCBP + 20)
    FP = *(PCBP + 24)
    r0 = *(PCBP + 28)
        ●
        ●
        ●
    r8 = *(PCBP + 60)

    /* If AR bit is set (1), move additional saved register values
(r16—r23) from new PCB into CPU registers. */
    if(PSW<AR>==1) {
        r16 = *(PCBP + 64)
            ●
            ●
            ●
        r23 = *(PCBP + 92)
    }
}
}

```

OPERATING SYSTEM CONSIDERATIONS

Privileged Instructions

RETPS

RETPS

{unforce kernel level on memory accesses}
{end of operation}

**Address
Modes**

None

**Condition
Flags**

Set by new PSW.

Exceptions

Normal exception (privileged opcode)
Process exception (old PCB and new PCB)
Reset exception (interrupt stack)

Example

RETPS

Notes

Opcode occupies 16 bits. There is no check of the interrupt stack.
Any exception in accessing this stack causes a reset.

RETQINT

RETQINT

Return From Quick Interrupt

Assembler Syntax RETQINT

Opcode 0x98

Description This instruction is used to return from a quick interrupt. The PC and PSW values to return to are popped from the interrupt stack and put into the PC and PSW registers.

Operands None

Operation if (mem_fault)
 reset(interrupt_stack_fault)
 /* Get old PSW value from interrupt stack. */
 tempa = *(ISP - 4)
 /* Based on tempa update: */
 /* PSW <EA, EX, X, CFD, QIE, CD, OE, NZVC, TE, IPL, CM,
 PM, I>. */
 PSW<EA> = *tempa<EA>
 PSW<EX> = *tempa<EX>
 PSW<X> = *tempa<X>
 PSW<CFD> = *tempa<CFD>
 PSW<QIE> = *tempa<QIE>
 PSW<CD> = *tempa<CD>
 PSW<OE> = *tempa<OE>
 PSW<NZVC> = *tempa<NZVC>
 PSW<TE> = *tempa<TE>
 PSW<IPL> = *tempa<IPL>
 PSW<CM> = *tempa<CM>
 PSW<I> = *tempa<I>
 PSW<PM> = PSW<CM>
 /* Get old PC value from interrupt stack. */
 tempa = *(ISP - 8)
 /* Put new PC value into PC register. */
 PC = tempa
 /* Finish pop of old PC and PSW. */
 ISP = ISP - 8

OPERATING SYSTEM CONSIDERATIONS

Privileged Instructions

RETQINT

RETQINT

```
/* Set new PSW ISC/TM/ET to 7/0/3. */  
PSW<ISC> = 7  
PSW<TM> = 0 /* Avoids RETQINT trace trap. */  
PSW<ET> = 3
```

{end of operation}

Example	RETQINT
Address	None
Modes	
Condition	Set by restored PSW
Flags	
Exceptions	Reset (interrupt_stack_fault)

WAIT

WAIT

Wait for Interrupt or Reset

Assembler Syntax **WAIT**

Opcode **0x2F**

Description This instruction halts the CPU, stopping instruction fetching and execution until an interrupt or external reset occurs.

The current PSW<IPL> value is used to determine whether each interrupt request will be granted during the WAIT instruction. Refer to the ON_INTERRUPT microsequence for details. When an interrupt occurs, the PC value saved is the next instruction after the WAIT so that when the interrupt handler returns, control goes to the next instruction instead of executing the WAIT instruction again.

Operands None

Operation if(!kernel-level)
 normal-exception (privileged-opcode)
 {Halt CPU until an interrupt occurs}

Address Modes None

Condition Flags Unchanged

Exceptions Normal exception (privileged opcode)

Example **WAIT**

Notes Opcode occupies 8 bits.

OPERATING SYSTEM CONSIDERATIONS

Nonprivileged Instructions

6.8.3 Nonprivileged Instructions

These instructions are executed in any execution level:

Instruction	Mnemonic
Gate	GATE
Move translated word	MOVTRW
Return from gate	RETG
User call process	UCALLPS

GATE and **RETG** were discussed previously in section 6.3.2 and section 6.3.3, respectively.

MOVTRW tells an enabled MMU to intercept the virtual address sent by the CPU, translate it, and return the physical address to the destination. If no MMU is enabled and the system treats the move translated word access as a read, then this instruction acts as a normal **MOVW** (i.e., the source is copied into the destination).

UCALLPS performs a process switch.

GATE

GATE

Gate

Assembler Syntax GATE

Opcode 0x3061

Description This instruction performs a system call, saving the current PSW and PC on the execution stack and using two levels of tables to obtain new PSW and PC values. The instruction performs the following:

1. Checks to make sure that the current stack pointer is within the stack bounds specified in the PCB. This is to insure that the routine called by the GATE instruction starts in a guaranteed safe stack area.
2. Pushes a return address (PC) and the current value of the PSW on the execution stack. The return address insures that the GATE instruction can be used like a subroutine call. The PSW on the stack will be used by RETG to restore the CPU to the state it was in before the GATE function was invoked.
3. Index1 is used as an offset into the first-level table, which starts at address 0. The word selected is the address of a second-level table.
4. Index2 is used as an offset into the second-level table selected. It is added to the word read from the first-level table, to obtain the address of the PSW and PC entry in the second-level table. The first word of the entry selected is a new PSW to be used by the gate-handling subroutine and the second word is the address (starting PC) of the gate routine.
5. The PSW is replaced by the new PSW from the second-level table, with the old execution level field set appropriately and some other fields changed (see operation below).
6. The PC is set to the address of the gate-handling routine.
7. GATE exits, going to the new PC.

OPERATING SYSTEM CONSIDERATIONS

Nonprivileged Instructions

GATE

GATE

Operands

r0 and r1 are implicit source operands (they should contain byte offsets within first-level and second-level tables, respectively).

Operation

/* When reading from the PCB in the following two operations, a memory exception causes a process exception (gate PCB).

Check SP against stack bounds in PCB. */

{force kernel level on memory accesses}

if(SP < *(PCBP + 12))

 stack-exception (stack-bound)

if(SP >= *(PCBP + 16))

 stack-exception (stack-bound)

{release kernel level on memory accesses}

/* When writing to the stack in the following two operations, a memory exception causes a stack exception (stack).

The address of the next instruction is always PC+2.

Save old PC and PSW on execution stack. */

*SP = address of next instruction

/* Set PSW ISC/TM/ET to 1/0/2, respectively. */

PSW<ISC> = 1

PSW<TM> = 0

PSW<ET> = 2

*(SP + 4) = PSW

/* Mask index values and put in registers. */

tempa = r0 & 0x7C /* index1 */

tempb = r1 & 0x7FF8 /* index2 */

/* A memory exception from here to the end of the microsequence causes a normal exception (gate vector).

Get new PC and PSW values from table. */

{force kernel level on memory accesses}

/* Get pointer to second-level table. */

tempa = *tempa

/* Add offset within second-level table. */

tempa = tempa + tempb

GATE

GATE

```
/* Get new PSW from second-level table. */
tempb = *tempa
/* Set PM in new PSW to CM in old PSW. */
tempb<PM> = PSW<CM>
/* New PSW same IPL/R/AR values as old PSW. */
tempb<IPL> = PSW<IPL>
tempb<R> = PSW<R>
tempb<AR> = PSW<AR>
/* Set new PSW ISC/TM/ET to 7/1/3, respectively. */
tempb<ISC> = 7
tempb<TM> = 1
tempb<ET> = 3

/* Put new PC/PSW values into PC/PSW registers
   get new PC from second-level table. */
PC = *(tempa + 4)
PSW = tempb

