

## A WORD ABOUT TRADEMARKS . . .

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

*WE*® 32001 Processor Module

*WE*® 32100 Microprocessor

*WE*® 32101 Memory Management Unit

*WE*® 32102 Clock

*WE*® 32106 Math Acceleration Unit

*WE*® 321AP Microprocessor Analysis Pod

*WE*® 321DS Microprocessor Development System

*WE*® 321EB Microprocessor Evaluation Board

*WE*® 321SD Development Software Programs

*WE*® 321SG Software Generation Programs

*UNIX*® Operating System, Microsystem

---

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).



Issue 2  
November 1986

## ***WE*<sup>®</sup> 32100 Microprocessor**

### **Information Manual**

Select Code 307-730

COMCODE 105195366

## **ACKNOWLEDGEMENTS**

Published by  
The AT&T Documentation Management Organization

for the

Microsystems Product Management  
AT&T Information Systems

and the

62045166 Microsystems Processor Development Department  
AT&T Information Systems, Holmdel

## FOREWORD

This manual contains information on the *WE* 32100 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 307-730, call:

□ **1-800-432-6600**



**WE 32100 MICROPROCESSOR**  
**INFORMATION MANUAL**  
**CONTENTS**

**CHAPTER 1. INTRODUCTION**

1 INTRODUCTION.....	1-1
1.1 DEVELOPMENT.....	1-1
1.2 ARCHITECTURE.....	1-2
1.3 INSTRUCTION SET.....	1-3
1.4 OPERATING SYSTEM SUPPORT.....	1-3
1.5 SUPPORT PRODUCTS.....	1-4
1.6 CHAPTER SUMMARY.....	1-7
1.6.1 Chapter 2. Registers, Data, and Instruction Formats.....	1-7
1.6.2 Chapter 3. Signal Descriptions.....	1-7
1.6.3 Chapter 4. Bus Operations.....	1-7
1.6.4 Chapter 5. Instruction Set and Addressing Modes.....	1-7
1.6.5 Chapter 6. Operating System Considerations.....	1-7

**CHAPTER 2. REGISTERS, DATA, AND INSTRUCTION FORMATS**

2. REGISTERS, DATA, AND INSTRUCTION FORMATS.....	2-1
2.1 USER REGISTERS.....	2-2
2.2 CONVENTIONAL REGISTER SET.....	2-3
2.2.1 Stack Pointer.....	2-3
2.2.2 Program Counter.....	2-4
2.3 HIGH-LEVEL LANGUAGE SUPPORT GROUP.....	2-4
2.3.1 Frame Pointer.....	2-4
2.3.2 Argument Pointer.....	2-5
2.4 OPERATING SYSTEM SUPPORT GROUP.....	2-5
2.4.1 Processor Status Word.....	2-5
2.4.2 Process Control Block Pointer.....	2-5
2.4.3 Interrupt Stack Pointer.....	2-5
2.5 DATA TYPES.....	2-6
2.6 SIGN AND ZERO EXTENSION.....	2-9
2.7 DATA STORAGE IN MEMORY.....	2-10
2.8 REGISTER DATA STORAGE.....	2-11
2.9 INSTRUCTION SET.....	2-11
2.10 INSTRUCTION STORAGE IN MEMORY.....	2-12
2.11 CONDITION FLAGS.....	2-12

**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 CLOCK SIGNALS .....	3-2
3.3 INTERRUPT SIGNALS .....	3-2
Autovector ( $\overline{\text{AVEC}}$ ) .....	3-2
Interrupt Option ( $\overline{\text{INTOPT}}$ ) .....	3-2
Interrupt Priority Level (IPL0–IPL3) .....	3-3
Nonmaskable Interrupt (NMINT) .....	3-3
3.4 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 ( $\overline{\text{DRDY}}$ ) .....	3-4
Data Strobe ( $\overline{\text{DS}}$ ) .....	3-4
Data Transfer Acknowledge ( $\overline{\text{DTACK}}$ ) .....	3-4
Synchronous Ready ( $\overline{\text{SRDY}}$ ) .....	3-4
3.5 ACCESS STATUS SIGNALS .....	3-4
Block (Double Word) Fetch ( $\overline{\text{BLKFTCH}}$ ) .....	3-4
Data Size (DSIZE0–DSIZE1) .....	3-5
Read/Write (R/W) .....	3-5
Access Status Codes (SAS0–SAS3) .....	3-5
Virtual Address ( $\overline{\text{VAD}}$ ) .....	3-6
Execution Mode (XMD0–XMD1) .....	3-6
3.6 ARBITRATION SIGNALS .....	3-6
Bus Arbiter ( $\overline{\text{BARB}}$ ) .....	3-6
Bus Request ( $\overline{\text{BUSRQ}}$ ) .....	3-7
Bus Request Acknowledge ( $\overline{\text{BRACK}}$ ) .....	3-7
3.7 BUS EXCEPTION SIGNALS .....	3-7
Access Abort ( $\overline{\text{ABORT}}$ ) .....	3-7
Data Bus Shadow ( $\overline{\text{DSHAD}}$ ) .....	3-7
Fault ( $\overline{\text{FAULT}}$ ) .....	3-8
Reset Acknowledge ( $\overline{\text{RESET}}$ ) .....	3-8
Reset Request ( $\overline{\text{RESETR}}$ ) .....	3-8
Retry ( $\overline{\text{RETRY}}$ ) .....	3-8
Relinquish and Retry Request Acknowledge ( $\overline{\text{RRRACK}}$ ) .....	3-8
Relinquish and Retry Request ( $\overline{\text{RRREQ}}$ ) .....	3-8
Stop ( $\overline{\text{STOP}}$ ) .....	3-9
3.8 DEVELOPMENT SYSTEM SUPPORT SIGNALS .....	3-9
High Impedance ( $\overline{\text{HIGHZ}}$ ) .....	3-9
Instruction Queue Status (IQS0–IQS1) .....	3-9
Start of Instruction ( $\overline{\text{SOI}}$ ) .....	3-9
3.9 PIN ASSIGNMENTS .....	3-9

## CHAPTER 4. BUS OPERATION

4. BUS OPERATION .....	4-1
4.1 SIGNAL SAMPLING POINTS .....	4-1

4.2 READ AND WRITE OPERATIONS .....	4-2
4.2.1 Read Transaction Using $\overline{\text{SRDY}}$ .....	4-3
4.2.2 Read Transaction Using $\overline{\text{DTACK}}$ .....	4-5
4.2.3 Read Transaction With Wait Cycle Using $\overline{\text{SRDY}}$ .....	4-6
4.2.4 Read Transaction With Two Wait Cycles Using $\overline{\text{DTACK}}$ .....	4-7
4.2.5 Write Transaction Using $\overline{\text{SRDY}}$ .....	4-8
4.2.6 Write Transaction Using $\overline{\text{DTACK}}$ .....	4-8
4.2.7 Write Transaction With Wait Cycle Using $\overline{\text{SRDY}}$ .....	4-8
4.2.8 Write Transaction With Wait Cycle Using $\overline{\text{DTACK}}$ .....	4-12
4.3 READ INTERLOCKED OPERATION.....	4-12
4.4 BLOCKFETCH OPERATION.....	4-15
4.4.1 Blockfetch Transaction Using $\overline{\text{SRDY}}$ .....	4-15
4.4.2 Blockfetch Transaction Using $\overline{\text{DTACK}}$ .....	4-17
4.4.3 Blockfetch Transaction Using $\overline{\text{DTACK}}$ With Wait Cycle on Second Word.....	4-18
4.4.4 Blockfetch Transaction Using $\overline{\text{SRDY}}$ With Wait Cycle on Both Words.....	4-19
4.5 BUS EXCEPTIONS.....	4-20
4.5.1 Faults.....	4-21
$\overline{\text{FAULT}}$ With $\overline{\text{SRDY}}$ .....	4-22
$\overline{\text{FAULT}}$ After $\overline{\text{DTACK}}$ .....	4-23
4.5.2 Retry .....	4-24
4.5.3 Relinquish and Retry.....	4-24
4.6 BLOCKFETCH SPECIAL CASES .....	4-27
4.6.1 Fault on First Word of Blockfetch With Status Code Other Than Prefetch.....	4-27
4.6.2 Fault on First Word of Blockfetch With Status Code of Prefetch.....	4-27
4.6.3 Retry on First Word of Blockfetch .....	4-27
4.6.4 Retry on Second Word of Blockfetch .....	4-27
4.6.5 Relinquish and Retry on Blockfetch .....	4-32
4.7 INTERRUPTS .....	4-32
4.7.1 Interrupt Acknowledge .....	4-32
4.7.2 Autovector Interrupt.....	4-35
4.7.3 Nonmaskable Interrupt.....	4-35
4.8 BUS ARBITRATION .....	4-38
4.8.1 Bus Request During a Bus Transaction.....	4-38
4.8.2 DMA Operation.....	4-41
4.9 RESET .....	4-42
4.9.1 System Reset.....	4-42
4.9.2 Internal Reset .....	4-42
4.9.3 Reset Sequence .....	4-44
4.10 ABORTED MEMORY ACCESSES.....	4-44
4.10.1 Aborted Access on PC Discontinuity With Instruction Cache Hit.....	4-44
4.10.2 Alignment Fault Bus Activity .....	4-46
4.11 SINGLE-STEP OPERATION.....	4-47
4.12 COPROCESSOR OPERATIONS.....	4-48
4.12.1 Coprocessor Broadcast.....	4-48
4.12.2 Coprocessor Operand Fetch.....	4-53
4.12.3 Coprocessor Status Fetch .....	4-54
4.12.4 Coprocessor Data Write .....	4-55
4.13 SUPPLEMENTARY PROTOCOL DIAGRAMS .....	4-56

## 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 Register Mode.....	5-8
5.2.2 Register Deferred Mode.....	5-9
5.2.3 Displacement Mode .....	5-10
5.2.4 Deferred Displacement Mode.....	5-11
5.2.5 Immediate Mode.....	5-12
5.2.6 Absolute Mode.....	5-13
5.2.7 Absolute Deferred Mode .....	5-13
5.2.8 Expanded Operand Mode.....	5-14
5.3 FUNCTIONAL GROUPS.....	5-16
5.3.1 Instruction Byte and Cycle Considerations.....	5-16
5.3.2 Data Transfer Instructions .....	5-17
5.3.3 Arithmetic Instructions.....	5-19
5.3.4 Logical Instructions .....	5-20
5.3.5 Program Control Instructions.....	5-22
5.3.6 Coprocessor Instructions.....	5-27
5.3.7 Stack and Miscellaneous Instructions .....	5-27
5.4 INSTRUCTION SET LISTINGS .....	5-29
5.4.1 Notation .....	5-29
5.4.2 Instruction Set Summary by Mnemonic.....	5-32
5.4.3 Instruction Set Summary by Opcode.....	5-36
5.4.4 Instruction Set Descriptions .....	5-40
Add (ADDB2, ADDH2, ADDW2) .....	5-41
Add, 3 Address (ADDB3, ADDH3, ADDW3).....	5-42
Arithmetic Left Shift (ALSW3).....	5-43
And (ANDB2, ANDH2, ANDW2) .....	5-44
And, 3 Address (ANDB3, ANDH3, ANDW3).....	5-45
Arithmetic Right Shift (ARSB3, ARSH3, ARSW3) .....	5-46
Branch on Carry Clear (BCCB, BCCH).....	5-47
Branch on Carry Set (BCSB, BCSH) .....	5-48
Branch on Equal (BEB, BEH).....	5-49
Branch on Greater Than (Signed) (BGB, BGH).....	5-50
Branch on Greater Than or Equal (Signed) (BGEG, BGEH) .....	5-51
Branch on Greater Than or Equal (Unsigned) (BGEUB, BGEUH) .....	5-52
Branch on Greater Than (Unsigned) (BGUB, BGUH) .....	5-53
Bit Test (BITB, BITH, BITW) .....	5-54
Branch on Less Than (Signed) (BLB, BLH) .....	5-55
Branch on Less Than or Equal (Signed) (BLEB, BLEH) .....	5-56
Branch on Less Than or Equal (Unsigned) (BLEUB, BLEUH) .....	5-57
Branch on Less Than (Unsigned) (BLUB, BLUH).....	5-58
Branch on Not Equal (BNEB, BNEH).....	5-59
Breakpoint Trap (BPT) .....	5-60

Branch (BRB, BRH) .....	5-61
Branch to Subroutine (BSBB, BSBH) .....	5-62
Branch on Overflow Clear (BVCB, BVCH) .....	5-63
Branch on Overflow Set (BVSB, BVSH) .....	5-64
Call Procedure (CALL) .....	5-65
Cache Flush (CFLUSH) .....	5-66
Clear (CLRB, CLRH, CLRW) .....	5-67
Compare (CMPB, CMPH, CMPW) .....	5-68
Decrement (DECB, DECH, DECW) .....	5-69
Divide (DIVB2, DIVH2, DIVW2) .....	5-70
Divide, 3 Address (DIVB3, DIVH3, DIVW3) .....	5-71
Extract Field (EXTFB, EXTFH, EXTFW) .....	5-72
Extended Opcode (EXTOP) .....	5-73
Increment (INCB, INCH, INCW) .....	5-74
Insert Field (INSFB, INSFH, INSFW) .....	5-75
Jump (JMP) .....	5-76
Jump to Subroutine (JSB) .....	5-77
Logical Left Shift (LLSB3, LLSH3, LLSW3) .....	5-78
Logical Right Shift (LRSW3) .....	5-79
Move Complemented (MCOMB, MCOMH, MCOMW) .....	5-80
Move Negated (MNEGB, MNEGH, MNEGW) .....	5-81
Modulo (MODB2, MODH2, MODW2) .....	5-82
Modulo, 3 Address (MODB3, MODH3, MODW3) .....	5-83
Move (MOVB, MOVH, MOVW) .....	5-84
Move Address (Word) (MOVAW) .....	5-85
Move Block (MOVBLW) .....	5-87
Multiply (MULB2, MULH2, MULW2) .....	5-89
Multiply, 3 Address (MULB3, MULH3, MULW3) .....	5-90
Move Version Number (MVERNO) .....	5-91
No Operation (NOP, NOP2, NOP3) .....	5-92
OR (ORB2, ORH2, ORW2) .....	5-93
OR, 3 Address (ORB3, ORH3, ORW3) .....	5-94
Pop (Word) (POPW) .....	5-95
Push Address (Word) (PUSHAW) .....	5-96
Push (Word) (PUSHW) .....	5-97
Return on Carry Clear (RCC) .....	5-98
Return on Carry Set (RCS) .....	5-99
Return on Equal (REQL, REQLU) .....	5-100
Restore Registers (RESTORE) .....	5-101
Return from Procedure (RET) .....	5-102
Return on Greater Than or Equal (Signed) (RGEQ) .....	5-103
Return on Greater Than or Equal (Unsigned) (RGEQU) .....	5-104
Return on Greater Than (Signed) (RETR) .....	5-105
Return on Greater Than (Unsigned) (RGTRU) .....	5-106
Return on Less Than or Equal (Signed) (RLEQ) .....	5-107
Return on Less Than or Equal (Unsigned) (RLEQU) .....	5-108
Return on Less Than (Signed) (RLSS) .....	5-109
Return on Less Than (Unsigned) (RLSSU) .....	5-110

Return on Not Equal (RNEQ, RNEQU) .....	5-111
Rotate (ROTW) .....	5-112
Return from Subroutine (RSB) .....	5-113
Return on Overflow Clear (RVC) .....	5-114
Return on Overflow Set (RVS) .....	5-115
Save Registers (SAVE) .....	5-116
Coprocessor Operation (No Operands) (SPOP) .....	5-117
Coprocessor Operation Read (SOPRS, SOPRD, SOPRT) .....	5-118
Coprocessor Operation, 2 Address (SPOPS2, SPOPD2, SPOPT2) .....	5-119
Coprocessor Operation Write (SPOPWS, SPOPWD, SPOPW) .....	5-120
String Copy (STRCPY) .....	5-121
String End (STREND) .....	5-123
Subtract (SUBB2, SUBH2, SUBW2) .....	5-124
Subtract, 3 Address (SUBB3, SUBH3, SUBW3) .....	5-125
Swap (Interlocked) (SWAPBI, SWAPHI, SWAPWI) .....	5-126
Test (TSTB, TSTH, TSTW) .....	5-127
Exclusive Or (XORB2, XORH2, XORW2) .....	5-128
Exclusive Or, 3 Address (XORB3, XORH3, XORW3) .....	5-129

## 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 .....	6-3
6.2 STRUCTURE OF A PROCESS .....	6-4
6.2.1 Execution Privilege .....	6-4
6.2.2 Execution Stack .....	6-5
6.2.3 Process Control Block .....	6-6
Initial Context for a Process .....	6-9
Saved Context for a Process .....	6-10
Memory Specifications .....	6-10
6.2.4 Processor Status Word .....	6-11
6.3 SYSTEM CALL .....	6-11
6.3.1 Gate Mechanism .....	6-14
Pointer Table .....	6-14
Handling-Routine Tables .....	6-14
6.3.2 GATE Instruction .....	6-15
First Entry Point .....	6-15
Second Entry Point - The Gate Mechanism .....	6-16
6.3.3 Return-From-Gate Instruction .....	6-18
6.4 PROCESS SWITCHING .....	6-18
6.4.1 Context Switching Strategy .....	6-19
R Bit .....	6-19
I Bit .....	6-19
6.4.2 Call Process Instruction .....	6-22
6.4.3 Return-to-Process Instruction .....	6-24
6.5 INTERRUPTS .....	6-25

6.5.1	Interrupt-Handler Model.....	6-25
6.5.2	Interrupt Mechanism.....	6-26
	Full-Interrupt Handler's PCB.....	6-27
	Interrupt Stack and ISP.....	6-28
	Interrupt-Vector Table.....	6-29
6.5.3	On-Interrupt Microsequence.....	6-30
6.5.4	Returning From an Interrupt.....	6-31
	Full Interrupts.....	6-31
	Quick Interrupts.....	6-31
6.6	EXCEPTIONS.....	6-32
6.6.1	Levels of Exception Severity.....	6-32
6.6.2	Exception Handler.....	6-32
6.6.3	Exception Microsequences.....	6-33
	Normal Exceptions.....	6-33
	Stack Exceptions.....	6-36
	Process Exceptions.....	6-37
	Reset Exceptions.....	6-37
6.7	MEMORY MANAGEMENT FOR VIRTUAL MEMORY SYSTEMS.....	6-38
6.7.1.	Initializing the Memory Management Unit.....	6-43
	Defining Virtual Memory.....	6-43
	Peripheral Mode.....	6-43
6.7.2	MMU Interactions.....	6-43
	MMU Exceptions.....	6-44
	Flushing.....	6-44
6.7.3	Efficient Mapping Strategies.....	6-44
6.7.4	Object Traps.....	6-45
6.7.5	Indirect Segment Descriptors.....	6-45
6.7.6	Using the Cacheable Bit.....	6-45
6.7.7	Using the Page-Write Fault.....	6-45
6.7.8	Access Protection.....	6-46
6.7.9	Using the Software Bits.....	6-46
6.8	OPERATING SYSTEM INSTRUCTIONS.....	6-46
6.8.1	Notation.....	6-46
6.8.2	Privileged Instructions.....	6-48
	Call Process (CALLPS).....	6-49
	Disable Virtual Pin and Jump (DISVJMP).....	6-51
	Enable Virtual Pin and Jump (ENBVJMP).....	6-52
	Interrupt Acknowledge (INTACK).....	6-53
	Return to Process (RETPS).....	6-54
	Wait (WAIT).....	6-56
6.8.3	Nonprivileged Instructions.....	6-57
	Gate (GATE).....	6-58
	Move Translated Word (MOVTRW).....	6-61
	Return from Gate (RETG).....	6-63
6.8.4	Microsequences.....	6-65
	On-Normal Exception.....	6-66
	On-Stack Exception.....	6-68

On-Process Exception .....	6-69
On-Reset Exception .....	6-70
On-Interrupt.....	6-71
XSWITCH Microsequence .....	6-74

## GLOSSARY AND ACRONYMS

## INDEX

### LIST OF FIGURES

Figure 1-1. WE 32100 Microprocessor .....	1-1
Figure 1-2. WE 321AP Microprocessor Analysis Pod .....	1-5
Figure 1-3. WE 321EB Microprocessor Evaluation Board .....	1-6
Figure 2-1. WE 32100 Microprocessor Block Diagram .....	2-1
Figure 2-2. Programmer's Model for User Registers .....	2-2
Figure 2-3. Conventional Register Set .....	2-3
Figure 2-4. WE 32100 Microprocessor Stack .....	2-4
Figure 2-5. High-Level Language Register Support Group .....	2-5
Figure 2-6. Operating System Register Support Group .....	2-6
Figure 2-7. Byte Data .....	2-7
Figure 2-8. Halfword Data .....	2-7
Figure 2-9. Word Data .....	2-7
Figure 2-10. Floating-Point Data Types .....	2-8
Figure 2-11. Extraction of a Bit Field .....	2-9
Figure 2-12. Extending Data to 32 Bits .....	2-10
Figure 2-13. Word Storage In Memory .....	2-11
Figure 2-14. Instruction Storage In Memory .....	2-13
Figure 2-15. Operand Format .....	2-13
Figure 2-16. Word Storage Within an Instruction .....	2-13
Figure 2-17. Condition Flags .....	2-14
Figure 3-1. Signal Grouping Diagram .....	3-1
Figure 3-2. 125-Pin Square, Ceramic Pin Grid Array .....	3-10
Figure 4-1. Signal Sampling Points .....	4-1
Figure 4-2. Read Transaction Using $\overline{\text{SRDY}}$ .....	4-4
Figure 4-3. Read Transaction Using $\overline{\text{DTACK}}$ .....	4-5
Figure 4-4. Read Transaction with One Wait Cycle Using $\overline{\text{SRDY}}$ .....	4-6
Figure 4-5. Read Transaction with Two Wait Cycles Using $\overline{\text{DTACK}}$ .....	4-7
Figure 4-6. Write Transaction Using $\overline{\text{SRDY}}$ .....	4-9
Figure 4-7. Write Transaction Using $\overline{\text{DTACK}}$ .....	4-10
Figure 4-8. Write Transaction With Two Wait Cycles Using $\overline{\text{SRDY}}$ .....	4-11
Figure 4-9. Write Transaction With Wait Cycle Using $\overline{\text{DTACK}}$ .....	4-13
Figure 4-10. Read Interlocked Transaction Using $\overline{\text{DTACK}}$ .....	4-14

Figure 4-11. Blockfetch Transaction Using $\overline{\text{SRDY}}$ .....	4-16
Figure 4-12. Blockfetch Transaction Using $\overline{\text{DTACK}}$ .....	4-17
Figure 4-13. Blockfetch Transaction Using $\overline{\text{DTACK}}$ With Wait Cycle on Second Word .....	4-18
Figure 4-14. Blockfetch Transaction Using $\overline{\text{SRDY}}$ With Wait Cycles on Both Words .....	4-19
Figure 4-15. Asynchronous Fault Without $\overline{\text{DTACK}}$ and $\overline{\text{SRDY}}$ (Read Transaction) .....	4-21
Figure 4-16. Fault With Synchronous $\overline{\text{Ready}}$ ( $\overline{\text{SRDY}}$ ); i.e., Synchronous Fault .....	4-22
Figure 4-17. Fault After Assertion of $\overline{\text{DTACK}}$ (Write Transaction is Shown) .....	4-23
Figure 4-18. Retry of Transaction (Read Transaction is Shown) .....	4-25
Figure 4-19. Relinquish and Retry .....	4-26
Figure 4-20. Fault on First Word of Blockfetch Transaction With Access Status Code Other Than Prefetch .....	4-28
Figure 4-21. Fault on First Word of Blockfetch Transaction With Access Status Code of Prefetch .....	4-29
Figure 4-22. Retry on First Word of Blockfetch Transaction .....	4-30
Figure 4-23. Retry on Second Word of Blockfetch Transaction .....	4-31
Figure 4-24. Interrupt Acknowledge .....	4-33
Figure 4-25. Autovector Interrupt Acknowledge .....	4-36
Figure 4-26. Nonmaskable Interrupt Acknowledge .....	4-37
Figure 4-27. Bus Request During a Transaction .....	4-39
Figure 4-28. Reset Sequence .....	4-44
Figure 4-29. Aborted Access on Instruction-Cache Hit With PC Discontinuity .....	4-45
Figure 4-30. Alignment Fault Bus Activity (Write Transaction is Shown) .....	4-46
Figure 4-31. Start of Single-Step Operation .....	4-47
Figure 4-32. Single-Step Operation .....	4-48
Figure 4-33. Coprocessor Command and ID Transfer .....	4-49
Figure 4-34. Coprocessor Command and ID Transfer (No Coprocessor Present) .....	4-52
Figure 4-35. Coprocessor Operand Fetch .....	4-53
Figure 4-36. Coprocessor Status Fetch Using $\overline{\text{SRDY}}$ .....	4-54
Figure 4-37. Coprocessor Data Write .....	4-55
Figure 4-38. Read Transaction Followed by a Read Transaction .....	4-57
Figure 4-39. Read Transaction Followed by a Write Transaction Using $\overline{\text{DTACK}}$ .....	4-58
Figure 4-40. Write Transaction Followed by a Write Transaction .....	4-59
Figure 4-41. Write Transaction Followed by a Read Transaction .....	4-60
Figure 4-42. Double-Word Program Fetch Without Blockfetch Transaction Using $\overline{\text{DTACK}}$ .....	4-61
Figure 4-43. Bus Arbitration During Relinquish and Retry .....	4-62
Figure 5-1. Instruction Format .....	5-3
Figure 5-2. Operand Format .....	5-4
Figure 5-3. Descriptor Byte Format .....	5-4
Figure 5-4. Register Mode Example .....	5-9
Figure 5-5. Deferred Addressing Using a Pointer .....	5-9
Figure 5-6. Register Deferred Mode Example .....	5-9
Figure 5-7. Example of a <code>MOVB 0x30 (% r2), % r3</code> .....	5-10
Figure 5-8. A Displacement Mode Source Operand .....	5-11

Figure 5-9. Deferred Displacement Addressing .....	5-11
Figure 5-10. A Deferred Displacement Mode Source Operand .....	5-12
Figure 5-11. A 32-Bit Immediate Source Operand .....	5-13
Figure 5-12. An Absolute Mode Source Operand .....	5-14
Figure 5-13. An Absolute Deferred Mode Source Operand .....	5-14
Figure 5-14. Expanded Operand Mode Descriptor Bytes .....	5-15
Figure 5-15. Expanded Operand Mode Example .....	5-16
Figure 5-16. Stack After CALL-SAVE Sequence .....	5-26
Figure 6-1. Memory Map .....	6-7
Figure 6-2. A Typical Process Control Block .....	6-9
Figure 6-3. Pointer and Handling Table Indexing .....	6-17
Figure 6-4. A PCB on an Initial Process Switch to a Process .....	6-21
Figure 6-5. A PCB on a Process Switch During Execution of a Process .....	6-22
Figure 6-6. An Interrupt Stack .....	6-29
Figure 6-7. On-Normal Exception Indexing .....	6-34
Figure 6-8. Virtual Memory Space for a Process .....	6-39
Figure 6-9. Virtual Address Fields for a Contiguous Segment .....	6-40
Figure 6-10. Virtual Address Fields for a Paged Segment .....	6-40
Figure 6-11. Virtual to Physical Translation for Contiguous Segments .....	6-41
Figure 6-12. Virtual to Physical Translation for Paged Segments .....	6-42

## LIST OF TABLES

Table 3-1. Pin Descriptions 125-Pin Package .....	3-10
Table 4-1. Simultaneously Assorted Exception Conditions .....	4-20
Table 4-2. Interrupt Level Code Assignments .....	4-34
Table 4-3. Interrupt Acknowledge Summary .....	4-40
Table 4-4. Output Signal States After DMA Request is Acknowledged .....	4-41
Table 4-5. Output States on Reset .....	4-43
Table 5-1. Register Set .....	5-2
Table 5-2. Addressing Modes .....	5-5
Table 5-3. Addressing Modes by Type .....	5-7
Table 5-4. Options for <i>type</i> in Expanded Operand Mode .....	5-15
Table 5-5. Data Transfer Instruction Group .....	5-18
Table 5-6. Arithmetic Instruction Group .....	5-19
Table 5-7. Logical Instruction Group .....	5-21
Table 5-8. Program Control Instruction Group .....	5-23
Table 5-9. Coprocessor Instruction Group .....	5-28
Table 5-10. Stack and Miscellaneous Instruction Group .....	5-28
Table 5-11. Condition Flag Code Assignments .....	5-30
Table 5-12. Assembly Language Operators and Symbols .....	5-31
Table 5-13. Instruction Set Summary by Mnemonic .....	5-32
Table 5-14. Instruction Set Summary by Opcode .....	5-36

Table 6-1.	Operating System Group .....	6-2
Table 6-2.	Processor Status Word Fields .....	6-12
Table 6-3.	Severity Levels for Exceptions .....	6-32
Table 6-4.	Normal Exceptions (ET=3) .....	6-35
Table 6-5.	Stack Exceptions (ET=2) .....	6-36
Table 6-6.	Process Exceptions (ET=1) .....	6-37
Table 6-7.	Reset Exceptions (ET=0) .....	6-38



## **Chapter 1**

### **Introduction**

## CHAPTER 1. INTRODUCTION

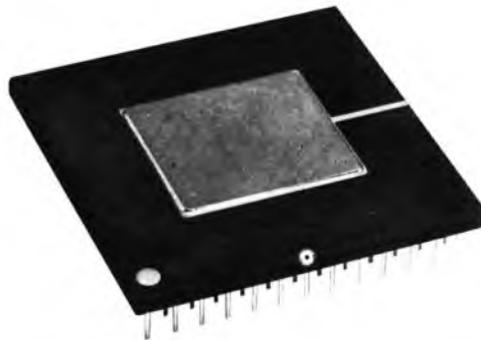
### CONTENTS

1. INTRODUCTION.....	1-1
1.1 DEVELOPMENT .....	1-1
1.2 ARCHITECTURE.....	1-2
1.3 INSTRUCTION SET.....	1-3
1.4 OPERATING SYSTEM SUPPORT.....	1-3
1.5 SUPPORT PRODUCTS .....	1-4
1.6 CHAPTER SUMMARY .....	1-7
1.6.1 Chapter 2. Registers, Data, and Instruction Formats .....	1-7
1.6.2 Chapter 3. Signal Descriptions .....	1-7
1.6.3 Chapter 4. Bus Operations .....	1-7
1.6.4 Chapter 5. Instruction Set and Addressing Modes .....	1-7
1.6.5 Chapter 6. Operating System Considerations.....	1-7

## **1. INTRODUCTION**

The *WE 32100* Microprocessor (CPU) is a high-performance, single-chip, 32-bit central processing unit designed for efficient operation in a high-level language environment. The *WE 32100* Microprocessor 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 32100* CPU is implemented in 1.5-micron CMOS technology and packaged in a 125-pin square, ceramic pin grid array. It is available in 10-, 14-, and 18-MHz frequency versions.



**Figure 1-1. The *WE 32100* Microprocessor**

The system memory space is addressed over a full 32-bit address bus 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.

The *WE 32100* Microprocessor is an efficient execution vehicle for operating systems and high-level languages. The included operating system instructions 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 32101* Memory Management Unit (MMU) for virtual memory systems.

### **1.1 DEVELOPMENT**

The AT&T family of 32-bit processors is a direct descendant of the 8-bit microprocessor developed at AT&T Bell Laboratories in the mid-1970s. This 8-bit microprocessor was

## INTRODUCTION

### Architecture

unique in its fabrication in CMOS technology. CMOS, which has since become the technology of choice for state-of-the-art microprocessors, was developed early in the 1970s at Bell Laboratories for use in all AT&T microprocessors. CMOS devices use significantly less power than equivalent devices designed in n-channel MOS (NMOS), are highly immune to signal interference, and can operate over a wide range of voltage and temperature. These characteristics are especially important for designing the higher density devices typified by 32-bit microprocessors.

The 8-bit microprocessor was followed by a single-chip, 4-bit microcomputer. The significance of the 4-bit microcomputer was the introduction and use of AT&T Bell Laboratories internally developed, computer-aided design (CAD) technologies in its design and development. The design, development, testing, and introduction of a new microprocessor typically requires two to three years; the computer-aided design and testing cycle developed for the 4-bit microcomputer enabled the *WE 32001 Processor Module* – the next CPU in the series – to be developed and operational in less than half of that time.

The *WE 32001 Processor Module* required still higher levels of software design, development, and hardware technology. The design of this first 32-bit microprocessor involved sophisticated innovations and refinements to both CMOS technology and CAD techniques that had been used previously on the 4-bit microcomputer. As a result, the *WE 32001 Processor Module* was developed and became successfully operational in thirteen months.

The computer-aided design and development tools used on the *WE 32001 Processor Module* had an equally significant impact on the introduction of the *WE 32100 Microprocessor*. The *WE 32100 Microprocessor* was developed and fabricated in eleven months, with the first device containing only one minor layout error in over 180,000 transistors.

## 1.2 ARCHITECTURE

The *WE 32100 Microprocessor* performs all the system address generation, control, memory access, and processing functions required in a 32-bit microcomputer system. Execution speed is enhanced by its unique pipelined architecture. Using this architecture, the microprocessor overlaps the execution of instructions while fetching subsequent instructions. As each is fetched from memory it is stored in an internal instruction cache, resulting in even greater operating efficiency.

The CPU utilizes a combination of address and data strobes and interface and control signals to provide the bus protocol required for efficient data transfer. The protocol facilitates interfacing to commercial memories and peripherals, as well as providing wait-state generation for handshaking with slow peripherals. In addition, the CPU provides special coprocessor signals for a high throughput coprocessing environment.

The architecture for the *WE 32100 Microprocessor* is described in detail in **Chapter 2. Registers, Data, and Instruction Formats.**

### **1.3 INSTRUCTION SET**

The *WE 32100* Microprocessor supports a powerful instruction set that includes standard data transfer, arithmetic, and logical operations for microprocessors, plus several unique operations. Its many program control instructions (branch, jump, return) provide flexibility for altering the sequence of execution. Other instructions are designed to aid in process switching for operating systems by manipulating the context of the processor with a minimum of code. In addition, special coprocessor instructions are included in the instruction set to implement a high-speed interface with the special purpose coprocessor available for the *WE 32100* Microprocessor.

Eighteen addressing modes are provided that include special high-level language support modes such as frame-pointer short offset and argument-pointer short offset. These modes are designed for referring to local variables of high-level functions and function arguments.

A detailed description for the *WE 32100* Microprocessor Instruction Set is provided in **Chapter 5. Instruction Set and Addressing Modes.**

### **1.4 OPERATING SYSTEM SUPPORT**

The *WE 32100* Microprocessor is designed for high-level language and operating system support. To aid in the design of process-oriented systems, it provides:

- Four execution privilege levels: kernel, executive, supervisor, and user
- Flexible transfer of execution control between privilege levels
- Capability of having the operating system contained within the address space of every process
- Support of explicit process switching by a scheduler
- Implicit switching of processes through the interrupt structure
- Nested exception handling structure, with different mechanisms used for different exceptions.

The processor groups each switchable process context into a compact area in memory called the process control block. This feature, with the special operating system instructions and microsequences, provides the programmer with an excellent tool for the creation and support of process-oriented systems.

A discussion of the techniques for efficient operating system design using the *WE 32100* Microprocessor and the use of the *WE 32101* MMU in a virtual memory operating system is provided in **Chapter 6. Operating System Considerations.**

## INTRODUCTION

### Support Products

#### 1.5 SUPPORT PRODUCTS

Software support for the *WE 32100* Microprocessor is available through the *WE 321SG* Software Generation Programs (SGP). This collection of programs and utilities provides everything necessary for rapid development of software in C language. The entire SGP resides in the *UNIX* Operating System and includes a C compiler, an assembler, a link editor, and various utility programs.

Development support is available through the *WE 321DS* Microprocessor Development System. This is a powerful development tool that expedites the integration of hardware and software into a finished application. It permits debugging of hardware and software to occur in parallel. The development system components include the *WE 321AP* Microprocessor Analysis Pod, the *WE 321SD* Development Software Programs, a *UNIX* System host, and a logic analyzer. The modular design of the development system enables the user to configure the system for maximum productivity from initial hardware debugging through the final stage of hardware and software integration. The *WE 321AP* Microprocessor Analysis Pod is shown on Figure 1-2.

Prototyping and performance evaluation support is available through the *WE 321EB* Microprocessor Evaluation Board. The evaluation board is a single-board microcomputer evaluation system that provides a prototyping vehicle to evaluate the hardware and software capabilities and performance of the *WE 32100* Microprocessor in an application environment. The board is supplied with a *WE 32100* Microprocessor CPU, a *WE 32101* MMU, a *WE 32102* Clock, a ROM-based monitor, read/write memory (RAM), and sockets for additional memory. Also included are address decoding circuitry, RS-232C ports, programmable parallel I/O lines, programmable interval timers, and an interrupt controller. The *WE 321EB* Microprocessor Evaluation Board is shown on Figure 1-3.

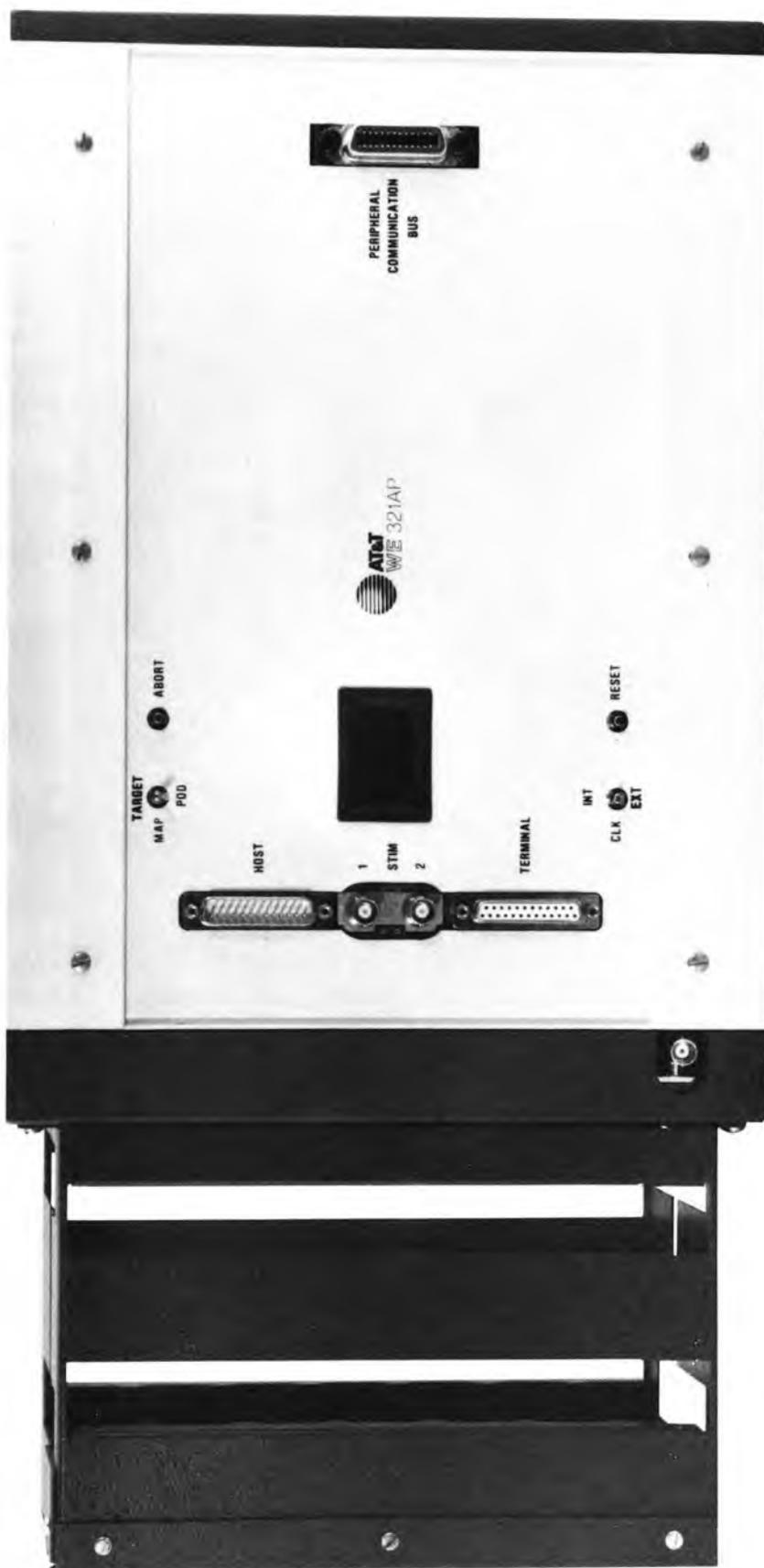
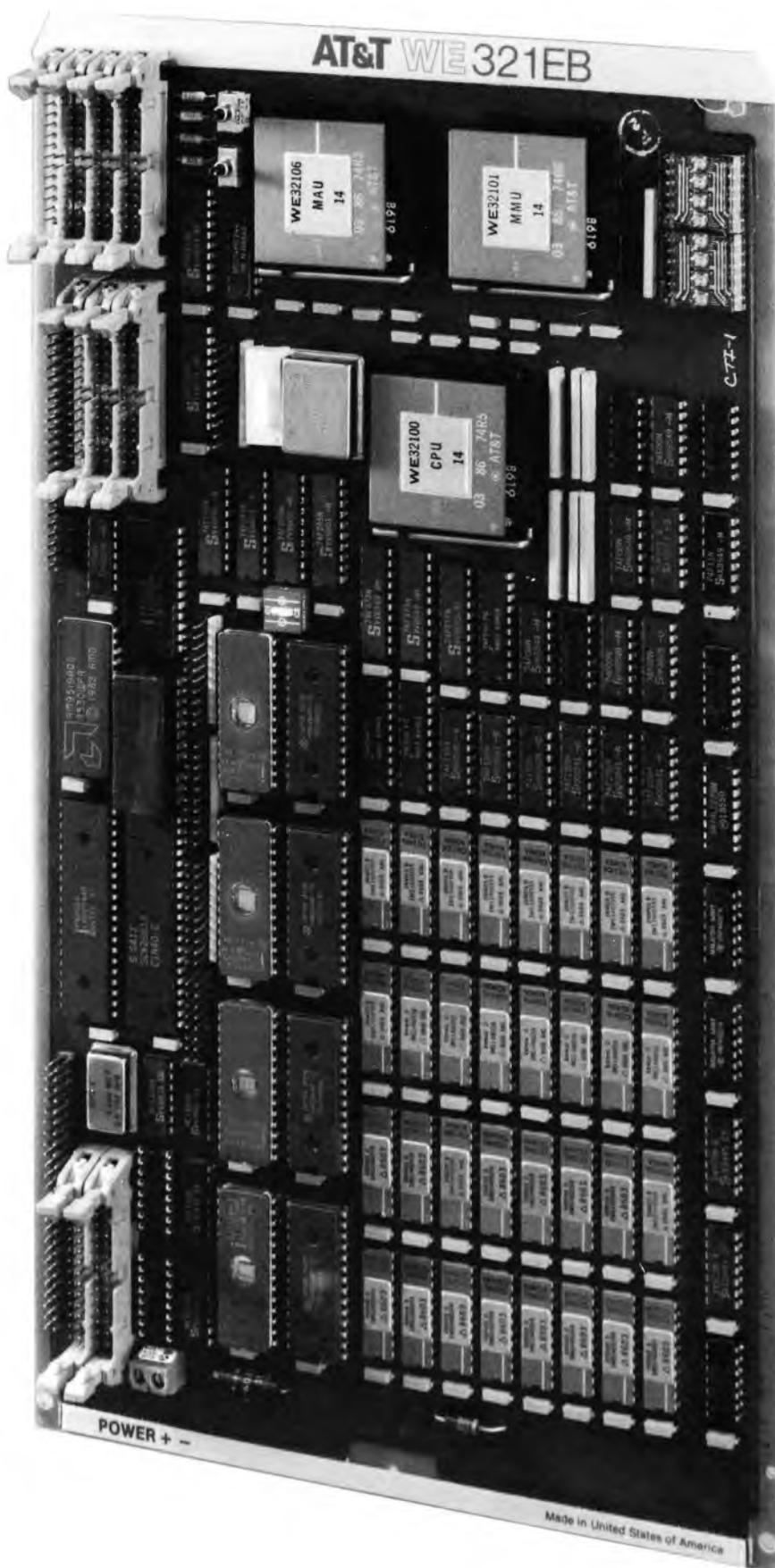


Figure 1-2. WE 321AP Microprocessor Analysis Pod

**INTRODUCTION**  
**Support Products**



**Figure 1-3. WE 321EB Microprocessor Evaluation Board**

## **1.6 CHAPTER SUMMARY**

This section provides a brief summary of all the remaining chapters in this manual. For a detailed description of the electrical, clock, timing, and thermal requirements refer to the *WE 32100 Microprocessor Data Sheet*.