/* Finish push of old PC and PSW. */
SP = SP + 8

{unforce kernel level on memory accesses}
{end of operation}
```

**Address
Modes**

None

**Condition
Flags**

Set by new PSW

Exceptions

Normal exception (gate vector)
Stack exception (stack bound and stack)
Process exception (gate PCB)
Reset exception (gate vector)

Example

GATE

Notes

Opcode occupies 16 bits.

The values of r0 and r1 should be byte-valued offsets. The value of

OPERATING SYSTEM CONSIDERATIONS

Nonprivileged Instructions

GATE

GATE

register r0 must be a multiple of 4; and the value of r1 must be a multiple of 8. These two registers are source operands only; GATE does not alter their contents.

MOVTRW

MOVTRW

Move Translated Word

Assembler Syntax *MOVTRW src,dst*

Opcode 0x0C

Description This instruction is intended for use with a memory management unit (MMU). An access using the address of the source operand (SAS = move translated word) is performed. It is expected that the MMU translates the address and returns the corresponding physical address.

Operands *src* – contains virtual address to be translated.
dst – contains the physical address after translation.

Operation {under move translated word status}
dst = physical address of source

Address Modes *src* – all modes except immediate, literal, or register.
dst – all modes except immediate or literal.

Condition Flags N = bit 31 of word returned
Z = 1, if word returned == 0
V = 0
C = 0

Exceptions Normal exception (invalid descriptor and external memory)

Example *MOVTRW X,%r0*

Notes Opcode occupies 8 bits.

When *MOVTRW* is executed in virtual mode with the *WE 32201* Memory Management Unit present, the address is translated to the corresponding physical address. If there is no exception, the MMU returns the translated physical address, which is then stored at the destination. If there is an exception, the MMU notifies the CPU in the normal fashion.

OPERATING SYSTEM CONSIDERATIONS

Nonprivileged Instructions

MOVTRW

MOVTRW

When MOVTRW is executed in physical mode with the WE 32201 Memory Management Unit present, the MMU behaves as if a read operation in physical mode is taking place.

In systems without an MMU, some other device must respond to the move translated word access.

The source operand is an address of operand. The destination operand is of the word type. If *&src* is not a word address, a normal exception (external memory) occurs.

During a MOVTRW instruction, the access status code signals identify the memory access as being a move translated word.

If the $\overline{\text{DYN16}}$ signal is asserted (signifying a 16-bit bus is being accessed) during the read cycle of the move translated instruction, the cycle is repeated with a DSIZE of halfword, and the second least significant bit of the address to be translated is complemented.

If the second least significant bit of the original address to be translated is 1, the data returned in the first cycle via DATA16—DATA31 is expected to be the lower 16 bits of the translated address. The data returned in the second cycle via DATA16—DATA31 is expected to be the upper 16 bits of the translated address.

If the second least significant bit of the original address to be translated is 0, the data returned in the first cycle via DATA16—DATA31 is expected to be the upper 16 bits of the translated address. The data returned in the second cycle via DATA16—DATA31 is expected to be the lower 16 bits of the translated address.

The WE 32201 MMU does not support dynamic bus sizing nor a 16-bit bus. A system that supports dynamic bus sizing needs external hardware to generate faults when the WE 32201 MMU accesses a 16-bit port or memory (e.g., during miss processing).

RETG

RETG

Return from Gate

Assembler Syntax **RETG**

Opcode **0x3045**

Description This instruction can be used to return from a GATE, normal exception, or quick interrupt. The PC and PSW values to return to are popped from the execution stack, the current and new execution levels are compared to prevent a return to a higher execution level, and then the new values are put into the PC and PSW registers.

Operands **None**

Operation */* Get old PC/PSW values from execution stack. */*
tempa = *(SP - 4)
tempb = *(SP - 8)
if(memory-exception)
 stack-exception(stack)

/ Compare execution levels to prevent return to a higher execution level. */*
if(tempa<CM> < PSW<CM>)
 normal-exception(illegal-level-change)

/ New PSW keeps same IPL/CFD/QIE/CD/R/AR/EX/EA values as current PSW. */*
tempa<IPL> = PSW<IPL>
tempa<CFD> = PSW<CFD>
tempa<QIE> = PSW<QIE>
tempa<CD> = PSW<CD>
tempa<R> = PSW<R>
tempa<AR> = PSW<AR>
tempa<EX> = PSW<EX>
tempa<EA> = PSW<EA>

/ Set new PSW ISC/TM/ET to 7/0/3, respectively. */*
tempa<ISC> = 7
tempa<TM> = 0 */* Avoids RETG trace trap. */*
tempa<ET> = 3

OPERATING SYSTEM CONSIDERATIONS

Nonprivileged Instructions

RETG

RETG

```
/* Put new PC/PSW values into PC/PSW registers. */
```

```
PSW = tempa
```

```
PC = tempb
```

```
/* Finish pop of old PC and PSW. */
```

```
SP = SP - 8
```

```
{end of operation}
```

Address Modes

None

Condition Flags

Set by new PSW

Exceptions

Normal exception (illegal level change)
Stack exception (stack)

Example

RETG

Notes

Opcode occupies 16 bits.

UCALLPS

UCALLPS

User Call Process

Assembler Syntax UCALLPS

Opcode 0x30C0

Description This instruction, when enabled, performs a process switch, saving the current process, pushing its PCBP onto the interrupt stack, and entering the new process. The PCBP of the new process is read from memory address 0x488, which is also the last location in the full interrupt PCBP table (vector number 255). The instruction does the following:

1. Saves the context (register contents) of the current process in the current PCB.
2. Pushes the current PCBP value onto the interrupt stack.
3. Puts the new PCBP value (from location 0x488) into the PCBP register.
4. Sets the PSW, PC, and SP registers from the new PCB.
5. Performs block moves (if any) for the new process.
6. Exits, going to the new process.

The instruction is enabled when the <EX> bit of the PSW is set (1). It is disabled when the <EX> bit is cleared (0). When disabled, use of this instruction causes an illegal opcode exception. The RETPS instruction can be used later to terminate the new process and return to the saved process.

Operands None

OPERATING SYSTEM CONSIDERATIONS

Nonprivileged Instructions

UCALLPS

Operation

```
If (PSW<EX> == 0)
    normal_exception (illegal_opcode)
    {force kernel level on memory accesses}
    /* Put new PCBP into tempa. */
    tempa = *1160 /* 488 hex */
    if (mem_fault)
        reset (system_data)

    /* Push old PCBP onto interrupt stack. */
    *ISP++ = PCBP
    if (mem_fault)
        reset (interrupt_stack_fault)

    /* Any memory fault in the XSWITCH_ONE subroutine will cause
       an on_process (old_pcb_fault) exception. The address of the
       next instruction is always PC + 2. */
    PC = address of next instruction

    /* Set old PSW ISC/TM/ET to 0/0/1. */
    PSW<ISC> = 0
    PSW<TM> = 0
    PSW<ET> = 1
```

UCALLPS

UCALLPS

UCALLPS

```

/* Save current registers in current PCB. */ XSWITCH_ONE();

/* Any memory fault in the XSWITCH_TWO or
XSWITCH_THREE subroutines causes an on_process
(new_pcb_fault) exception. */ /* Put new PCBP in PCBP
register and get new PC, PSW, and SP. */

XSWITCH_TWO();

/* Set new PSW ISC/TM/ET to 7/0/3. */ PSW<ISC> = 7
PSW<TM> = 0 PSW<ET> = 3

/* Do block moves if PSW<R> == 1 and PSW<AR> == 0. */
XSWITCH_THREE()

{unforce kernel level on memory accesses} {end of operation}

```

Address	None
Modes	
Condition	Set by the new PSW
Flags	
Exceptions	Normal exception (illegal opcode) On_process (old_pcb_fault) On_process (new_pcb_fault) Reset (interrupt_stack_fault) Reset (system_data)
Example	{load new PCBP into 0x488} UCALLPS
Notes	This is a 16-bit opcode.

OPERATING SYSTEM CONSIDERATIONS

Microsequences

6.8.4 Microsequences

The microsequences represent built-in microprocessor functions. These are executed automatically when the processor accepts an interrupt, generates an exception, or acknowledges a reset request. The XSWITCH functions are called by some operating system instructions and the microsequences.

ON-NORMAL EXCEPTION

ON-NORMAL EXCEPTION

On-Normal Exception

Description A normal exception is caused by some action of the current process, such as execution of an illegal opcode, and it causes the CPU to perform the following gate-like sequence or implicit process switch.

If PSW<EX> == 0, this sequence is identical to that of GATE except that zero (instead of r0) is used as the offset into the first-level table (offset1) and the ISC value (instead of r1) is used as the offset into the second-level table (offset2). An RETG instruction is used to return from this gate-like normal exception.

If PSW<EX> == 1, the sequence is identical to that of a process switch except that the PCBP is located at the second to last entry in the full-interrupt table (see Figure 6-1), memory location 1156 (0x484). An RETPS instruction is used to return from this implicit process switch normal exception.

Operation if (PSW<EX> == 0) {

/* When reading from the PCB in the following two operations, a memory exception causes a process exception (gate PCB).
 Check SP against stack bounds in PCB. */
 {force kernel level on memory accesses}
 if(SP < *(PCBP + 12))
 stack-exception(stack-bound)
 if(SP >= *(PCBP + 16))
 stack-exception(stack-bound)
 {release kernel level on memory accesses}

/* When writing to the stack in the following two operations, a memory exception causes a stack exception (stack).
 Save old PC and PSW on execution stack. */
 *SP = PC
 /* Set PSW TM/ET to 0/3, respectively. */
 PSW<TM> = 0
 PSW<ET> = 3 /* Normal exception. */

OPERATING SYSTEM CONSIDERATIONS

Microsequences

ON-NORMAL EXCEPTION

ON-NORMAL EXCEPTION

```
*(SP + 4) = PSW
/* Set temp registers to gate table index values. */
tempa = 0
tempb = PSW<ISC> << 3

/* A memory exception from here to the end of the
microsequence causes a reset exception (gate vector).