### **1.6.1 Chapter 2. Registers, Data, and Instruction Formats**

Chapter 2 describes in detail the *WE 32100* Microprocessor's sixteen 32-bit registers (r0—r15). Also described are the data types which are supported by the microprocessor, i.e., byte, halfword, word, and bit field. A brief description of the microprocessor instruction set containing information about the syntax of the instructions is also included.

### **1.6.2 Chapter 3. Signal Descriptions**

The *WE 32100* Microprocessor input and output signals are described in this chapter.

### **1.6.3 Chapter 4. Bus Operations**

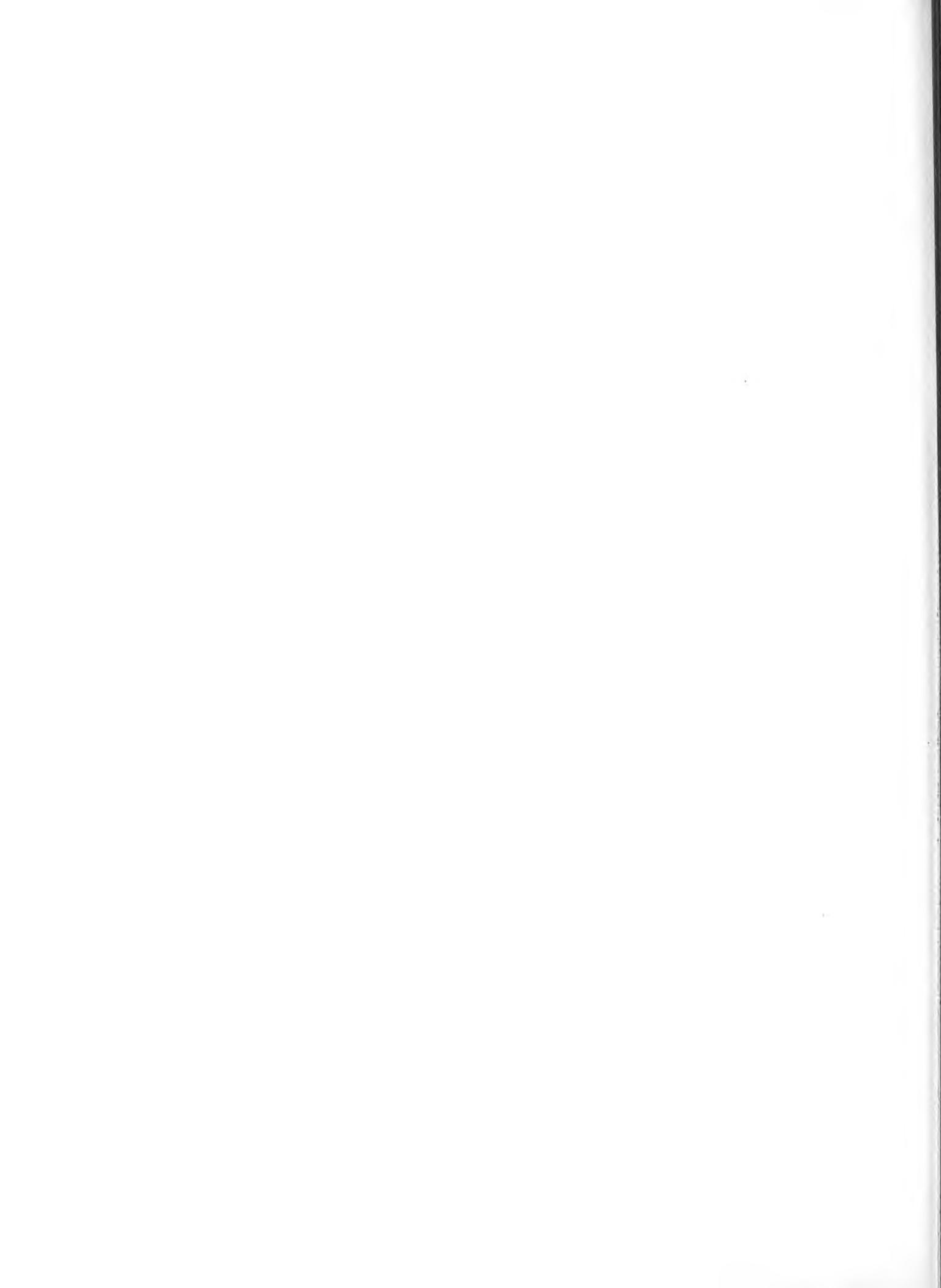
This chapter covers the bus protocol for the *WE 32100* Microprocessor. Among the topics discussed are: signal sampling points, read and write operations, read interlocked operation, blockfetch operation, bus exceptions, blockfetch special cases, interrupts, bus arbitration, reset, aborted memory accesses, single-step operation, and coprocessor operations. Additional protocol diagrams are also provided.

### **1.6.4 Chapter 5. Instruction Set and Addressing Modes**

Chapter 5 details the *WE 32100* Microprocessor instruction set. A listing of the syntax, opcodes, addressing mode, condition flags, exceptions, and examples are provided for each instruction.

### **1.6.5 Chapter 6. Operating System Considerations**

To make full use of the power of the *WE 32100* Microprocessor as an execution vehicle for today's efficient process-oriented operating systems, this chapter presents the operating system considerations important to the system designer.



**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 .....	2-2
2.2 CONVENTIONAL REGISTER SET .....	2-3
2.2.1 Stack Pointer .....	2-3
2.2.2 Program Counter .....	2-4
2.3 HIGH-LEVEL LANGUAGE SUPPORT GROUP .....	2-4
2.3.1 Frame Pointer .....	2-4
2.3.2 Argument Pointer .....	2-5
2.4 OPERATING SYSTEM SUPPORT GROUP .....	2-5
2.4.1 Processor Status Word .....	2-5
2.4.2 Process Control Block Pointer .....	2-5
2.4.3 Interrupt Stack Pointer .....	2-5
2.5 DATA TYPES .....	2-6
2.6 SIGN AND ZERO EXTENSION .....	2-9
2.7 DATA STORAGE IN MEMORY .....	2-10
2.8 REGISTER DATA STORAGE .....	2-11
2.9 INSTRUCTION SET .....	2-11
2.10 INSTRUCTION STORAGE IN MEMORY .....	2-12
2.11 CONDITION FLAGS .....	2-12

2. REGISTERS, DATA AND INSTRUCTION FORMATS

The WE 32100 Microprocessor was the first 32-bit microprocessor with separate 32-bit address and data buses. Using either physical or virtual addresses, the 32-bit address bus can access over 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 32100 Microprocessor illustrating its four major sections—main controller, fetch unit, executive unit, and bus interface control—is shown on Figure 2-1.

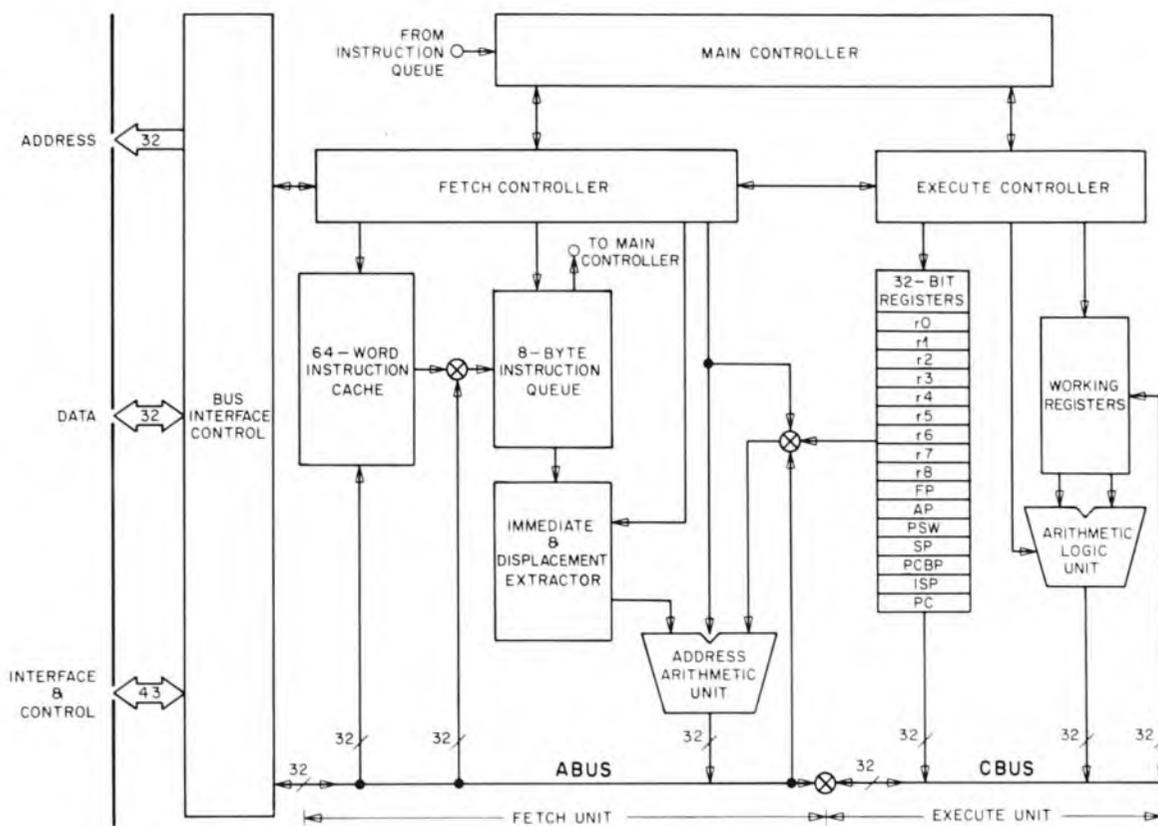


Figure 2-1. WE 32100 Microprocessor Block Diagram

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 which significantly enhance the performance of the WE 32100 Microprocessor. One of these features is a 64-word instruction cache which stores prefetched instructions from memory.

# REGISTERS, DATA, AND INSTRUCTION FORMATS

## User Registers

The prefetched instructions are read from memory at the same time as instructions are executed. This technique is known as pipelining. Thus, the normal suspension of execution, while the processor waits for an instruction to be fetched, is avoided when 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 sixteen 32-bit user-accessible registers, consisting of nine general-purpose (r0-r8) and seven special-purpose registers (r9-r15).

### 2.1 USER REGISTERS

Figure 2-2 shows the programming model for the microprocessor's sixteen 32-bit registers (r0-r15). This register set is designed for efficient support of high-level language program execution. All of these registers, except the program counter (r15) and processor status word (r11), may be accessed in any addressing mode. The process status word (r11), process control block pointer (r13), and interrupt stack pointer (r14) 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.

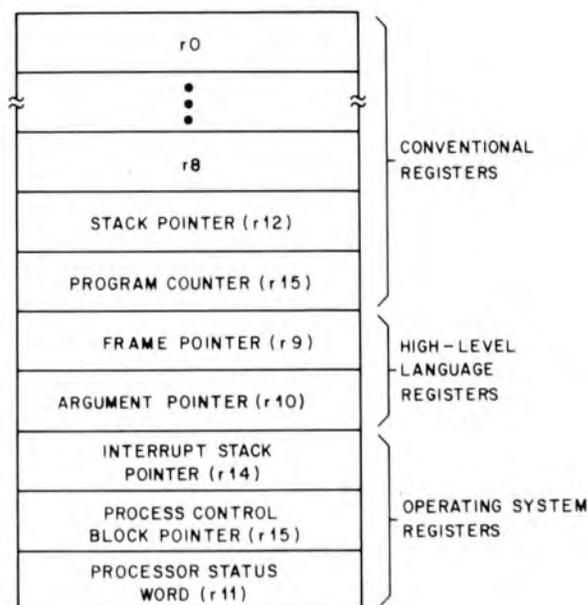
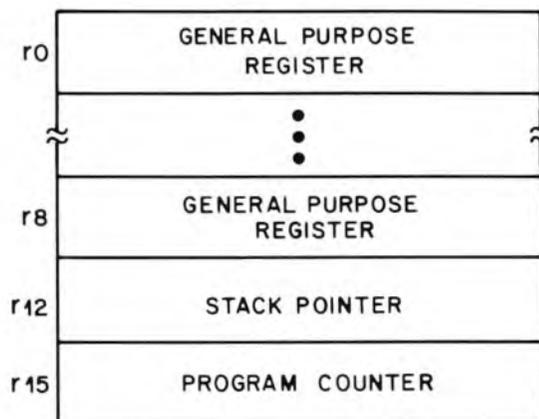


Figure 2-2. Programmer's Model for User Registers

The WE 32100 Microprocessor registers support an extremely powerful assembly language. These registers were also designed for the efficient support of procedure-oriented high-level languages, such as C, and process-oriented operating systems, such as the UNIX Operating System.

Although all of the sixteen registers are available to assembly language programmers, it is useful to separate the register 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.



**Figure 2-3. Conventional Register Set**

## 2.2 CONVENTIONAL REGISTER SET

The conventional registers consist of the nine general-purpose registers, the stack pointer, and the program counter. This is illustrated on Figure 2-3. This set, including condition flags, is typically associated with an assembly language programming model of a microprocessor. In the *WE 32100* Microprocessor, the condition flags — indicators of the processor's current status — are contained within the processor status word register.

The nine general-purpose registers, r0 through r8, 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. Instruction Set and Addressing Modes**).

### 2.2.1 Stack Pointer

The stack pointer (SP), 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

## REGISTERS, DATA, AND INSTRUCTION FORMATS

### Program Counter

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.

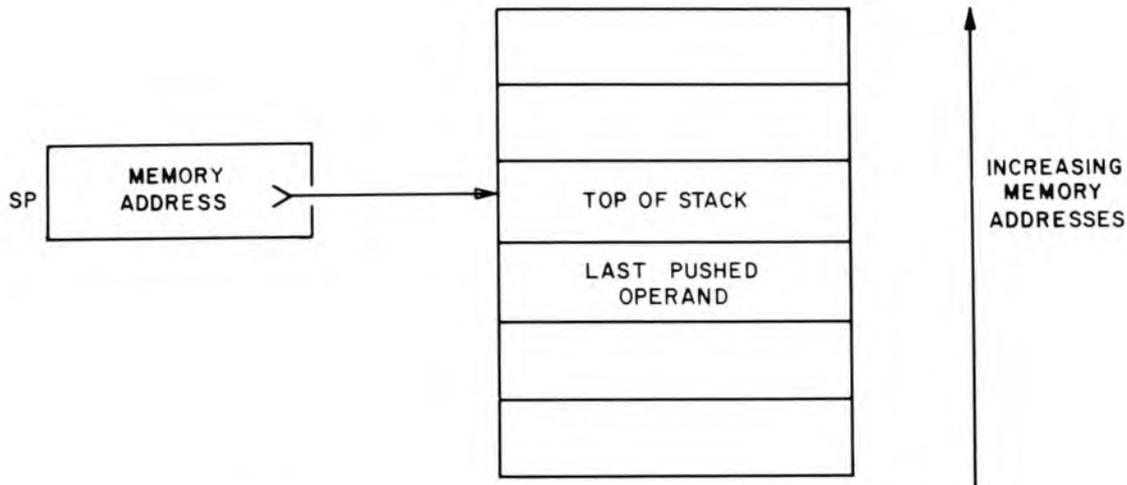


Figure 2-4. The WE 32100 Microprocessor Stack

### 2.2.2 Program Counter

The program counter (PC), 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

The frame pointer (FP), r9, points to the beginning stack location in which the local variables of the currently running program, procedure, or function are stored. The frame pointer is implicitly changed by the save (SAVE) and restore registers' (RESTORE) assembly instructions.

**2.3.2 Argument Pointer**

The argument pointer (AP), 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.

r9	Frame Pointer
r10	Argument Pointer

**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**

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

**2.4.2 Process Control Block Pointer**

The process control block pointer (PCBP), 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; boundaries for an execution stack; and block move specifications (and possible memory specifications) for the process. The PCBP may only be written when the microprocessor is in the kernel mode (see **Chapter 6. Operating System Considerations**).

**2.4.3 Interrupt Stack Pointer**

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

# REGISTERS, DATA, AND INSTRUCTION FORMATS

## Data Types

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 32100* Microprocessor are byte, halfword, word, floating point (word, double word, and double extended word) and bit field data. Floating point data is provided only when the *WE 32106* Math Acceleration Unit (MAU) is used with the *WE 32100* CPU. 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 divisible by two. Bits are numbered from right to left starting with zero, the LSB, and ending with 15, the MSB, as illustrated on Figure 2-8.

A word is a 32-bit quantity. Data words may appear at any address divisible by four. Bits are numbered from right to left starting with zero, the LSB, and ending with 31, the MSB, as illustrated on Figure 2-9.

Floating point data types may appear at any address in memory divisible by four. Figure 2-10 illustrates the floating point data types supported by the assembler.

Each of these four data types may be interpreted as either a signed or unsigned quantity, with signed data represented in two's complement form.

A bit field is a sequence of 1 to 32 bits extracted from a 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 2-11 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

## Data Types

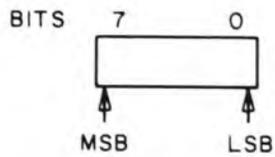


Figure 2-7. Byte Data

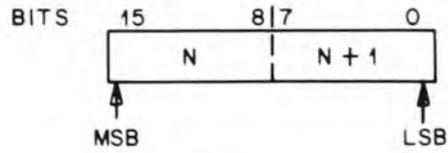


Figure 2-8. Halfword Data

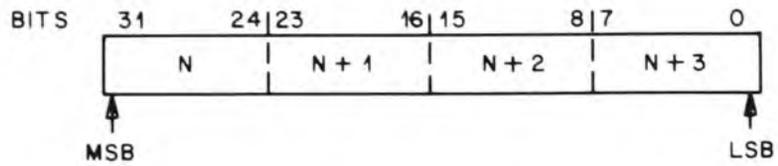


Figure 2-9. Word Data

# REGISTERS, DATA, AND INSTRUCTION FORMATS

## Data Types

Bit	31	30	23	22	0
Field	Sign	Exponent		Fraction	

### A. Single-Precision Floating-Point Data Type

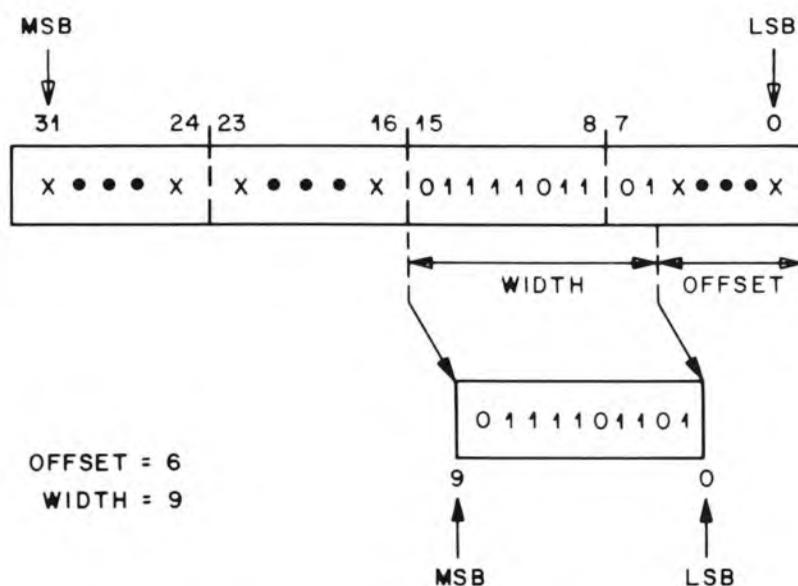
Bit	63	62	52	51	0
Field	Sign	Exponent		Fraction	

### 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

Figure 2-10. Floating-Point Data Types



**Figure 2-11. Extraction of a Bit Field**

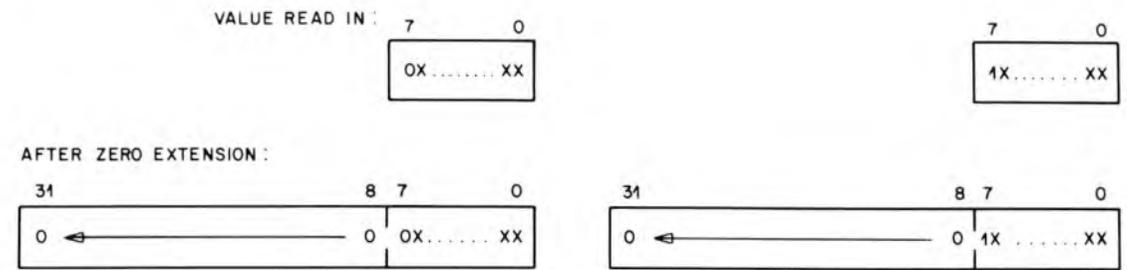
## 2.6 SIGN AND ZERO EXTENSION

All CPU operations are performed on 32-bit quantities even though an instruction may specify a byte or halfword operand. The *WE 32100* 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 data or halfwords and *zero extension* when reading unsigned data or bytes (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 applied can be changed using the expanded-operand type mode described in **5.2.8 Expanded-Operand Type Mode**. For sign extension, the value of the MSB is replicated to fill the 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 2-12 illustrates sign and zero extension.

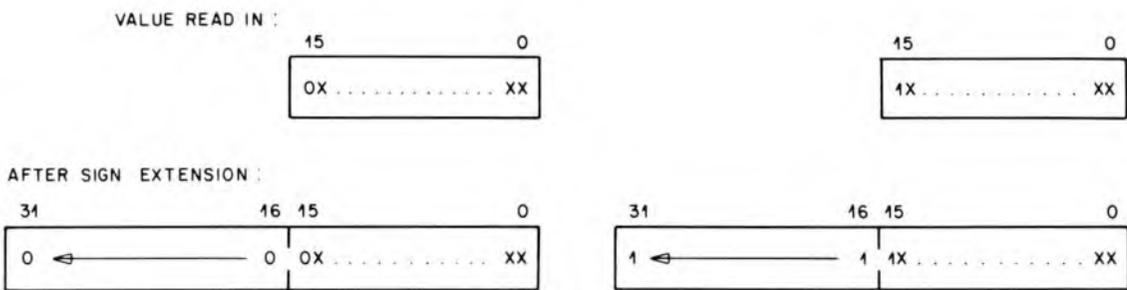
An arithmetic, logical, data transfer, or bit field operation always yields an intermediate result that is 32 bits in length. If the result is to be stored in a register, the processor writes all 32 bits to that register. The processor automatically strips any surplus high-order bits from a result when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

# REGISTERS, DATA, AND INSTRUCTION FORMATS

## Data Storage In Memory



A. BYTE DATA (UNSIGNED)



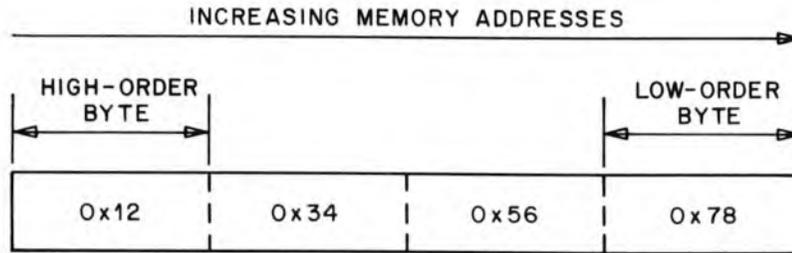
B. HALFWORD DATA (SIGNED)

Figure 2-12. Extending Data to 32 Bits

## 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. Boundary restrictions apply to the starting location of halfwords and words. Halfwords may only be stored at addresses divisible by 2, and words only at addresses divisible by 4. The microprocessor generates a fault if these boundaries are violated.

Figure 2-13 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 2-13. Word Storage in Memory**

## 2.8 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* 32100 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. When storing byte operands into a register unsigned data is assumed, 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 using an expanded operand type addressing mode (this addressing mode is described in **Chapter 5. Instruction Set and Addressing Modes**). 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 using an expanded operand type addressing mode.

## 2.9 INSTRUCTION SET

The *WE* 32100 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 discussed in **Chapter 6. Operating System Considerations**.

## REGISTERS, DATA, AND INSTRUCTION FORMATS

### Instruction Storage in Memory

The microprocessor instructions are mnemonic-based assembly language statements. However, programs may be written in C language and translated into assembly language by its C compiler.

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.

### 2.10 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 2-14 illustrates the general format of an assembly instruction as it is stored in memory. Each individual operand shown on Figure 2-14 consists of a descriptor byte, followed by up to four bytes of data (see Figure 2-15).

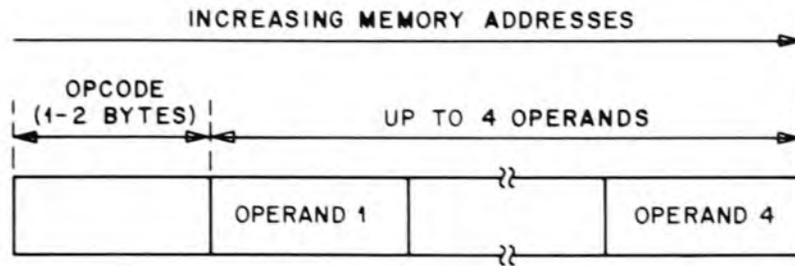
The descriptor byte defines an operand's addressing mode and register fields, which are covered in the next chapter. Immediate data stored within an instruction is stored with lower-order bytes located at lower-order addresses. For example, the hexadecimal value, 0x12345678, would be stored within an instruction as illustrated on Figure 2-16.

Notice that data storage within an instruction, as shown on Figure 2-16, is the reverse of the data storage within a memory location shown on Figure 2-13.

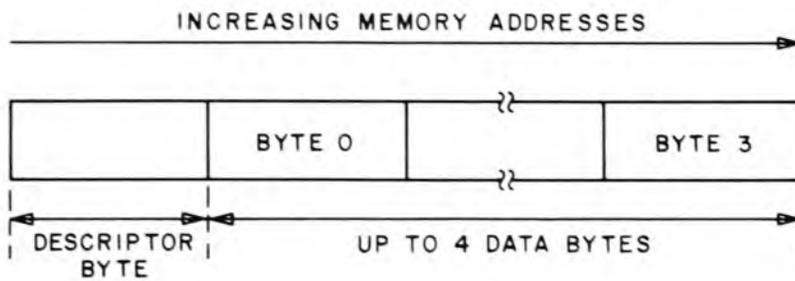
### 2.11 CONDITION FLAGS

Bits 21 to 18 of the processor status word (PSW), r11, contain four condition flags (N, Z, V, and C) that are set by most instructions. The order is shown on Figure 2-17. The conditional program-control instructions check one or more of these flags before executing the branch, jump, or return. In general, these flags reflect the result of the most recently executed instruction. Most instructions set the flags according to standard criteria. Before defining that criteria, the following terms are defined:

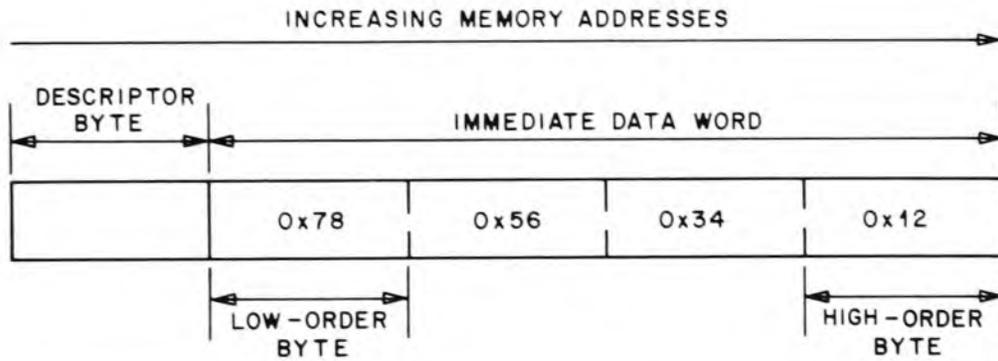
- *Result* refers to the internal result of the operation as if it were performed in an infinite-precision machine. The microprocessor operates on 32-bit data internally and uses a 33-bit space for the internal result. Bytes and halfwords read in are extended to 32 bits before the operation. The destination operand determines the *type* (signed or unsigned) and size (byte, halfword, or word) of this result.



**Figure 2-14. Instruction Storage in Memory**



**Figure 2-15. Operand Format**



**Figure 2-16. Word Storage Within an Instruction**

## REGISTERS, DATA, AND INSTRUCTION FORMATS

### Condition Flags

- *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.

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

- N** *Negative* (PSW bit 21) - Logical instructions change the N flag to the setting of the output value of the MSB: bit 31 for words, bit 15 for halfwords, and bit 7 for bytes. For all other instructions, N is set if the sign of the result is negative. If truncation occurs, the N flag may be set even though the sign bit of the output value is zero. Zero is considered positive.
- Z** *Zero* (PSW bit 20) - Logical instructions set the Z flag if the output value is zero. For all other instructions Z is set if the result is equal to zero. If truncation occurs, the Z flag may not be set even though all bits of the output value are zero.
- V** *Overflow* (PSW bit 19) - For instructions with a signed destination, the V flag is set if the sign bit of the output value is different from any truncated bit of the result. For instructions with an unsigned destination, V is set if any truncated bit is a 1. The arithmetic left shift operation sets the V bit only if a truncation error occurs. Bit, compare, and test instructions always reset V.
- C** *Carry/Borrow* (PSW bit 18) - Logical instructions clear this bit. For all other instructions, the type of the result determines the state of the C bit. C is set if a *carry* occurs into the 33rd bit for word operations, into the 17th bit for halfword operations, or into the 9th bit for byte operations. The C bit is set if a *borrow* occurs from these bits for subtract, negate, and decrement. For example, consider A minus B where A and B are unsigned. If A is greater than or equal to B after both are extended to 32 bits, then C is cleared. Otherwise, the C flag is set.

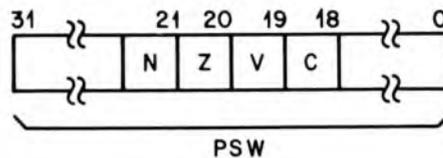


Figure 2-17. Condition Flags

**Note:** If a memory-write fault occurs, the flags are set as if the instruction was completed normally.

The instruction descriptions given in **Chapter 5. Instruction Set and Addressing Modes** include the effect that each instruction has on the condition flags.

## **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 CLOCK SIGNALS .....	3-2
3.3 INTERRUPT SIGNALS .....	3-2
Autovector (AVEC) .....	3-2
Interrupt Option (INTOPT) .....	3-2
Interrupt Priority Level (IPL0—IPL3) .....	3-3
Nonmaskable Interrupt (NMINT) .....	3-3
3.4 INTERFACE AND CONTROL SIGNALS .....	3-3
Address Strobe (AS) .....	3-3
Cycle Initiate (CYCLE) .....	3-3
Coprocessor Done (DONE) .....	3-3
Data Ready (DRDY) .....	3-4
Data Strobe (DS) .....	3-4
Data Transfer Acknowledge (DTACK) .....	3-4
Synchronous Ready (SRDY) .....	3-4
3.5 ACCESS STATUS SIGNALS .....	3-4
Block (Double Word) Fetch (BLKFTCH) .....	3-4
Data Size (DSIZE0—DSIZE1) .....	3-5
Read/Write (R/W) .....	3-5
Access Status Codes (SAS0—SAS3) .....	3-5
Virtual Address (VAD) .....	3-6
Execution Mode (XMD0—XMD1) .....	3-6
3.6 ARBITRATION SIGNALS .....	3-6
Bus Arbiter (BARB) .....	3-6
Bus Request (BUSRQ) .....	3-7
Bus Request Acknowledge (BRACK) .....	3-7
3.7 BUS EXCEPTION SIGNALS .....	3-7
Access Abort (ABORT) .....	3-7
Data Bus Shadow (DSHAD) .....	3-7
Fault (FAULT) .....	3-8
Reset Acknowledge (RESET) .....	3-8
Reset Request (RESETR) .....	3-8
Retry (RETRY) .....	3-8
Relinquish and Retry Request Acknowledge (RRRACK) .....	3-8
Relinquish and Retry Request (RRREQ) .....	3-8
Stop (STOP) .....	3-9
3.8 DEVELOPMENT SYSTEM SUPPORT SIGNALS .....	3-9
High Impedance (HIGHZ) .....	3-9
Instruction Queue Status (IQS0—IQS1) .....	3-9
Start of Instruction (SOI) .....	3-9
3.9 PIN ASSIGNMENTS .....	3-9

**3. SIGNAL DESCRIPTIONS**

The WE 32100 Microprocessor input and output signals are described briefly in this chapter.

The term "asserted" means that a 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 that the signal is active low.

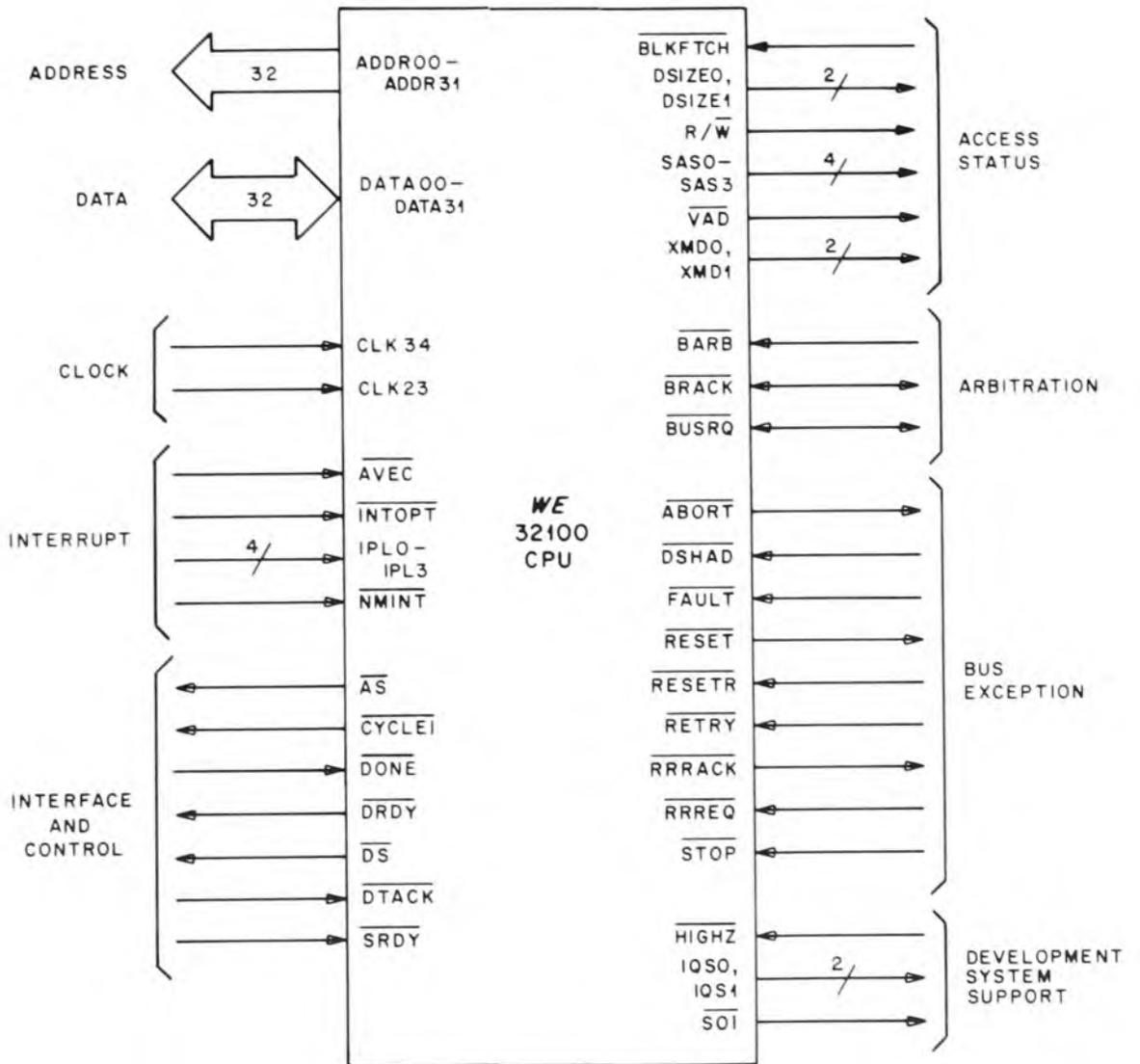


Figure 3-1. Signal Grouping Diagram

## 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 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).

##### 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 CLOCK SIGNALS

The microprocessor requires two clock inputs, both operating at the same frequency. The falling edge of input clock, CLK34 signifies the beginning of a machine cycle. This clock input lags input clock CLK23, by 90°.

#### 3.3 INTERRUPT SIGNALS

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

##### 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 32100 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 no interrupts are pending. This type of interrupt request can be masked by using the interrupt priority level (IPL) field (bits 13–16) of the processor status word (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 ( $\overline{\text{NMINT}}$ )**

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 auto-vector interrupt with vector number 0. All address bus bits are driven low during the acknowledge cycle; this is what distinguishes the nonmaskable from all other interrupts.

## **3.4 INTERFACE AND CONTROL SIGNALS**

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

### **Address Strobe ( $\overline{\text{AS}}$ )**

When low (0), this output signal indicates the presence of a valid physical address on the address pins. If the address is virtual, then the falling edge of  $\overline{\text{AS}}$  indicates a valid address, and the virtual address will be 3-stated subsequent to the falling edge of  $\overline{\text{AS}}$ .

### **Cycle Initiate ( $\overline{\text{CYCLEI}}$ )**

This 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.  $\overline{\text{CYCLEI}}$  is negated at the end of the fourth clock state (second clock cycle).  $\overline{\text{CYCLEI}}$  is asserted in both the read and write halves of an interlocked read transaction.

### **Coprocessor Done ( $\overline{\text{DONE}}$ )**

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

## SIGNAL DESCRIPTIONS

### Data Ready

#### Data Ready ( $\overline{\text{DRDY}}$ )

When active, this output indicates that the microprocessor (acting as bus master) has not detected any bus exceptions ( $\overline{\text{FAULT}}$ ,  $\overline{\text{RETRY}}$ , and  $\overline{\text{RRREQ}}$  signals) during the current bus transaction. This signal may be tied common to other devices that have this signal as an output (i.e., 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's data shadow signal  $\overline{\text{DSHAD}}$  is inactive, the MMU is not the bus master, and its  $\overline{\text{DRDY}}$  is 3-stated.  $\overline{\text{DRDY}}$ 's trailing edge marks the end of a bus cycle which has no bus exceptions.

#### Data Strobe ( $\overline{\text{DS}}$ )

During a read operation this output signal, when low, indicates that a coprocessor or slave device can place data on the data bus. During a write operation this signal, when low, indicates that the microprocessor has placed valid data on the data bus.

#### Data Transfer Acknowledge ( $\overline{\text{DTACK}}$ )

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  $\overline{\text{DTACK}}$  is driven low by the coprocessor or slave device. During a write operation, the transaction is terminated when the coprocessor or slave device drives  $\overline{\text{DTACK}}$  low. If  $\overline{\text{DTACK}}$  is high, wait states are inserted in the current cycle.  $\overline{\text{DTACK}}$  is ignored if the data bus shadow signal  $\overline{\text{DSHAD}}$  is asserted. The  $\overline{\text{DTACK}}$  is an asynchronous input and is latched at two different times (i.e., double latched) to maintain stability.

#### Synchronous Ready ( $\overline{\text{SRDY}}$ )

This signal is a synchronous input that begins the termination of 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  $\overline{\text{SRDY}}$  is not asserted at this time and  $\overline{\text{DTACK}}$  was not asserted during the previous clock state, then wait state cycles are inserted until either signal is asserted.  $\overline{\text{SRDY}}$  is ignored if the  $\overline{\text{DSHAD}}$  signal was previously asserted.