Get new PC and PSW values from table. */
{force kernel level on memory accesses}
/* Get pointer into second-level table. */
tempa = *tempa
/* Add offset within second-level table. */
tempa = tempa + tempb
/* Get new PSW from second-level table. */
tempb = *tempa
/* Set PM in new PSW. */
tempb<PM> = PSW<CM>
/* New PSW, same IPL/R/AR values as old PSW. */
/* Set new PSW ISC/TM/ET to 7/1/3, respectively. */
tempb<ISC> = 7
tempb<TM> = 1
tempb<ET> = 3
tempb<IPL> = PSW<IPL>
tempb<R> = PSW<R>
tempb<AR> = PSW<AR>

/* Put new PC/PSW values into PC/PSW registers. */
PC = *(tempa + 4) /* Get new PC. */
PSW = tempb

/* Increment the stack pointer. */
SP = SP + 8

{release kernel level on memory accesses}
{end of operation}
}
```

ON-NORMAL EXCEPTION

ON-NORMAL EXCEPTION

```

if PSW<EX> == 1 {
    {force kernel level on memory accesses}
    /* Put new PCBP into tempa. */
    tempa = *1156 /* 484 hex */
    if (memory-exception)
        reset (system_data)
    /* Push old PCBP onto interrupt stack. */
    *ISP++ = PCBP
    if (memory-exception)
        reset (interrupt-stack)
    /* Any memory exception in the XSWITCH_ONE
       subroutine causes an on-process (old PCB) exception. */

    /* Save current registers in current PCB. */
    XSWITCH_ONE();

    /* Any memory exception in the XSWITCH_TWO
       subroutine causes an on-process (new PCB) exception. */

    /* Put new PCBP in PCBP register and get new PC, PSW,
       and SP. */
    XSWITCH_TWO( );

    /* Set new PSW ISC/TM/ET to 7/0/3, respectively. */
    PSW<ISC> = 7
    PSW<TM> = 0 /* Avoid CALLPS trace trap */
    PSW<ET> = 3

    {release kernel level on memory accesses}
    {end of operation}
}

```

Condition Set by new PSW
Flags

Exceptions Possible exceptions if PSW<EX> == 0:
 Stack exception (stack-bound and stack)
 Process exception (gate PCB)
 Reset exception (gate vector)

OPERATING SYSTEM CONSIDERATIONS

Microsequences

ON-NORMAL EXCEPTION

ON-NORMAL EXCEPTION

Possible exceptions if PSW<EX> == 1:

On-process (old PCB) On-process (new PCB) Reset (interrupt-stack and system-data)

Notes

The value of the ISC field of the PSW saved by the on-normal microsequence contains the identity of the normal exception. When this microsequence completes, the PSW<ISC> field is set to 7. See Table 6-4 for a list of normal exceptions.

Some exceptions set the condition flags as if the instruction that caused the exception was successfully completed.

ON-STACK EXCEPTION

ON-STACK EXCEPTION

On-Stack Exception

Description A stack exception is caused by discovery of a stack-bound violation during a GATE or normal exception. Such an event causes the CPU to perform the following process switching action, similar to a CALLPS instruction except that the new PCBP is obtained from a fixed address instead of from r0.

A RETPS instruction can be used to return from the stack exception handler process.

Operation */* Get new PCBP value from fixed address. */*
{force kernel level on memory accesses}
*tempa = *136 /* 88 hex */*
if(memory-exception)
 reset-exception(system-data)

/ Push old PCBP onto interrupt stack. */*
**ISP++ = PCBP*
if(memory-exception)
 reset-exception(interrupt-stack)

/ Any memory exception in the XSWITCH_ONE microsequence will cause a process exception (old PCB). */*
XSWITCH_ONE()

/ Any memory exception in the following XSWITCH_TWO microsequence will cause a process exception (new PCB).*
XSWITCH_TWO()

/ Set new PSW ISC/TM/ET to 7/0/3, respectively. */*
PSW<ISC> = 7
PSW<TM> = 0 / Prevent trace trap. */*
PSW<ET> = 3

{release kernel level on memory accesses}
{end of operation}

OPERATING SYSTEM CONSIDERATIONS

Microsequences

ON-STACK EXCEPTION

ON-STACK EXCEPTION

Condition Set by the new PSW
Flags

Exceptions Process exception (old PCB and new PCB)
Reset exception (interrupt stack and system data)

Notes The ISC field of the saved PSW contains the code that caused the stack exception. Refer to Table 6-5 for a list of stack exceptions.

ON-PROCESS EXCEPTION

ON-PROCESS EXCEPTION

On-Process Exception

Description A process exception is caused by a memory exception while accessing a PCB. Such an event causes the CPU to perform the following process switching action, similar to a CALLPS instruction except that there is no attempt to save the context of the current process (except for its PCBP value) and the new PCBP value is obtained from a fixed address instead of from r0.

There is no automatic way to return from a process exception because the exception is caused when there is a fatal error in the old process. The operating system is expected to choose some other process to invoke or return to.

Operation `/* Get new PCBP from fixed address. */`
`{force kernel level on memory accesses}`
`tempa = *132 /* 84 hex */`
`if(memory-exception)`
`reset-exception(system-data)`

`/* Push old PCBP onto interrupt stack. */`
`*ISP++ = PCBP`
`if(memory-exception)`
`reset-exception(interrupt-stack)`

`/* Any memory exception in the XSWITCH_TWO microsequence`
`will cause a reset exception (new PCB).`

`XSWITCH_TWO()`

`/* Set new PSW TM/ET to 0/3, respectively. */`
`PSW<TM> = 0 /* Prevent trace trap. */`
`PSW<ET> = 3`

`{release kernel level on memory accesses}`
`{end of operation}`

OPERATING SYSTEM CONSIDERATIONS

Microsequences

ON-PROCESS EXCEPTION

ON-PROCESS EXCEPTION

Condition Set by new PSW
Flags

Exceptions Reset exception (system data, interrupt stack, and new PCB)

Notes The ISC field of the PSW presented to the exception handling process contains the code corresponding to the condition that caused the process exception. Refer to Table 6-6 for a list of process exceptions.

ON-RESET EXCEPTION

ON-RESET EXCEPTION

On-Reset Exception

Description A reset exception is caused by an external reset request or by an exception while accessing the interrupt stack, the gate tables, or the interrupt tables. Such an event causes the CPU to go to physical addressing mode, obtain a new PCBP value from a fixed address, and set the PSW, PC, and SP registers from values in the new PCB. No information from the current (old) context is saved because the CPU may be powering up for the first time or else the old software context was so damaged that it caused a reset exception.

Operation

{flush instruction cache}

if(external-reset)

PSW<R> = 0

PSW<AR> = 0

PSW<EX> = 0

{force kernel level on memory accesses}

/* Force physical mode. */

{Negate VAD signal}

/* Get new PCBP from fixed address. */

tempa = *128 /* 80 hex */

if(memory-exception)

reset-exception(system-data)

/* Any memory exception in the XSWITCH_TWO microsequence will cause a reset exception (new PCB).

Put new PCBP in PCBP register and get new PC, PSW, and SP.

XSWITCH_TWO()

/* Set new PSW TM/ET to 0/3, respectively. */

PSW<TM> = 0 /* Prevent trace trap. */

PSW<ET> = 3

OPERATING SYSTEM CONSIDERATIONS

Microsequences

ON-RESET EXCEPTION

ON-RESET EXCEPTION

{release kernel level on memory accesses}
{end of operation}

**Condition
Flags**

Set by new PSW

Exceptions

Reset exception (system data and new PCB)

Notes

The ISC field of the PSW presented to the exception handling process contains the code corresponding to the condition that caused the reset exception. Refer to Table 6-7 for a list of reset exceptions.

ON-INTERRUPT

ON-INTERRUPT

On-Interrupt

Description

An interrupt is triggered by a request from external hardware and causes the CPU to perform a process switch or a gate-like action (depending on the value of $PSW\langle QIE \rangle$).

An interrupt occurs if the priority level requested is greater than the priority level in the $PSW\langle IPL \rangle$ field. Therefore, if $PSW\langle IPL \rangle == 15$, no interrupts will be acknowledged (except for a non-maskable interrupt).

For full-interrupts ($QIE == 0$), the on-interrupt microsequence implements a process switch to the process represented by the PCBP value stored at location $(140 + (4 * \text{Interrupt-ID}))$, where Interrupt-ID is an 8-bit value fetched during an interrupt acknowledge access.

For quick-interrupts ($QIE == 1$), the on-interrupt microsequence implements a gate-like PSW/PC switch, pushing the old PSW and PC onto the interrupt stack (a gate instruction saves the PC/PSW pair on the execution stack) and fetching new PSW and PC values from locations $(1164 + (8 * \text{Interrupt-ID}))$ and $(1164 + (8 * \text{Interrupt-ID}) + 4)$, respectively.

If an interrupt request is granted and autovectoring is requested (when \overline{AVEC} is asserted), an autovector interrupt acknowledge cycle is performed and no Interrupt-ID is fetched. The complement of the value of the interrupt option pin concatenated with the priority level at which the interrupt was requested is used as the Interrupt-ID. That is, bits 0—3 of the Interrupt-ID correspond to the requested level, bit 4 corresponds to the interrupt option pin, and bits 5—7 are zeros.

If a nonmaskable interrupt request is received (when the \overline{NMINT} signal is asserted), an autovector interrupt acknowledge cycle is performed (as if an autovector interrupt at level 0 was being acknowledged) and no Interrupt-ID is fetched. The value 0 is used as the Interrupt-ID.

OPERATING SYSTEM CONSIDERATIONS

Microsequences

ON-INTERRUPT

ON-INTERRUPT

```
Operation      /* Test for full or quick interrupt. */
               if(PSW<QIE> == 1)
                 goto QINT /* Quick interrupt. */
                 {Get interrupt-ID value via interrupt acknowledge bus cycle}
tempa = interrupt-ID
if(memory-exception)
    stack-exception(interrupt-ID-fetch)

/* It is a full interrupt. */
/* Get new PCBP from full interrupt table. */
{force kernel level on memory accesses}
tempa = *(140 + tempa * 4) /* 8C + tempa * 4 hex */
if(memory-exception)
    reset-exception(system-data)

/* Push old PCBP onto interrupt stack. */

*ISP++ = PCBP
if(memory-exception)
    reset-exception(interrupt-stack)

/*Set old PSW ISC/TM/ET to 0/0/1, respectively. */
PSW<ISC> = 0
PSW<TM> = 0
PSW<ET> = 1

/* Any memory exception in the XSWITCH_ONE microsequence
will cause a process exception (old PCB).*/

/* Save current registers in current PCB.*/

XSWITCH_ONE()

/* Any memory exception in the XSWITCH_TWO or
XSWITCH_THREE microsequences will cause a process exception
(new PCB).*/

/* Put new PCBP in PCBP register and get new PC, PSW, and
SP.*/

XSWITCH_TWO( )
```

ON-INTERRUPT

ON-INTERRUPT

```
/* Set new PSW ISC/TM/ET to 7/0/3, respectively. */
PSW<ISC> = 7
PSW<TM> = 0 /* Prevent trace trap. */
PSW<ET> = 3
/* Do block moves if PSW<R> == 1 and PSW<AR> == 0. */

XSWITCH_THREE( )

{release kernel level on memory accesses}
{end of operation}

QINT: /* It is a quick interrupt. */

/* Get Interrupt-ID value via an interrupt acknowledge bus cycle. */

tempa = Interrupt-ID * 8

if (memory exception)
    stack exception (Interrupt-ID fetch)

/* Put address of new PC and PSW pair in quick interrupt table into
tempa. */
tempa = 1164 + tempa

/* When writing to the stack in the following two operations, a
memory exception causes a reset exception (Interrupt stack). */

/* Set PSW ISC/TM/ET to 2/0/0, respectively. */
PSW<ISC> = 2
PSW<TM> = 0
PSW<ET> = 0

{force kernel level on memory accesses}

/* Save old PC and PSW on interrupt stack. */

*ISP = address of the next instruction /* Push PC. */
*(ISP + 4) = PSW /*Push PSW */
/* A memory exception from here until the end of the
microsequence causes a reset exception (system data).
```

OPERATING SYSTEM CONSIDERATIONS

Microsequences

ON-INTERRUPT

```
Get new PC and PSW values from table.  
Set PSW ISC/TM/ET to 1/0/0 respectively. */  
PSW<ISC> = 1  
PSW<TM> = 0  
PSW<ET> = 0  
tempb = tempa
```

```
/* Based on tempb, update:  
PSW<EA, EX, X, CFD, QIE, CD, OE, NZVC, TE, IPL, CM, PM, I>. */
```

```
/* Adjust previous execution level in new PSW. */
```

```
PSW<PM> = PSW<CM>  
PSW<EA> = *tempb<EA>  
PSW<EX> = *tempb<EX>  
PSW<X> = *tempb<X>  
PSW<CFD> = *tempb<CFD>  
PSW<QIE> = *tempb<QIE>  
PSW<CD> = *tempb<CD>  
PSW<OE> = *tempb<OE>  
PSW<NZVC> = *tempb<NZVC>  
PSW<TE> = *tempb<TE>  
PSW<IPL> = *tempb<IPL>  
PSW<CM> = *tempb<CM>  
PSW<I> = *tempb<I>
```

```
/* Put new PC value into temporary register. */  
tempa = *(tempa + 4)
```

```
/* Finish push of old PC and PSW. */  
ISP = ISP + 8  
PSW<ISC> = 7 */ Avoids trace trap */  
PSW<ET> = 3  
PSW<TM> = 0  
PC = tempa
```

```
{release kernel level on memory accesses}  
{end of operation}
```

ON-INTERRUPT

ON-INTERRUPT

ON-INTERRUPT

Condition Flags	Set by new PSW
Exceptions	Reset exception (system data and interrupt stack) on-stack exception (interrupt-ID fetch)
Notes	The interrupt-ID fetch is 8 bits and zero-extended to 32 bits in tempa.

OPERATING SYSTEM CONSIDERATIONS

Microsequences

XSWITCH

XSWITCH

XSWITCH Microsequences

Description These microsequences implement context switching. They are used, in various combinations, by the instructions CALLPS and RETPS and by the implicit microsequences On-Interrupt, On-Process, On-Stack, and On-Reset.

XSWITCH_ONE() performs a context save. The current register values that are saved depends on the setting of the PSW<R> and PSW<AR> bits. Table 6-8 indicates the registers that are saved in the current PCB for the various combinations.

XSWITCH_TWO() performs a context switch, putting a new value in the PCBP register and reading the new PSW, SP, and PC values from the new PCB.

XSWITCH_THREE() performs block moves specified in the PCB depending on the values of the PSW<R> and PSW<AR> bits (see Table 6-8).

		XSWITCH_ONE		XSWITCH_THREE
R	AR	AP, FP, R0, ..., R8	R16, ..., R23	Block Move
0	0	Not saved	Not saved	No
0	1	Reserved	Reserved	Reserved
1	0	Saved	Not saved	Yes
1	1	Saved	Saved	No

The action taken when a memory exception is encountered in the XSWITCH microsequences is determined by the calling sequence.

XSWITCH

XSWITCH

Operation */* Save current registers in current PCB. One argument: tempa is expected to contain new PCBP value. */*

XSWITCH_ONE:

```

/* Save current PC in PCB. */
*(PCBP + 4) = PC

/* Copy R-bit and AR-bit from new PSW to current PSW. */
PSW<R> = *tempa<R>
PSW<AR> = *tempa<AR>

/* Save current PSW and SP in PCB. */
*PCBP = PSW
*(PCBP + 8) = SP

/* If R-bit == 1, save current r0—r8/FP/AP in PCB. */
if(PSW<R> == 1) {
    *(PCBP + 20) = AP
    *(PCBP + 24) = FP
    *(PCBP + 28) = r0
        •
        •
        •
    *(PCBP + 60) = r8
    FP = PCBP + 52
}
/* If AR-bit == 1, save current r16—r23 in PCB. */
if(PSW<AR> == 1) {
    *(PCBP + 64) = r16
    *(PCBP + 68) = r17
    *(PCBP + 72) = r18
        •
        •
        •
    *(PCBP + 92) = r23
}