### 3.5 ACCESS STATUS SIGNALS

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

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

This input, 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 ( $\overline{\text{DSIZE0}}$  and  $\overline{\text{DSIZE1}}$ ) signals will show a double word access on all instruction fetches. If the input is negated, the microprocessor fetches a block of instruction by two consecutive reads.

**Data Size (DSIZE0–DSIZE1)**

This 2-bit output is used to indicate the size of the data in the current bus transaction. These pins are used to indicate whether the CPU is transferring byte, halfword, word, or double word pieces of data. On all instruction fetches the DSIZE signals will show a double word access. For prefetches, the DSIZE signals will have the value for either a double word or word access. The DSIZE signals are interpreted as follows:

DSIZE1	DSIZE0	Transaction Size
0	0	word
0	1	double word
1	0	halfword
1	1	byte

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

This 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	Auto-vector 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

## SIGNAL DESCRIPTIONS

### Virtual Address

#### Virtual Address ( $\overline{\text{VAD}}$ )

When asserted, this output indicates that the address is virtual. When negated, the address is physical. When the address is virtual, the ADDR (00–31) will be 3-stated in the middle of the third clock state. The  $\overline{\text{VAD}}$  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.  $\overline{\text{VAD}}$  is a level rather than 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:

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

The XMD0–XMD1 pins 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.

### 3.6 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 input 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 does a bus transaction:

$\overline{\text{ABORT}}$	ADDR00–ADDR31
$\overline{\text{DS}}$	DSIZE0–DSIZE1
$\overline{\text{AS}}$	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 which are 3-stated when the  $\overline{\text{BRACK}}$  is issued are:

$\overline{\text{ABORT}}$	ADDR00—ADDR31
$\overline{\text{DS}}$	DSIZE0—DSIZE1
$\overline{\text{AS}}$	R/ $\overline{\text{W}}$
$\overline{\text{CYCLEI}}$	SAS0—SAS3
$\overline{\text{VAD}}$	DATA00—DATA31
$\overline{\text{DRDY}}$	XMD0—XMD1

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

**3.7 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.

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

This signal is asserted on an access which is ignored by the memory system. This occurs when the microprocessor has a program counter discontinuity with an instruction cache hit or an alignment fault.

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

This signal is used by the MMU to remove the microprocessor from the data bus. The  $\overline{\text{DATA00—DATA31}}$ ,  $\overline{\text{DRDY}}$ ,  $\overline{\text{DSIZE0—DSIZE1}}$ , and  $\overline{\text{R/W}}$  are 3-stated and the  $\overline{\text{DTACK}}$ ,  $\overline{\text{SRDY}}$ , and  $\overline{\text{FAULT}}$  inputs are ignored when this input is asserted.

## SIGNAL DESCRIPTIONS

### Fault

#### 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 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 signal is used to reset the microprocessor.  $\overline{\text{RESETR}}$  must be active for three clock cycles in order to be acknowledged. The microprocessor acknowledges the request by immediately asserting  $\overline{\text{RESETR}}$ .

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

When this 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 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 which 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 3-state the following signals:

$\overline{\text{ABORT}}$	ADDR00—ADDR31
$\overline{\text{DS}}$	DSIZE0—DSIZE1
$\overline{\text{AS}}$	R/ $\overline{\text{W}}$
$\overline{\text{CYCLEI}}$	SAS0—SAS3
$\overline{\text{VAD}}$	DATA00—DATA31
$\overline{\text{DRDY}}$	XMD0—XDM1

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.

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

When asserted, this asynchronous 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.

**3.8 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 ( $\overline{\text{SOI}}$ )**

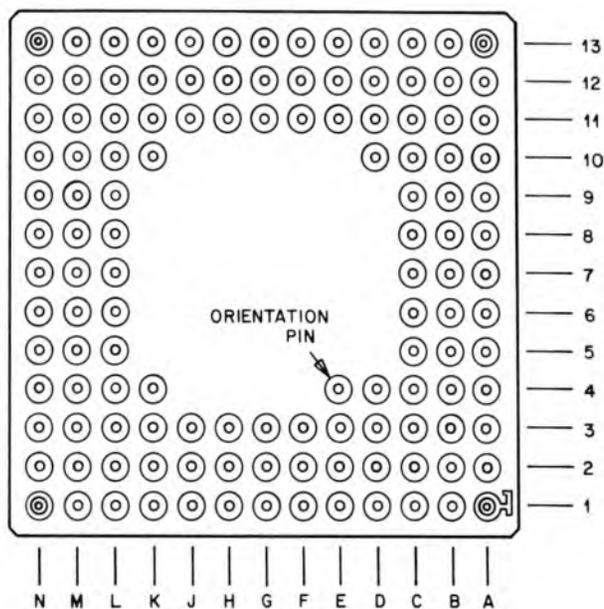
When asserted, this output signal indicates that the microprocessor's internal control has fetched the opcode for the next instruction from the internal instruction queue. Since the instructions are pipelined, it does not mean the end of the previous instruction execution.

**3.9 PIN ASSIGNMENTS**

The *WE 32100* Microprocessor is available in a 125-pin, square, ceramic pin grid array. Figure 3-2 with Table 3-1 describes the pin assignments.

## SIGNAL DESCRIPTIONS

### Pin Assignments



**Figure 3-2. 125-Pin, Square Ceramic Pin Grid Array Bottom View**

**Table 3-1. WE 32100 Microprocessor 125-Pin PGA Package 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	IQSO	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 0
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

**SIGNAL DESCRIPTIONS**  
**Pin Assignments**

<b>Table 3-1. WE 32100 Microprocessor 125-Pin PGA Package Pin Descriptions (Continued)</b>			
<b>Pin</b>	<b>Name</b>	<b>Type</b>	<b>Description</b>
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	—	Power
C7	GRD	—	Ground
C8	+5V	—	Power
C9	GRD	—	Ground
C10	DONE	I	Coprocessor Done
C11	HIGHZ	I	High Impedance
C12	RRRACK	O	<b>WARNING:</b> This pin must be tied high (5Vdc). Relinquish and Retry Request Acknowledge
C13	FAULT	I	
D1	ADDR24	O	Microprocessor Address 24
D2	ADDR20	O	Microprocessor Address 20
D3	ADDR25	O	Microprocessor Address 25
D4	DATA27	I/O	Microprocessor Data 27
D10	SRDY	I	Synchronous Ready
D11	RETRY	I	Retry
D12	BARB	I	Bus Arbiter
D13	RESET	O	Reset Acknowledge
E1	ADDR18	O	Microprocessor Address 18
E2	ADDR23	O	Microprocessor Address 23
E3	+5V	—	Power
E4	—	—	Device Socket Orientation Pin
E11	+5V	—	Power
E12	DTACK	I	Data Transfer Acknowledge
E13	BLKFTCH	I	Block (Double Word) Fetch
F1	ADDR21	O	Microprocessor Address 21
F2	ADDR22	O	Microprocessor Address 22
F3	GRD	—	Ground

## SIGNAL DESCRIPTIONS

### Pin Assignments

Table 3-1. WE 32100 Microprocessor 125-Pin PGA Package Pin Descriptions (Continued)			
Pin	Name	Type	Description
F11	<u>CLK23</u>	I	Input Clock 23
F12	<u>DSHAD</u>	I	Data Bus Shadow
F13	<u>RESETR</u>	I	Reset Request
G1	DATA21	I/O	Microprocessor Data 21
G2	ADDR17	O	Microprocessor Address 17
G3	DATA20	I/O	Microprocessor Data 20
G11	<u>CLK34</u>	I	Input Clock 34
G12	<u>CYCLEI</u>	O	Cycle Initiate
G13	<u>STOP</u>	I	Microprocessor Stop
H1	ADDR15	O	Microprocessor Address 15
H2	DATA18	I/O	Microprocessor Data 18
H3	GRD	—	Ground
H11	<u>GRD</u>	—	Ground
H12	<u>AS</u>	O	Address Strobe
H13	<u>DS</u>	O	Data Strobe
J1	DATA19	I/O	Microprocessor Data 19
J2	DATA17	I/O	Microprocessor Data 17
J3	+5V	—	Power
J11	<u>+5V</u>	—	Power
J12	<u>ABORT</u>	O	Access Abort
J13	<u>DRDY</u>	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
K10	DATA00	I/O	Microprocessor Data 00
K11	GRD	—	Ground
K12	—	—	<b>WARNING:</b> This pin is for manufacture use only and must be tied high (5 Vdc).
K13	<u>NMINT</u>	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	GRD	—	Ground
L6	ADDR08	O	Microprocessor Address 08
L7	+5V	—	Power
L8	GRD	—	Ground

**SIGNAL DESCRIPTIONS**  
**Pin Assignments**

<b>Table 3-1. WE 32100 Microprocessor 125-Pin PGA Package Pin Descriptions (Continued)</b>			
<b>Pin</b>	<b>Name</b>	<b>Type</b>	<b>Description</b>
L9	+5V	—	Ground
L10	ADDR00	O	Microprocessor Address 00
L11	INTOPT	I	Interrupt Option
L12	IPL3	I	Interrupt Priority Level 3
L13	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	VAD	O	Virtual Address
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	ADDR07	O	Microprocessor Address 07
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

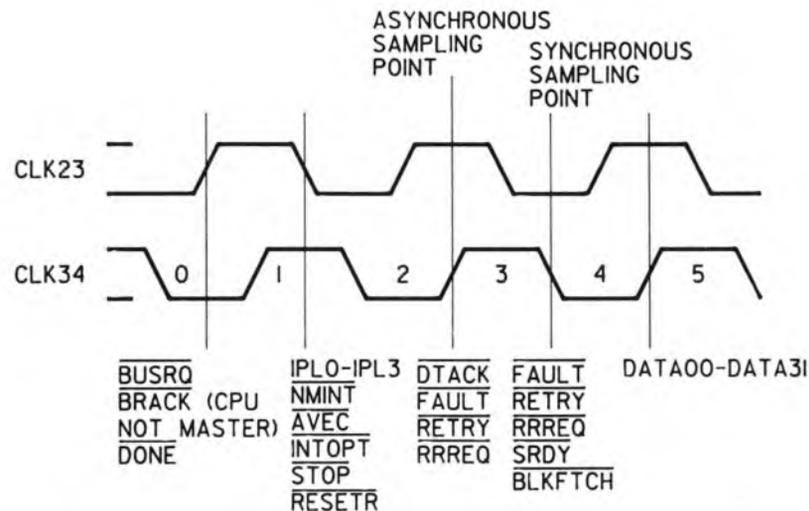
4. BUS OPERATION .....	4-1	4.6.1 Fault on First Word of Blockfetch With Status Code Other Than Prefetch .....	4-27
4.1 SIGNAL SAMPLING POINTS .....	4-1	4.6.2 Fault on First Word of Blockfetch With Status Code of Prefetch .....	4-27
4.2 READ AND WRITE OPERATIONS .....	4-2	4.6.3 Retry on First Word of Blockfetch .....	4-27
4.2.1 Read Transaction Using SRDY .....	4-3	4.6.4 Retry on Second Word of Blockfetch .....	4-27
4.2.2 Read Transaction Using DTACK .....	4-5	4.6.5 Relinquish and Retry on Blockfetch .....	4-32
4.2.3 Read Transaction With Wait Cycle Using SRDY .....	4-6	4.7 INTERRUPTS .....	4-32
4.2.4 Read Transaction With Two Wait Cycles Using DTACK .....	4-7	4.7.1 Interrupt Acknowledge .....	4-32
4.2.5 Write Transaction Using SRDY .....	4-8	4.7.2 Autovector Interrupt .....	4-35
4.2.6 Write Transaction Using DTACK .....	4-8	4.7.3 Nonmaskable Interrupt .....	4-35
4.2.7 Write Transaction With Wait Cycle Using SRDY .....	4-8	4.8 BUS ARBITRATION .....	4-38
4.2.8 Write Transaction With Wait Cycle Using DTACK .....	4-12	4.8.1 Bus Request During a Bus Transaction .....	4-38
4.3 READ INTERLOCKED OPERATION .....	4-12	4.8.2 DMA Operation .....	4-41
4.4 BLOCKFETCH OPERATION .....	4-15	4.9 RESET .....	4-42
4.4.1 Blockfetch Transaction Using SRDY .....	4-15	4.9.1 System Reset .....	4-42
4.4.2 Blockfetch Transaction Using DTACK .....	4-17	4.9.2 Internal Reset .....	4-42
4.4.3 Blockfetch Transaction Using DTACK With Wait Cycle on Second Word .....	4-18	4.9.3 Reset Sequence .....	4-44
4.4.4 Blockfetch Transaction Using SRDY With Wait Cycles on Both Words .....	4-19	4.10 ABORTED MEMORY ACCESSES .....	4-44
4.5 BUS EXCEPTIONS .....	4-20	4.10.1 Aborted Access on PC Discontinuity With Instruction Cache Hit .....	4-45
4.5.1 Faults .....	4-21	4.10.2 Alignment Fault Bus Activity .....	4-46
FAULT With SRDY .....	4-22	4.11 SINGLE-STEP OPERATION .....	4-47
FAULT After DTACK .....	4-23	4.12 COPROCESSOR OPERATIONS .....	4-48
4.5.2 Retry .....	4-24	4.12.1 Coprocessor Broadcast .....	4-48
4.5.3 Relinquish and Retry .....	4-24	4.12.2 Coprocessor Operand Fetch .....	4-53
4.6 BLOCKFETCH SPECIAL CASES .....	4-27	4.12.3 Coprocessor Status Fetch .....	4-54
		4.12.4 Coprocessor Data Write .....	4-55
		4.13 SUPPLEMENTARY PROTOCOL DIAGRAMS .....	4-56

## 4. BUS OPERATION

This chapter covers bus protocol for the *WE 32100* Microprocessor.

### 4.1 SIGNAL SAMPLING POINTS

The *WE 32100* Microprocessor utilizes two phase-shifted input clocks (CLK23 and CLK34) as depicted 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.



#### Notes:

1.  $\overline{\text{BUSRQ}}$ ,  $\overline{\text{BRACK}}$ ,  $\overline{\text{IPL0-IPL3}}$ ,  $\overline{\text{NMINT}}$ ,  $\overline{\text{AVEC}}$ ,  $\overline{\text{INTOPT}}$ ,  $\overline{\text{STOP}}$ ,  $\overline{\text{RESETR}}$  are sampled repetitively one CLK34 cycle apart (i.e., on every clock cycle).
2. After  $\overline{\text{DTACK}}$  is asserted,  $\overline{\text{FAULT}}$ ,  $\overline{\text{RETRY}}$ ,  $\overline{\text{RRREQ}}$  and  $\overline{\text{BLKFTCH}}$  are sampled once at the synchronous sampling point. If  $\overline{\text{FAULT}}$ ,  $\overline{\text{RETRY}}$ , 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  $\overline{\text{FAULT}}$ ,  $\overline{\text{RETRY}}$ ,  $\overline{\text{RRREQ}}$  and  $\overline{\text{BLKFTCH}}$  are sampled once at the synchronous sampling point.
3.  $\overline{\text{BLKFTCH}}$  must remain asserted until negation of data strobe ( $\overline{\text{DS}}$ ).
4.  $\overline{\text{DSHAD}}$  is not latched and can be asserted at any time subject to the following conditions:  $\overline{\text{DSHAD}}$  should only be asserted during a CPU-initiated transaction while  $\overline{\text{AS}}$  is active and  $\overline{\text{DTACK}}$ ,  $\overline{\text{SRDY}}$ , and  $\overline{\text{FAULT}}$  are inactive. Unless  $\overline{\text{RETRY}}$  or  $\overline{\text{RRREQ}}$  is active,  $\overline{\text{DSHAD}}$  should only be negated while  $\overline{\text{AS}}$  is still active and  $\overline{\text{DTACK}}$ ,  $\overline{\text{SRDY}}$  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.

**Figure 4-1. Signal Sampling Points**

## BUS OPERATION

### Read & Write Operations

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 ( $CYCLEI$ ) 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 the virtual mode, the CPU only drives the address bus 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 ( $DSIZE0-DSIZE1$ ) indicate the size of the transaction (byte, halfword, word, or 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. At clock state four, the access status code is changed to reflect the next operation if it is a bus transaction, or to "no operation" if it is not. The leading edge of  $CYCLEI$  can be used to latch the access status code.

#### 4.2 READ AND WRITE OPERATIONS

The *WE* 32100 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 (MMU) to release the shared address bus.

Two 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 ( $\overline{SRDY}$ ) input and the data transfer acknowledge ( $\overline{DTACK}$ ) input. Wait cycles prolong a bus transaction, which allows slave devices more time to place data on the bus during a read transaction and more time to pick up data from the bus during a write transaction.

During bus transactions the CPU samples the  $\overline{DTACK}$  and  $\overline{SRDY}$  inputs at their respective sampling points as shown on Figure 4-1. If either input is active (low) at its sampling point, no wait cycles are inserted and the transaction completes in three clock cycles. However, if neither  $\overline{DTACK}$  nor  $\overline{SRDY}$  is sampled active, wait cycles are inserted, and the CPU continues sampling until either 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.

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

#### **4.2.1 Read Transaction Using $\overline{\text{SRDY}}$**

Figure 4-2 illustrates a read transaction with zero wait cycles (3 cycle access) using  $\overline{\text{SRDY}}$  to terminate the access. The read transaction starts with the CPU driving the address bus ( $\text{ADDR00}–\text{ADDR31}$ ) and the data size outputs ( $\text{DSIZE0}–\text{DSIZE1}$ ), driving the  $\overline{\text{R/W}}$  output high to indicate that a read operation is being performed, and asserting the  $\overline{\text{CYCLE1}}$  output at the beginning of clock state zero.

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

The transaction illustrated on Figure 4-2 is terminated by the assertion of  $\overline{\text{SRDY}}$  by the addressed device.  $\overline{\text{SRDY}}$  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.

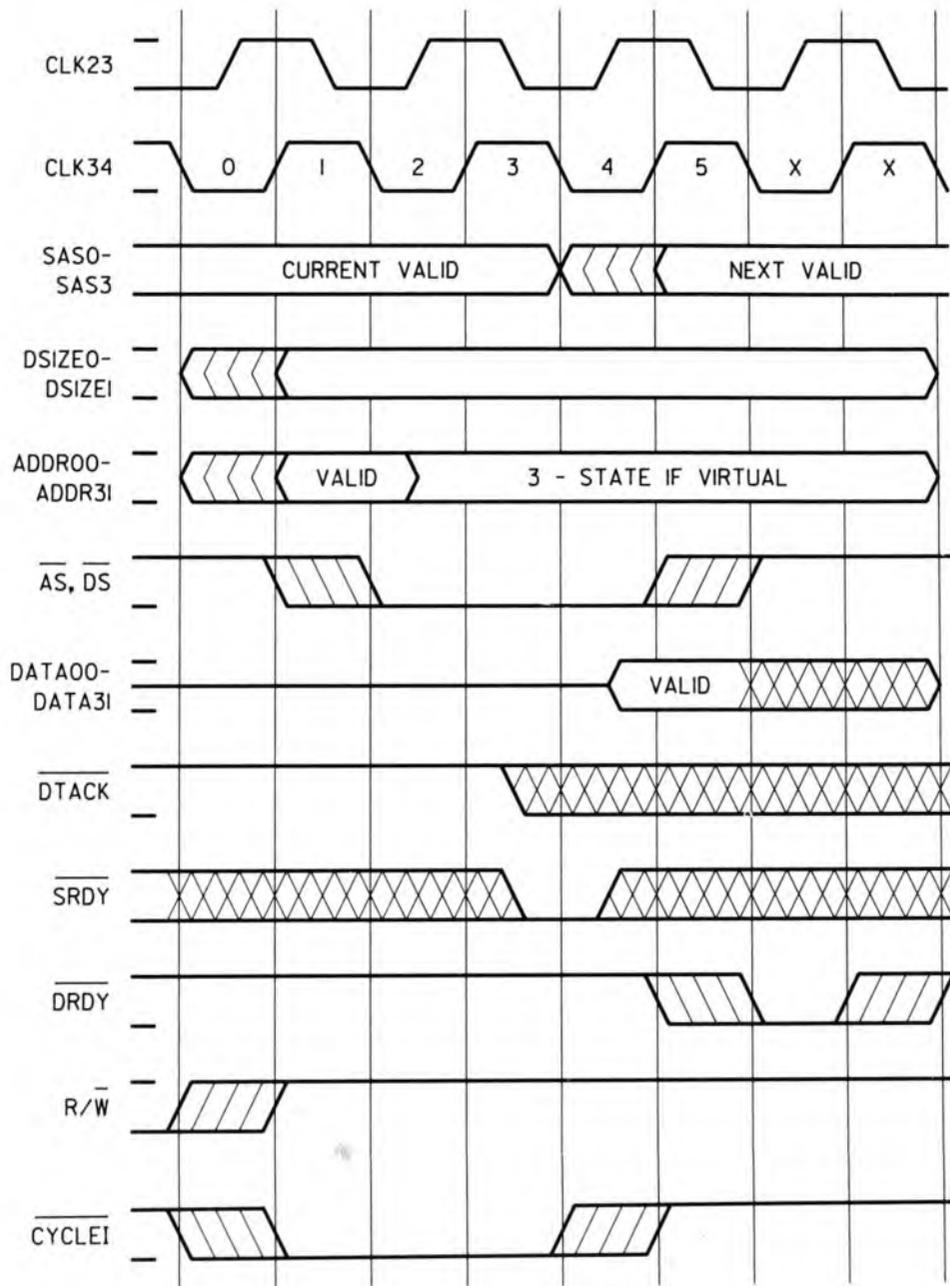
$\overline{\text{SRDY}}$  is synchronously sampled at the end of clock state three. The read transaction depicted on Figure 4-2 completes in three clock cycles (zero wait cycles) because  $\overline{\text{SRDY}}$  is active when sampled at the end of the clock state three.

#### **4.2.2 Read Transaction Using $\overline{\text{DTACK}}$**

The read transaction using  $\overline{\text{DTACK}}$  is identical to the that using  $\overline{\text{SRDY}}$ , except that the addressed device asserts  $\overline{\text{DTACK}}$  instead of  $\overline{\text{SRDY}}$  to acknowledge that it is putting data on the data bus (see Figure 4-3).  $\overline{\text{DTACK}}$  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  $\overline{\text{DTACK}}$  active at the end of clock state two. Upon sampling  $\overline{\text{DTACK}}$  active, the CPU latches the data and terminates the transaction.

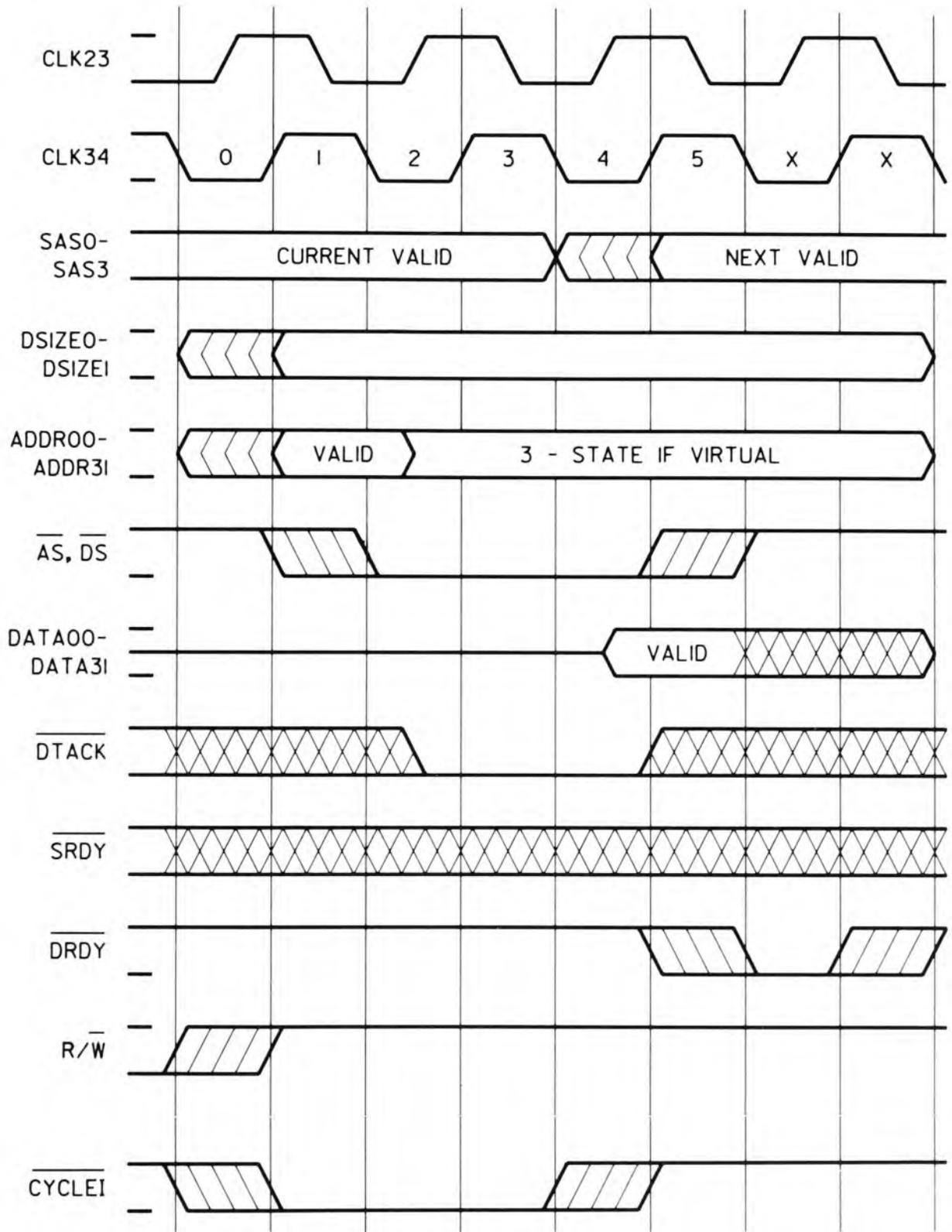
**BUS OPERATION**  
**Read Transaction Using  $\overline{\text{SRDY}}$**



Note: Zero wait cycles.

**Figure 4-2. Read Transaction Using  $\overline{\text{SRDY}}$**

**BUS OPERATION**  
**Read Transaction Using DTACK**



Note: Zero wait cycles.

**Figure 4-3. Read Transaction Using DTACK**

## BUS OPERATION

### Read Transaction With Wait Cycle Using $\overline{\text{SRDY}}$

#### 4.2.3 Read Transaction With Wait Cycle Using $\overline{\text{SRDY}}$

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, and no bus exceptions occur. As illustrated on Figure 4-4, 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.

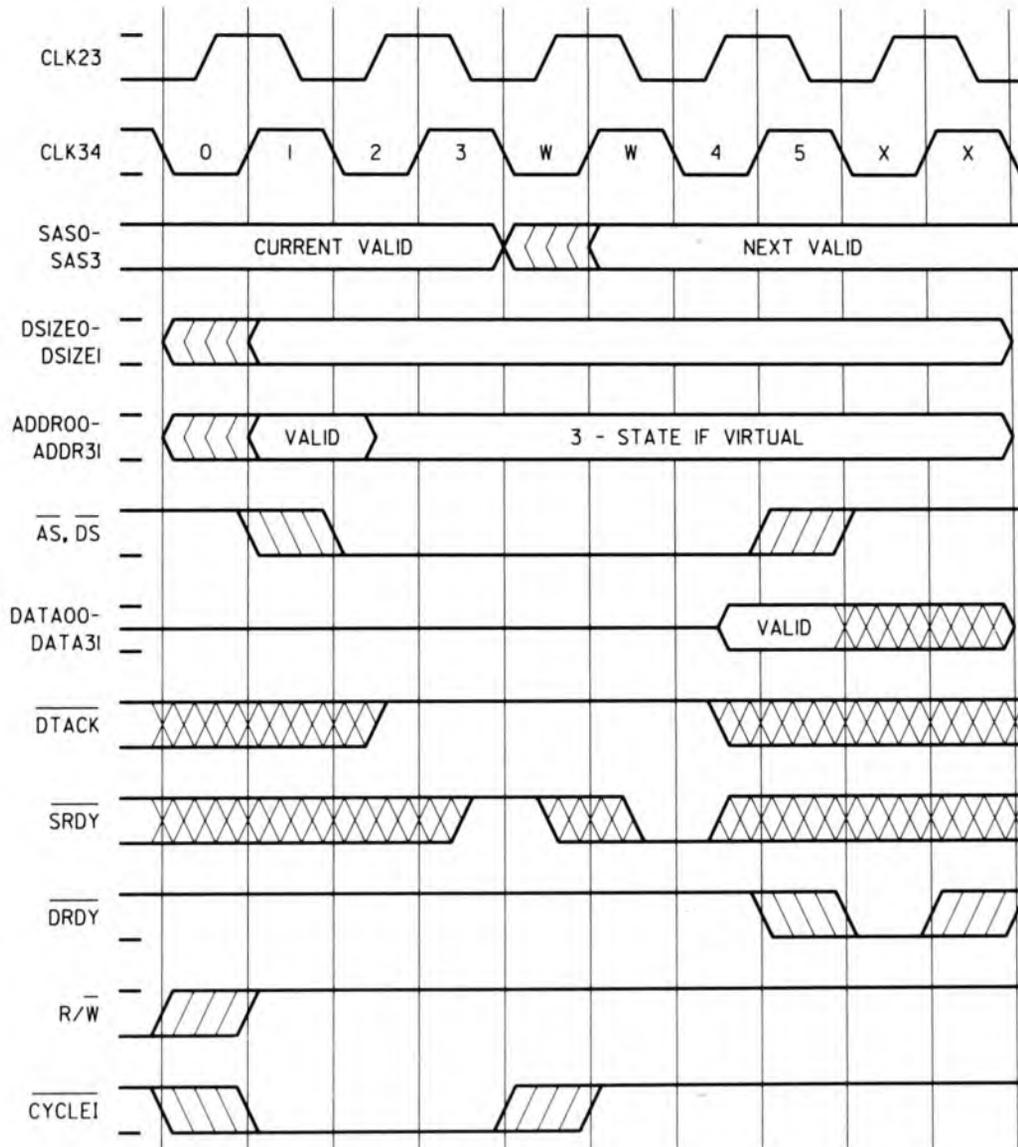
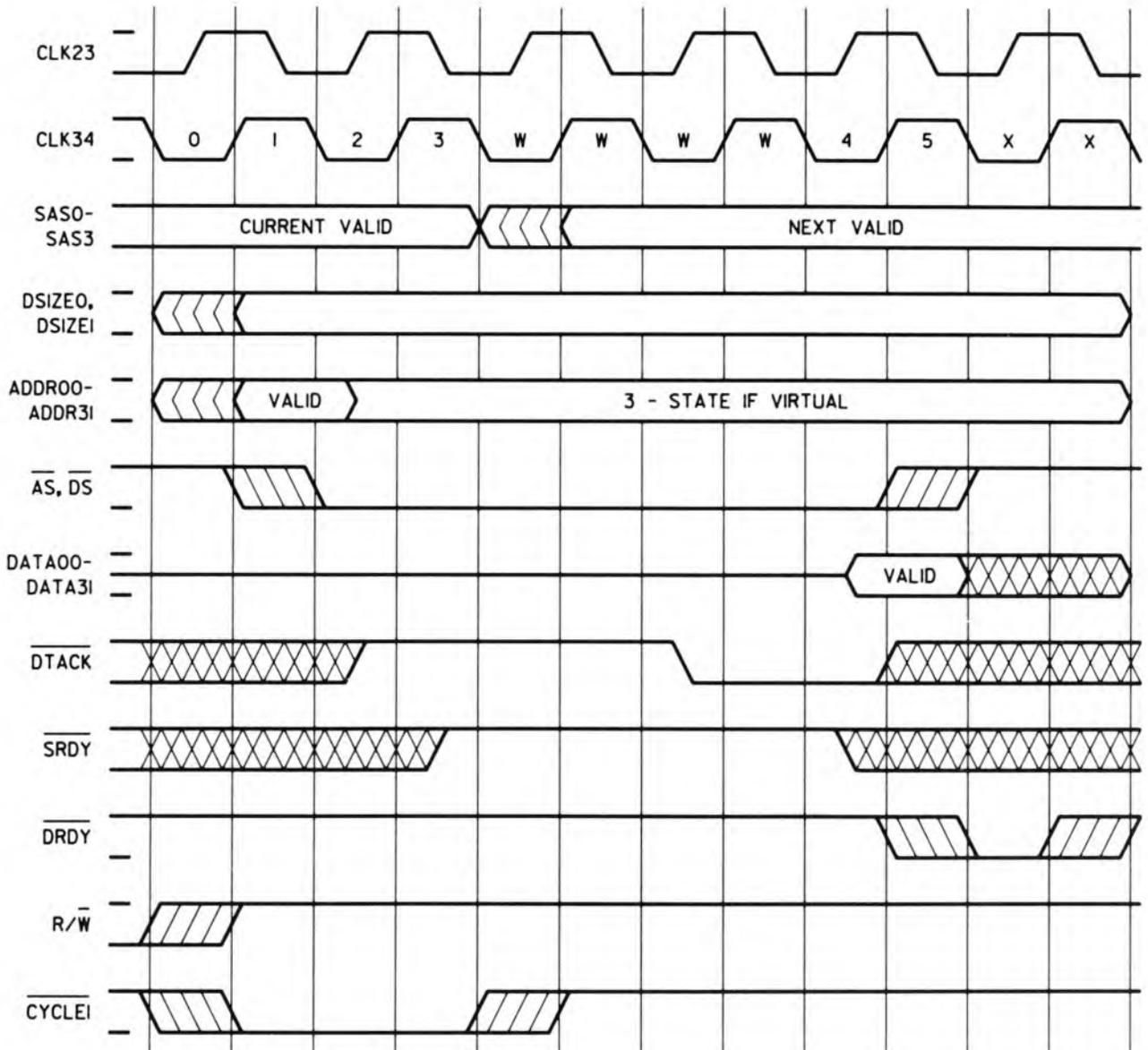


Figure 4-4. Read Transaction With One Wait Cycle Using  $\overline{\text{SRDY}}$

**4.2.4 Read Transaction With Two Wait Cycles Using  $\overline{DTACK}$**

The CPU can insert multiple wait cycles during bus transactions as illustrated on Figure 4-5. In this figure the CPU does not receive an acknowledge ( $\overline{DTACK}$  or  $\overline{SRDY}$ ) for two clock cycles. Neither  $\overline{DTACK}$  nor  $\overline{SRDY}$  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 4-5. Read Transaction With Two Wait Cycles Using  $\overline{DTACK}$**

## BUS OPERATION

### Write Transaction Using $\overline{\text{SRDY}}$

#### 4.2.5 Write Transaction Using $\overline{\text{SRDY}}$

During write transactions the  $\overline{\text{R/W}}$  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" ( $\text{SAS3-SAS0} = 1010$ ).

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

Figure 4-6 illustrates a write transaction with the addressed device using  $\overline{\text{SRDY}}$  as the acknowledgement. By asserting  $\overline{\text{SRDY}}$  the addressed device indicates to the CPU that it is ready to latch the data on the data bus.  $\overline{\text{SRDY}}$  is synchronously sampled at the end of clock state three. On Figure 4-6 the CPU sampled  $\overline{\text{DTACK}}$  inactive at the end of clock state two; however, it sampled  $\overline{\text{SRDY}}$  active at the end of clock state three. As a result, the CPU terminates the transaction.

#### 4.2.6 Write Transaction Using $\overline{\text{DTACK}}$

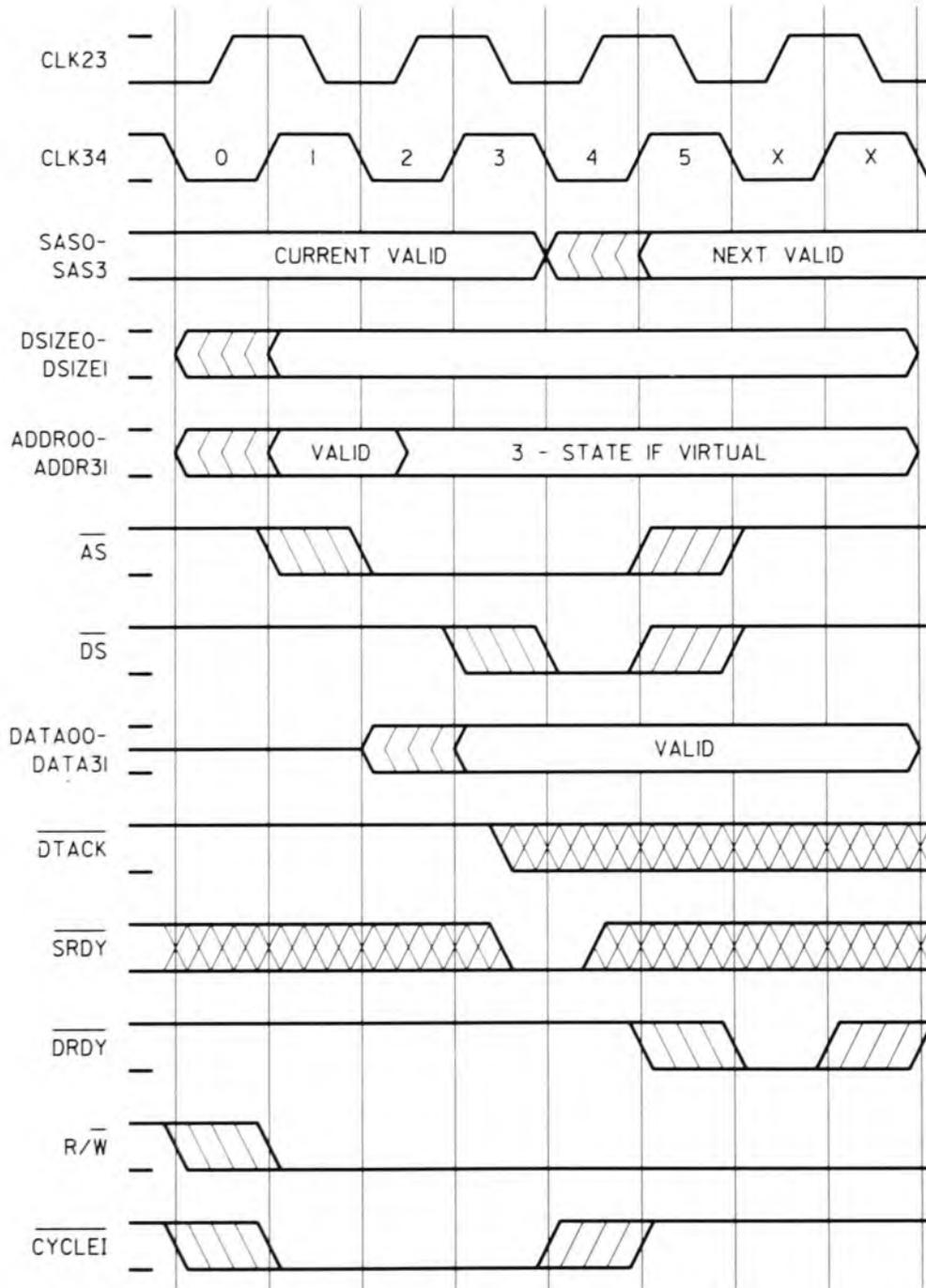
The write transaction using  $\overline{\text{DTACK}}$  is identical to that using  $\overline{\text{SRDY}}$ , except that the addressed device asserts  $\overline{\text{DTACK}}$  to indicate that it is ready to latch the data on the data bus.  $\overline{\text{DTACK}}$  is sampled asynchronously at the end of clock state two. On Figure 4-7 the CPU samples  $\overline{\text{DTACK}}$  active at this time and proceeds to terminate the transaction.

#### 4.2.7 Write Transaction With Wait Cycle Using $\overline{\text{SRDY}}$

Wait cycle insertion for write transactions is similar to wait cycle insertion for read transactions. Just as in read transactions, the CPU inserts wait cycles if  $\overline{\text{DTACK}}$  is not active when sampled at the end of clock state two,  $\overline{\text{SRDY}}$  is not active when sampled at the end of clock state three, and no bus exceptions occur.

Figure 4-8 illustrates a write transaction with two wait cycles. The CPU begins wait cycle insertion because  $\overline{\text{DTACK}}$  is not active at the end of state two and  $\overline{\text{SRDY}}$  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  $\overline{\text{SRDY}}$  at the end of the second wait cycle, and the CPU terminates the transaction.

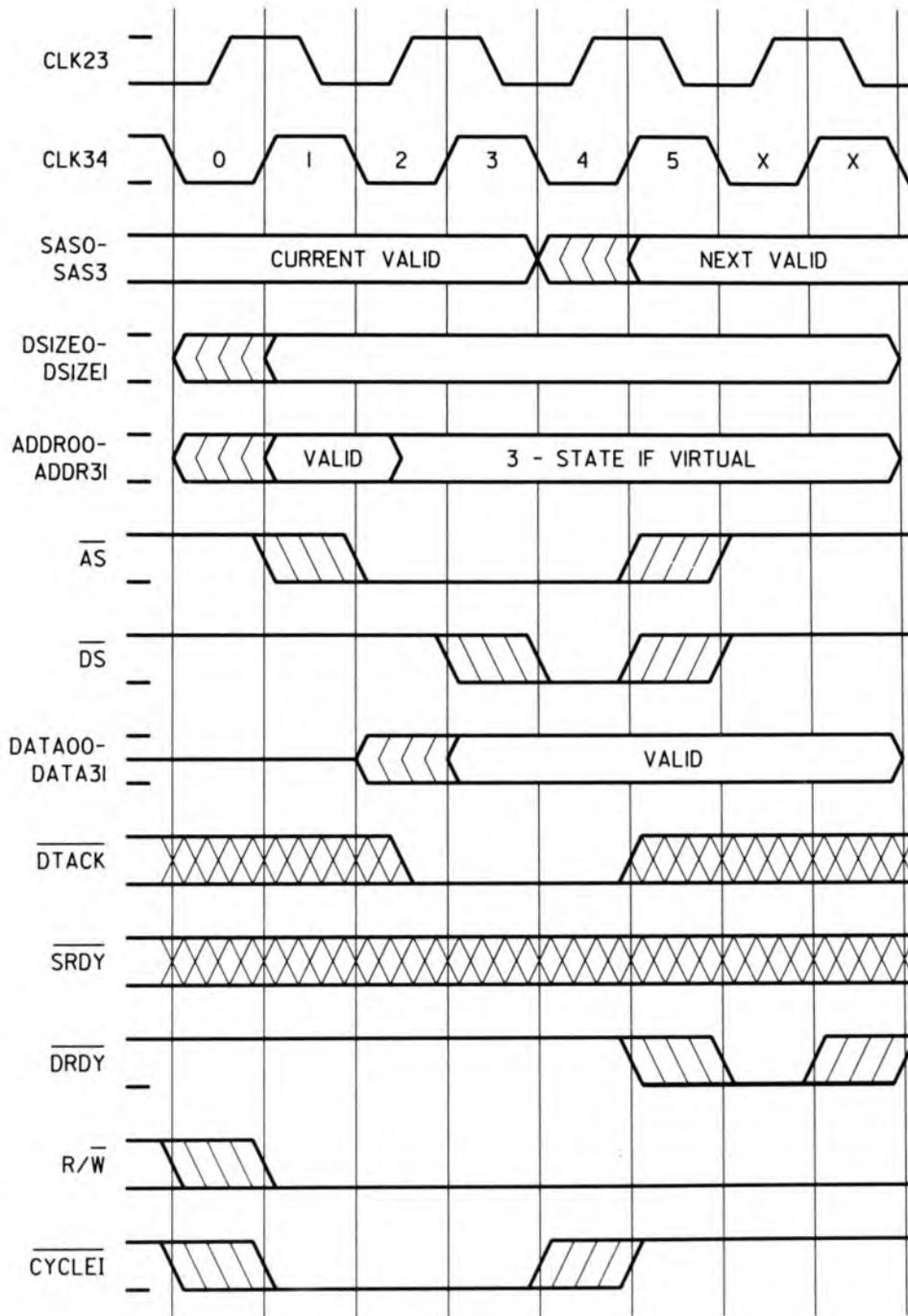
**BUS OPERATION**  
**Write Transaction Using  $\overline{\text{SRDY}}$**



Note: Zero wait cycles.

**Figure 4-6. Write Transaction Using  $\overline{\text{SRDY}}$**

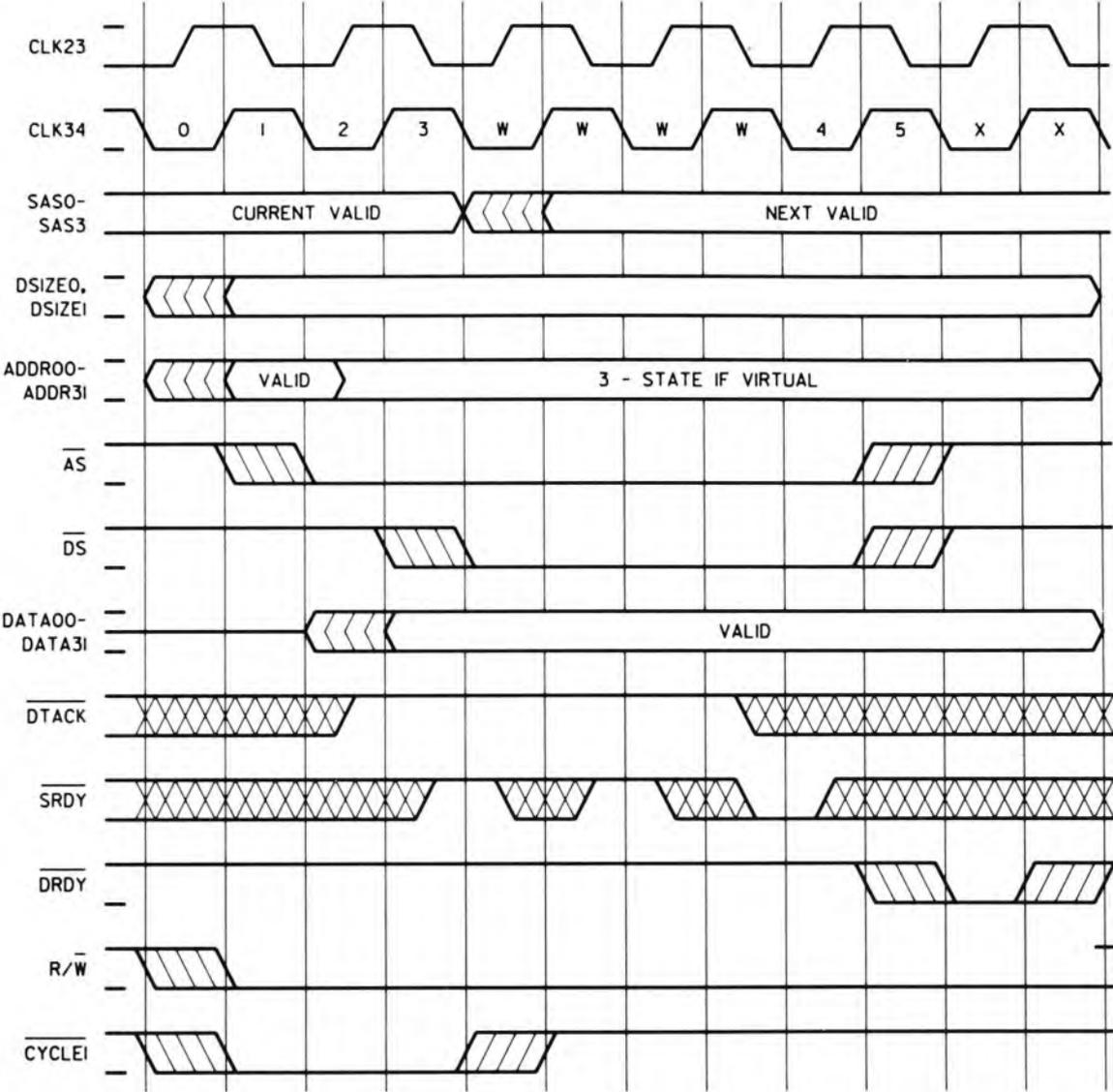
**BUS OPERATION**  
**Write Transaction Using  $\overline{DTACK}$**



Note: Zero wait cycles.

**Figure 4-7. Write Transaction Using  $\overline{DTACK}$**

**BUS OPERATION**  
**Write Transaction With Two Wait Cycles Using  $\overline{\text{SRDY}}$**



**Figure 4-8. Write Transaction With Two Wait Cycles Using  $\overline{\text{SRDY}}$**

## BUS OPERATION

### Write Transaction With Wait Cycle Using $\overline{DTACK}$

#### 4.2.8 Write Transaction With Wait Cycle Using $\overline{DTACK}$

The write transaction shown on Figure 4-9 is another example of wait cycle insertion. In this transaction the addressed device asserts  $\overline{DTACK}$  to indicate that it is ready to latch the data, and that therefore, no more wait cycles are to be inserted.

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

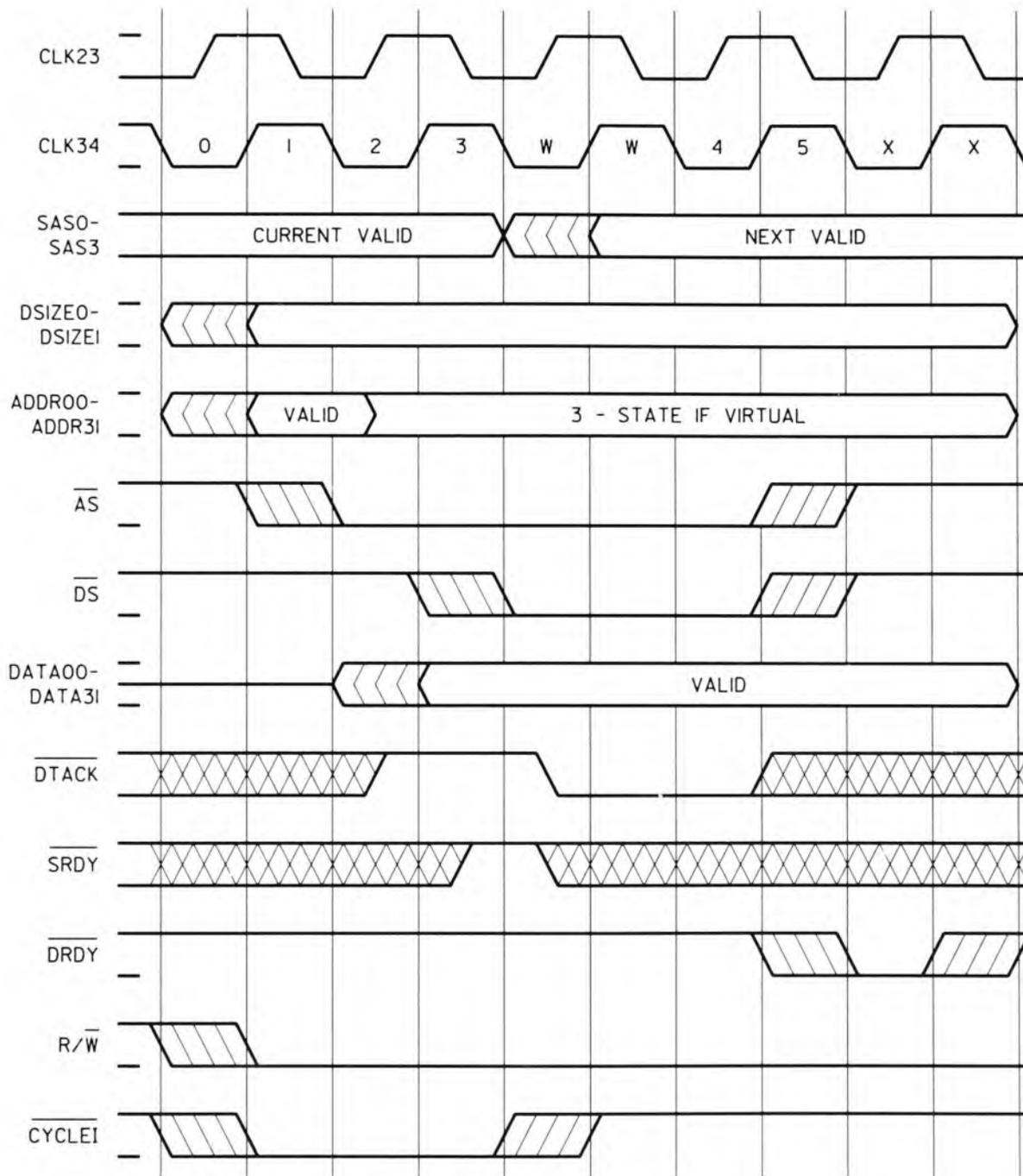
Additional protocol diagrams for read and write operations are in **4.13 Supplementary Protocol Diagrams**.

### 4.3 READ INTERLOCKED OPERATION

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 other than by a reset. This prevents another process from altering data in memory which 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 4-10 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.

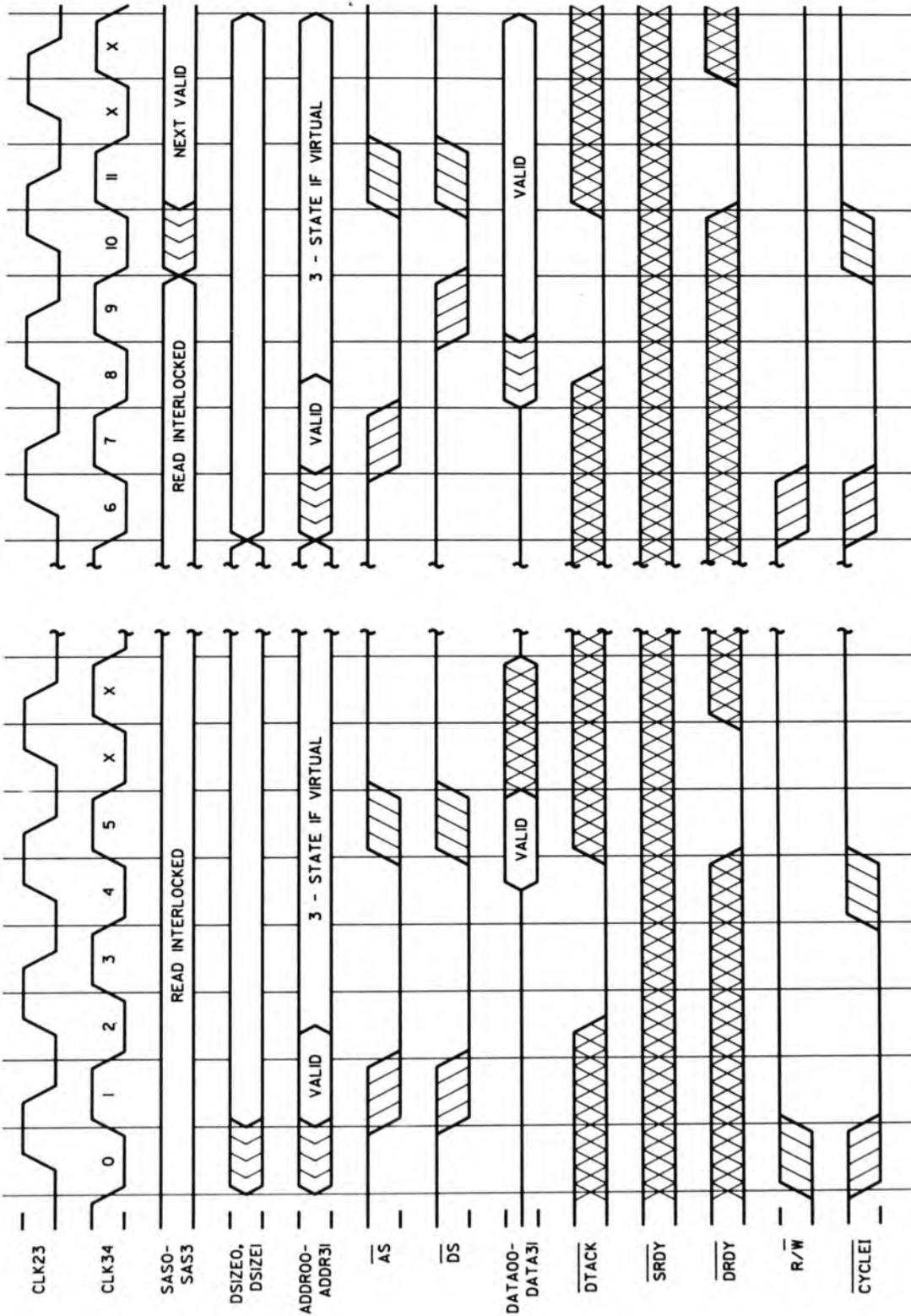
**BUS OPERATION**  
**Write Transaction With Wait Cycle Using  $\overline{DTACK}$**



**Figure 4-9. Write Transaction With Wait Cycle Using  $\overline{DTACK}$**

# BUS OPERATION

## Read Interlocked Transaction



- Notes:
1. 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.
  2. Zero wait cycles.

**Figure 4-10. Read Interlocked Transaction Using DTACK**

#### **4.4 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 the  $\overline{\text{DSIZE}}$  of double word which indicates that it is ready to perform a blockfetch.

If the memory is designed to handle blockfetch, it will respond with the blockfetch ( $\overline{\text{BLKFTCH}}$ ) signal and an acknowledge signal, either  $\overline{\text{SRDY}}$  or  $\overline{\text{DTACK}}$ .

##### **4.4.1 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, removes  $\overline{\text{DS}}$  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 a  $\overline{\text{SRDY}}$ . The CPU samples the  $\overline{\text{SRDY}}$  at the end of clock state seven, then latches the data during clock state eight and terminates the transaction. This operation is shown on Figure 4-11.

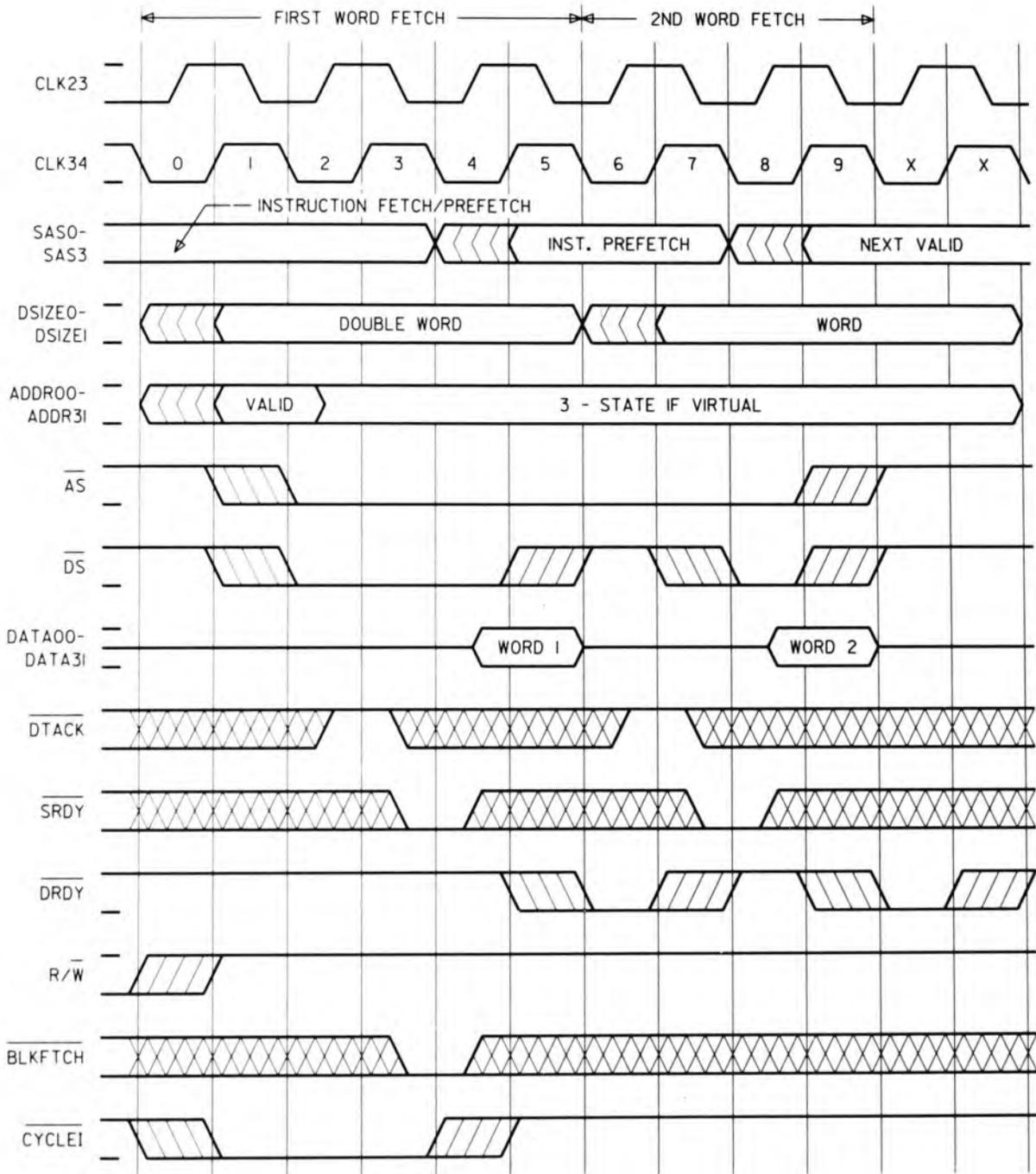
$\overline{\text{AS}}$  stays low for both words fetched.  $\overline{\text{DS}}$  goes inactive for one cycle in between the first and second words.  $\overline{\text{DSIZE}}$  changes from double word to word at clock state six.  $\overline{\text{R/W}}$  is held in the read mode for the entire transaction. Only one  $\overline{\text{CYCLEI}}$  is issued for this transaction. Two  $\overline{\text{DRDY}}$ s are issued, one for each word. The  $\overline{\text{BLKFTCH}}$  pin is sampled only with the first  $\overline{\text{SRDY}}$ . It 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 then the CPU will latch the data and terminate the transaction by removing both  $\overline{\text{AS}}$  and  $\overline{\text{DS}}$ . It will then proceed to start up a second read with SAS of "prefetch" and issue 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 is fetching the two words from a double word address block. It will ask 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 four above that of the first word. This is the other corresponding word from the double word block.

For example, assuming physical addressing, the CPU drives  $\overline{\text{ADDR}}$  with 0003C000. Memory provides data for the first word corresponding to location 0003C000. Memory provides data for the second word corresponding to location 0003C004.

Another example of physical mode, consider the CPU driving  $\overline{\text{ADDR}}$  with 00078004. Memory provides data for the first word corresponding to location 00078004. Memory provides data for the second word corresponding to location 00078008.

**BUS OPERATION**  
**Blockfetch Transaction Using  $\overline{\text{SRDY}}$**

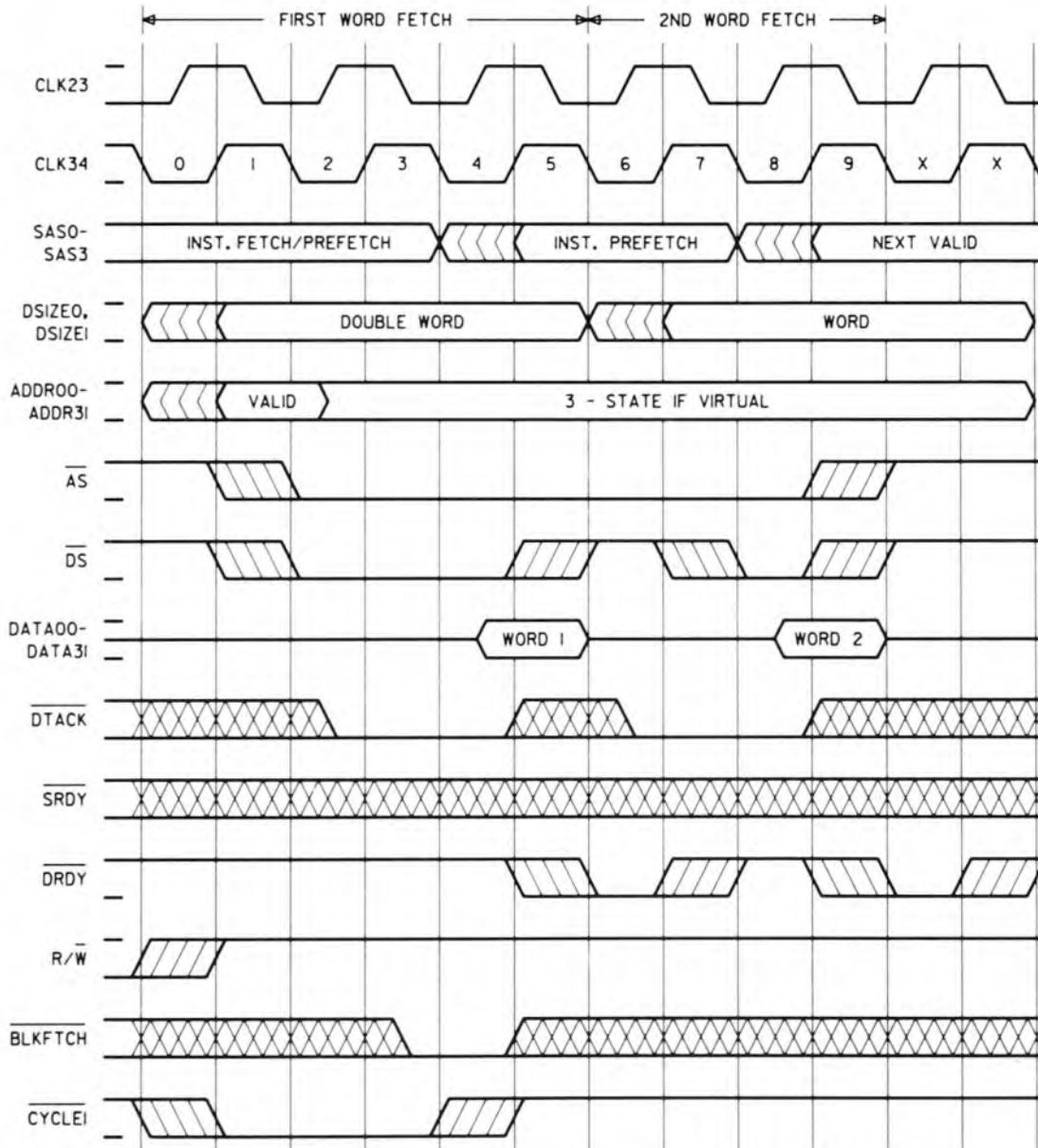


Note: Zero wait cycles.

**Figure 4-11. Blockfetch Transaction Using  $\overline{\text{SRDY}}$**

**4.4.2 Blockfetch Transaction Using  $\overline{DTACK}$**

This transaction (see Figure 4-12) is the same as Figure 4-11 except 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.  $\overline{BLKFETCH}$  is sampled at the end of clock state three, the same as on Figure 4-11. For the second word,  $\overline{DTACK}$  is sampled at the end of clock state six.



Note: Zero wait cycles.

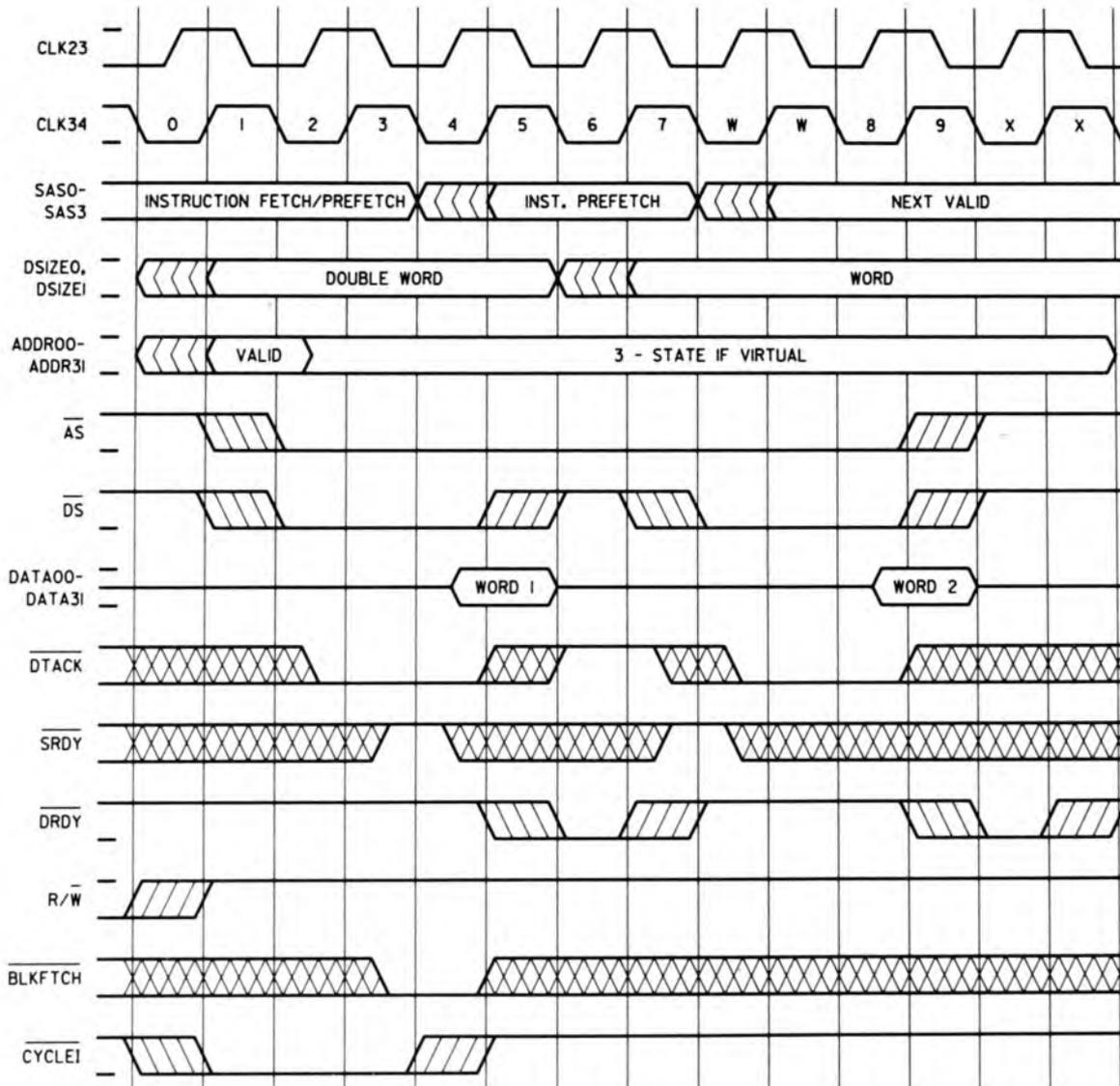
**Figure 4-12. Blockfetch Transaction Using  $\overline{DTACK}$**

## BUS OPERATION

### Blockfetch Transaction Using $\overline{\text{DTACK}}$ With Wait Cycle On Second Word

#### 4.4.3 Blockfetch Transaction Using $\overline{\text{DTACK}}$ With Wait Cycle On Second Word

In this case (see Figure 4-13), during the fetch of the second word there was no  $\overline{\text{DTACK}}$  during clock state six nor  $\overline{\text{SRDY}}$  during clock state seven. Therefore, the CPU inserted a wait cycle. It sampled  $\overline{\text{DTACK}}$  during the first clock state "W", then latched the data and terminated the transaction.

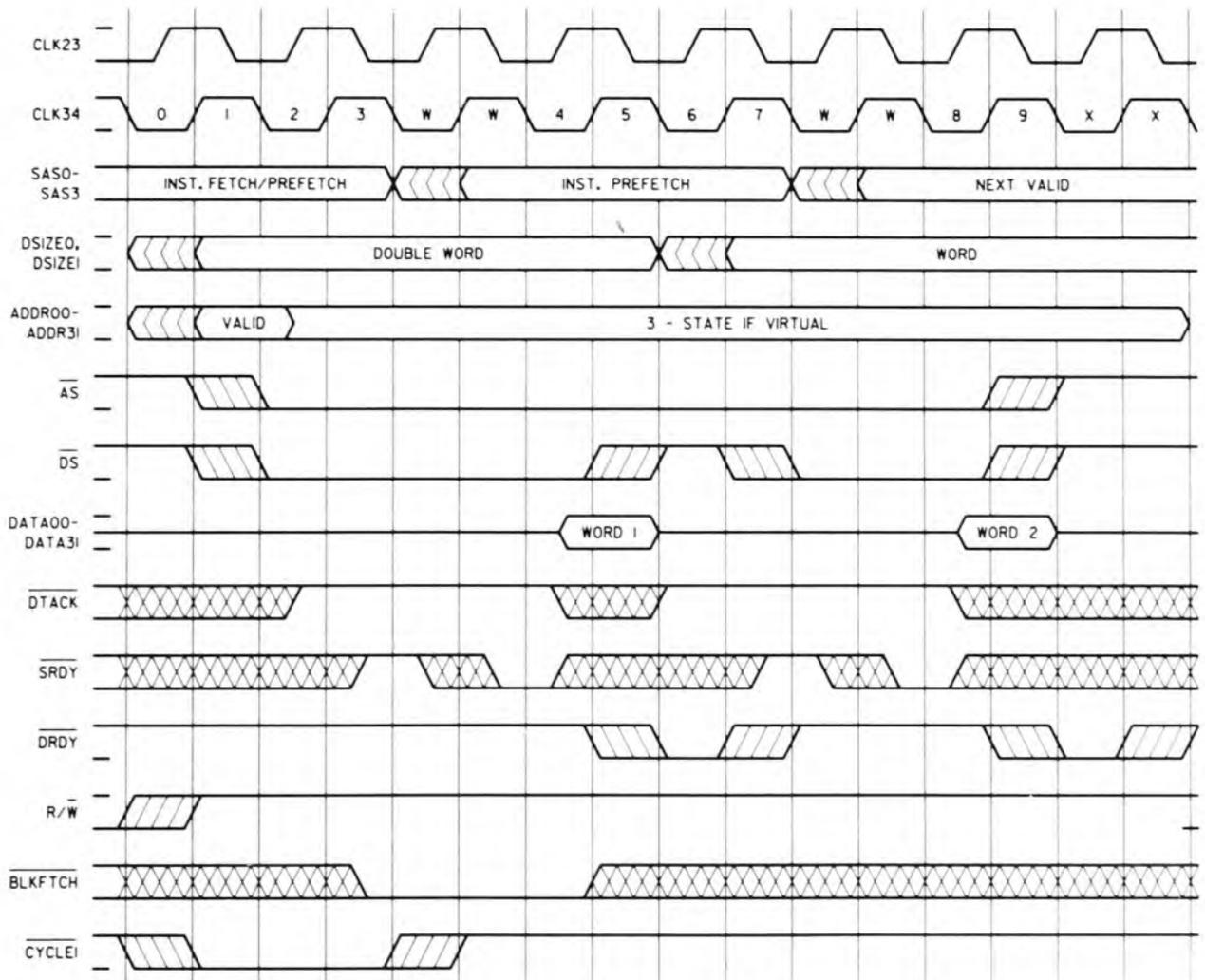


Note: Zero wait cycles.

Figure 4-13. Blockfetch Transaction Using  $\overline{\text{DTACK}}$  With Wait Cycle on Second Word

**4.4.4 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 a  $\overline{\text{DTACK}}$  during clock state two nor a  $\overline{\text{SRDY}}$  during clock state three (see Figure 4-14). It inserted a wait cycle. During the second clock state "W", it sampled  $\overline{\text{SRDY}}$  and  $\overline{\text{BLKFTCH}}$ , then latched the data and terminated  $\overline{\text{DS}}$ . The CPU proceeded to the second fetch. During clock state six it did not sample  $\overline{\text{DTACK}}$  nor a  $\overline{\text{SRDY}}$  during clock state seven. The CPU inserted a wait cycle. During the second clock state "W" it sampled  $\overline{\text{SRDY}}$ , then latched the data and terminated the transaction.



Note: Wait cycle on both words.

**Figure 4-14. Blockfetch Transaction Using  $\overline{\text{SRDY}}$  With Wait Cycles on Both Words**

## BUS OPERATION

### Bus Exceptions

#### 4.5 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 input ( $\overline{\text{FAULT}}$ ). This causes the CPU to terminate the access and possibly execute a fault handling routine. The WE 32101 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  $\overline{\text{RETRY}}$  input. A relinquish and retry causes the microprocessor to give up its bus and 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
$\overline{\text{RRREQ}}$ , $\overline{\text{RETRY}}$ , $\overline{\text{FAULT}}$	The relinquish and retry request ( $\overline{\text{RRREQ}}$ ) 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 fault ( $\overline{\text{FAULT}}$ ) input is still asserted during the retried access, the fault will be honored (recognized). The fault input will be recognized only during the retried access.
$\overline{\text{RRREQ}}$ , $\overline{\text{RETRY}}$	The relinquish and retry request ( $\overline{\text{RRREQ}}$ ) is honored first. The microprocessor 3-states the appropriate signals and then asserts the relinquish and retry acknowledge output ( $\overline{\text{RRRACK}}$ ). 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 $\overline{\text{FAULT}}$ will be recognized on the retried access if it is still asserted.

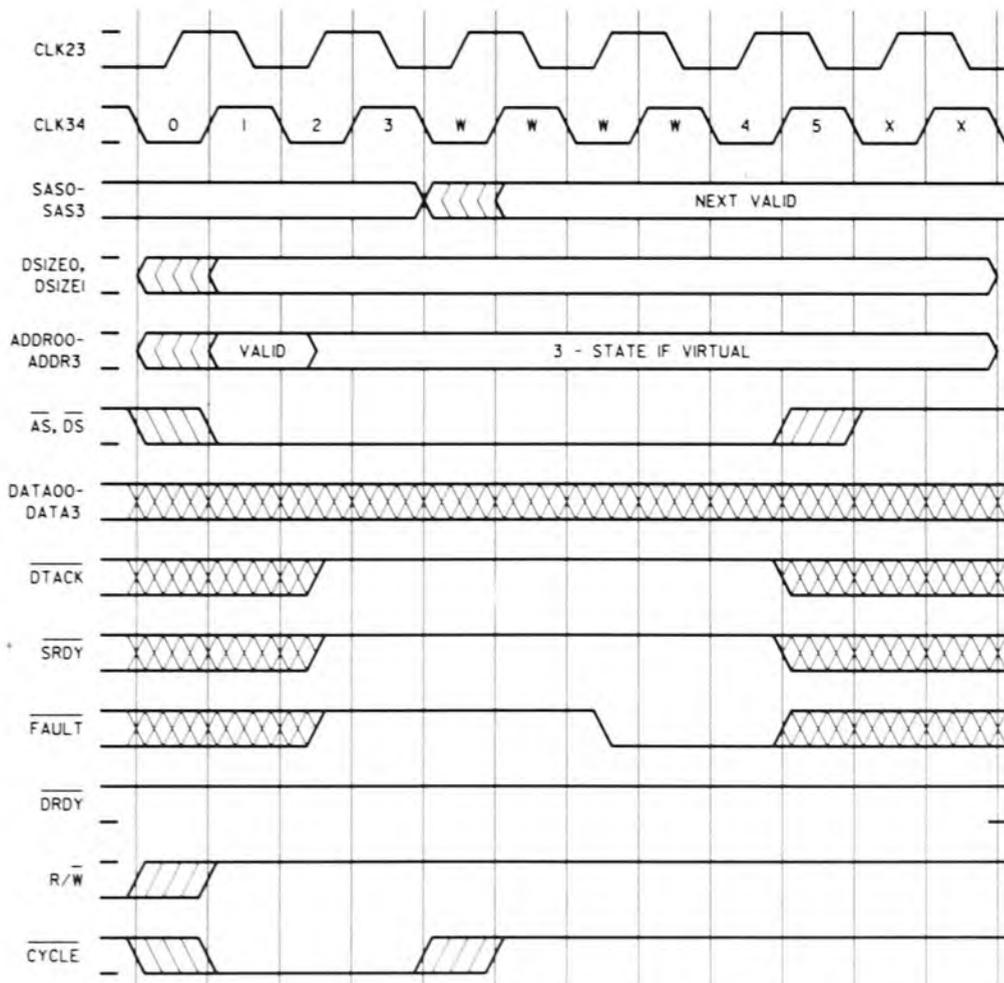
\* Table 4-1 applies only when the microprocessor is the bus master.

**4.5.1 Faults**

A bus transaction can be terminated by a bus exception: in this case,  $\overline{\text{FAULT}}$  without a  $\overline{\text{DTACK}}$  or  $\overline{\text{SRDY}}$  (see Figure 4-15). On Figure 4-15, the CPU inserted two wait cycles because it did not receive an acknowledge or a bus exception. During the third clock state "W", the CPU asynchronously sampled  $\overline{\text{FAULT}}$  and terminated the transaction. Note that if a  $\overline{\text{DTACK}}$  was also sampled with the  $\overline{\text{FAULT}}$  during the third clock state "W", the figure would not change.

Upon the faulted transaction, the CPU will proceed 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 will do an instruction fetch, and if this is also faulted, the CPU will proceed 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  $\overline{\text{FAULT}}$ . If this bus transaction is a write, the CPU will sample  $\overline{\text{FAULT}}$  in the same way as in a read case. There are some differences for a blockfetch which can be seen in **4.6 Blockfetch Special Cases**.