return

```

OPERATING SYSTEM CONSIDERATIONS

Microsequences

XSWITCH

XSWITCH

/* Put new PCBP in PCBP register and get new PC, PSW, and SP.
One argument: tempa is expected to contain new PCBP value. */

XSWITCH_TWO:

/* Put new PCBP value into PCBP register. */

PCBP = tempa

/* Put new PSW, PC, and SP values from PCB into registers. */

PSW = *PCBP /* PSW<AR/R/ISC/TM/ET> unchanged here. */

PSW<TM> = 0

PC = *(PCBP + 4)

SP = *(PCBP + 8)

/* If I-bit == 1, increment PCBP past initial context area. */

if(PSW<I> == 1) {

/* Clear I-bit in PSW register. */

PSW<I> = 0

/* Increment PCBP past initial context area. */

PCBP = PCBP + 12

}

/* If cache flushing not disabled, flush cache. */

if(PSW<CFD> == 0)

{flush instruction cache}

return

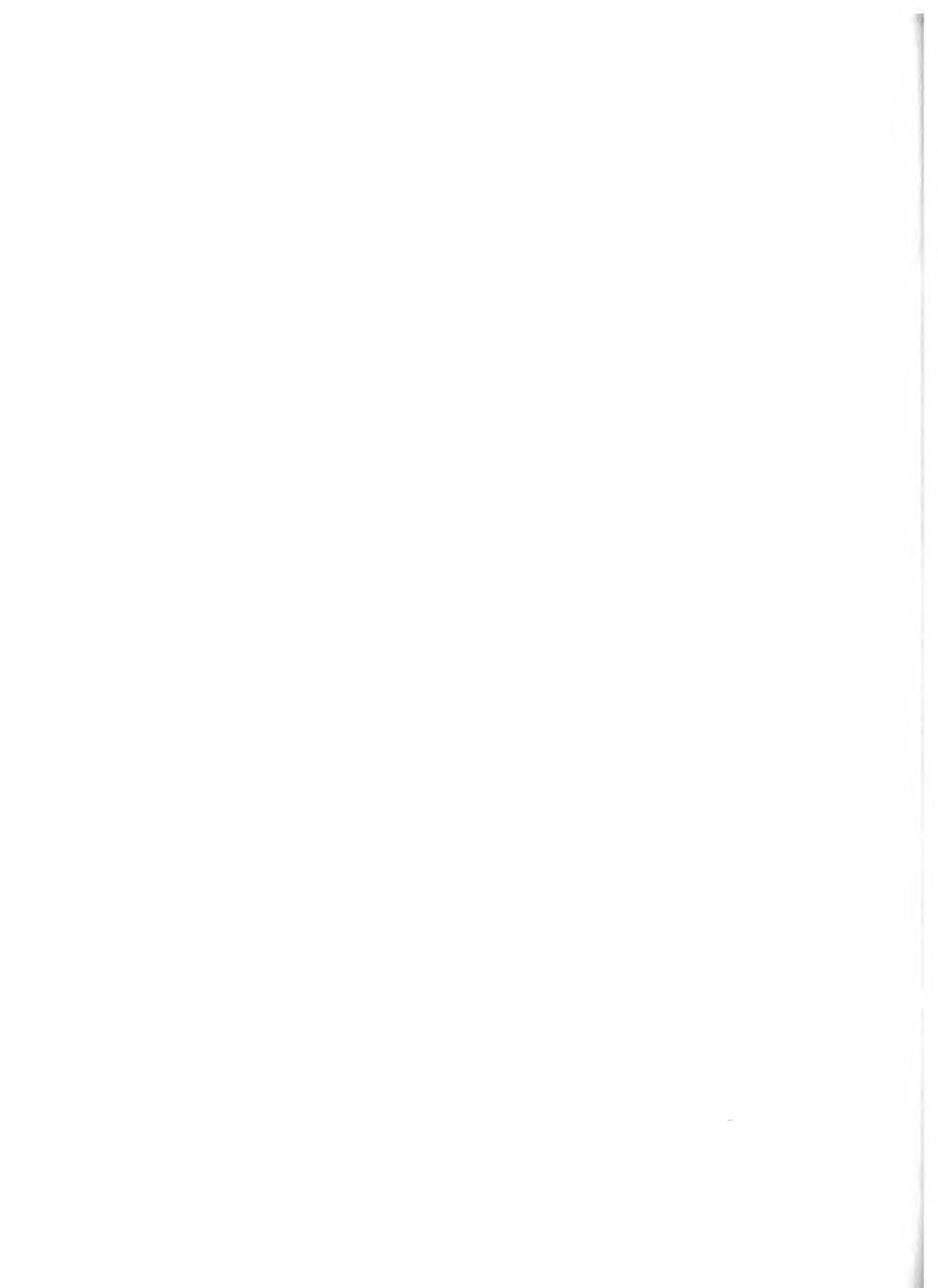
XSWITCH

XSWITCH

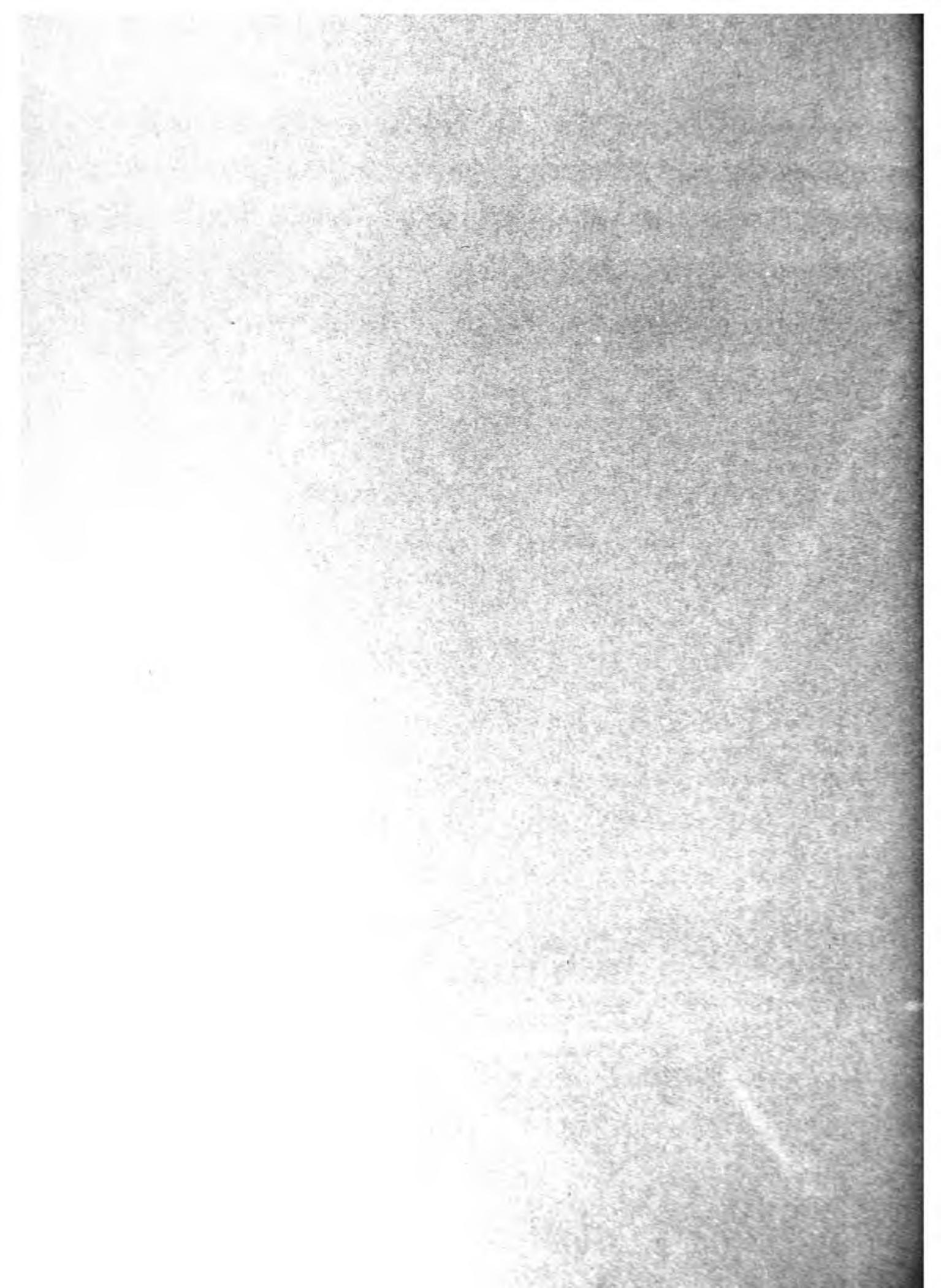
```
/* Do block moves, if PSW<R> == 1 and PSW<AR> == 0. */
```

```
XSWITCH_THREE:
```

```
    if(PSW<R> == 1 && PSW<AR> == 0) {  
        /* Get address of block0 size. */  
        r0 = PCBP + 64  
  
        /* Get block0 size. */  
        r2 = *r0++  
  
        /* While block size != 0. */  
        while(r2 != 0) {  
            /* Get destination start address. */  
            r1 = *r0++  
            /* Do one block copy. */  
            {execute MOVBLW instruction}  
            /* Get size of next block. */  
            r2 = *r0++  
        }  
  
        r0 = r0 + 4  
    }  
  
    return
```



Glossary



Assert - To drive a signal to its active state.

Bit field - A sequence of 1 to 32 bits contained in a base word. The field is specified by the address of its base word, a bit offset, and a width.

Bit offset - Identifies the starting bit of the field in its base word. The offset ranges from 0 to 31.

Bus interface control - Provides all the strobes and control signals necessary to implement the interface with peripherals.

Byte - An 8-bit quantity that may appear at any address in memory.

Coprocessor - A support processor that operates synchronously with the CPU to provide greater throughput in arithmetic or I/O functions.

Exceptional conditions - Events other than interrupts and reset requests that may interrupt the execution of a program. The four classes of exceptional conditions are normal exceptions, stack exceptions, process exceptions, and reset exceptions.

Execute unit - The elements in this unit perform all arithmetic and logic operations, perform all shift and all rotate operations, and compute the condition flags.

Faults - Error conditions that are detected outside the microprocessor and conveyed to the microprocessor over its fault input !FAULT bar!.

Fetch unit - The elements in this unit handle the instruction stream and perform memory-based operand accesses.

Full interrupt - Interrupt whose handling routine implements a process switch to the interrupt's handler. All interrupts are handled via the full

interrupt sequence if the QIE bit in the PSW is cleared (0).

Halfword - 16-bit quantity that may appear at any address in memory that is divisible by 2.

Instruction cache - A 64- by 32-bit on-chip cache used to increase the microprocessor's performance by reducing external memory reads for instruction fetches.

Instruction queue - An 8-byte, first-in-first-out (FIFO) on-chip queue that stores prefetched instructions.

Interrupt - A means by which external devices may request service by the microprocessor.

Main controller - The microprocessor's central control unit. It is responsible for acquiring and decoding instruction opcodes and directing the action of the fetch and execute controllers.

Memory management unit (MMU) - A software or hardware unit, or combination of both, that translates virtual addresses into physical addresses and verifies access authorization. The WE 32101 Memory Management Unit provides this function for the CPU.

Negate - To drive a signal to its inactive state.

Nonmaskable interrupt - Type of interrupt that interrupts the microprocessor regardless of the priority level in the IPL field of the PSW.

Normal exceptions - A class of exceptional conditions generated by the microprocessor when it detects a condition such as a trap, invalid opcode, or illegal operation.

Pipelining - Overlapping the execution of instructions to increase the

GLOSSARY

microprocessor's performance.

Prefetch - A technique where the CPU fetches an instruction prior to the completion of previous instructions.

Privileged instruction - An operating system group instruction that can execute only in kernel execution level.

Process control block (PCB) - A process data structure in external memory that saves the context of a process when the process is not running. Different contexts are saved depending on the settings of the PSW R and AR bits.

Process control block pointer (PCBP) - User register that points to the starting address of the process control block for the current process.

Process exceptions - A class of exceptional conditions that may occur during a process switch.

Processor status word (PSW) - User register that contains status information about the microprocessor and the current process.

Program counter (PC) - User register that contains the 32-bit memory address of the instruction being executed or, upon completion, contains the starting address of the next instruction to be executed.

Read interlocked operation - An operation which consists of a memory fetch (read access), one or more internal microprocessor operations, and then a write access to the same memory location.

Reset exceptions - A class of exceptional conditions that is triggered by an error condition in accessing critical system data.

Sign extension - Automatic extension of a byte or halfword value to 32 bits by

filling the high-order bits with the value of the sign bit.

Stack exceptions - A class of exceptional conditions that may occur during a process switch or a GATE sequence.

Stack pointer (SP) - User register that contains the current 32-bit address of the top of the execution stack; i.e., the memory address of the next item to be stored on (pushed onto) the stack or the last item retrieved (popped) from the stack.

Trace mechanism - An interpretive diagnostic trace trap using two bits in the PSW, trace enable (TE) and trace mask (TM), to analyze each executed instruction.

Wait-state - Idle periods that may be generated during a bus transaction to allow slow peripherals to handshake with the microprocessor.

Width - The size of a bit field. Width plus one is the number of bits in the field. The width ranges from 0 to 31.

Word - A 32-bit quantity that may appear at any address divisible by 4.

Working registers - Registers that are used exclusively by the microprocessor and are not user-accessible.

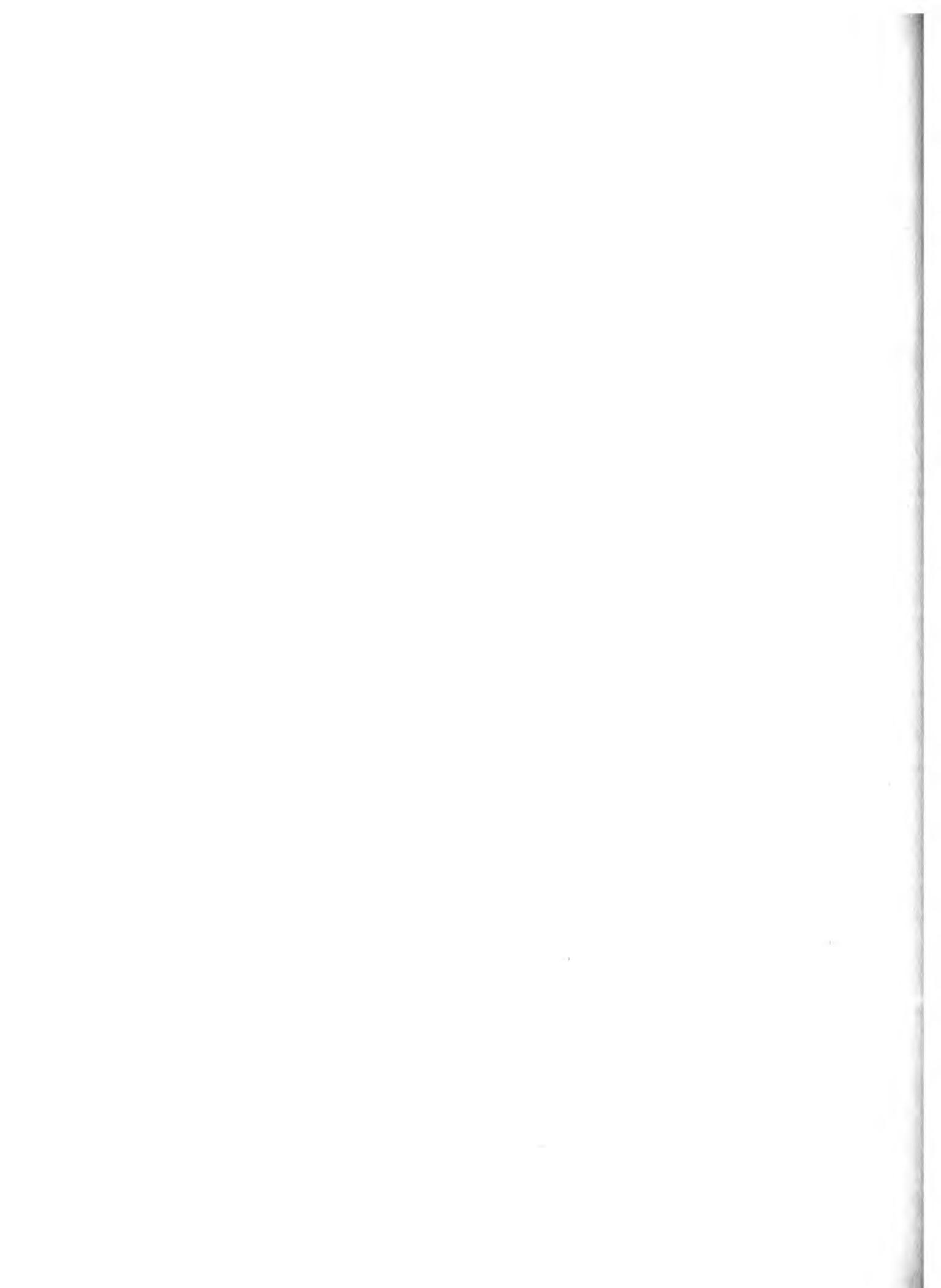
Zero extension - Automatically extending a byte or halfword value to 32 bits by filling the high-order bits with zeros.

3-state - To place an input in a high impedance state.

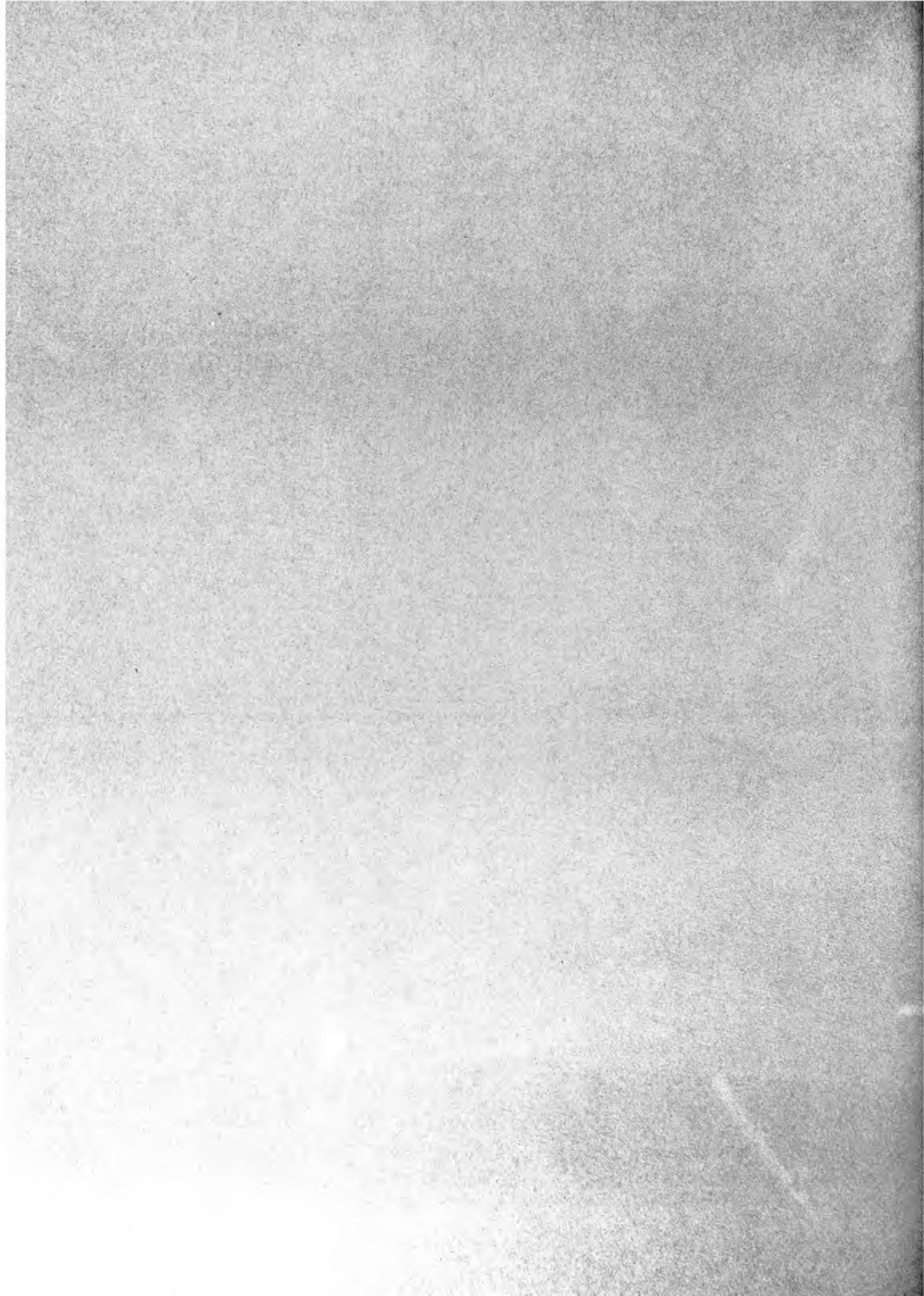
Acronyms



AP - Argument pointer	PPC - Prefetch counter
BPT - Breakpoint trap	PSL - Page select field
C - Condition flag bit carry	PSW - Processor status word
CALLPS - Call process	QIE - Quick interrupt enable
CD - Cache disable	RAM - Random access read/write memory
CFD - Cache flush disable	RI - Register-initial
CM - Current execution level	ROM - Read-only memory
CMOS - Complimentary metal-oxide semiconductor	rrrr - Register field
CPU - Central processing unit	RSB - Return from subroutine
CR - Configuration register	SD - Segment descriptors
DMA - Direct memory access	SDP - Software demand paging
EPROM - Erasable programmable ROM	SDT - Segment descriptor table
ET - Exception type	SGP - Software generation programs
FLTAR - Fault address register	SID - Section ID field
FLTCR - Fault code register	SOT - Segment offset field
FP - Frame pointer	SP - Stack pointer
I/O - Input/output	SSL - Segment select field
IPL - Interrupt priority level	TE - Trace enable