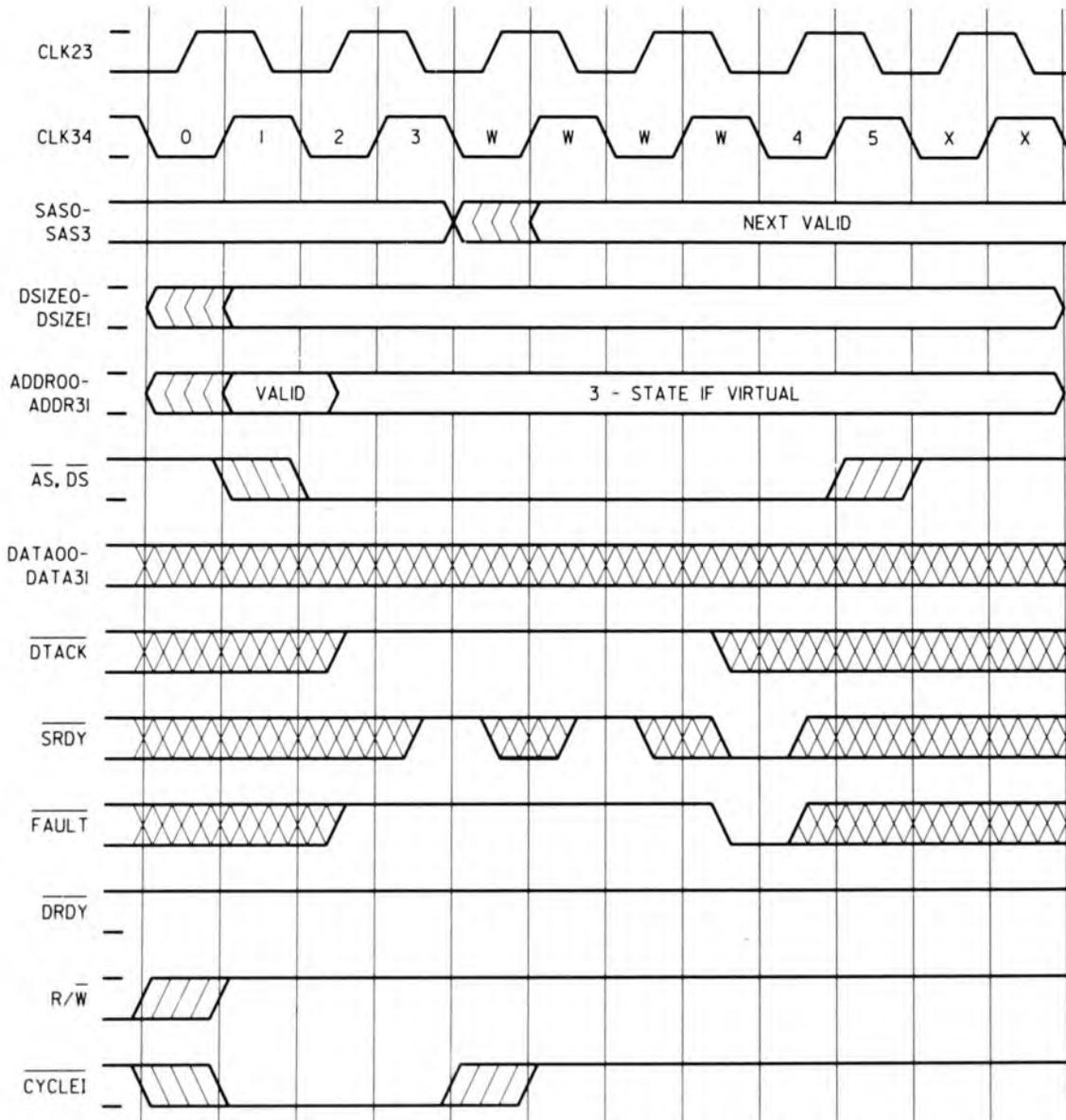


**Figure 4-15. Asynchronous Fault Without  $\overline{\text{DTACK}}$  and  $\overline{\text{SRDY}}$  (Read Transaction)**

**BUS OPERATION**  
**FAULT With SRDY**

**FAULT With SRDY**

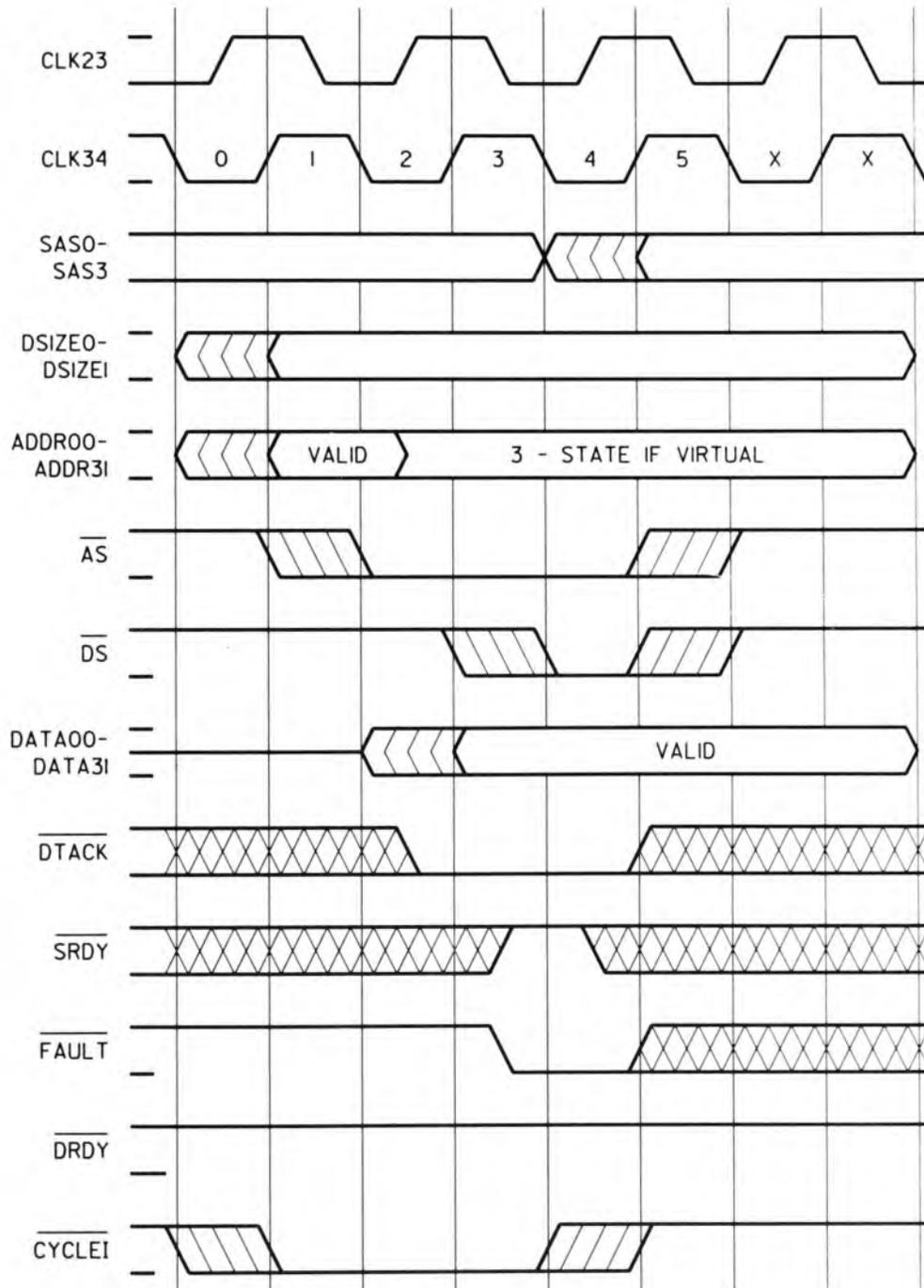
The CPU can sample  $\overline{\text{FAULT}}$  synchronously if it has a  $\overline{\text{SRDY}}$  with it. Figure 4-16 shows both  $\overline{\text{SRDY}}$  and  $\overline{\text{FAULT}}$  being sampled during the last clock state "W". The CPU then terminates the transaction and does not issue a  $\overline{\text{DRDY}}$ . For reads and writes, the CPU samples the  $\overline{\text{SRDY}}$  and  $\overline{\text{FAULT}}$  in the same way.



**Figure 4-16. Fault with Synchronous Ready ( $\overline{\text{SRDY}}$ ); i.e., Synchronous Fault**

**FAULT After DTACK**

The CPU can also sample  $\overline{\text{FAULT}}$  synchronously if it has sampled a  $\overline{\text{DTACK}}$  asynchronously in the same clock cycle. Figure 4-17 shows  $\overline{\text{DTACK}}$  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  $\overline{\text{DTACK}}$  and  $\overline{\text{FAULT}}$  is the same for reads.



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

**Figure 4-17. Fault After Assertion of  $\overline{\text{DTACK}}$  (Write Transaction is Shown)**

## BUS OPERATION

### Retry

#### 4.5.2 Retry

$\overline{\text{RETRY}}$  is sampled the same way the  $\overline{\text{FAULT}}$  is sampled. The previous figures on how  $\overline{\text{FAULT}}$  is sampled can have the words  $\overline{\text{FAULT}}$  replaced by  $\overline{\text{RETRY}}$  as far as the sampling is concerned. Figure 4-18 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 will redo the entire transaction. The SAS code will be the same as the first transaction as well as the address, DSIZE, and R/ $\overline{\text{W}}$ .

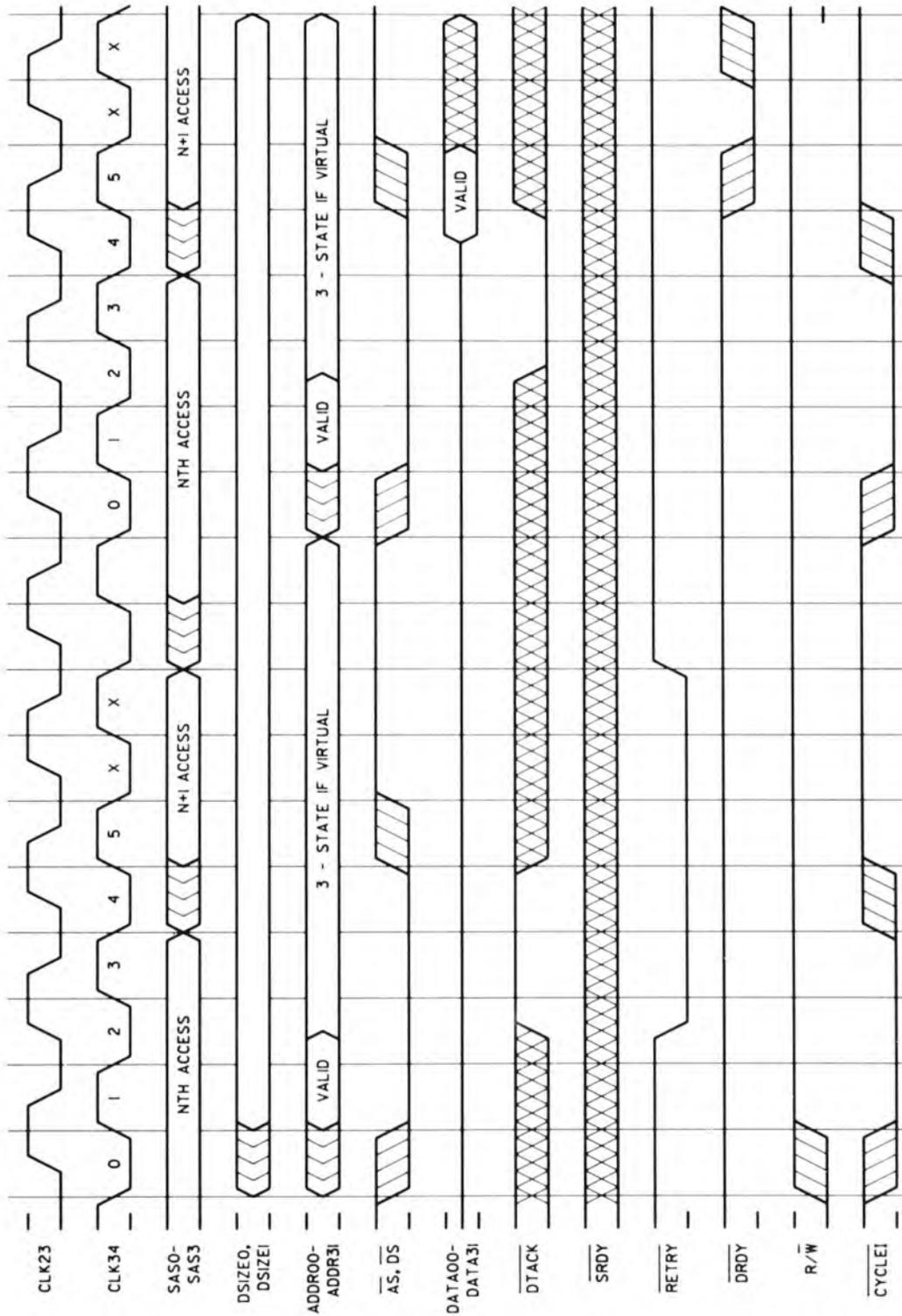
$\overline{\text{RETRY}}$  operates on reads and writes in the same way. There are some differences if the transaction is a blockfetch, and these can be seen in the  $\overline{\text{RETRY}}$  with blockfetch figures in **4.6 Blockfetch Special Cases**.

#### 4.5.3 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 4-19.

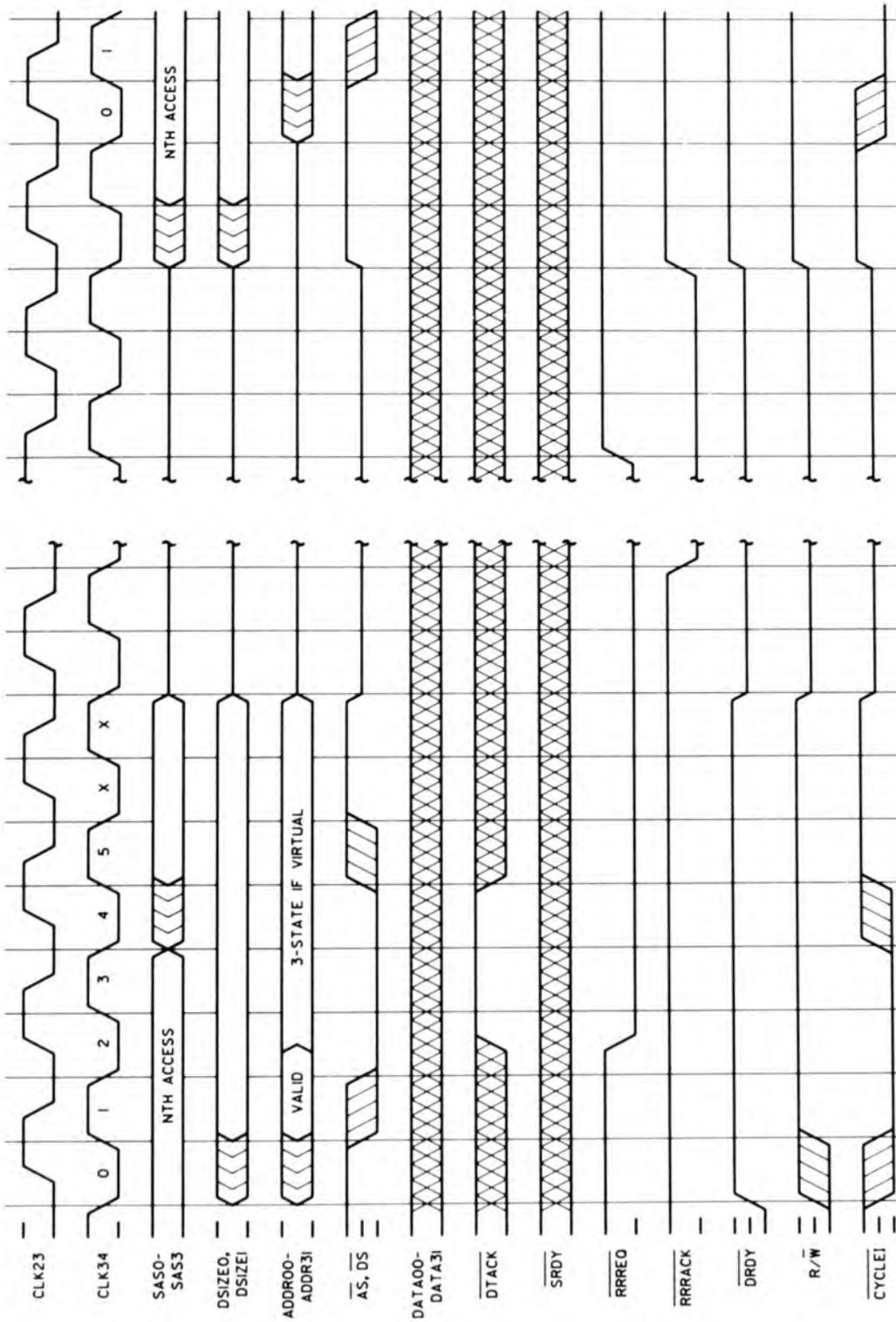
When the CPU samples  $\overline{\text{RRREQ}}$ , it terminates the transaction and does not issue a  $\overline{\text{DRDY}}$ . After the second clock state "X", the CPU 3-states the address and data buses as well as most of the control signals in order to allow some other device to use the bus. A cycle later the CPU issues  $\overline{\text{RRRACK}}$ . This indicates that the device that had issued  $\overline{\text{RRREQ}}$  can get onto the bus for a bus transaction. The CPU will continue 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 will take back the bus and redo the entire transaction. As in the  $\overline{\text{RETRY}}$  case, the SAS code, address, DSIZE, and R/ $\overline{\text{W}}$  will be 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 can be seen in the blockfetch with relinquish and retry figure in **4.6. Blockfetch Special Cases**.



**Figure 4-18. Retry of Transaction (Read Transaction is Shown)**

**BUS OPERATION**  
**Relinquish & Retry**



**Figure 4-19. Relinquish and Retry**

## **4.6 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 of bus exception. The following descriptions address these special cases.

### **4.6.1 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 where the SAS code is "instruction fetch" or "instruction fetch after PC discontinuity," the blockfetch transaction is altered as on Figure 4-20. 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 a bus exception. Upon seeing blockfetch and 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.

### **4.6.2 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 on with its current execution. This is illustrated on Figure 4-21. 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 on to the second transaction. The second transaction operates normally.

### **4.6.3 Retry on First Word of Blockfetch**

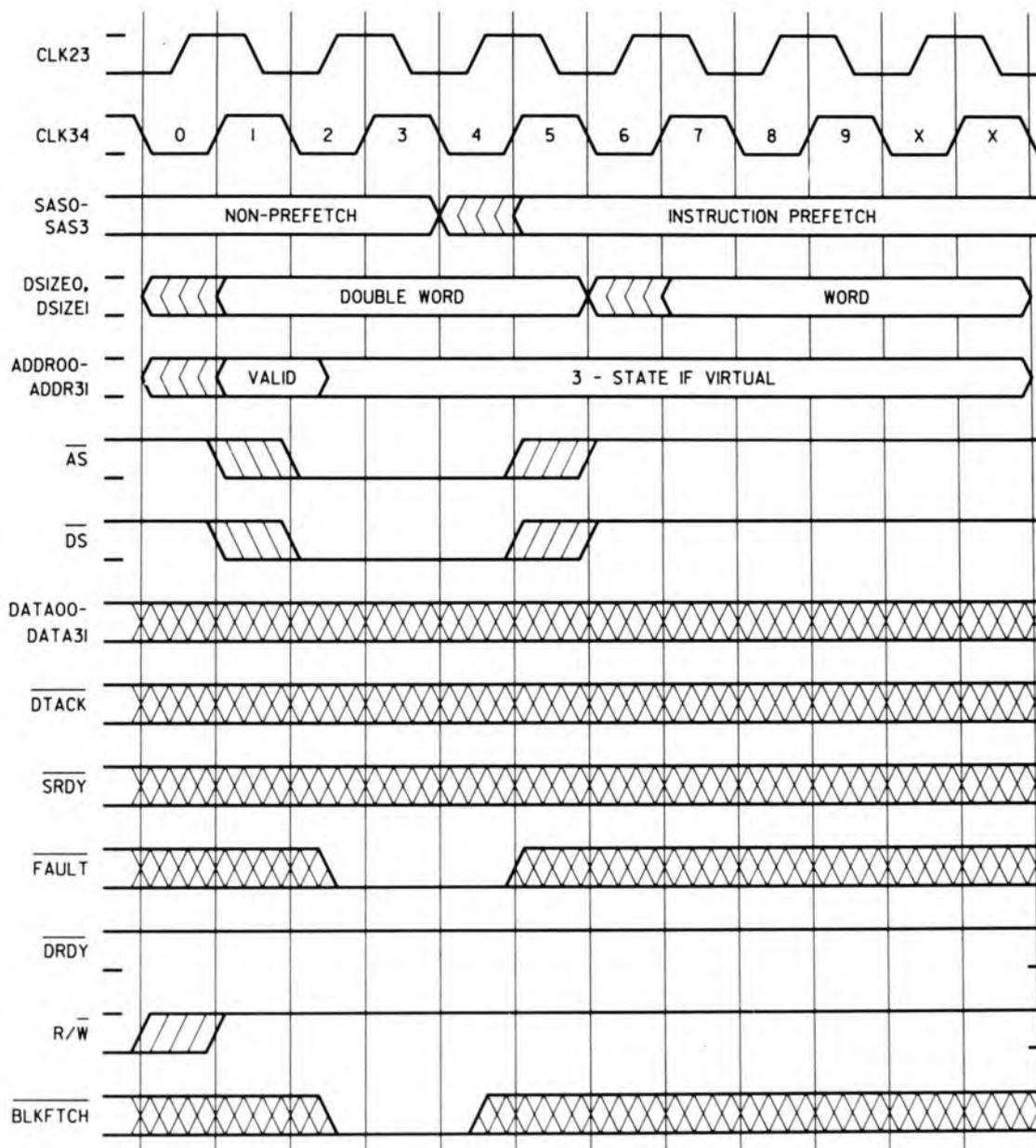
The CPU samples  $\overline{\text{RETRY}}$  during clock state three and the  $\overline{\text{BLKFTCH}}$  during clock state four. The CPU then terminates the transaction by removing  $\overline{\text{DS}}$  and  $\overline{\text{AS}}$  at clock state five. At this point the CPU waits for  $\overline{\text{RETRY}}$  to go away. When it does the CPU retries the entire blockfetch transaction. This process is illustrated on Figure 4-22.

### **4.6.4 Retry on Second Word of Blockfetch**

In this case (see Figure 4-23), the CPU samples  $\overline{\text{BLKFTCH}}$  and  $\overline{\text{DTACK}}$ , clocks the data from the data bus, issues a  $\overline{\text{DRDY}}$ , and continues on to the second word. During the first clock 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 removed. 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 removed, the CPU continues on with its current execution. If the CPU wants to do a new bus transaction it will proceed with this one since it will not retry the blockfetch.

# BUS OPERATION

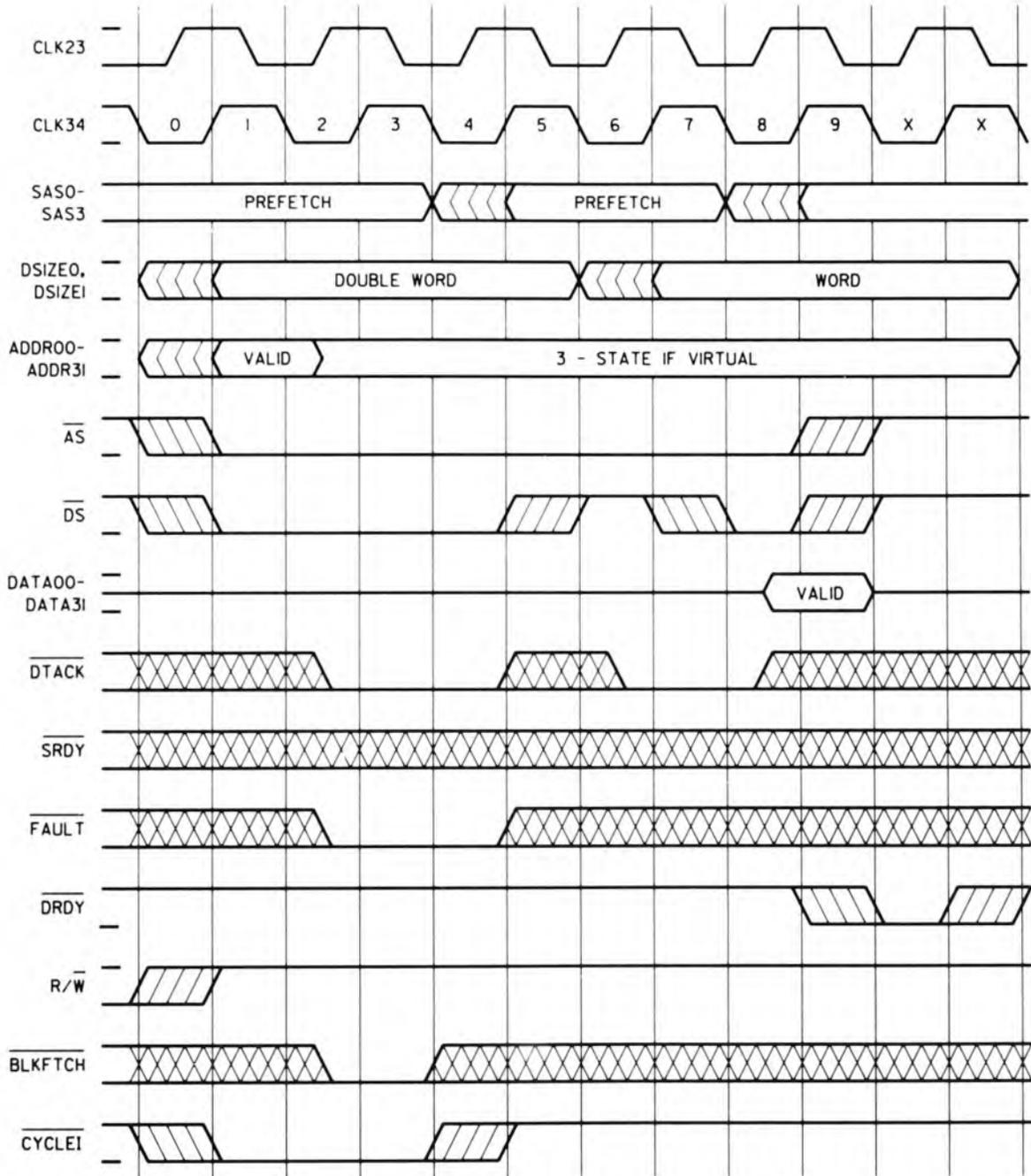
## Retry on Second Word of Blockfetch



**Figure 4-20. Fault on First Word of Blockfetch Transaction With Access Status Code Other Than Prefetch**

## BUS OPERATION

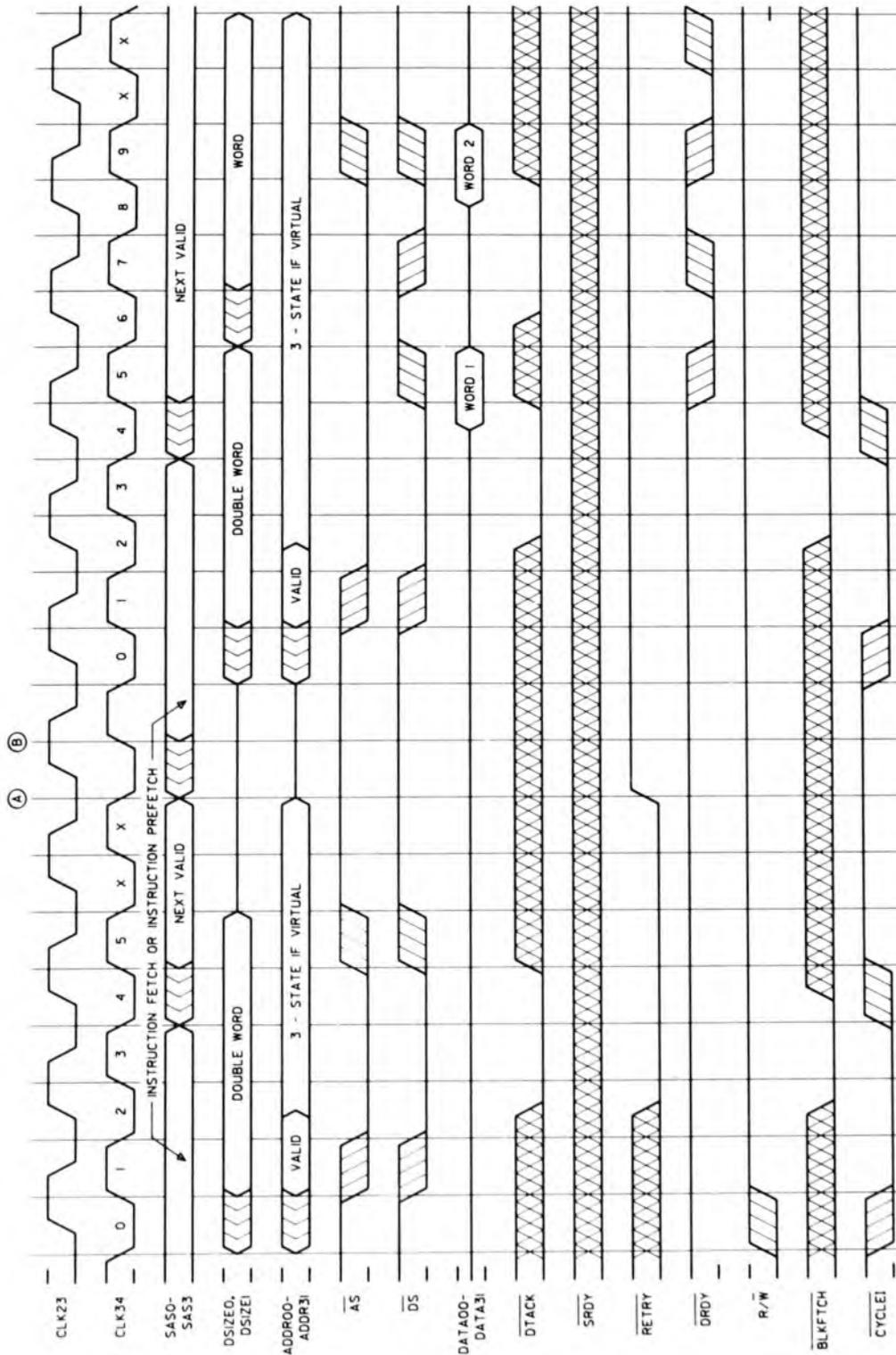
### Retry on Second Word of Blockfetch



**Figure 4-21. Fault on First Word of Blockfetch Transaction  
With Access Status Code of Prefetch**

# BUS OPERATION

## Retry on Second Word of Blockfetch



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

Figure 4-22. Retry on First Word of Blockfetch Transaction



## BUS OPERATION

### Relinquish and Retry on Blockfetch

#### 4.6.5 Relinquish and Retry on Blockfetch

Figures 4-22 and 4-23 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 4-22 will look the same if the bus exception is  $\overline{\text{RRREQ}}$  rather than  $\overline{\text{RETRY}}$ . However, the CPU will release the bus before doing the retried transaction. Additionally, the CPU will 3-state the bus at the end of the second clock state "X" (indicated by A on the diagram). One cycle later it will issue a relinquish and retry request acknowledge to tell the requesting device that it can now use the bus. When  $\overline{\text{RRREQ}}$  is removed, the CPU will continue with the retried transaction starting at point B.

The same explanation applies for a  $\overline{\text{RRREQ}}$  on the second word of a blockfetch (Figure 4-23). As above, the CPU would 3-state the bus at point A and issue a  $\overline{\text{RRRACK}}$ . Once the  $\overline{\text{RRREQ}}$  is removed, the CPU will continue with the next bus transaction starting at point B and will not retry the blockfetch.

## 4.7 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 pins (IPL0—IPL3) 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 will always interrupt the microprocessor regardless of the IPL field value.

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

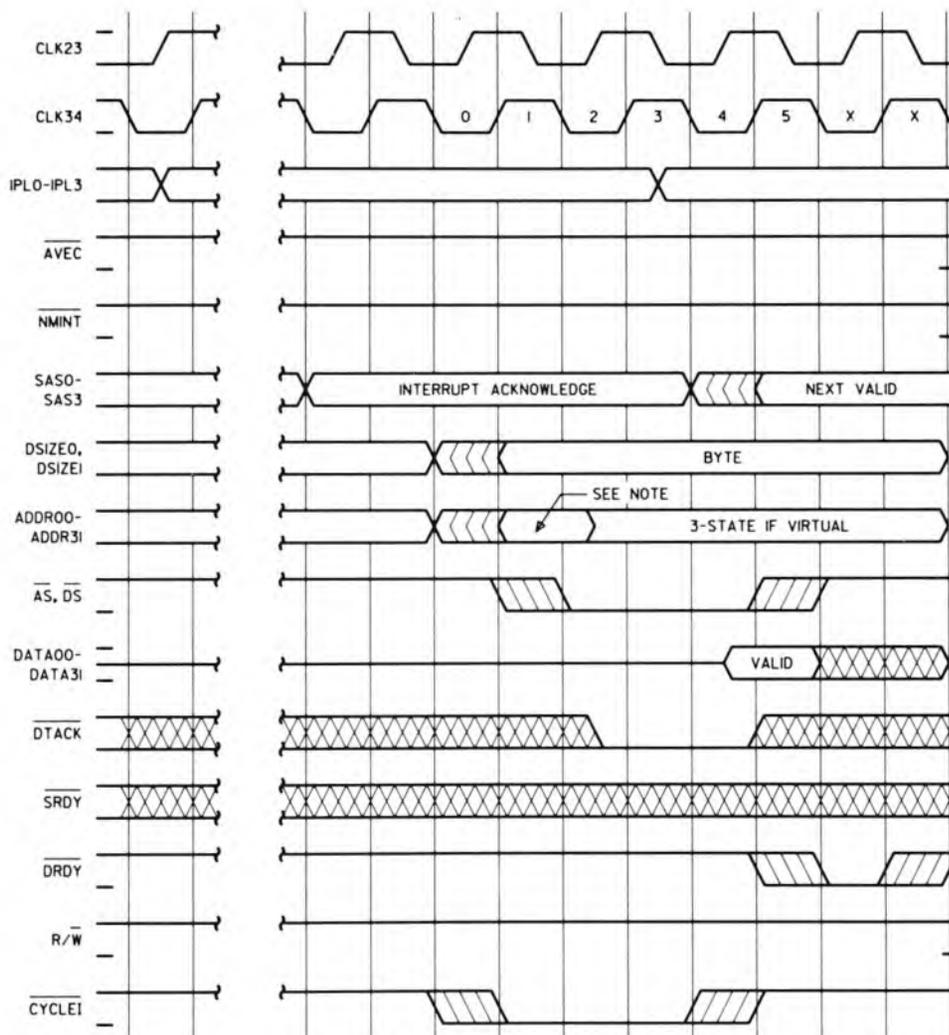
### 4.7.1 Interrupt Acknowledge

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.

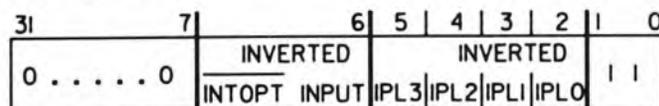
## BUS OPERATION

### Interrupt Acknowledge

The interrupt acknowledge transaction, is illustrated on Figure 4-24 which depicts the case in which a value placed on the IPL0—IPL3 inputs causes an interrupt. In this case, the interrupt acknowledge is issued in response to the application of the IPL pins and INTOPT pin with no AVEC or 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 Figure 4-24. The interrupting device drives the data bus with the 8-bit offset and a memory acknowledge; in this case a DTACK. The bus exceptions are accepted during this bus transaction. The IPL input values should be removed once the corresponding interrupt acknowledge has occurred.



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



**Figure 4-24. Interrupt Acknowledge**

**BUS OPERATION**  
**Interrupt Acknowledge**

Table 4-2 summarizes how the interrupt priority levels are to be interpreted and shows the corresponding acknowledge for each level.

Table 4-2. 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	

x signifies no value placed on address bus.

### 4.7.2 Autovector Interrupt

If the autovector ( $\overline{AVEC}$ ) input is active during an interrupt request, the microprocessor will not fetch a vector number from the interrupting device. Instead, the microprocessor provides the interrupt vector by treating the inverted  $\overline{INTOPT}$  input, concatenated with the interrupt priority level input (IPL0—IPL3), 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 4-25 for an illustration of the auto-vector interrupt acknowledge transaction. In this transaction, an auto-vector acknowledge is issued in response to the application of the IPL pins and  $\overline{INTOPT}$  pin with  $\overline{AVEC}$  active and no  $\overline{NMINT}$ . Since the CPU does not need to read in an external value, it does an auto-vector 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  $\overline{AVEC}$  input values. No  $\overline{DRDY}$  is issued because there is no latching of data.

Because the IPL signals are double sampled and compared for greater noise immunity, the single sampled  $\overline{AVEC}$  signal must be applied on the second sample of the IPL signals. This accounts for the one cycle difference between the two signals.

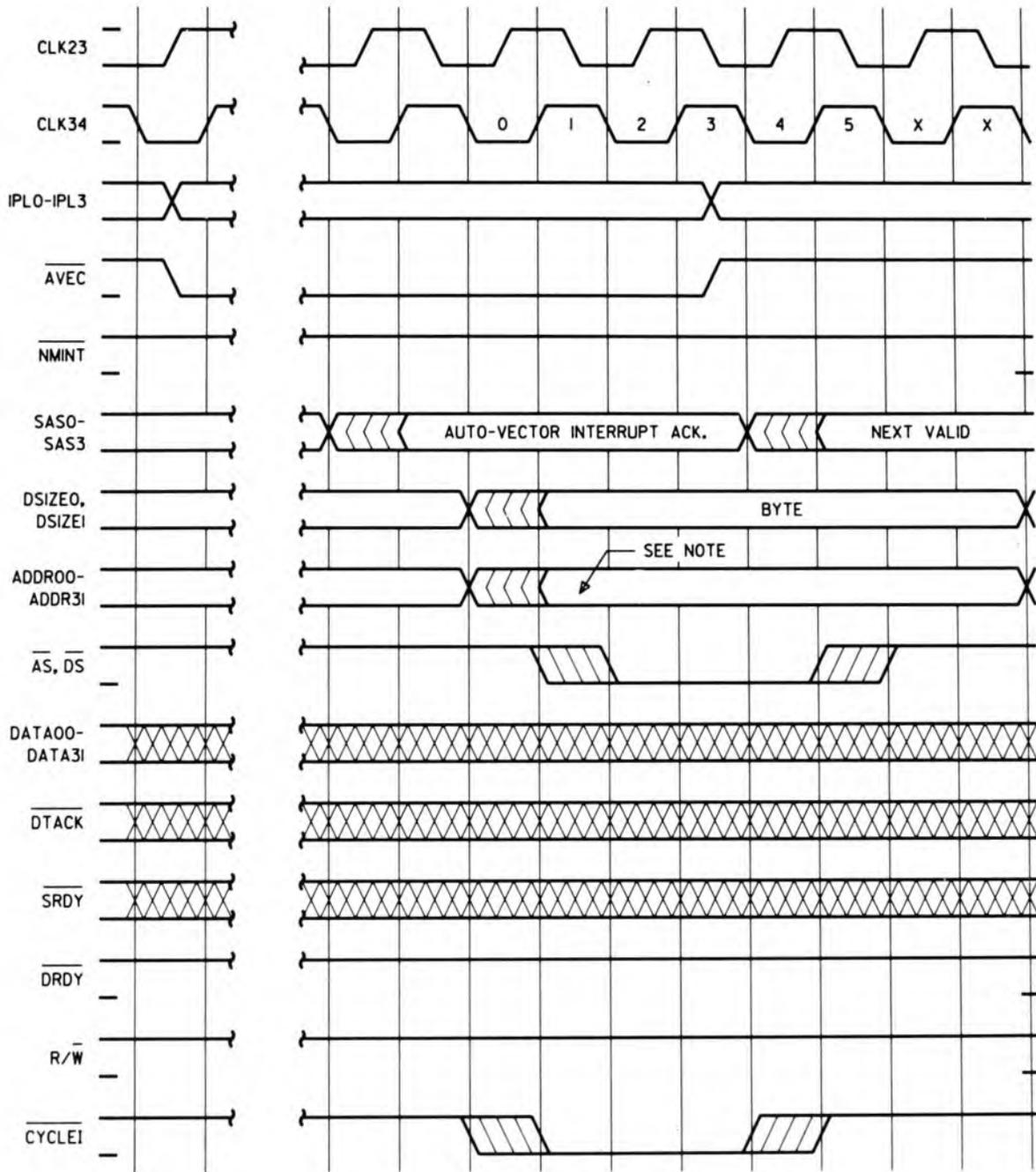
### 4.7.3 Nonmaskable Interrupt

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 input ( $\overline{NMINT}$ ) 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 4-26 illustrates the nonmaskable interrupt acknowledge transaction. Here, a nonmaskable interrupt acknowledge is issued in response to the application of the  $\overline{NMINT}$  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  $\overline{NMINT}$  when it sees the acknowledge. The SAS code is "auto-vector acknowledge," but the interrupt vector is 0. ADDR00 can be used to determine the difference between the  $\overline{AVEC}$  and  $\overline{NMINT}$  interrupts. It is a 1 for auto-vector and a 0 for nonmaskable interrupt.

# BUS OPERATION

## Nonmaskable 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				1	1
		INTOPT INPUT		IPL3	IPL2	IPL1	IPL0		

Figure 4-25. Autovector Interrupt Acknowledge



## BUS OPERATION

### Bus Arbitration

#### 4.8 BUS ARBITRATION

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

The relinquish and retry request has priority over a bus request. The microprocessor will only acknowledge a relinquish and retry request during bus transactions; however, it will ignore 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.

##### 4.8.1 Bus Request During a Bus Transaction

BUSRQ is sampled independently of bus transactions at the beginning of every clock cycle. On Figure 4-27 it is sampled for the first time at the beginning of clock state two. After sampling 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, BRACK. At this point the device requesting the bus can perform its operations. When finished, the device drops the BUSRQ. After seeing this drop, the CPU removes 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 4-13. **Supplementary Protocol Diagrams** for an example.

For a bus request that does not occur during a bus transaction, the CPU will 3-state the bus a cycle after sampling BUSRQ and issue BRACK a cycle after that.

# BUS OPERATION

## Bus Request During a Bus Transaction

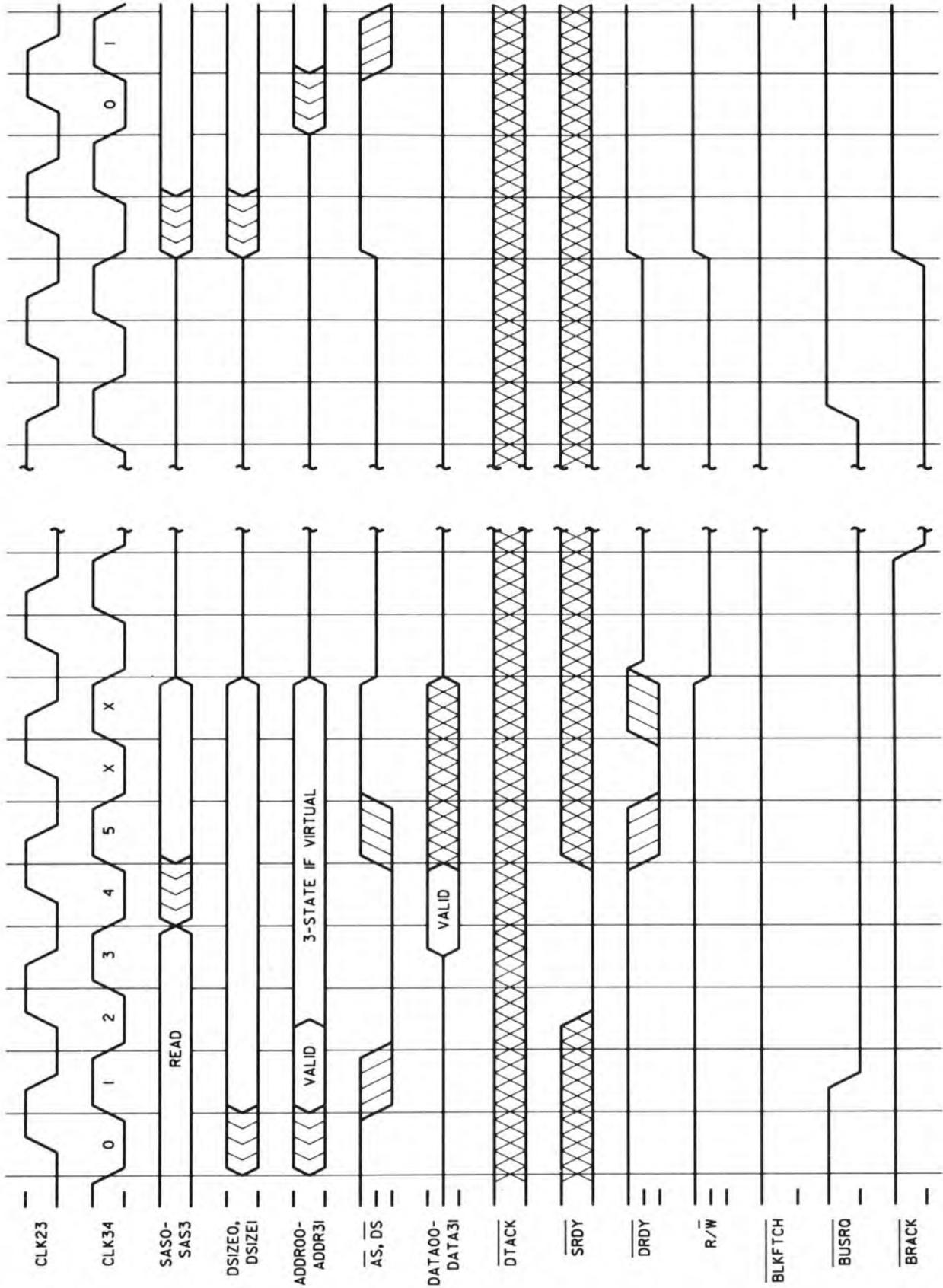


Figure 4-27. Bus Request During a Transaction

## BUS OPERATION

### Bus Request During a Bus Transaction

**Table 4-3. Interrupt Acknowledge Summary**

<b>Interrupt Priority</b>	<b>Interrupt Acknowledge</b>	$\overline{\text{AVEC}}$	$\overline{\text{NMINT}}$	<b>QIE</b>	<b>Result</b>
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 auto-vector 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.
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 auto-vector interrupt at vector number 0. The address bus contains all zeros during the acknowledge.