ISC - Internal state code	TM - Trace mask
ISP - Interrupt stack pointer	TT - Trace trap
LSB - Least significant bit	TTL - Transistor-transistor logic
mmmm - Mode field	V - Condition flag bit overflow
MMU - Memory management unit	Z - Condition flag bit zero
MSB - Most significant bit	
N - Condition flag bit negative	
NOP - No operation	
OE - Overflow enable	
PC - Program counter	
PCB - Process control block	
PCBP - Process control block pointer	
PD - Page descriptors	
PDT - Page descriptor table	
PM - Previous execution level	
POT - Page offset field	



Index



A

Absolute

- address modes, 5-8
- deferred mode, 5-16
- mode, 5-16

Access abort, 3-6

Access protection, 6-52

Access status

- code, 3-5
- signals, 3-8

Acknowledge

- dynamic 16-bit port, 3-4, 4-3, 4-4, 4-6, 4-10, 4-11, 4-12, 4-22, 4-28, 4-41, 4-50

Address

- and data bus, 2-1
- fault, 4-27
- modes, 5-3
- mode syntax, 5-7
- signals, 3-2
- strobe, 3-3
- translation, 6-44
- virtual, 6-44

Addressing modes, Chapt. 5

- absolute, 5-16
- absolute deferred, 5-16
- auto pre/post
 - increment/decrement, 5-20
- deferred displacement, 5-14
- displacement, 5-12
- expanded operand, 5-18
- format 1, 5-5
 - for registers 16-31, 5-25
- format 2, 5-20
- immediate, 5-15
- indexed register, 5-21
- negative literal, 5-7
- positive literal, 5-6
- register, 5-9
- register deferred, 5-11
- short offset, 5-6
- special, 5-8
- syntax, 5-7, 5-20

Alignment

- fault bus activity, 4-54

AP. See argument pointer.

AR bit, 6-22

Arbitration signals, 3-4, 4-47

Arbitrary Byte Alignment, 2-12, 2-13, 4-1, 4-54, 6-16

Architecture, 1-2, 6-1

- pipelining, 2-1, 4-55

Argument pointer (AP), 2-6

- short offset mode, 5-6

Arithmetic instructions, 5-29

Assembler syntax, 5-42

Assembly language

- operators, 5-43

- symbols, 5-43

- syntax, 5-7, 5-20, 5-25

Asserted signal, 3-1, 4-3

Asynchronous read, 4-6

Asynchronous write, 4-12

Autovector interrupt, 3-2, 4-44, 6-31

B

Bit

- cacheable (\$), 6-50

- software, 6-52

Bit field

- base word, 2-9

- defined, 2-9

- extraction of, 2-9

- instructions, 5-1

- offset, 2-9

- width, 2-9

Block diagram, 2-2

Blockfetch, 3-7

- operation, 4-22

Borrow, 2-22

Branch instructions, 5-34

Bus

- address, 2-1, 4-1

- arbiter, 3-5

- arbitration, 4-47

- exception signals, 3-5

- exceptions, 4-27

- relinquish and retry, 4-31

- operation, Chapt. 4

- request, 3-5, 4-49, 4-55

- acknowledge, 3-5

Byte

- data, 2-7, 5-1

INDEX

- deferred displacement mode, 5-7
- displacement mode, 5-7
- replication, 2-13

C

- Carry, 2-22

Cache

- disable, 6-15, 6-51
- flush disable, 6-15
- instruction cache flush, 6-15
- instruction cache hit, 4-53
- memory, 6-50
- memory management unit descriptor flushing, 6-49

- Call process instruction, 6-21, 6-27, 6-55

- Call-save sequence, 5-38

Clock

- input, 4-1
- signals, 3-10
- state, 4-1

- CM. See current execution level.

- Condition codes, 6-15

- Condition flags, 2-21, 5-42

- Conventional registers, 2-4

- Context switching, strategy, 6-22

- Control signals, 3-3

- Control-register save area, 6-9

Coprocessor, 4-56

- broadcast, 4-56
- data write, 4-63
- done, 3-3
- instructions, 5-40
- interface, 4-56
- operand fetch, 4-61
- status fetch, 4-62

- Current execution level (CM), 6-14

- Cycle initiate, 3-3

D

Data

- bus shadow, 3-6
- cache (MMU), 6-50
- in memory, 2-12
- ready, 3-3
- signals, 3-2

- size, 3-7
- storage, 2-12
- strobe, 3-3
- transfer acknowledge, 3-4
- transfer instructions, 5-28
- types, 2-7, 5-1

- Deferred address modes
 - defined, 5-11

- Deferred displacement mode, 5-14

- Descriptor byte format, 5-5, 5-9, 5-18

Development

- microprocessor, 1-2
- system support signals, 3-10

- Direct memory access (DMA), 4-49

- Disable Virtual Pin and Jump, 6-57

- Displacement modes, 5-12, 5-14

- DMA. See direct memory access.

- Double word (blockfetch), 3-7

- Dynamic Bus Sizing, 1-1, 2-16, 4-3

E

- EA bit, 6-13, 6-16

- Enable overflow trap, 6-15

- Enable Virtual Pin and Jump, 6-58

- Entry point, 6-18

Exception

- breakpoint trap, 6-41
- conditions, 6-37
- defined 6-37
- external memory, 6-41
- gate vector, 6-44
- handler, 6-37
- illegal level change, 6-41
- illegal opcode, 6-41
- integer overflow, 6-41
- integer zero divide, 6-41
- interrupt-stack fault, 6-44
- invalid descriptor, 6-41
- level of severity, 6-37
- memory management unit, 6-49
- new-PCB fault, 6-43, 6-44
- normal, 6-38
- old-PCB fault, 6-43, 6-44
- on-normal, 6-38, 6-79
- on-process, 6-43, 6-85
- on-reset, 6-44, 6-87

- on-stack, 6-42, 6-83
 - privileged-opcode, 6-41
 - privileged-register, 6-41
 - process, 6-43
 - reserved data type, 6-41
 - reserved opcode, 6-41
 - reset, 6-44
 - stack, 6-42
 - system data, 6-44
 - trace (trap), 6-41
 - type, 6-13
- Execution**
- modes, 3-9
 - modes, levels, 6-1, 6-5, 6-14
 - privilege, 6-5
 - stack, 6-6
- Executive mode (level 1), 6-1
- Expanded-operand type mode, 5-7, 5-18
- Explicit process switch, 6-4, 6-21
- Extending data, 2-11
- F**
- Fault**, 3-6, 4-28
- blockfetch, 4-34
 - defined, 4-27
 - exception mechanism, 6-37
 - external, 6-37
 - gate-PCB, 6-43
 - gate vector, 6-19, 6-44
 - internal, 6-37
 - interrupt-stack, 6-44
 - memory, 4-27, 6-49
 - new-PCB, 6-43, 6-44
 - old-PCB, 6-43, 6-44
 - page-write, 6-51
 - stack, 6-42
- Fetch unit, 2-1
- Fields**
- PSW, 6-13
- First entry point, 6-18
- Floating-point data types, 2-7, 2-10
- Flushing**
- instruction cache, 6-15
 - memory management unit
 - descriptor cache, 6-49
 - data cache, 6-51
- FP. See frame pointer.
- Frame pointer (FP), 2-5
- short offset mode, 5-6
- Full interrupts, 4-39, 6-36
- Full-interrupt handler's PCB, 6-33
- Functional group instructions, 5-27
- G**
- Gate**, 6-13
- instruction, 6-18, 6-67
 - mechanism, 6-17, 6-19
 - PCB fault, 6-43
 - return from, 6-21, 6-73
 - vector fault, 6-19, 6-44
- General-purpose registers, 2-4, 5-1
- General-register save area, 6-8
- H**
- Halfword**
- boundary, 2-12
 - data, 2-7, 5-1
 - deferred displacement mode, 5-7
 - displacement mode, 5-6
 - immediate mode, 5-8
- Handling-routine tables, 6-17
- High impedance, 3-10
- High level support group, 2-5
- I**
- I bit, 6-23
- Immediate modes, 5-8, 5-15
- Implicit process switch, 6-4, 6-30, 6-35, 6-42 thru 6-44
- Indexing**
- on-normal exception, 6-39
 - pointer and handling table, 6-20
- Indirect segment descriptors, 6-50
- Initial context for a process, 6-11, 6-22
- Initialize**
- memory management unit, 6-48
- Instruction**
- format, Chapt. 2, 2-19, 5-4
 - queue status, 3-10
- Instruction set, 1-3, 2-19, Chapt. 5

INDEX

- functional groups, 5-27
 - listings, 5-42
 - operating system, 6-2, 6-52