### 4.8.2 DMA Operation

The microprocessor provides the 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:

$\overline{\text{ABORT}}$	$\text{DATA00} - \text{DATA31}$	$\text{R}/\overline{\text{W}}$
$\text{ADDR00} - \text{ADDR31}$	$\overline{\text{DRDY}}$	$\text{SAS0} - \text{SAS3}$
$\overline{\text{AS}}$	$\overline{\text{DS}}$	$\overline{\text{VAD}}$
$\overline{\text{CYCLEI}}$	$\text{DSIZE0} - \text{DSIZE1}$	$\text{XMD0} - \text{XMD1}$

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.

Terminating a DMA operation reverses the start of DMA. 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.

<b>Table 4-4. Output Signal States After DMA Request is Acknowledged</b>			
<b>Output Signal</b>	<b>Signal State</b>	<b>Output Signal</b>	<b>Signal State</b>
$\overline{\text{ABORT}}$	Z*	$\text{DSIZE0} - \text{DSIZE1}$	Z
$\text{ADDR00} - \text{ADDR31}$	Z	$\text{R}/\overline{\text{W}}$	Z*
$\overline{\text{AS}}$	Z*	$\overline{\text{RESET}}$	Logic 1
$\overline{\text{BRACK}}$	Logic 0	$\overline{\text{RRRACK}}$	Logic 1
$\overline{\text{CYCLEI}}$	Z*	$\text{SAS0} - \text{SAS3}$	Z*
$\text{DATA00} - \text{DATA31}$	Z	$\overline{\text{VAD}}$	Z
$\overline{\text{DRDY}}$	Z*	$\text{XMD0} - \text{XMD1}$	Z
$\overline{\text{DS}}$	Z*		

Where:

Z High impedance state

Z\* High impedance. Held at logic 1 with external passive hold resistor.

## BUS OPERATION

### Reset

#### 4.9 RESET

The microprocessor handles two types of reset requests; system and internal. A reset has the highest priority and will preempt any ongoing microprocessor operation.

##### 4.9.1 System Reset

A system reset is initiated when the system drives the reset request input ( $\overline{\text{RESETR}}$ ) 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 pins once  $\overline{\text{RESET}}$  is negated. During this sequence the microprocessor performs the following register initialization to restart operations.

- The microprocessor changes to physical addressing mode.
- The microprocessor 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).
- The microprocessor fetches a word at the PCB address and stores it in the processor status word.
- The microprocessor 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.
- The microprocessor 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 microprocessor clears the bit (0), fetches a word at the location twelve bytes from the initial PCBP, and stores it as the new PCBP.
- The microprocessor begins execution at the address specified by the PC.

##### 4.9.2 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 will still go active for 128 clock cycles after  $\overline{\text{RESETR}}$  is released.

<b>Table 4-5. Output States on Reset</b>		
<b>Output Signal</b>	<b>Signal State</b>	
	<b>CPU is Bus Arbiter</b>	<b>CPU is Not Bus Arbiter</b>
$\overline{\text{ABORT}}$	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–DSIZE1	Logic 0	Z
IQS0–IQS1	Logic 1	Logic 1
R/ $\overline{\text{W}}$	Logic 1	Z
$\overline{\text{RRACK}}$ ,	Z*	Z*
SAS0–SAS3	Logic 1	Z
$\overline{\text{SOI}}$	Logic 1	Logic 1
$\overline{\text{VAD}}$	**	Z
XMD0–XMD1	†	Z

Where:

Z High impedance state

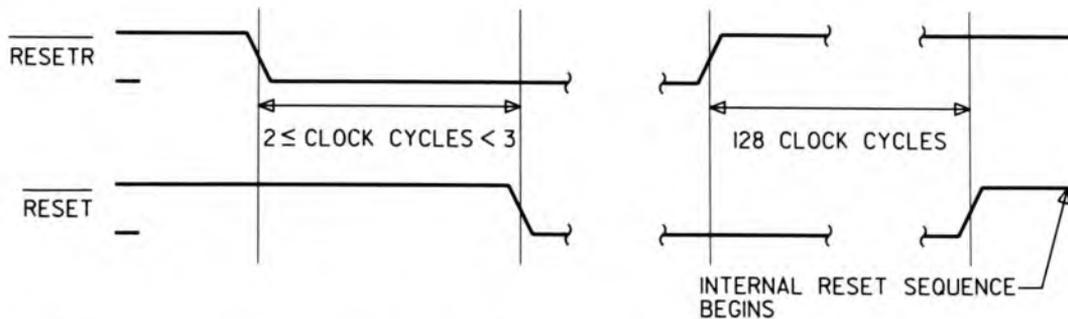
\* Open drain output not actively driven under this condition.

\*\* Not guaranteed to be logic 1 (i.e., physical address) until approximately 38 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 two clock cycles to be recognized.  
 $\overline{\text{RESET}}$  is negated 128 clock cycles after negation of  $\overline{\text{RESETR}}$ .

Figure 4-28. Reset Sequence

### 4.9.3 Reset Sequence

The reset sequence is depicted on Figure 4-28. 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. Operating System Considerations).

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

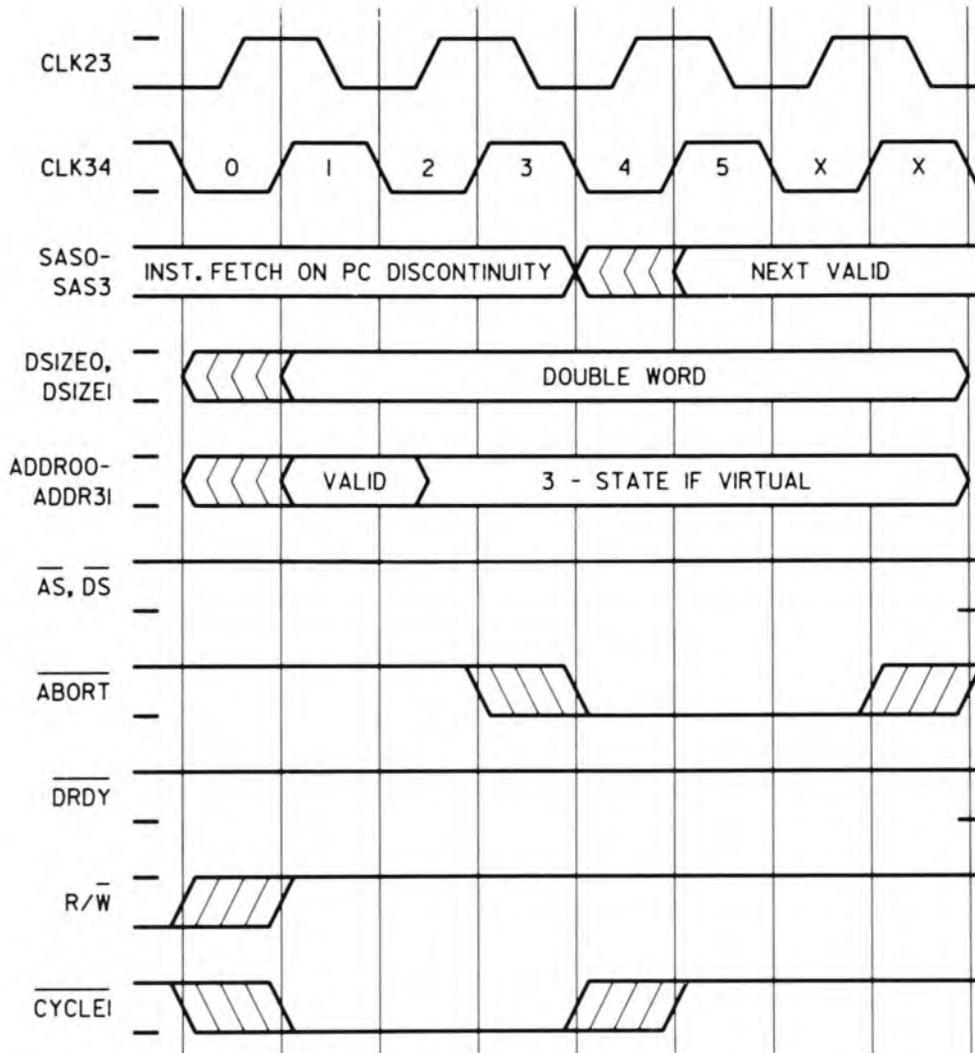
## 4.10 ABORTED MEMORY ACCESSES

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.

### 4.10.1 Aborted Access on PC Discontinuity with Instruction Cache Hit

Figure 4-29 depicts 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.

**BUS OPERATION**  
**Aborted Access on PC Discontinuity with Instruction Cache Hit**



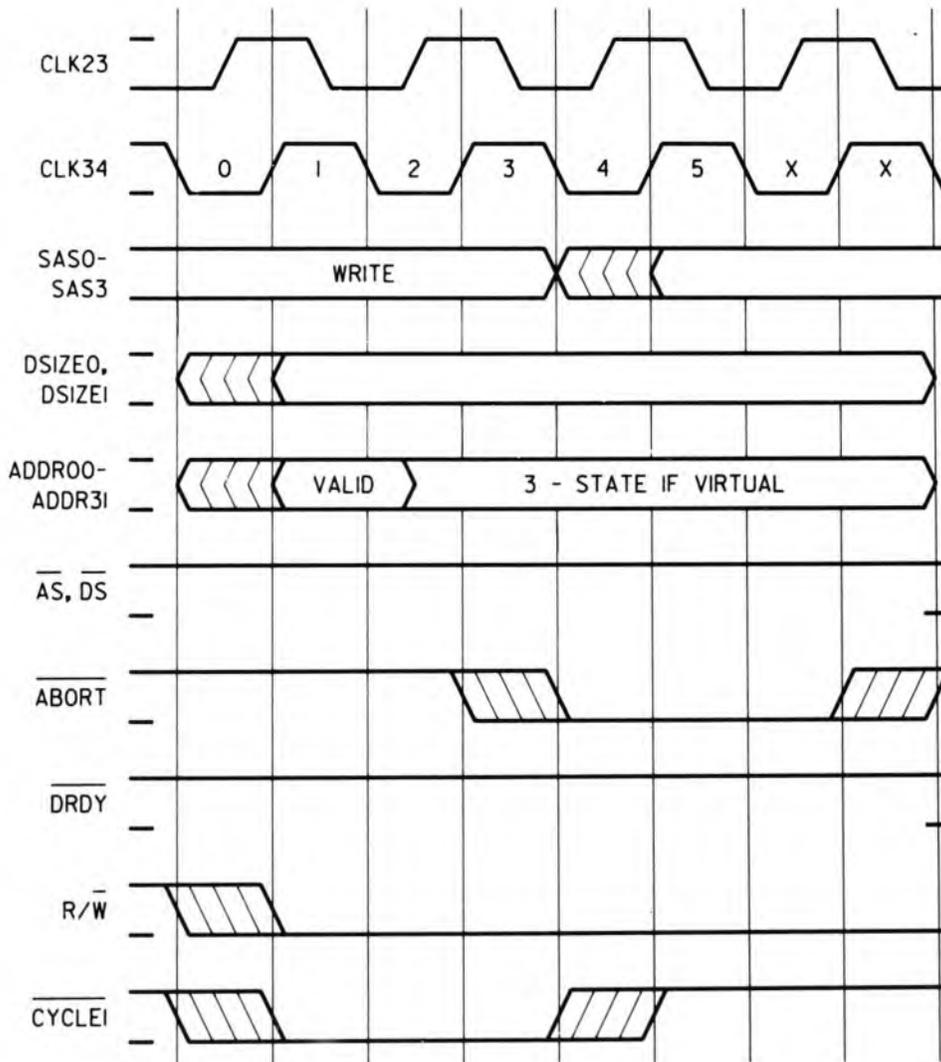
Note:  $\overline{\text{BLKFTCH}}$ ,  $\text{DATA00} - \text{DATA31}$ ,  $\overline{\text{DTACK}}$ ,  $\overline{\text{FAULT}}$ ,  $\overline{\text{RETRY}}$ ,  $\overline{\text{RRREQ}}$ , and  $\overline{\text{SRDY}}$  are ignored.

**Figure 4-29. Aborted Access on Instruction-Cache Hit With PC Discontinuity**

**BUS OPERATION**  
**Alignment Fault Bus Activity**

**4.10.2 Alignment Fault Bus Activity**

If the CPU detects an alignment fault on an intended CPU-generated bus transaction, it will terminate the transaction and proceed to the fault handler. The write transaction on Figure 4-30 started with the address bus, as well as the DSIZE, SAS, R/W, and CYCLEI being driven by the CPU. The CPU detects the alignment fault and does not issue  $\overline{AS}$  and  $\overline{DS}$ . It issues  $\overline{ABORT}$ , starting at clock state three, to indicate that it is terminating the transaction. The CPU ignores memory, acknowledges and bus exceptions during this time.  $\overline{DRDY}$  is not issued.



Notes:

1.  $\overline{DATA00}$ – $\overline{DATA31}$ ,  $\overline{DTACK}$ ,  $\overline{FAULT}$ ,  $\overline{RETRY}$ ,  $\overline{RRREQ}$ , and  $\overline{SRDY}$  are ignored.
2. Protocol is the same for a read transaction

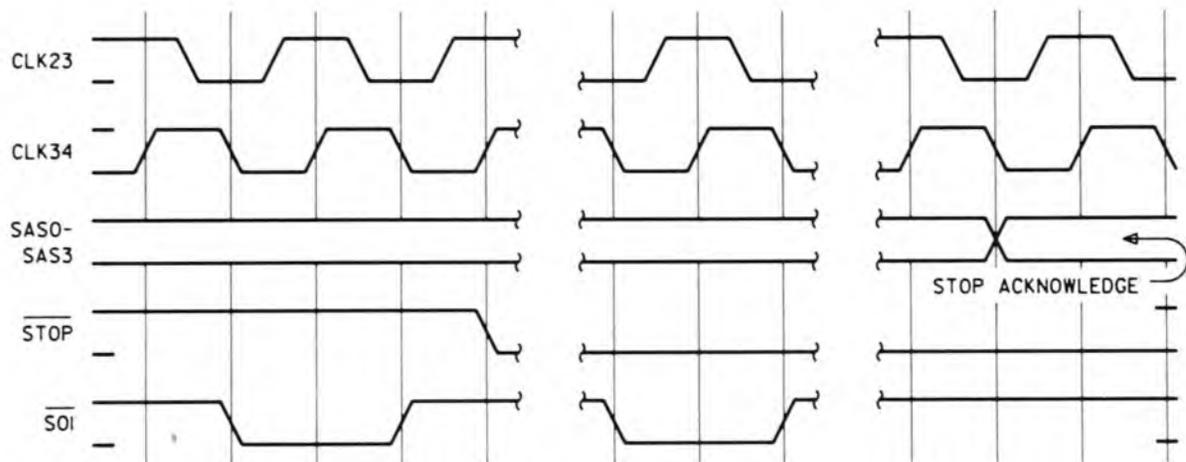
**Figure 4-30. Alignment Fault Bus Activity (Write Transaction is Shown)**

**4.11 SINGLE-STEP OPERATION**

Hardware single-step can be performed by use of the stop input ( $\overline{\text{STOP}}$ ). 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 4-31 depicts the start of single-step operation. The operation is started by the assertion of  $\overline{\text{STOP}}$ . The CPU will complete the current instruction and execute, at most, one more instruction. After this the CPU stops execution and issues the SAS code "stop acknowledge." The CPU will remain in this state until  $\overline{\text{STOP}}$  is released.



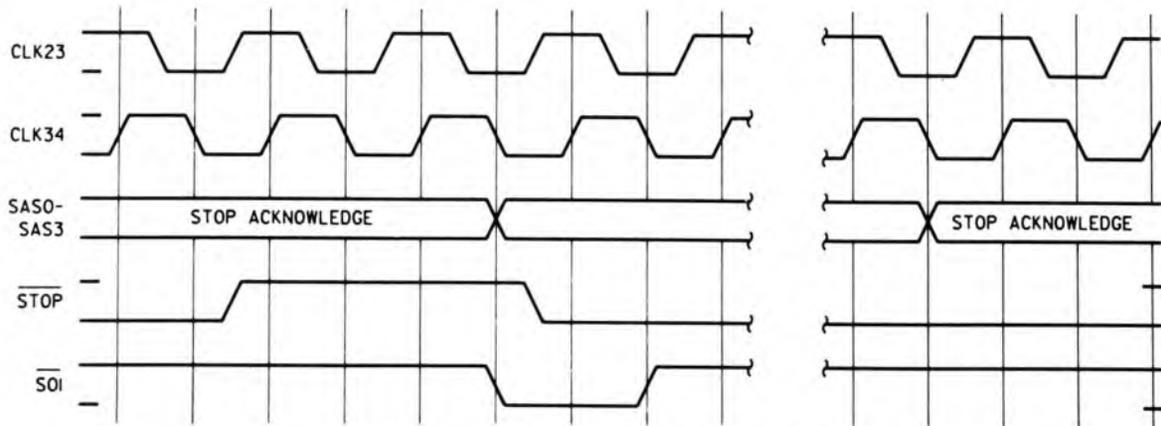
Notes:

1. At most, one full assertion of  $\overline{\text{SOI}}$  may appear before  $\overline{\text{STOP}}$  is acknowledged.
2.  $\overline{\text{BARB}} = 0$  and  $\overline{\text{BRACK}} = 1$  in order to see stop acknowledge access status code.

**Figure 4-31. 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 4-32. Single-Step Operation**

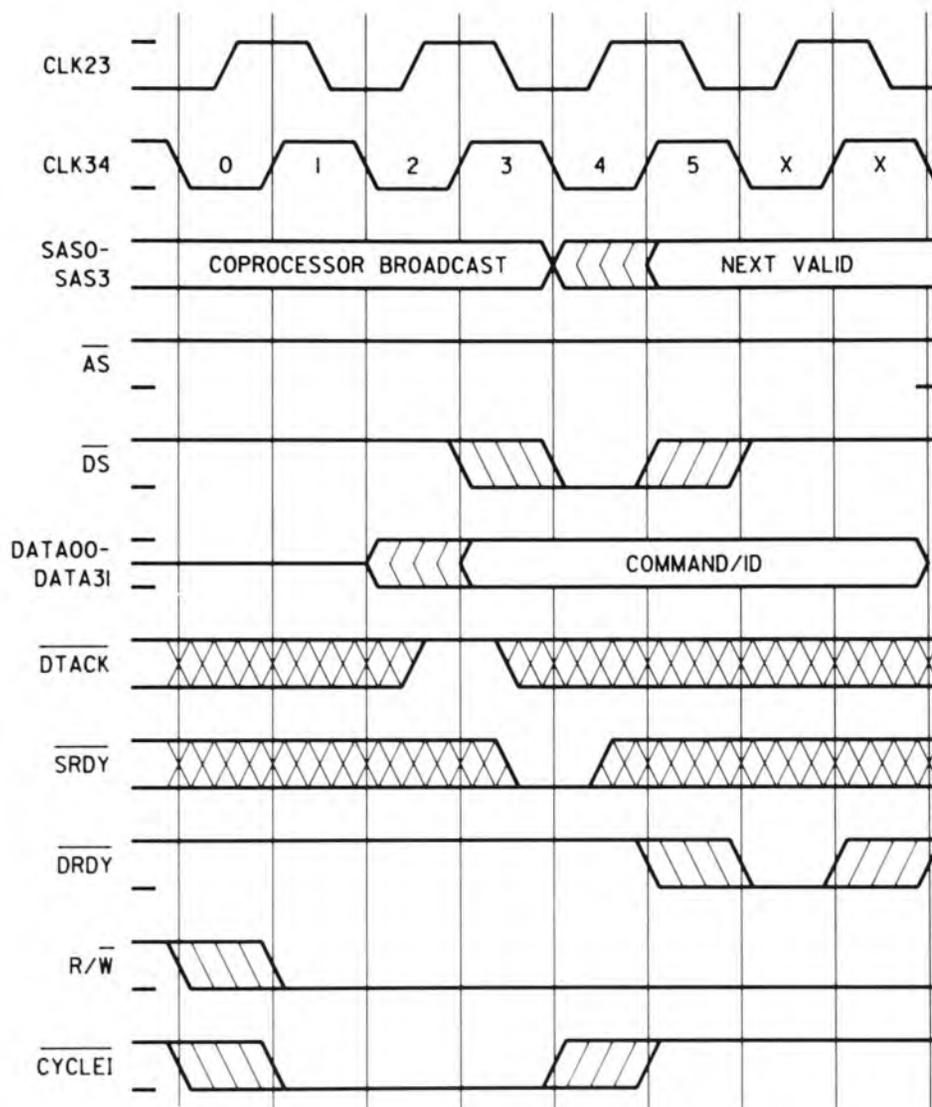
After the CPU has stopped, and until a start of instruction output ( $\overline{SOI}$ ) is issued, instruction by instruction execution can be performed by releasing  $\overline{STOP}$  and keeping it released until an  $\overline{SOI}$  output is issued. At this point, immediately application and holding of  $\overline{STOP}$  will prevent a second instruction from starting. With  $\overline{STOP}$  asserted, the CPU will complete the instruction and issue the stop acknowledge SAS code. To resume normal execution,  $\overline{STOP}$  must be completely released. The single-step operation is shown on Figure 4-32.

## 4.12 COPROCESSOR OPERATIONS

The WE 32100 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 occurs.

### 4.12.1 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/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. The coprocessor responds with a memory acknowledge. The CPU then terminates the transaction and goes on to the next one. The CPU will insert 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 4-33 shows the zero, one, and two wait cycle cases before the coprocessor responds with a memory acknowledge (in this case,  $\overline{SRDY}$ ).



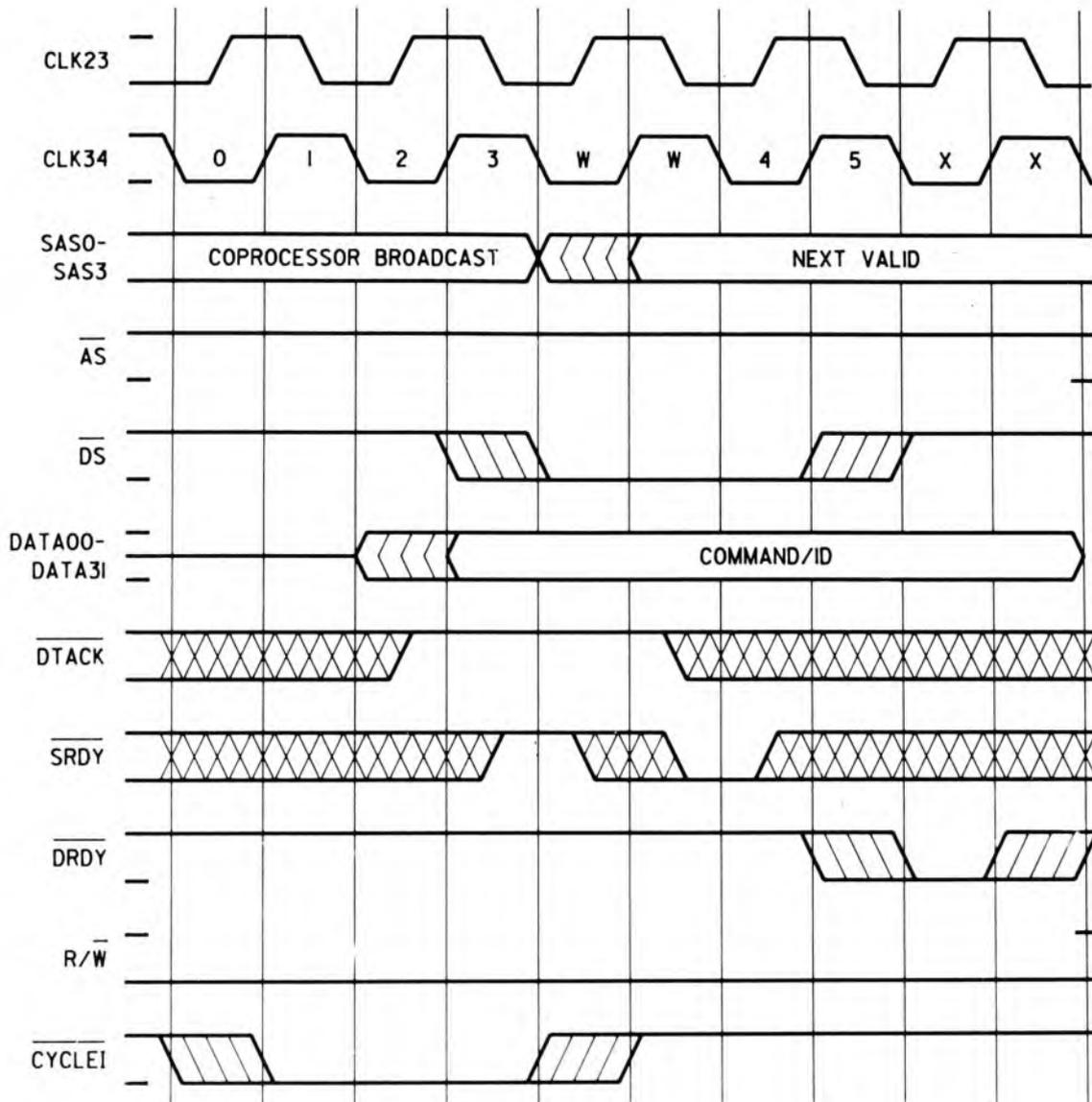
A. Zero Wait Cycles

Notes:

1. Zero, one, and two wait cycles using  $\overline{\text{SRDY}}$ .
2. Greater than two wait cycles causes internal CPU memory fault.

**Figure 4-33. Coprocessor Command and ID Transfer**  
**(Sheet 1 of 3)**

**BUS OPERATION**  
**Coprocessor Broadcast**

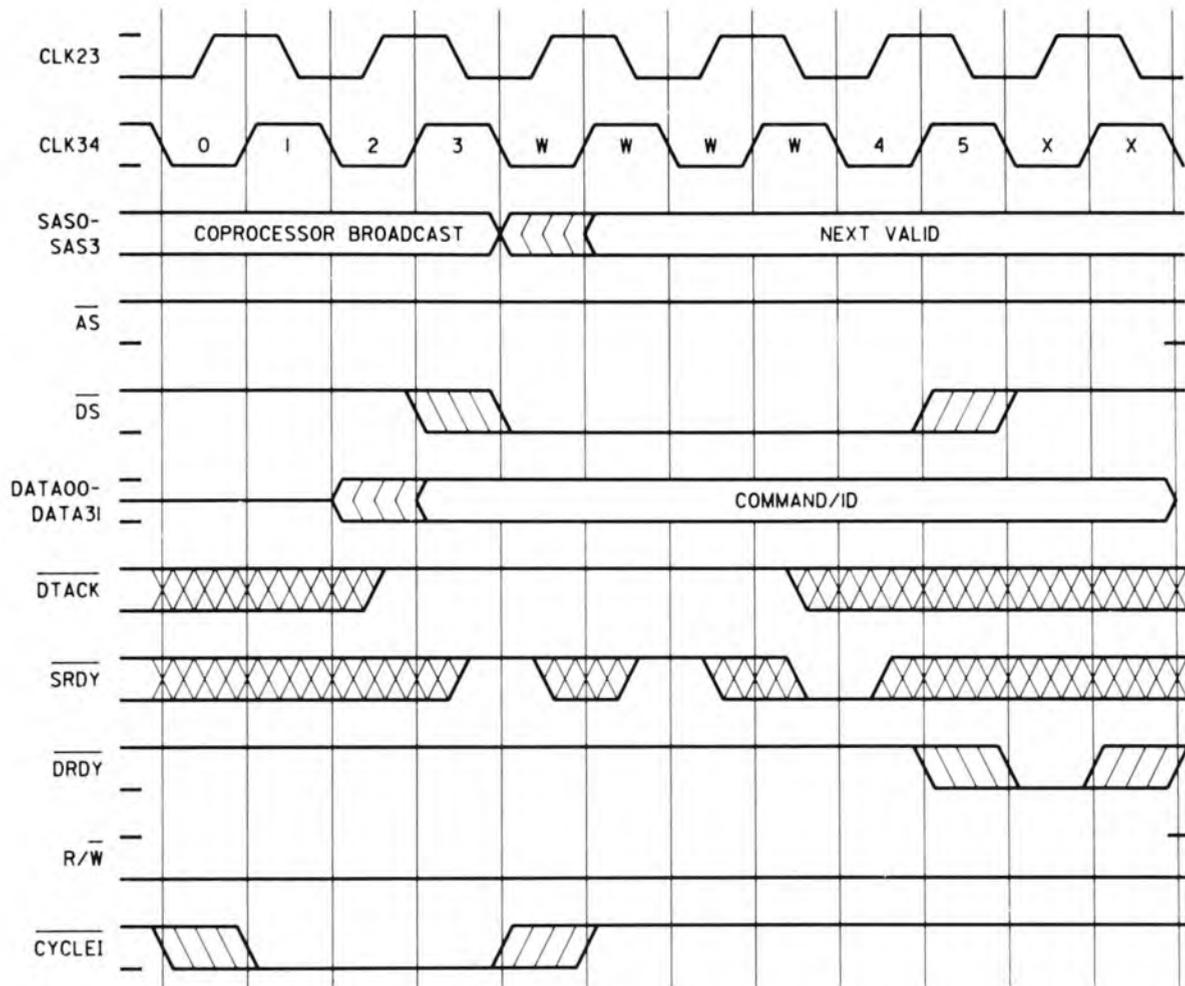


**B. One Wait Cycle**

**Notes:**

1. Zero, one, and two wait cycles using  $\overline{\text{SRDY}}$ .
2. Greater than two wait cycles causes internal CPU memory fault.

**Figure 4-33. Coprocessor Command and ID Transfer**  
**(Sheet 2 of 3)**



C. Two Wait Cycles

Notes:

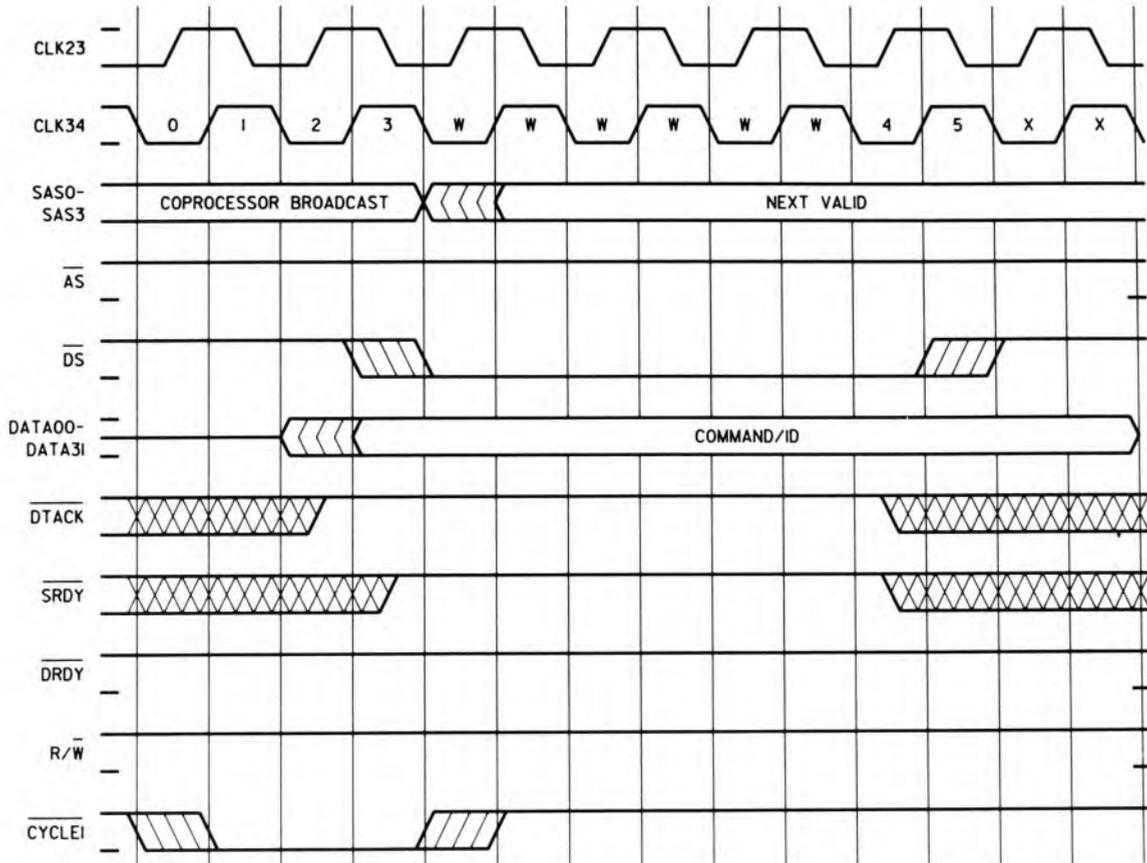
1. Zero, one, and two wait cycles using  $\overline{\text{SRDY}}$ .
2. Greater than two wait cycles causes internal CPU memory fault.

**Figure 4-33. 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 be done in software if the hardware is not in the system. The fault takes the CPU to the fault handler which then proceeds to the software version of the coprocessor function. Figure 4-34 depicts a case where 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.



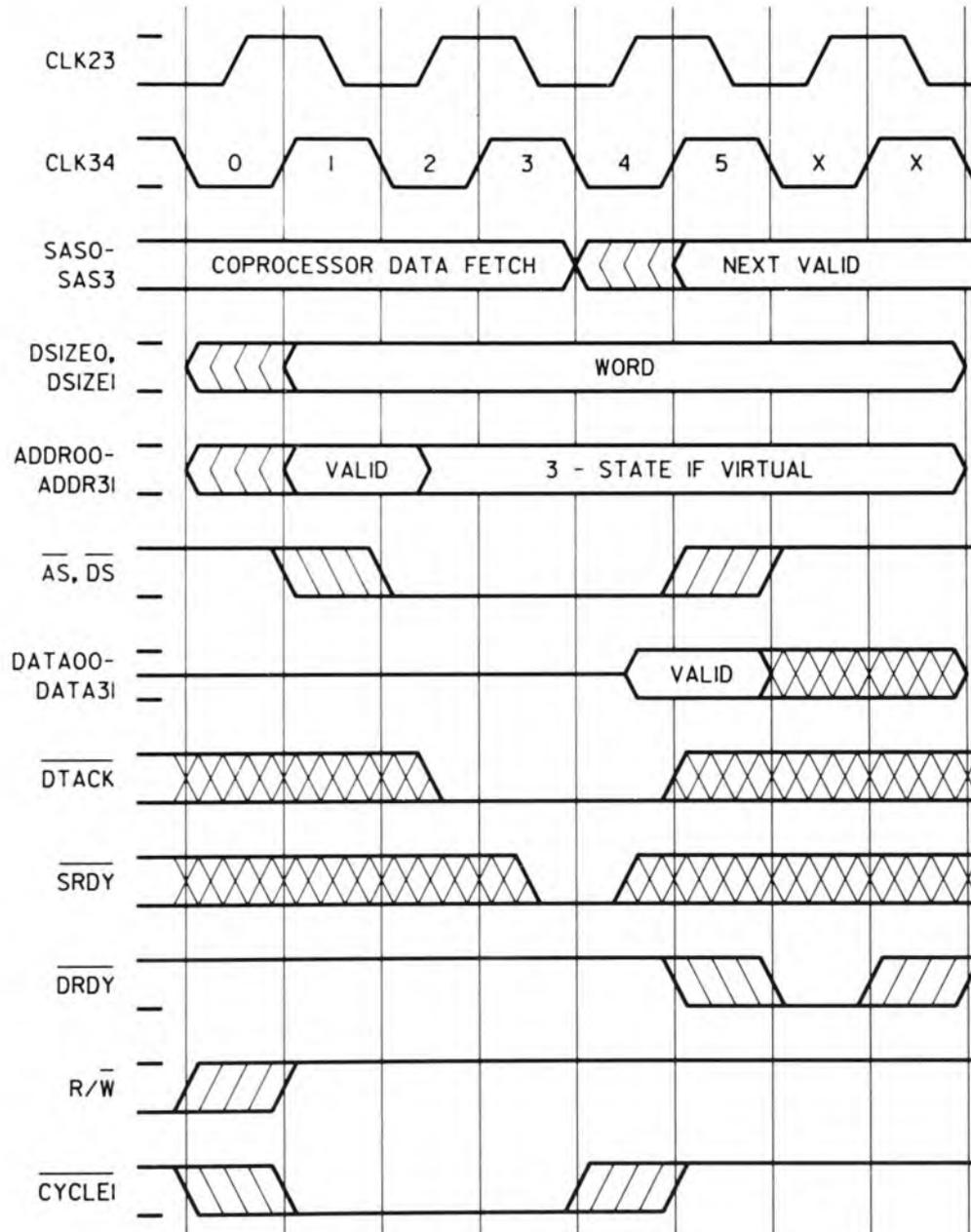
#### Notes:

1.  $\overline{RRREQ} = 1$ ,  $\overline{RETRY} = 1$ ,  $\overline{FAULT} = 1$ .
2. No acknowledge ( $\overline{DTACK}$ ,  $\overline{SRDY}$ ) and no bus exceptions ( $\overline{RETRY}$ ,  $\overline{RRREQ}$ ,  $\overline{FAULT}$ ).

Figure 4-34. Coprocessor Command and ID transfer (No Coprocessor Present)

**4.12.2 Coprocessor Operand Fetch**

After doing a broadcast, the CPU will perform 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 as it sees it on the bus. The SAS is "coprocessor data fetch" and DSIZE is a word. The memory issues the acknowledge for this transaction. Figure 4-35 shows the protocol for a single coprocessor operand fetch.



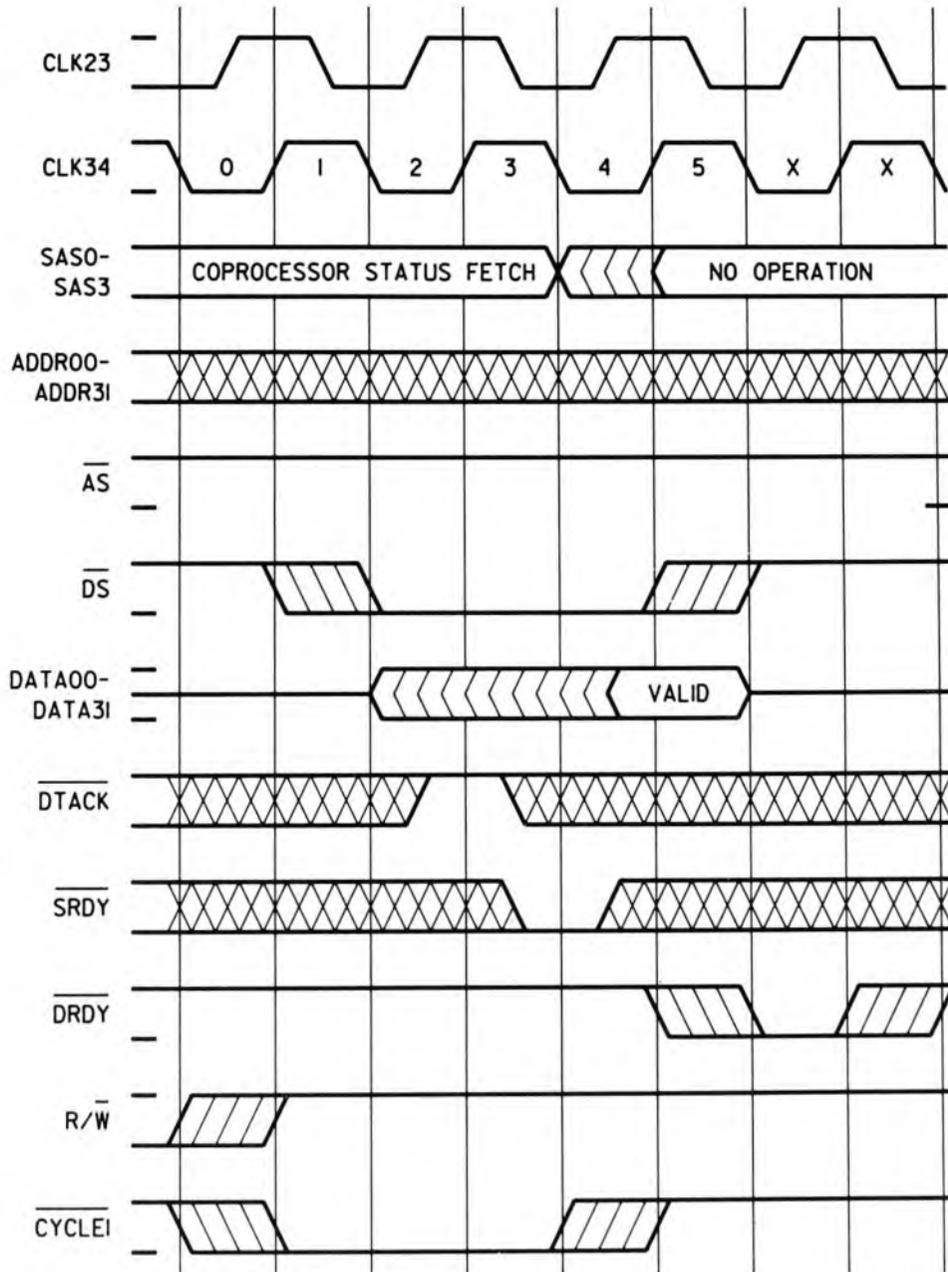
Note: Zero wait cycles use of  $\overline{DTACK}$  or  $\overline{SRDY}$

**Figure 4-35. Coprocessor Operand Fetch**

**BUS OPERATION**  
**Coprocessor Status Fetch**

**4.12.3 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 input ( $\overline{DONE}$ ) of the CPU. Approximately two clock cycles later, the CPU initiates the coprocessor status fetch transaction shown on Figure 4-36. This is a read type transaction where the coprocessor drives the data bus with status information. There is no  $\overline{AS}$  issued to keep the memory from being accessed. The SAS code is "coprocessor status fetch".

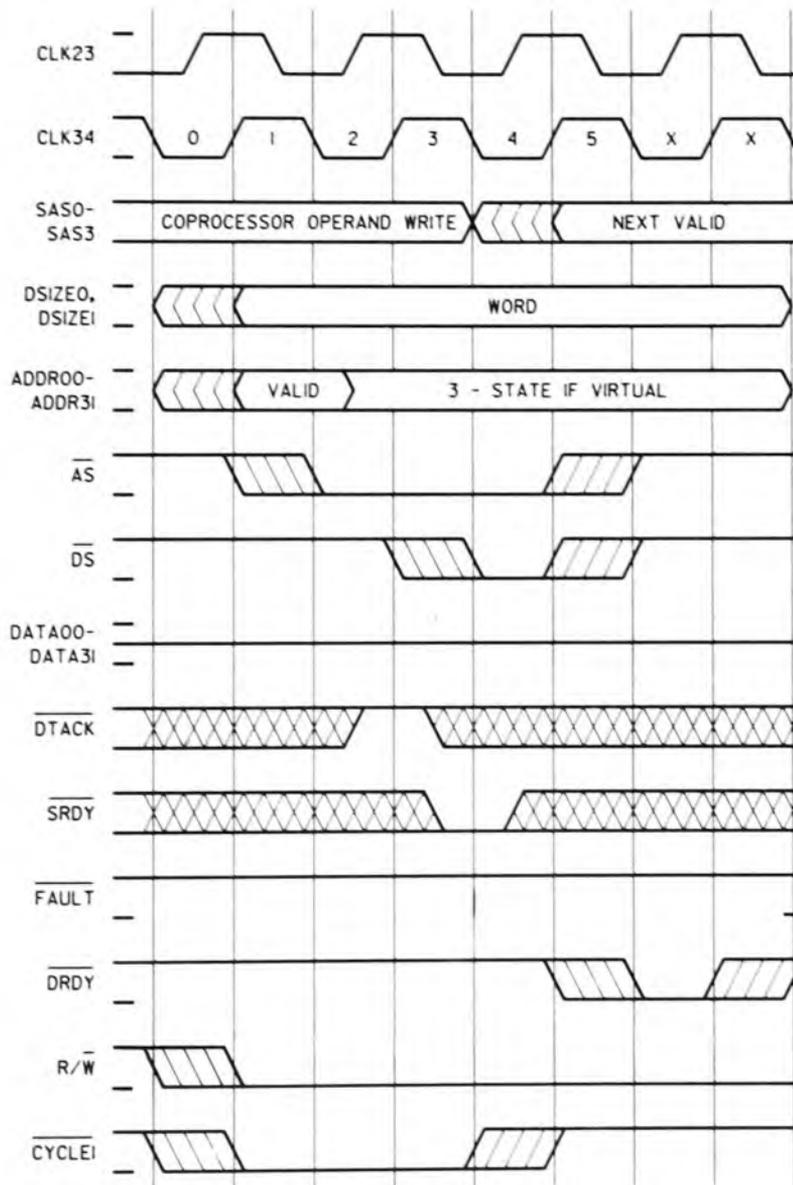


Note: Coprocessor status fetch begins approximately two clock cycles after the microprocessor's processor done input ( $\overline{DONE}$ ) has been driven low.

**Figure 4-36. Coprocessor Status Fetch Using  $\overline{SRDY}$**

**4.12.4 Coprocessor Data Write**

After doing the coprocessor status fetch, the CPU will perform from zero to three coprocessor data write transactions, depending on what coprocessor instruction is being executed. For this transaction the CPU goes through the motions of doing a write to the memory but the coprocessor drives the data bus with the results that it wants to send to the memory. The CPU does not drive the data bus during this transaction. The SAS is "coprocessor data write", and DSIZE is a word. The memory issues the acknowledge for this transaction. Figure 4-37 shows the protocol for a single coprocessor data write.



**Notes:**

1. Zero wait cycles using  $\overline{\text{SRDY}}$ .
2. DATA00-DATA31 supplied by coprocessor.

**Figure 4-37. Coprocessor Data Write**

## **BUS OPERATION**

### **Supplementary Protocol Diagrams**

#### **4.13 SUPPLEMENTARY PROTOCOL DIAGRAMS**

The following supplementary protocol diagrams are provided:

Figure 4-38. Read Transaction Followed by a Read Transaction.

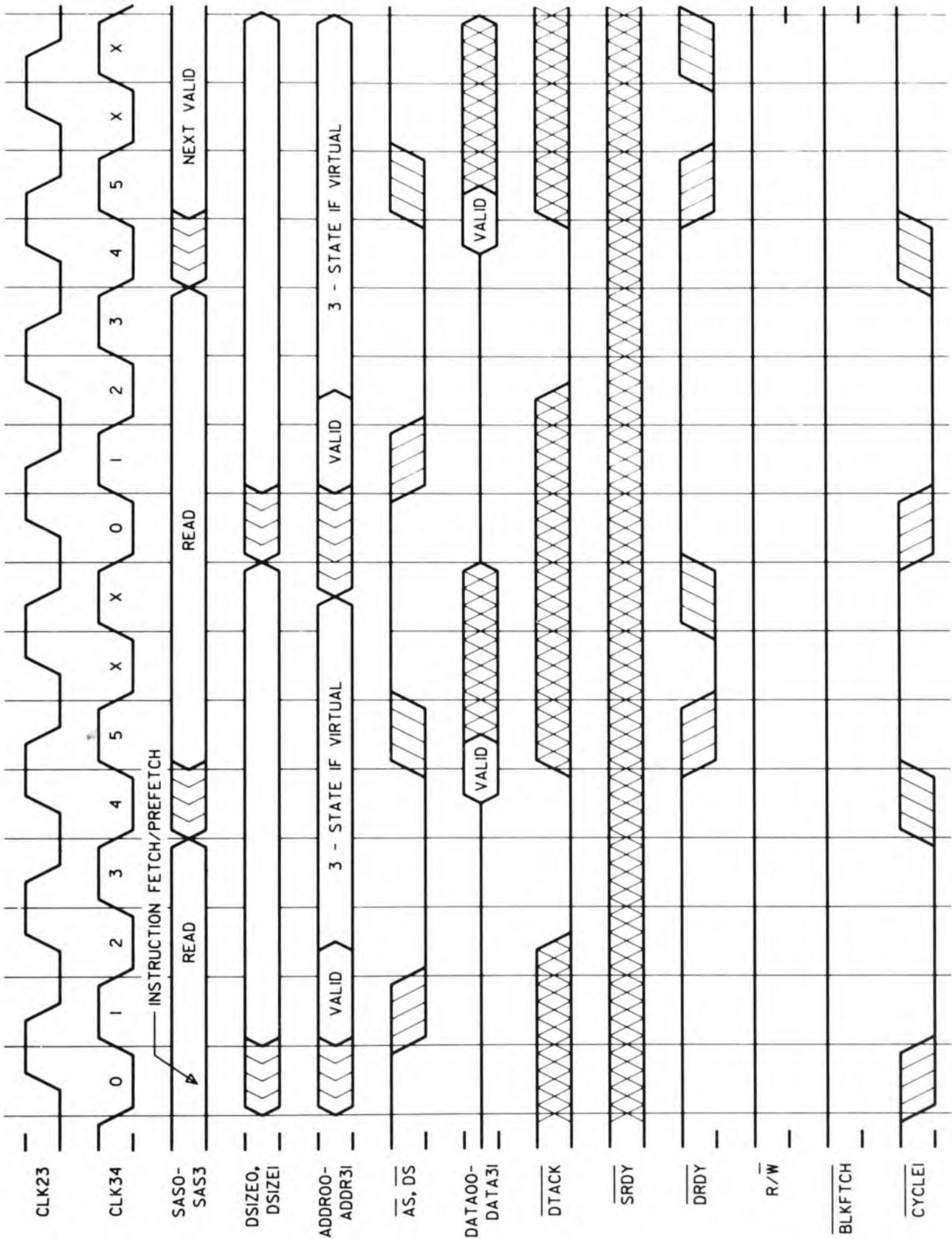
Figure 4-39. Read Transaction Followed by a Write Transaction Using  $\overline{\text{DTACK}}$

Figure 4-40. Write Transaction Followed by a Write Transaction.

Figure 4-41. Write Transaction Followed by a Read Transaction.

Figure 4-42. Double-Word Program Fetch Without Blockfetch Transaction Using  $\overline{\text{DTACK}}$ .

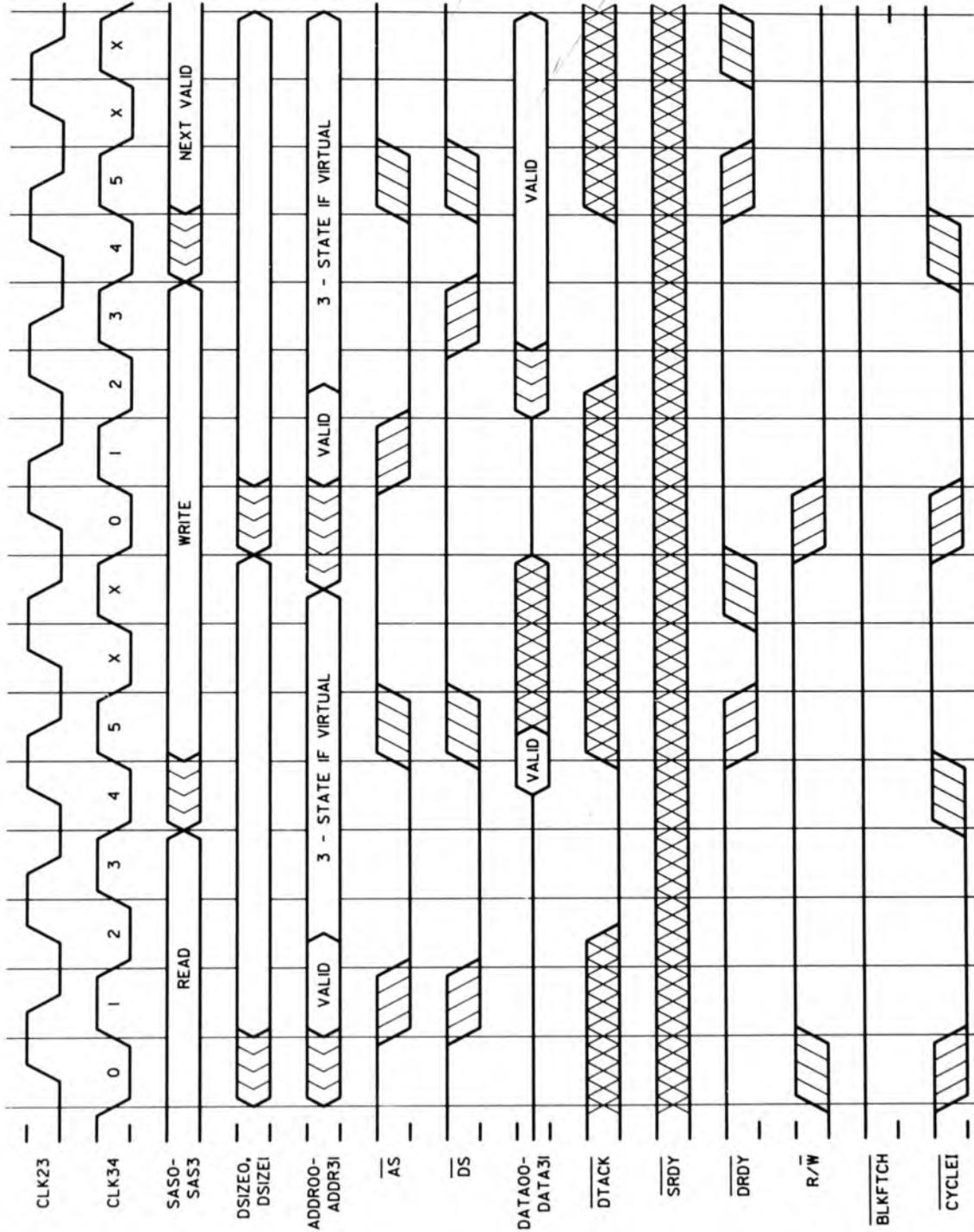
Figure 4-43. Bus Arbitration During Relinquish and Retry.



Note: Zero wait cycles.

Figure 4-38. Read Transaction Followed by a Read Transaction

**BUS OPERATION**  
**Supplementary Protocol Diagrams**

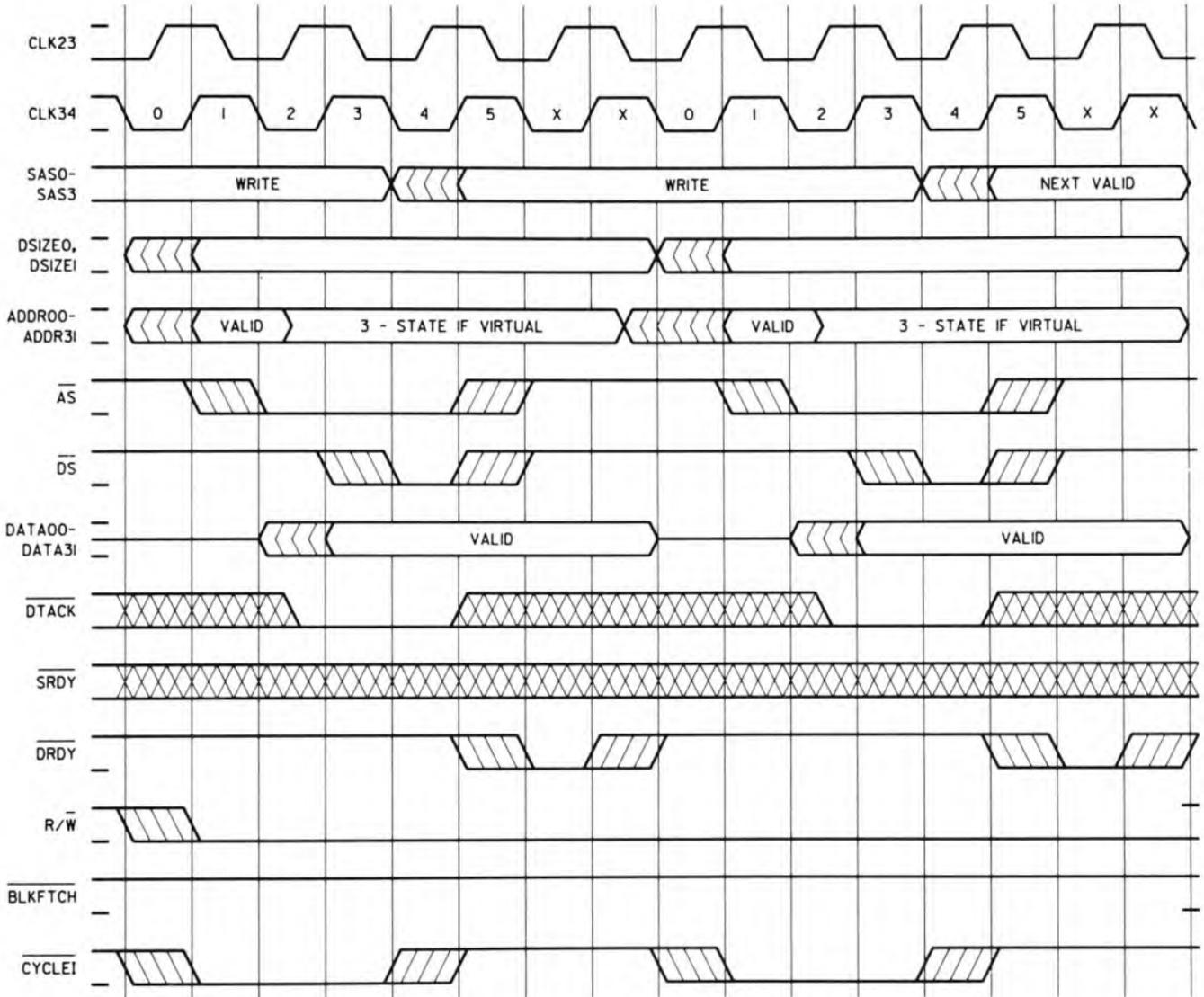


Note: Zero wait cycles.

**Figure 4-39. Read Transaction Followed by a Write Transaction Using DTACK**

# BUS OPERATION

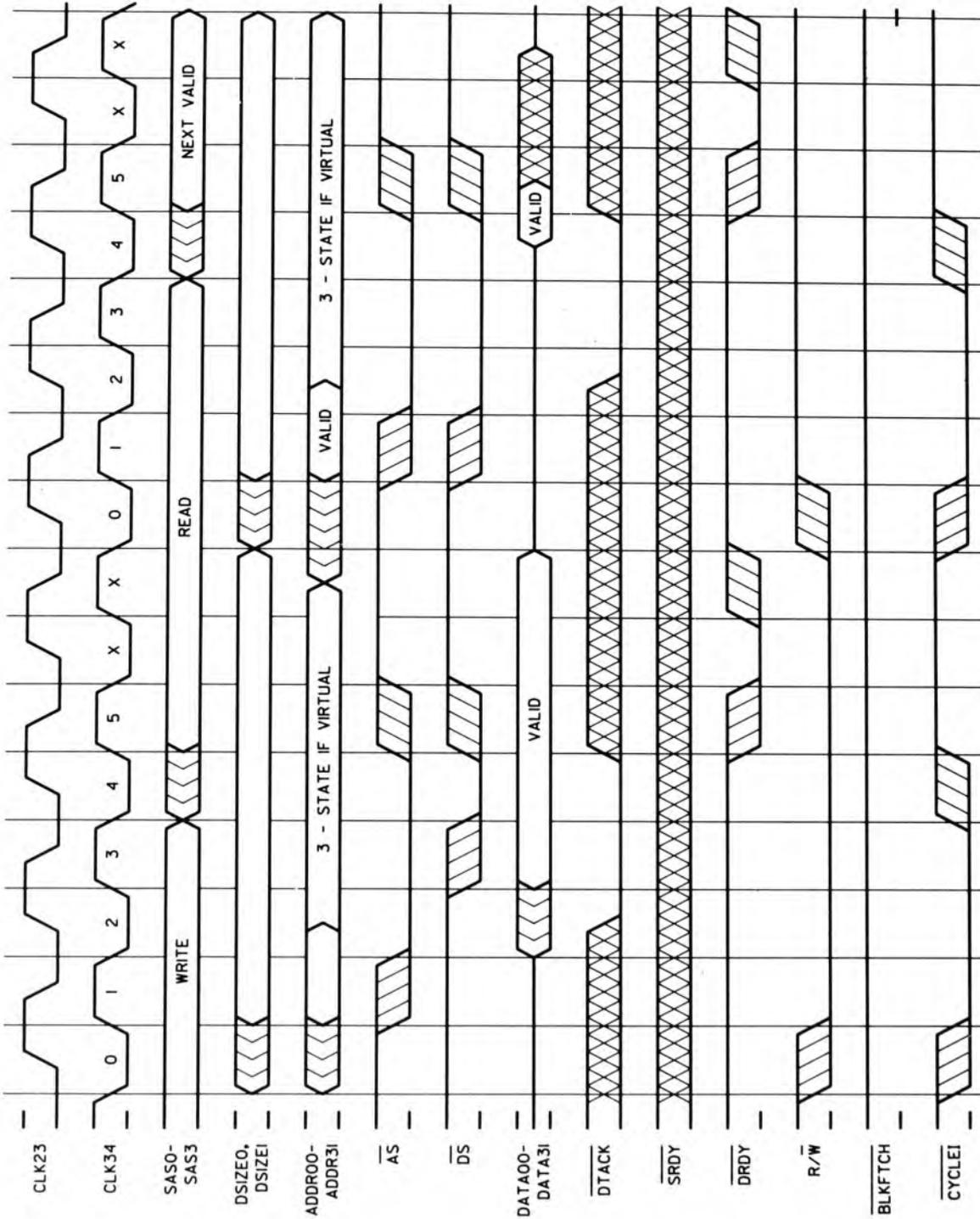
## Supplementary Protocol Diagrams



Note: Zero wait cycles.

**Figure 4-40. Write Transaction Followed by a Write Transaction**

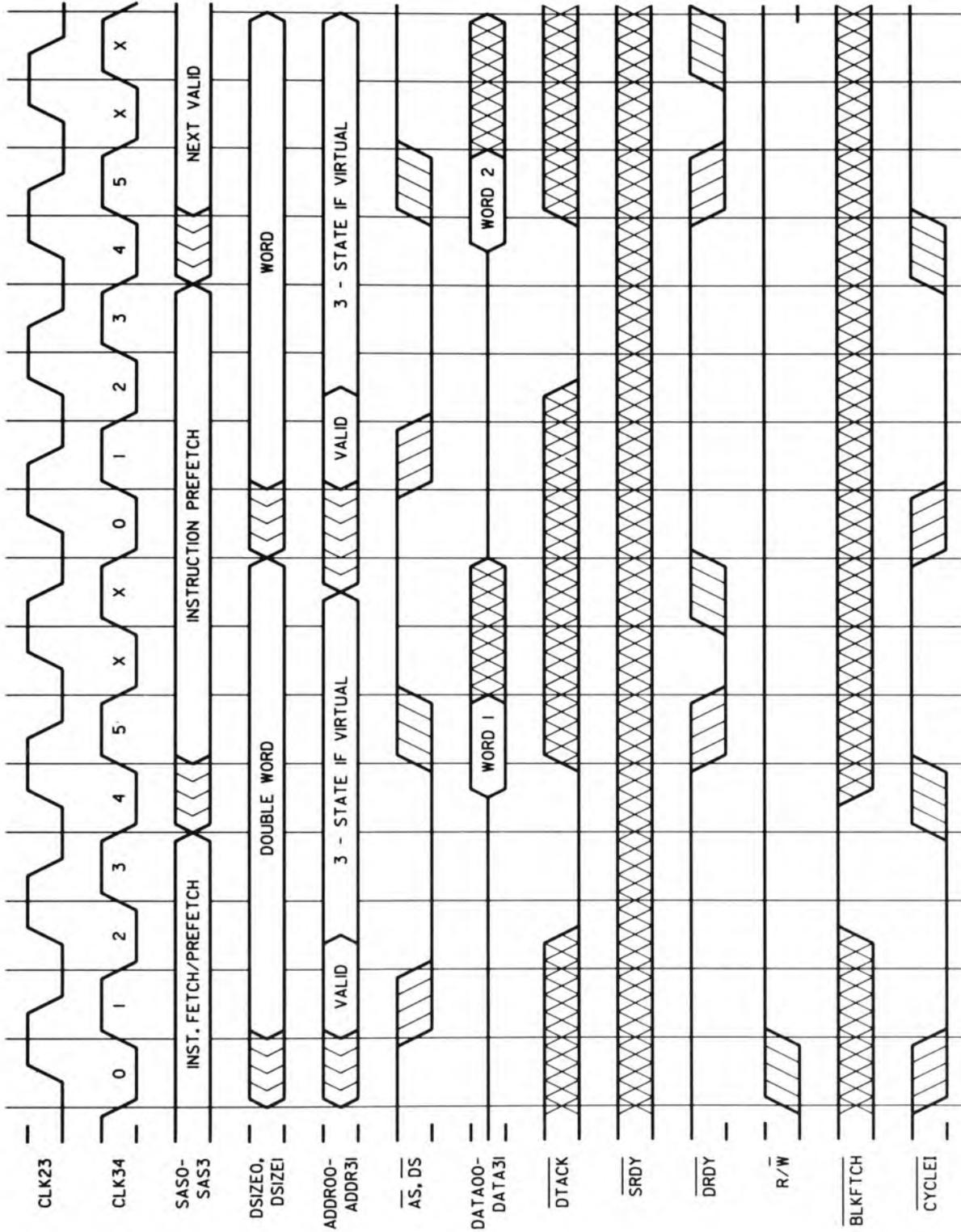
**BUS OPERATION**  
**Supplementary Protocol Diagrams**



Note: Zero wait cycles.

Figure 4-41. Write Transaction Followed by a Read Transaction

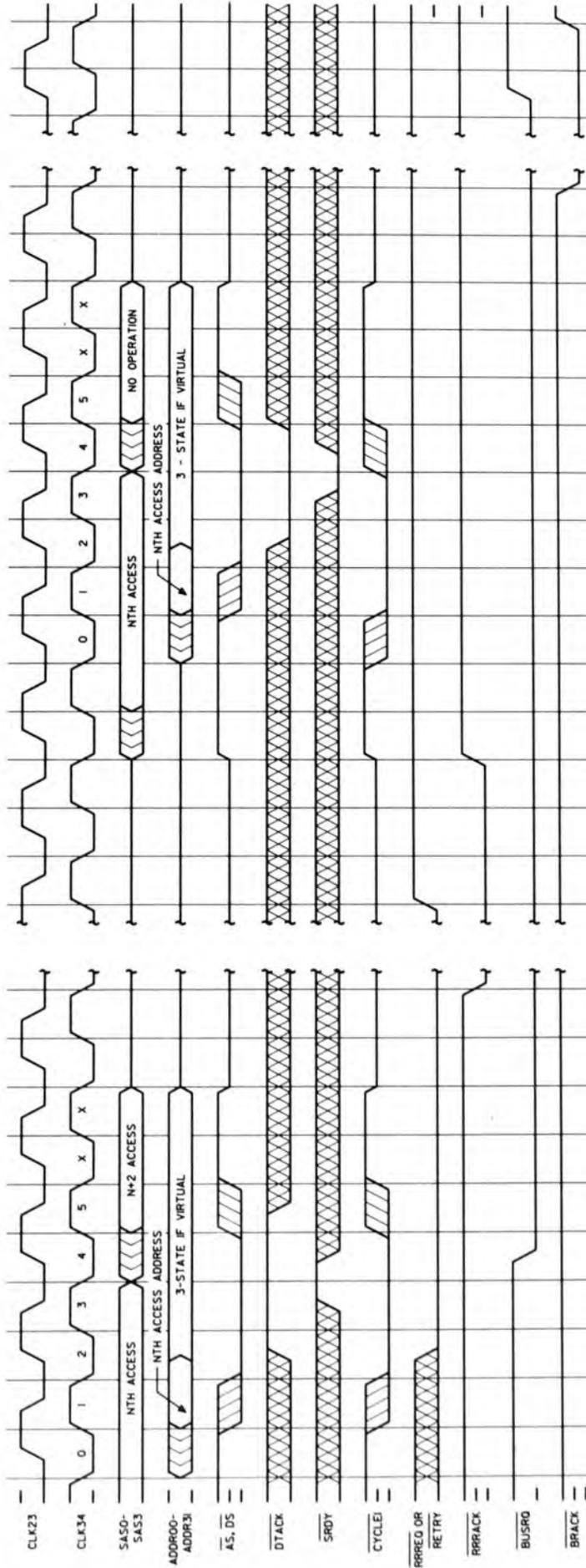
**BUS OPERATION**  
**Supplementary Protocol Diagrams**



Note: Zero wait cycles.

Figure 4-42. Double-Word Program Fetch Without Blockfetch Transaction Using DTACK

**BUS OPERATION**  
**Supplementary Protocol Diagrams**



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 4-43. Bus Arbitration During Relinquish and Retry**

## **Chapter 5**

# **Instruction Set and Addressing Modes**

## CHAPTER 5. INSTRUCTION SET AND ADDRESSING MODES

### CONTENTS

5. INSTRUCTIONSET AND ADDRESSING MODES .....	5-1
5.1 REGISTERS .....	5-1
5.2 ADDRESSING MODES.....	5-3
5.2.1 Register Mode.....	5-8
5.2.2 Register Deferred Mode.....	5-9
5.2.3 Displacement Mode .....	5-10
5.2.4 Deferred Displacement Mode.....	5-11
5.2.5 Immediate Mode.....	5-12
5.2.6 Absolute Mode.....	5-13
5.2.7 Absolute Deferred Mode .....	5-13
5.2.8 Expanded Operand Mode.....	5-14
5.3 FUNCTIONAL GROUPS.....	5-16
5.3.1 Instruction Byte and Cycle Considerations.....	5-16
5.3.2 Data Transfer Instructions .....	5-17
5.3.3 Arithmetic Instructions.....	5-19
5.3.4 Logical Instructions .....	5-20
5.3.5 Program Control Instructions.....	5-22
5.3.6 Coprocessor Instructions.....	5-27
5.3.7 Stack and Miscellaneous Instructions .....	5-27
5.4 INSTRUCTION SET LISTINGS .....	5-29
5.4.1 Notation .....	5-29
5.4.2 Instruction Set Summary by Mnemonic.....	5-32
5.4.3 Instruction Set Summary by Opcode.....	5-36
5.4.4 Instruction Set Descriptions .....	5-40
Add (ADDB2, ADDH2, ADDW2) .....	5-41
Add, 3 Address (ADDB3, ADDH3, ADDW3) .....	5-42
Arithmetic Left Shift (ALSW3) .....	5-43
And (ANDB2, ANDH2, ANDW2) .....	5-44
And, 3 Address (ANDB3, ANDH3, ANDW3) .....	5-45
Arithmetic Right Shift (ARSB3, ARSH3, ARSW3) .....	5-46
Branch on Carry Clear (BCCB, BCCH) .....	5-47
Branch on Carry Set (BCSB, BCSH) .....	5-48
Branch on Equal (BEB, BEH) .....	5-49
Branch on Greater Than (Signed) (BGB, BGH) .....	5-50
Branch on Greater Than or Equal (Signed) (BGEG, BGEH).....	5-51
Branch on Greater Than or Equal (Unsigned) (BGEUB, BGEUH).....	5-52
Branch on Greater Than (Unsigned) (BGUB, BGUH) .....	5-53
Bit Test (BITB, BITH, BITW) .....	5-54
Branch on Less Than (Signed) (BLB, BLH).....	5-55
Branch on Less Than or Equal (Signed) (BLEB, BLEH) .....	5-56
Branch on Less Than or Equal (Unsigned) (BLEUB, BLEUH).....	5-57
Branch on Less Than (Unsigned) (BLUB, BLUH) .....	5-58
Branch on Not Equal (BNEB, BNEH) .....	5-59

Breakpoint Trap (BPT) .....	5-60
Branch (BRB, BRH) .....	5-61
Branch to Subroutine (BSBB, BSBH) .....	5-62
Branch on Overflow Clear (BVCB, BVCH) .....	5-63
Branch on Overflow Set (BVSB, BVSH) .....	5-64
Call Procedure (CALL) .....	5-65
Cache Flush (CFLUSH) .....	5-66
Clear (CLRB, CLRH, CLRW) .....	5-67
Compare (CMPB, CMPH, CMPW) .....	5-68
Decrement (DECB, DECH, DECW) .....	5-69
Divide (DIVB2, DIVH2, DIVW2) .....	5-70
Divide, 3 Address (DIVB3, DIVH3, DIVW3) .....	5-71
Extract Field (EXTFB, EXTFH, EXTFW) .....	5-72
Extended Opcode (EXTOP) .....	5-73
Increment (INCB, INCH, INCW) .....	5-74
Insert Field (INSFB, INSFH, INSFW) .....	5-75
Jump (JMP) .....	5-76
Jump to Subroutine (JSB) .....	5-77
Logical Left Shift (LLSB3, LLSH3, LLSW3) .....	5-78
Logical Right Shift (LRSW3) .....	5-79
Move Complemented (MCOMB, MCOMH, MCOMW) .....	5-80
Move Negated (MNEGB, MNEGH, MNEGW) .....	5-81
Modulo (MODB2, MODH2, MODW2) .....	5-82
Modulo, 3 Address (MODB3, MODH3, MODW3) .....	5-83
Move (MOVB, MOVH, MOVW) .....	5-84
Move Address (Word) (MOVAW) .....	5-85
Move Block (MOVBLW) .....	5-87
Multiply (MULB2, MULH2, MULW2) .....	5-89
Multiply, 3 Address (MULB3, MULH3, MULW3) .....	5-90
Move Version Number (MVERNO) .....	5-91
No Operation (NOP, NOP2, NOP3) .....	5-92
OR (ORB2, ORH2, ORW2) .....	5-93
OR, 3 Address (ORB3, ORH3, ORW3) .....	5-94
Pop (Word) (POPW) .....	5-95
Push Address (Word) (PUSHAW) .....	5-96
Push (Word) (PUSHW) .....	5-97
Return on Carry Clear (RCC) .....	5-98
Return on Carry Set (RCS) .....	5-99
Return on Equal (REQL, REQLU) .....	5-100
Restore Registers (RESTORE) .....	5-101
Return from Procedure (RET) .....	5-102
Return on Greater Than or Equal (Signed) (RGEQ) .....	5-103
Return on Greater Than or Equal (Unsigned) (RGEQU) .....	5-104
Return on Greater Than (Signed) (RETR) .....	5-105
Return on Greater Than (Unsigned) (RGTRU) .....	5-106
Return on Less Than or Equal (Signed) (RLEQ) .....	5-107

Return on Less Than or Equal (Unsigned) (RLEQU) .....	5-108
Return on Less Than (Signed) (RLSS) .....	5-109
Return on Less Than (Unsigned) (RLSSU) .....	5-110
Return on Not Equal (RNEQ, RNEQU) .....	5-111
Rotate (ROTW) .....	5-112
Return from Subroutine (RSB) .....	5-113
Return on Overflow Clear (RVC) .....	5-114
Return on Overflow Set (RVS) .....	5-115
Save Registers (SAVE) .....	5-116
Coprocessor Operation (No Operands) (SPOP) .....	5-117
Coprocessor Operation Read (SPOPRS, SPOPRD, SPOPRT) .....	5-118
Coprocessor Operation, 2 Address (SPOPS2, SPOPD2, SPOPT2) .....	5-119
Coprocessor Operation Write (SPOPWS, SPOPWD, SPOPWT) .....	5-120
String Copy (STRCPY) .....	5-121
String End (STREND) .....	5-123
Subtract (SUBB2, SUBH2, SUBW2) .....	5-124
Subtract, 3 Address (SUBB3, SUBH3, SUBW3) .....	5-125
Swap (Interlocked) (SWAPBI, SWAPHI, SWAPWI) .....	5-126
Test (TSTB, TSTH, TSTW) .....	5-127
Exclusive Or (XORB2, XORH2, XORW2) .....	5-128
Exclusive Or, 3 Address (XORB3, XORH3, XORW3) .....	5-129

## 5. INSTRUCTION SET AND ADDRESSING MODES

The WE 32100 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 discussed in **Chapter 6. Operating System Considerations.**

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 **2.5 Data Types.**

### 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 address modes, other than absolute, immediate, or literal, reference a processor register. In general, any of the sixteen processor registers may be used as an operand in all of the address modes. Table 5-1 lists the registers and assigned functions.

General-purpose registers r0 through r8 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)

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.

## INSTRUCTION SET AND ADDRESSING MODES

### Registers

Register	Name	Assembler Syntax	Assigned Function
0	r0	%r0	General-purpose <sup>1</sup>
1	r1	%r1	General-purpose <sup>1</sup>
2	r2	%r2	General-purpose <sup>1</sup>
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 <sup>2,3</sup>
12	SP	%sp or %r12	Stack pointer
13	PCBP	%pcbp or %r13	Processor control block pointer <sup>2</sup>
14	ISP	%isp or %r14	Interrupt stack pointer <sup>2</sup>
15	PC	%pc or %r15	Program counter <sup>3</sup>

<sup>1</sup> Block or string instructions may use this register as an implied argument for indexing or addressing. Operating system instructions also use this register.

<sup>2</sup> Privileged register. Writing to this register when the processor is not in kernel execution level causes a privileged-register exception (see **6.2.1 Execution Privilege**).

<sup>3</sup> Registers 11 and 15 may not be used in some address modes (see **5.2 Addressing Modes**).

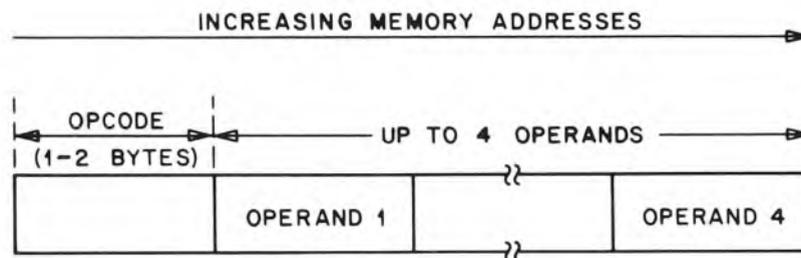
Some of the registers have restrictions on usage in instructions. Because registers 11, 13, and 14 (r11, r13, and r14) are privileged, these may be written only when kernel execution level is in effect. Register 11, the processor status word (PSW), contains status information about the current instruction 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.

The last register is the program counter, register 15 (r15). This register and register 11 may not be referenced in some address modes (see **5.2 Addressing Modes**). In addition, it is referenced implicitly in all program-control instructions and for all function calls and returns.

## 5.2 ADDRESSING MODES

An assembly language instruction for the *WE 32100* microprocessor consists of a mnemonic, such as *ADDW*, *MOVH*, *INCB*, followed by up to four operands. Each operand is physically located in either one of the 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.

# INSTRUCTION SET AND ADDRESSING MODES

## Addressing Modes

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 **mmm**, 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. Table 5-2 lists all mode field values (0–15) and their corresponding addressing modes.

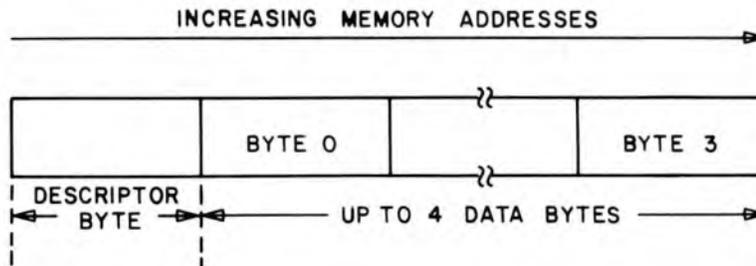


Figure 5-2. Operand Format

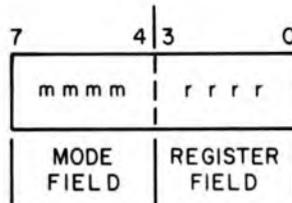


Figure 5-3. Descriptor Byte Format

# INSTRUCTION SET AND ADDRESSING MODES

## Addressing Modes

<b>Table 5-2. Addressing Modes</b>		
<b>Mode Field Value</b>	<b>Addressing Mode</b>	<b>Description</b>
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 becomes the word immediate mode.
5	Register Deferred	The register specified in the register field contains the operand's address. If register 15 is specified in the register field, this 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's 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 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's 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's 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's 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's address.
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's address.

## INSTRUCTION SET AND ADDRESSING MODES

### Addressing Modes

Mode Field Value	Addressing Mode	Description
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's 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's 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 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 *OXnnn*
- 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  $(+2^{31}-1)$
- 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: *sbyte* (for signed byte), *byte* or *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 **5.2.8 Expanded-Operand Type** for more details.

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

Notes: Mode field has special meaning if register field is 15; see absolute or immediate mode.

Mode may not be used for a destination operand.

Mode may not be used if the instruction takes effective address of the operand.

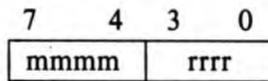
*type* overrides instruction type; *type* determines the operand type, except that it does not determine the length for immediates or literals or whether literals are signed or unsigned. *opnd* determines actual address mode. For total bytes, add 1 to byte count for address mode determined by *opnd*.

## INSTRUCTION SET AND ADDRESSING MODES

### Register Mode

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.

### 5.2.1 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 (%).

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

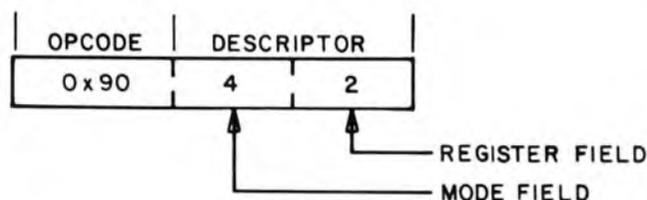
The general syntax, mode, and register fields used to signify the register addressing mode are:

*syntax:* %rn where n is a register number  
m m m m: 4  
r r r r: 0 to 14

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

## INSTRUCTION SET AND ADDRESSING MODES

### Register Deferred Mode



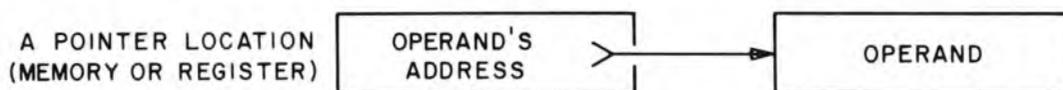
**Figure 5-4. Register Mode Example**

### 5.2.2 Register Deferred Mode

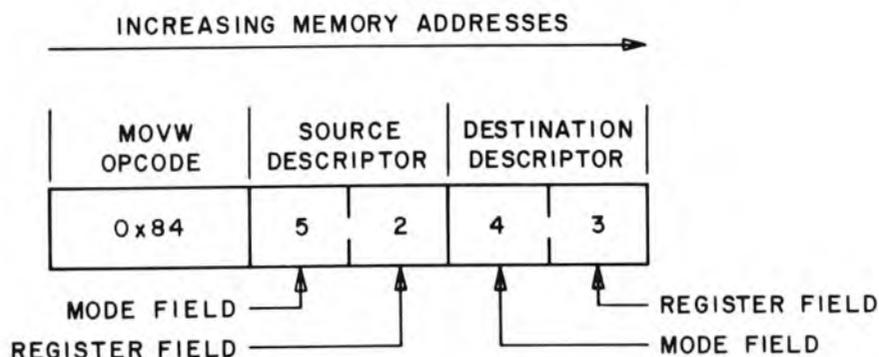
Deferred addressing mode involves indirect addressing using pointers. A pointer is either a register or memory location containing an address. Figure 5-5 illustrates the relationship between the address contained in a pointer and the operand ultimately obtained. The term *deferred* is used to describe 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 will be 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**



**Figure 5-6. Register Deferred Mode Example**

## INSTRUCTION SET AND ADDRESSING MODES

### Displacement Mode

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.