 - storage, 2-19
 - summary by mnemonic, 5-44
 - summary by opcode, 5-49
- Instruction cache
- hit, 4-53
 - on-chip, 2-1
- Instructions
- branch, 5-34
 - jump, 5-34
 - procedure-call, 5-38
- Interface signals, 3-3
- Interlocked operation, read, 4-19
- Internal reset, 4-50
- Internal State Code (ISC), 6-14, 6-39 thru 6-44
- Interrupt
- acknowledge, 4-41, 6-31, 6-59
 - autovector, 4-44, 6-31
 - handler model, 6-30
 - handler's PCB, 6-33
 - mechanism, 6-31
 - nonmaskable, 4-44
 - on-interrupt microsequence, 6-35, 6-89
 - option, 3-2
 - priority level, 3-2, 6-14
 - quick, 6-37
 - request and acknowledge codes, 4-43
 - returning from, 6-36
 - signals, 3-2
 - stack 6-33
 - stack pointer (ISP), 2-4, 6-33
 - structure, 6-30
 - vector table, 6-35
- ISC. See internal state code.
- J**
- Jump instructions, 5-34
- K**
- Kernel mode (level 0), 6-1
- L**
- Language support group
- high-level, 2-5
- Least significant bit (LSB), 2-7
- Level
- priority of interrupts, 3-2
- Levels of exception severity, 6-37
- Listing
- instruction set, 5-42
- Logical instructions, 5-32
- LSB. See least significant bit.
- M**
- Map, memory, 6-8
- Mapping strategy, 6-49
- Memory
- data in, 2-12
 - instructions in, 2-19
 - management, 6-44
 - management, virtual, 6-4, 6-44
 - map, 6-8
 - options, 6-45
 - PCB specifications, 6-12
 - translation, 6-45
 - virtual, 6-44
- Memory management unit (MMU), 6-44
- exceptions, 6-49
 - flushing, 6-49
 - initialization, 6-48
 - interactions, 6-48
 - mapping strategies, 6-49
 - peripheral mode, 6-48
 - translation, 6-46
 - translation probe, 6-49
- Microsequence, 6-2, 6-78
- defined, 6-2
 - on-interrupt, 6-35, 6-89
 - on-normal, 6-39, 6-79
 - on-process, 6-43, 6-85
 - on-reset, 6-44, 6-87
 - on-stack, 6-42, 6-83
 - XSWITCH, 6-27 thru 6-30, 6-94
- Miscellaneous instructions, 5-41
- mmmm field (address mode field), 5-4, 5-9
- MMU. See memory management unit.

Mnemonic, 2-19, 5-4
 instructions by, 5-44
 Modes, addressing. See addressing modes.
 Most significant bit (MSB), 2-7
 Move Translated Word, 6-71
 MSB. See most significant bit.

N

Negated signal, 3-1, 4-3,
 Negative literal mode, 5-7
 New-PCB fault, 6-43, 6-44
 Nonmaskable interrupt, 3-3, 4-44
 Nonprivileged instructions, 6-66
 Normal exceptions, 6-39
 Notation, 5-42
 operation, 6-52

O

Old-PCB fault, 6-43, 6-44
 On-chip instruction cache, 2-1
 On-interrupt microsequence, 6-35, 6-89
 On-normal exception, 6-39, 6-79
 On-process exception, 6-43, 6-85
 On-reset exception, 6-44, 6-87
 On-stack exception, 6-42, 6-83
 Opcodes,
 instruction set, 5-49
 Operand, 5-18
 descriptor, 5-18
 format, 2-20, 5-2
 in instruction format, 5-18
 syntax, 5-7, 5-18
 See also addressing modes.
 Operating system
 features, 6-1
 instructions, 6-2, 6-52
 support, 1-3, 2-6, Chapt. 6
 Operation, Chapt. 4
 read, 4-1
 write, 4-1
 OPND. See operand.
 Outputs
 during DMA, 4-49
 during reset, 4-51
 Overflow, 2-22

P

PC. See program counter.
 PCB. See process control block.
 PCBP. See process control block pointer.
 Peripheral mode, 6-48
 Physical
 address, 6-44
 data cache, 6-50
 memory, 6-17, 6-44
 Pin assignments, 3-11
 Pipelining, 4-55
 defined, 2-1
 PM. See previous execution level.
 Pointer
 defined, 5-11
 Pointer table, 6-17
 Positive literal mode, 5-6
 Previous execution level (PM), 6-14
 Privileged
 execution modes, 6-1, 6-5
 instructions, 6-2, 6-53
 register, 2-1, 5-1
 Procedure transfer, 5-22
 Process
 defined, 6-1
 exceptions, 6-43
 structure, 6-5
 switching, 6-1, 6-21
 explicit, 6-4, 6-21
 implicit 6-4, 6-30, 6-35, 6-42 thru 6-44
 Process control block (PCB), 6-5, 6-7
 Process control block pointer (PCBP), 2-6,
 locations, 6-8
 register, 5-1, 6-6
 Processor status word (PSW), 2-6, 6-13
 fields, 6-13
 register, 2-1, 5-1
 Program control instructions, 5-34
 Program counter (PC), 2-5, 5-2
 PSW. See processor status word.

Q

Quick interrupt, 6-30, 6-37
 enable, 6-15

INDEX

R

R bit, 6-22
Read operation, 4-1
Read/Write (R/W), 3-8
Registers, Chapt. 2, 5-1
 assembler syntax, 5-2
 conventional, 2-4
 data storage, 2-18
 deferred mode, 5-11
 indexed modes, 5-21
 initial context, 6-14
 modes, 5-9, 5-11, 5-21, 5-25
 set, 5-2
 syntax, 5-2
 user, 2-1, 5-1
Relinquish and retry, 4-31
 of blockfetch, 4-39
 request, 3-7
 request acknowledge, 3-6
Reserved
 data type exception, 6-41
 opcode, exception, 6-41
Reset, 4-50
 acknowledge, 3-6
 exceptions, 6-44, 6-87
 internal, 4-50
 MMU, 4-51
 request, 3-6
 sequence, 4-52
 signal, 3-6, 4-50
 states, 4-51
Retry, 3-6, 4-31
Return
 from gate, 6-21, 6-73
 from interrupt, 6-36
 from quick interrupt, 6-63
 to process, 6-28, 6-60
rrrr field (register field), 5-4, 5-9
R/W. See Read/Write.

S

Save-context area, 6-8
Saved context for a process, 6-12, 6-22
Second entry point, 6-19
Segment descriptors

 indirect, 6-50
Selective caching, 6-51
Short offset address modes, 5-6
Sign extension, 2-11
Signals, Chapt. 3
 address and data, 3-2
 access status, 3-7
 arbitration, 3-4
 bus exception, 3-5
 clock, 3-10
 descriptions, 3-1
 development system support, 3-10
 interface and control, 3-3
 interrupt, 3-2
 sampling points, 4-2
SP. See stack pointer.
Special address modes, 5-8
Stack, 2-5
 bounds, 6-7, 6-42
 exception handler, 6-7, 6-42
 exceptions, 6-42, 6-83
 execution, 6-6
 fault, 6-42
 instructions, 5-41
 interrupt, 6-7, 6-33
 pointer (SP), 2-4
Stack-bound, 6-10, 6-11, 6-42
 exception, 6-42
 fault, 6-42
Start of instruction, 3-10
Status codes, 3-8
Stop, 3-10
Storage
 data, 2-12
 of instructions, 2-19
 register data, 2-18
Structure of a process, 6-5
Subroutine transfer, 5-34
Supervisor mode (level 2), 6-1
Support
 high-level language, 2-5
 operating system, 2-6
Symbols,
 assembly language, 5-43
Synchronous
 read, 4-1, 4-4, 4-10
 write, 4-1, 4-12
Syntax

- address mode, 5-7, 5-20
- assembler, 5-2, 5-42
- assembly language, 5-18
- operand, 5-7, 5-18, 5-20
- register, 5-2
- System call, 6-13
- System reset, 4-50

T

- Trace enable, 6-15
- Trace mask, 6-14
- Transfer
 - procedure, 5-34
 - subroutine, 5-34
- Translation, virtual, 6-44
- Trap, 6-37
 - enable overflow, 6-15

U

- UNIX[®] Operating System, 2-1
- User
 - call process, 6-30, 6-75
 - mode (level 3), 6-2
 - registers, 2-1, 5-1

V

- Virtual
 - address, 3-9, 6-44
 - address space, 6-45
 - memory, 6-44, 6-48
 - memory, division of, 6-45
 - memory space, 6-44

W

- Wait, 6-65
- Word, 2-7
 - address modes, 5-6
 - data, 2-7
 - boundary, 2-9
 - deferred displacement mode, 5-6
 - displacement mode, 5-6
 - storage, 2-13, 2-21

- Write operation, 4-1

X

- XSWITCH function, 6-27 thru 6-30, 6-94
- XSWITCH_ONE, 6-95
- XSWITCH_TWO, 6-96
- XSWITCH_THREE, 6-97

Z

- Zero, 2-22
 - extension, 2-11