### 5.2.3 Displacement Mode

The displacement mode forms an operand's address by adding an offset to the contents of a *WE 32100* Microprocessor register. For example, the instruction `MOVB 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 `MOVB` instruction.

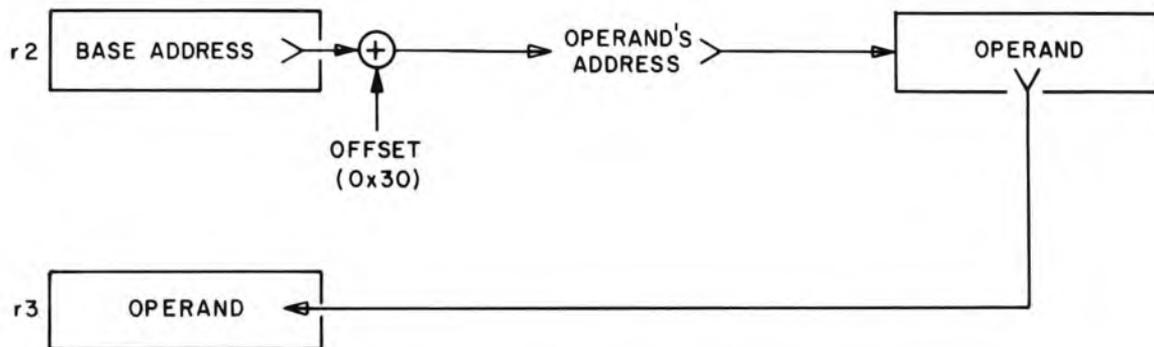


Figure 5-7. Example of `MOV 0x30(%r2),%r3`

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

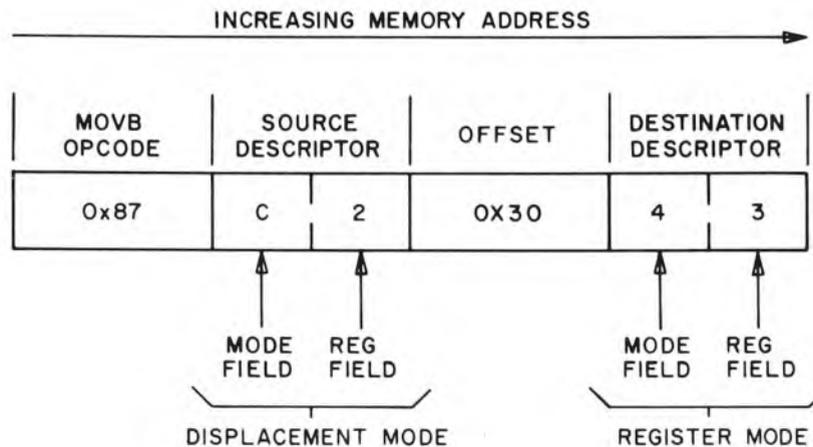
Using the appropriate mode and register fields, the instruction `MOVB 0x30(%r2),%r3` is stored in memory as shown on Figure 5-8.

The offset used in the displacement mode may be either a byte (8-bits), halfword (16-bits), or word (32-bits), or an expression yielding such a value. Two's complement, negative offsets are also valid. Negative byte and halfword offsets are first 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.

## INSTRUCTION SET AND ADDRESSING MODES

### Deferred Displacement Mode



**Figure 5-8. A Displacement Mode Source Operand**



**Figure 5-9. Deferred Displacement Addressing**

#### 5.2.4 Deferred Displacement Mode

The deferred displacement mode uses the contents of the address calculated in the displacement mode as a pointer which 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.

## INSTRUCTION SET AND ADDRESSING MODES

### Immediate Mode

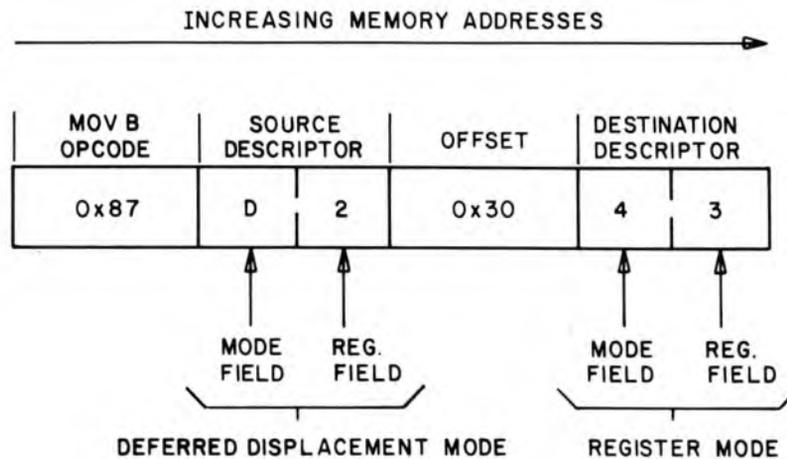


Figure 5-10. A Deferred Displacement Mode Source Operand

### 5.2.5 Immediate Mode

In the immediate addressing mode, the operand is contained within the instruction. The ampersand symbol 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 `&` symbol signifies that the data immediately following is to be treated as immediate data. The `%` symbol, 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

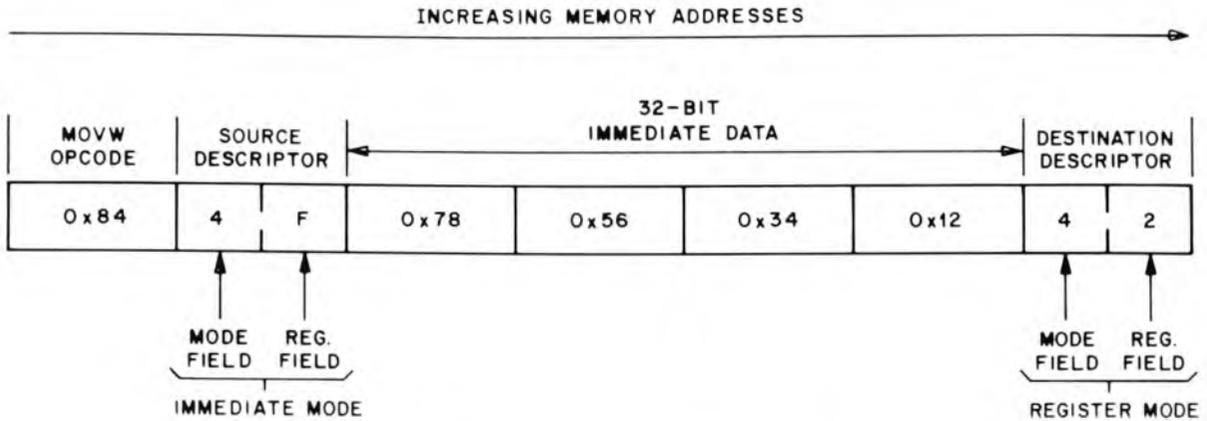
A mode field of 4 indicates that the immediate data is 32-bits long, while mode fields of 5 and 6 are used for 16-bit and 8-bit immediate data, respectively. Figure 5-11 illustrates the storage of the instruction `MOVW &0x12345678,%r2` in memory. This instruction causes the immediate data, 0x12345678,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

## INSTRUCTION SET AND ADDRESSING MODES

### Absolute Deferred Mode



**Figure 5-11. A 32-Bit Immediate Source Operand**

### 5.2.6 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)
mmm: 7
rrr: 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.

### 5.2.7 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:

```

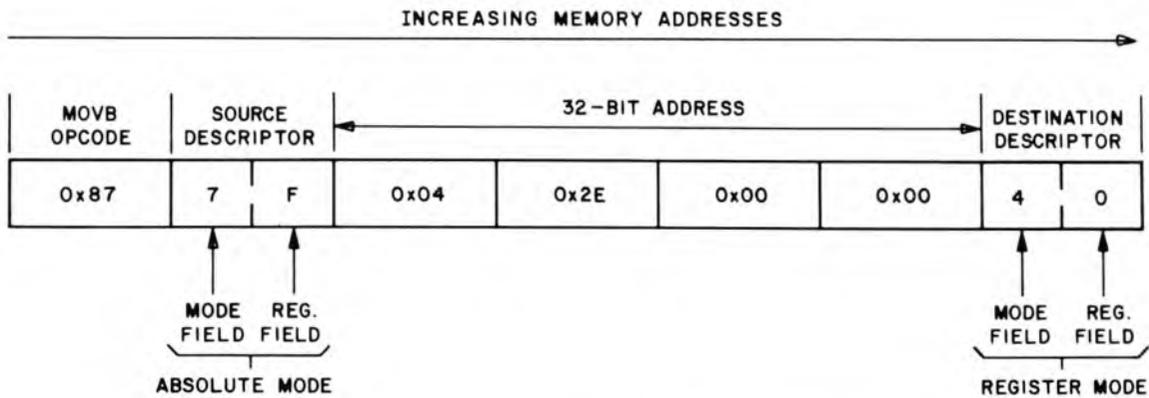
syntax: *$exp  (exp must evaluate to a byte, halfword, or word)
mmm: 14
rrr: 15

```

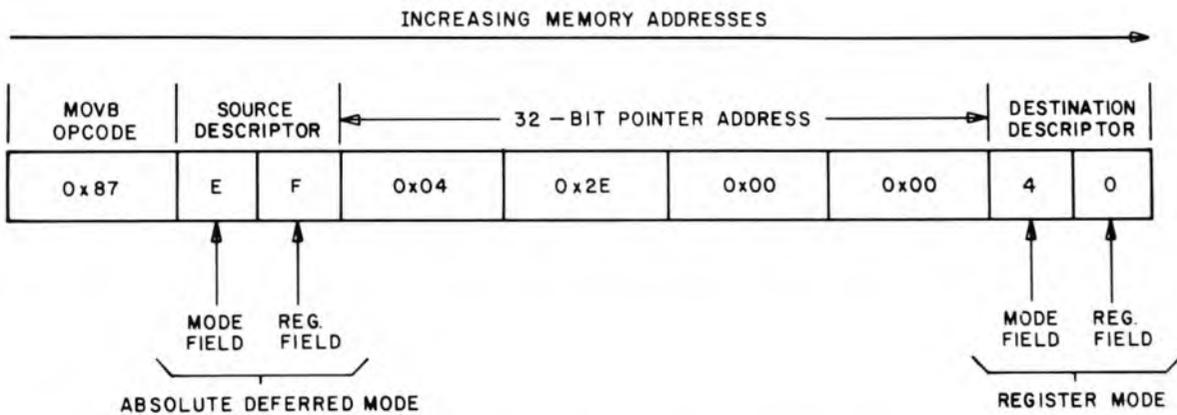
Thus, the instruction `MOVB *$0x2E04,%r0` is stored in memory as illustrated on Figure 5-13.

# 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**

### 5.2.8 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*.

## INSTRUCTION SET AND ADDRESSING MODES

### Expanded Operand Mode

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**

The first operand's real mode is register, the second operand 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.

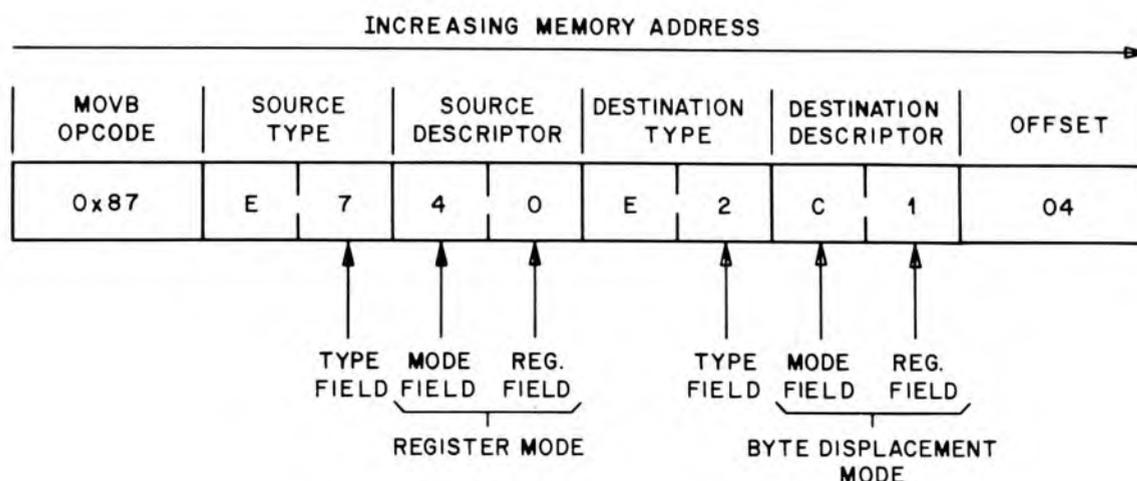
The expanded operand mode is illegal with coprocessor instructions CALL, SAVE, RESTORE, SWAP INTERLOCKED, PUSHW, PUSHAW, POPW, and JSB instructions and will generate an illegal operand fault.

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

Note: Type fields E1, E5, E8–E14 are reserved data types.  
Type field EF is an absolute deferred data type.

# INSTRUCTION SET AND ADDRESSING MODES

## Functional Groups



**Figure 5-15. Expanded Operand Mode Example**

## 5.3 FUNCTIONAL GROUPS

The *WE* 32100 Microprocessor instruction set may be separated into six functional groups: data transfer, arithmetic, logical, program control, coprocessor, and stack and miscellaneous instructions. This section 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 5-11.

### 5.3.1 Instruction Byte and Cycle Considerations

The architecture of the *WE* 32100 CPU makes exact instruction timing calculations difficult due to the following effects:

- Addressing modes of operands
- On-chip instruction cache
- Instruction pipelining
- Instruction and data alignment
- Data dependencies

The entries in the cycle count column in Tables 5-5 through 5-10 contain the ranges, from practical best to worst case, derived from tests taking all of the above effects into consideration. It is recommended that actual benchmarks be run to more accurately measure performance. The following discussion describes the timing differences due to the above effects.

**Addressing Modes of Operands.** Since the instruction set is orthogonal to the addressing modes of its operands, tests were done on combinations of the five basic addressing mode classes (register, absolute address, register deferred, immediate, and absolute deferred) for each instruction. In all cases, register operands take the least time, while the absolute deferred operands take the most time to access.

**On-Chip Instruction Cache.** Tests were run with all instruction cache hits (best case) and all instruction misses (worst case). The instruction cache improvements ranged from 20–60% for arithmetic and logical instructions, by eliminating instruction fetches.

**Instruction Pipelining.** Test instructions were placed between two others that had potential for overlapped execution in the CPU's pipeline (best case). They were also placed between two branch-taken instructions, to eliminate pipeline overlap (worst case). These tests showed that pipelining saved between 2 to 6 cycles per instruction.

**Instruction and Data Alignment.** In test runs taking this effect into account, performance increases of an average of 2 to 6 cycles were encountered for optimal alignment. Optimal alignment was obtained by placing as many of the test instruction's opcode and associated operands as possible on word boundaries. Worst case alignment minimizes alignment of the opcode and operands.

**Data Dependencies.** This effect was found only in five instructions: `MULW2`, `DIVW2`, `MOVBLW`, `STRCPY`, and `STREND`. In the test involving the `MULW2` and `DIVW2` instructions timing is optimized if at least one operand is zero or one. For the string instructions, the length of the string has a large impact on the instruction time. Since string lengths are not limited, test runs were done on strings of one byte (best case) and four bytes (worst case) to determine best and worst case timings.

### **5.3.2 Data Transfer Instructions**

These instructions (listed in Table 5-5) transfer data to and from 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 type mode changes an operand's type. The destination operand (*dst*) result determines how the condition flags are set.

**INSTRUCTION SET AND ADDRESSING MODES**  
**Data Transfer Instructions**

<b>Table 5-5. Data Transfer Instruction Group</b>					
<b>Instruction</b>	<b>Mnemonic</b>	<b>Opcode</b>	<b>Bytes</b>	<b>Cycles</b>	<b>Conditions*</b>
<b>Move:</b>					
Move byte	MOVB	0x87	3-11	2-31	Case 1
Move halfword	MOVH	0x86	3-11	2-31	
Move word	MOVW	0x84	3-11	1-27	
Move address (word)	MOVAW	0x04	3-11	2-22	
Move complemented byte	MCOMB	0x8B	3-11	2-31	
Move complemented halfword	MCOMH	0x8A	3-11	2-31	
Move complemented word	MCOMW	0x88	3-11	1-27	
Move negated byte	MNEGB	0x8F	3-11	2-31	Case 2
Move negated halfword	MNEGH	0x8E	3-11	2-31	
Move negated word	MNEGW	0x8C	3-11	1-27	
Move version number	MVERNO	0x3009	2	—	Unchanged
<b>Swap (Interlocked):</b>					
Swap byte interlocked	SWAPBI	0x1F	2-6	22-33	Case 1
Swap halfword interlocked	SWAPHI	0x1E	2-6	22-33	
Swap word interlocked	SWAPWI	0x1C	2-6	18-28	
<b>Block Operations:</b>					
Move block of words	MOVBLW	0x3019	2	—	Unchanged
<b>Field Operations:</b>					
Extract field byte	EXTFB	0xCF	5-21	7-55	Case 1
Extract field halfword	EXTFH	0xCE	5-21	7-55	
Extract field word	EXTFW	0xCC	5-21	4-54	
Insert field byte	INSFB	0xCB	5-21	18-72	
Insert field halfword	INSFH	0xCA	5-21	18-72	
Insert field word	INSFW	0xC8	5-21	14-71	
<b>String Operations:</b>					
String copy	STRCPY	0x3035	2	83-182**	Unchanged
String end	STREND	0x301F	2	54-120**	

\*Refer to Table 5-11 for condition flag code assignments.

\*\*Dependent on string length. See 5.3.1 Instruction Byte and Cycle Consideration.

### 5.3.3 Arithmetic Instructions

Arithmetic instructions (listed in Table 5-6) perform arithmetic operations on data in registers and memory. Most of these instructions have three types (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 expanded-operand type 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.

<b>Table 5-6. Arithmetic Instruction Group</b>						
Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*	
<b>Add:</b>						
Add byte	ADDB2	0x9F	3-11	4-33	Case 2	
Add halfword	ADDH2	0x9E	3-11	4-33		
Add word	ADDW2	0x9C	3-11	2-31		
Add byte, 3-address	ADDB3	0xDF	4-16	4-44		
Add halfword, 3-address	ADDH3	0xDE	4-16	4-44		
Add word, 3-address	ADDW3	0xDC	4-16	4-43		
<b>Subtract:</b>						
Subtract byte	SUBB2	0xBF	3-11	4-33		
Subtract halfword	SUBH2	0xBE	3-11	4-33		
Subtract word	SUBW2	0xBC	3-11	2-31		
Subtract byte, 3-address	SUBB3	0xFF	4-16	4-44		
Subtract halfword, 3-address	SUBH3	0xFE	4-16	4-43		
Subtract word, 3-address	SUBW3	0xFC	4-16	4-43		
<b>Increment:</b>						
Increment byte	INCB	0x93	2-6	2-24		
Increment halfword	INCH	0x92	2-6	2-24		
Increment word	INCW	0x90	2-6	1-22		
<b>Decrement:</b>						
Decrement byte	DECB	0x97	2-6	2-24		
Decrement halfword	DECH	0x96	2-6	2-24		
Decrement word	DECW	0x94	2-6	1-22		
<b>Multiply:</b>						
Multiply byte	MULB2	0xAB	3-11	20-91	Case 3	
Multiply halfword	MULH2	0xAA	3-11	20-130		
Multiply word	MULW2	0xA8	3-11	18-210		

\*Refer to Table 5-11 for condition flag code assignments.

# INSTRUCTION SET AND ADDRESSING MODES

## Logical Instructions

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
Multiply byte, 3-address	MULB3	0xEB	4-16	22-204	Case 4
Multiply halfword, 3-address	MULH3	0xEA	4-16	22-200	
Multiply word, 3-address	MULW3	0xE8	4-16	20-205	
<b>Divide:</b>					
Divide byte	DIVB2	0xAF	3-11	21-154	Case 3
Divide halfword	DIVH2	0xAE	3-11	21-194	
Divide word	DIVW2	0xAC	3-11	19-275	
Divide byte, 3-address	DIVB3	0xEF	4-16	23-270	Case 4
Divide halfword, 3-address	DIVH3	0xEE	4-16	23-263	
Divide word, 3-address	DIVW3	0xEC	4-16	21-275	
<b>Modulo:</b>					
Modulo byte	MODB2	0xA7	3-11	21-154	Case 3
Modulo halfword	MODH2	0xA6	3-11	21-194	
Modulo word	MODW2	0xA4	3-11	19-275	
Modulo byte, 3-address	MODB3	0xE7	4-16	23-270	Case 4
Modulo halfword, 3-address	MODH3	0xE6	4-16	23-263	
Modulo word, 3-address	MODW3	0xE4	4-16	21-275	
<b>Arithmetic Shift:</b>					
Arithmetic left shift word	ALSW3	0xC0	4-16	5-43	Case 5
Arithmetic right shift byte	ARSB3	0xC7	4-16	5-44	Case 3
Arithmetic right shift halfword	ARSH3	0xC6	4-16	5-44	
Arithmetic right shift word	ARSW3	0xC4	4-16	5-43	

### 5.3.4 Logical Instructions

Logical instructions (listed in Table 5-7) perform logical operations on data in registers and memory. Most of these instructions have three types (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 type 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*). Three-address instructions have two source operands (*src1*, *src2*) and a destination operand. A few instructions have a read-only count operand (*count*).

## INSTRUCTION SET AND ADDRESSING MODES

### Logical Instructions

<b>Table 5-7. Logical Instruction Group</b>					
Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
<b>AND:</b>					
AND byte	ANDB2	0xBB	3-11	4-33	Case 1
AND halfword	ANDH2	0xBA	3-11	4-33	
AND word	ANDW2	0xB8	3-11	2-31	
AND byte, 3-address	ANDB3	0xFB	4-16	4-44	
AND halfword, 3-address	ANDH3	0xFA	4-16	4-44	
AND word, 3-address	ANDW3	0xF8	4-16	4-43	
<b>Exclusive OR (XOR):</b>					
Exclusive OR byte	XORB2	0xB7	3-11	4-33	
Exclusive OR halfword	XORH2	0xB6	3-11	4-33	
Exclusive OR word	XORW2	0xB4	3-11	2-31	
Exclusive OR byte 3-address	XORB3	0xF7	4-16	4-44	
Exclusive OR halfword 3-address	XORH3	0xF6	4-16	4-44	
Exclusive OR word, 3-address	XORW3	0xF4	4-16	4-43	
<b>OR:</b>					
OR byte	ORB2	0xB3	3-11	4-33	
OR halfword	ORH2	0xB2	3-11	4-33	
OR word	ORW2	0xB0	3-11	2-31	
OR byte, 3-address	ORB3	0xF3	4-16	4-44	
OR halfword, 3-address	ORH2	0xF2	4-16	4-44	
OR word, 3-address	ORW3	0xF0	4-16	4-43	
<b>Compare or Test:</b>					
Compare byte	CMPB	0x3F	3-11	4-33	Case 2
Compare halfword	CMPH	0x3E	3-11	4-33	
Compare word	CMPW	0x3C	3-11	2-31	
Test byte	TSTB	0x2B	2-6	2-24	Case 6
Test halfword	TSTH	0x2A	2-6	2-24	
Test word	TSTW	0x28	2-6	1-18	
Bit test byte	BITB	0x3B	3-11	4-31	Case 1
Bit test halfword	BITH	0x3A	3-11	4-31	
Bit test word	BITW	0x38	3-11	2-30	
<b>Clear:</b>					
Clear byte	CLRB	0x83	2-6	2-21	Case 2
Clear halfword	CLRH	0x82	2-6	2-21	
Clear word	CLRW	0x80	2-6	1-19	
<b>Rotate or Logical Shift:</b>					
Rotate word	ROTW	0xD8	4-16	5-43	Case 1
Logical left shift byte	LLSB3	0xD3	4-16	5-44	
Logical left shift halfword	LLSH3	0xD2	4-16	5-44	
Logical left shift word	LLSW3	0xD0	4-16	5-43	
Logical right shift word	LRSW3	0xD4	4-16	5-43	

\*Refer to Table 5-11 for condition flag code assignments.

## INSTRUCTION SET AND ADDRESSING MODES

### Program Control Instructions

#### 5.3.5 Program Control Instructions

Program control instructions (listed in Table 5-8) change the program sequence, but generally do not alter the condition flags.

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 a 16-bit displacement is embedded in the instruction. This displacement (*disp8*, *disp16*) is read, its sign is extended through 32 bits, and the result is 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 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 transfer is different from a normal transfer. Before transferring to a subroutine, it saves the address of the next instruction.

Call and return instructions for subroutines always implicitly affect the stack pointer (SP). For subroutines, 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. Return from subroutine decrements the SP by 4, retrieves the saved address from the stack, and writes it to 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.

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 the six 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.

**INSTRUCTION SET AND ADDRESSING MODES**  
**Program Control Instructions**

Table 5-8. Program Control Instruction Group					
Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions
<b>Unconditional Transfer:</b> Branch with byte displacement	BRB	0x7B	2	5–16	Unchanged
Branch with halfword displacement	BRH	0x7A	3	5–14	
Jump	JMP	0x24	2–6	7–17	
<b>Conditional Transfers:</b> Branch on carry clear byte	BCCB	0x53*	2	See Note 1	
Branch on carry clear halfword	BCCH	0x52*	3	See Note 2	
Branch on carry set byte	BCSB	0x5B*	2	See Note 1	
Branch on carry set halfword	BCSH	0x5A*	3	See Note 2	
Branch on overflow clear, byte displacement	BVCB	0x63	2	See Note 1	
Branch on overflow clear, halfword displacement	BVCH	0x62	3	See Note 2	
Branch on overflow set, byte displacement	BVSB	0x6B	2	See Note 1	
Branch on overflow set, halfword displacement	BVSH	0x6A	3	See Note 2	
Branch on equal byte (duplicate)	BEB	0x6F	2	See Note 1	
Branch on equal byte	BEB	0x7F	2	See Note 1	
Branch on equal halfword (duplicate)	BEH	0x6E	3	See Note 2	
Branch on equal halfword	BEH	0x7E	3	See Note 2	
Branch on not equal byte (duplicate)	BNEB	0x67	2	See Note 1	
Branch on not equal byte	BNEB	0x77	2	See Note 1	
Branch on not equal halfword (duplicate)	BNEH	0x66	3	See Note 2	
Branch on not equal halfword	BNEH	0x76	3	See Note 2	

Notes:

- \* Indicates that opcode matches another instruction mnemonic with the same operation.
- 1. 5–10 cycles during a branch not taken; 7–14 cycles during a branch taken.
- 2. 5–10 cycles during a branch not taken; 7–12 cycles during a branch taken.

**INSTRUCTION SET AND ADDRESSING MODES**  
**Program Control Instructions**

Table 5-8. Program Control Instruction Group (Continued)					
Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions
Branch on less than byte (signed)	BLB	0x4B	2	See Note 1	Unchanged
Branch on less than halfword (signed)	BLH	0x4A	3	See Note 2	
Branch on less than byte (unsigned)	BLUB	0x5B*	2	See Note 1	
Branch on less than halfword (unsigned)	BLUH	0x5A*	3	See Note 2	
Branch on less than or equal byte (signed)	BLEB	0x4F	2	See Note 1	
Branch on less than or equal halfword (signed)	BLEH	0x4E	3	See Note 2	
Branch on less than or equal byte (unsigned)	BLEUB	0x5F	2	See Note 1	
Branch on less than or equal halfword (unsigned)	BLEUH	0x5E	3	See Note 2	
Branch on greater than byte (signed)	BGB	0x47	2	See Note 1	
Branch on greater than halfword (signed)	BGH	0x46	3	See Note 2	
Branch on greater than byte (unsigned)	BGUB	0x57	2	See Note 1	
Branch on greater than halfword (unsigned)	BGUH	0x56	3	See Note 2	
Branch on greater than or equal byte (signed)	BGEB	0x43	2	See Note 1	
Branch on greater than or equal halfword (signed)	BGEH	0x42	3	See Note 2	
Branch on greater than or equal byte (unsigned)	BGEUB	0x53*	2	See Note 1	
Branch on greater than or equal halfword (unsigned)	BGEUH	0x52*	3	See Note 2	
Return on carry clear	RCC	0x50*	1	See Note 3	
Return on carry set (signed)	RCS	0x58*	1	See Note 3	
Return on overflow clear	RVC	0x60	1	See Note 3	
Return on overflow set (signed)	RVS	0x68	1	See Note 3	

Notes:

\* Indicates that opcode matches another instruction mnemonic with the same operation.

1. 5–10 cycles during a branch not taken; 7–14 cycles during a branch taken.
2. 5–10 cycles during a branch not taken; 7–12 cycles during a branch taken.
3. 4–5 cycles during a return not taken; 13–14 cycles during a branch taken.

## INSTRUCTION SET AND ADDRESSING MODES

### Program Control Instructions

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions
Return on equal (unsigned)	REQLU	0x6C*	1	See Note 3	Unchanged
Return on equal (signed)	REQL	0x7C*	1	See Note 3	
Return on not equal (unsigned)	RNEQU	0x64*	1	See Note 3	
Return on not equal (signed)	RNEQ	0x74*	1	See Note 3	
Return on less than (signed)	RLSS	0x48	1	See Note 3	
Return on less than (unsigned)	RLSSU	0x58*	1	See Note 3	
Return on less than or equal (signed)	RLEQ	0x4C	1	See Note 3	
Return on less than or equal (unsigned)	RLEQU	0x5C	1	See Note 3	
Return on greater than (signed)	RGTR	0x44	1	See Note 3	
Return on greater than (unsigned)	RGTRU	0x54	1	See Note 3	
Return on greater than or equal (signed)	RGEQ	0x40	1	See Note 3	
Return on greater than or equal (unsigned)	RGEQU	0x50*	1	See Note 3	
<b>Subroutine Transfer:</b> Branch to subroutine, byte displacement	BSBB	0x37	2	See Note 1	
Branch to subroutine, halfword displacement	BSBH	0x36	3	See Note 2	
Jump to subroutine	JSB	0x34	2–6	7–17	
Return from subroutine	RSB	0x78	1	13–14	
<b>Procedure Transfer:</b> Save registers	SAVE	0x10	2	11–36**	
Restore registers	RESTORE	0x18	2	12–38**	
Call procedure	CALL	0x2C	7	25–36	
Return from procedure	RET	0x08	1	21–23	

Notes:

\* Indicates that opcode matches another instruction mnemonic with the same operation.

\*\* Dependent on number of registers saved/restored.

1. 5–10 cycles during a branch not taken; 7–14 cycles during a branch taken.
2. 5–10 cycles during a branch not taken; 7–12 cycles during a branch taken.
3. 4–5 cycles during a return not taken; 13–14 cycles during a branch taken.

Procedure-call instructions explicitly manipulate four registers:

1. **PC** – The call instruction saves the old PC as the return address (RA) and sets 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).
2. **SP** - These instructions adjust SP automatically to point to the top of the stack whenever they store or retrieve items.

# INSTRUCTION SET AND ADDRESSING MODES

## Program Control Instructions

3. FP - The save instruction sets 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.
4. AP - The call instruction adjusts AP to the beginning of a list of arguments for the function.

On a function call, the calling function contains a call instruction; the SAVE instruction should be the first statement of the called function. For a return, a RESTORE and a RETURN appear in the function being exited.

Figure 5-16 shows the stack after the CALL-SAVE sequence:

```

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

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

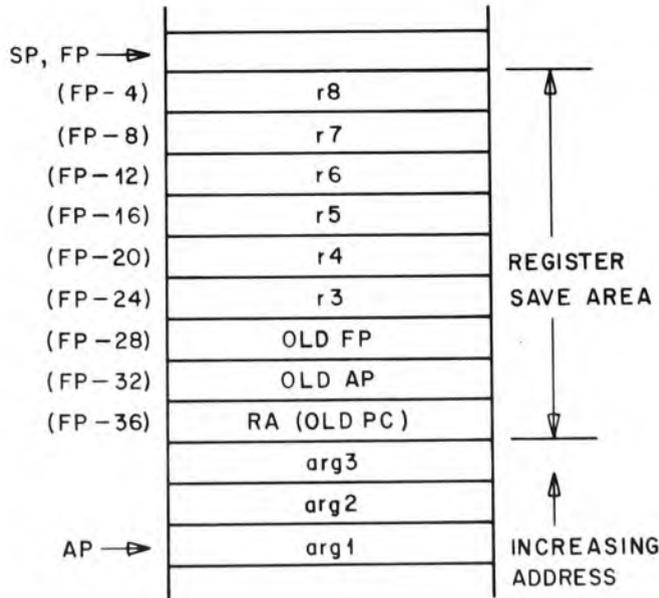


Figure 5-16. Stack After CALL-SAVE Sequence

## INSTRUCTION SET AND ADDRESSING MODES

### Stack and Miscellaneous Instructions

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

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

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

### 5.3.6 Coprocessor Instructions

These instructions (listed in Table 5-9) implement the interface with coprocessors. Most programmers will find it convenient to access the math acceleration unit (MAU) 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.

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” 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.

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).

### 5.3.7 Stack and Miscellaneous Instructions

The stack instructions (listed in Table 5-10) 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 **m32as** 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.

**INSTRUCTION SET AND ADDRESSING MODES**  
**Stack and Miscellaneous Instructions**

<b>Table 5-9. Coprocessor Instruction Group</b>					
<b>Instuction</b>	<b>Mnemonic</b>	<b>Opcode</b>	<b>Bytes</b>	<b>Cycles</b>	<b>Conditions*</b>
Coprocessor operation	SPOP	0x32	5	Not Applicable	Case 10
Coprocessor operation read single	SPOPRS	0x22	6—10	Not Applicable	
Coprocessor operation read double	SPOPRD	0x02	6—10	Not Applicable	
Coprocessor operation read triple	SPOPRT	0x06	6—10	Not Applicable	
Coprocessor operation single 2-address	SPOPS2	0x23	7—15	Not Applicable	
Coprocessor operation double 2-address	SPOPD2	0x03	6—15	Not Applicable	
Coprocessor operation triple 2-address	SPOPT2	0x07	7—15	Not Applicable	
Coprocessor operation write single	SPOPWS	0x33	6—10	Not Applicable	
Coprocessor operation write double	SPOPWD	0x13	6—10	Not Applicable	
Coprocessor operation write triple	SPOPWT	0x17	6—10	Not Applicable	

\*Refer to Table 5-11 for condition flag code assignments.

<b>Table 5-10. Stack and Miscellaneous Instruction Group</b>					
<b>Instuction</b>	<b>Mnemonic</b>	<b>Opcode</b>	<b>Bytes</b>	<b>Cycles</b>	<b>Conditions*</b>
<b>Stack Operations:</b> Push address word Push word Pop word	PUSHAW PUSHW POPW	0xE0 0xA0 0x20	2—6 2—6 2—6	9—20 8—23 9—23	Case 1
<b>Miscellaneous:</b> No operation, 1 byte No operation, 2 byte No operation, 3 byte	NOP NOP2 NOP3	0x70 0x73 0x72	1 2 3	4—11 4—10 4—10	
Breakpoint trap	BPT	0x2E	1	—	Unchanged
Cache flush	CFLUSH	0x27	1	—	
Extended opcode	EXTOP	0x14	1, 2	—	

\*Refer to Table 5-11 for condition flag code assignments.

## 5.4 INSTRUCTION SET LISTINGS

Section 5.4.4 **Instruction Set Descriptions** at the end of this section presents descriptions of each member of the instruction set for the *WE* 32100 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 5.3 **Functional Groups** (Tables 5-5 through 5-10), 5.4.2 **Instruction Set Summary by Mnemonic**, and 5.4.3 **Instruction Set Summary by Opcode**.)

### 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 5-12.

**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, it will give the contents of registers before and after execution. 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.

# INSTRUCTION SET AND ADDRESSING MODES

## Notation

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 type 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 type, 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 type 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.
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.

Notes: 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.

Table 5-12. 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
-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
x y	Bitwise inclusive OR x and y
x^y	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
←	Assigns the value on the right to the location identified on the left (same as the C language assignment operator '=')
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 Summary by Mnemonic

### 5.4.2 Instruction Set Summary by Mnemonic

A mnemonic listing of instruction sets is given in Table 5-13.

Table 5-13. 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
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 (duplicate)
BEB	0x7F	Branch on equal byte
BEH	0x6E	Branch on equal halfword (duplicate)
BEH	0x7E	Branch on equal halfword
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)
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)

\* Indicates that opcode matches another instruction mnemonic with the same operation.

## INSTRUCTION SET AND ADDRESSING MODES

### Instruction Set Summary by Mnemonic

<b>Table 5-13. Instruction Set Summary by Mnemonic (Continued)</b>		
<b>Mnemonic</b>	<b>Opcode</b>	<b>Instruction</b>
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 (8-bit) displacement
BRH	0x7A	Branch with halfword (16-bit) 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
CFLUSH	0x27	Cache flush
CLRB	0x83	Clear byte
CLRH	0x82	Clear halfword
CLRW	0x80	Clear word
CMPB	0x3F	Compare byte
CMPH	0x3E	Compare halfword
CMPW	0x3C	Compare word
DECB	0x97	Decrement byte
DECH	0x96	Decrement halfword
DECW	0x94	Decrement word
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
EXTFB	0xCF	Extract field byte
EXTFH	0xCE	Extract field halfword
EXTFW	0xCC	Extract field word
EXTOP	0x14	Extended opcode
INCB	0x93	Increment byte
INCH	0x92	Increment halfword
INCW	0x90	Increment word
INSFB	0xCB	Insert field byte

\*Indicates that opcode matches another instruction mnemonic with the same operation.

# INSTRUCTION SET AND ADDRESSING MODES

## Instruction Set Summary by Mnemonic

<b>Table 5-13. Instruction Set Summary by Mnemonic (Continued)</b>		
<b>Mnemonic</b>	<b>Opcode</b>	<b>Instruction</b>
INSFH	0xCA	Insert field halfword
INSFW	0xC8	Insert field word
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
MOVAW	0x04	Move address (word)
MOVB	0x87	Move byte
MOVBLW	0x3019	Move block of words
MOVH	0x86	Move halfword
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 bytes
NOP3	0x72	No operation, 3 bytes
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
POPW	0x20	Pop word
PUSHAW	0xE0	Push address word
PUSHW	0xA0	Push word

**INSTRUCTION SET AND ADDRESSING MODES**  
**Instruction Set Summary by Mnemonic**

<b>Table 5-13. Instruction Set Summary by Mnemonic (Continued)</b>		
<b>Mnemonic</b>	<b>Opcode</b>	<b>Instruction</b>
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
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)
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
SPOP	0x32	Coprocessor operation
SPOPRS	0x22	Coprocessor operation read single
SPOPRD	0x02	Coprocessor operation read double
SPOPRT	0x06	Coprocessor operation read triple
SPOPS2	0x23	Coprocessor operation single 2-address
SPOPD2	0x03	Coprocessor operation double 2-address
SPOPT2	0x07	Coprocessor operation triple 2-address
SPOPWS	0x33	Coprocessor operation write single
SPOPWD	0x13	Coprocessor operation write double
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
SUBW2	0xBC	Subtract word
SUBW3	0xFC	Subtract word, 3-address

\*Indicates that opcode matches another instruction mnemonic with the same operation.

## INSTRUCTION SET AND ADDRESSING MODES

### Instruction Set Summary by Opcode

<b>Mnemonic</b>	<b>Opcode</b>	<b>Instruction</b>
SWAPBI	0x1F	Swap byte interlocked
SWAPHI	0x1E	Swap halfword interlocked
SWAPWI	0x1C	Swap word interlocked
TSTB	0x2B	Test byte
TSTH	0x2A	Test halfword
TSTW	0x28	Test word
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

### 5.4.3 Instruction Set Summary by Opcode

The instruction sets are listed by opcode in Table 5-14.

<b>Menemonic</b>	<b>Opcode</b>	<b>Instruction</b>
SPOPRD	0x02	Coprocessor operation read double
SPOPD2	0x03	Coprocessor operation double, 2-address
MOVAW	0x04	Move address (word)
SPOPRT	0x06	Coprocessor operation read triple
SPOPT2	0x07	Coprocessor operation triple, 2-address
RET	0x08	Return from procedure
SAVE	0x10	Save registers
SPOPWD	0x13	Coprocessor operation write double
EXTOP	0x14	Extended opcode
SPOPWT	0x17	Coprocessor operation write triple
RESTORE	0x18	Restore registers
SWAPWI	0x1C	Swap word interlocked
SWAPHI	0x1E	Swap halfword interlocked
SWAPBI	0x1F	Swap byte interlocked
POPW	0x20	Pop word
SOPPRS	0x22	Coprocessor operation read single
SOPPS2	0x23	Coprocessor operation single, 2-address
JMP	0x24	Jump
CFLUSH	0x27	Cache Flush
TSTW	0x28	Test word
TSTH	0x2A	Test halfword
TSTB	0x2B	Test byte
CALL	0x2C	Call procedure
BPT	0x2E	Breakpoint trap

**INSTRUCTION SET AND ADDRESSING MODES**  
**Instruction Set Summary by Opcode**

<b>Table 5-14. Instruction Set Summary by Opcode (Continued)</b>		
<b>Mnemonic</b>	<b>Opcode</b>	<b>Instruction</b>
MVERNO	0x3009	Move version number
MOVBLW	0x3019	Move block of words
STREND	0x301F	String end
STRCPY	0x3035	String copy
SPOP	0x32	Coprocessor operation
SPOPWS	0x33	Coprocessor operation write single
JSB	0x34	Jump to subroutine
BSBH	0x36	Branch to subroutine, halfword displacement
BSBB	0x37	Branch to subroutine, byte displacement
BITW	0x38	Bit test word
BITH	0x3A	Bit test halfword
BITB	0x3B	Bit test byte
CMPW	0x3C	Compare word
CMPH	0x3E	Compare halfword
CMPB	0x3F	Compare byte
RGEQ	0x40	Return on greater than or equal (signed)
BGEH	0x42	Branch on greater than or equal halfword (signed)
BGEB	0x43	Branch on greater than or equal byte (signed)
RGTR	0x44	Return on greater than (signed)
BGH	0x46	Branch on greater than halfword (signed)
BGB	0x47	Branch on greater than byte (signed)
RLSS	0x48	Return on less than (signed)
BLH	0x4A	Branch on less than halfword (signed)
BLB	0x4B	Branch on less than byte (signed)
RLEQ	0x4C	Return on less than or equal (signed)
BLEH	0x4E	Branch on less than or equal halfword (signed)
BLEB	0x4F	Branch on less than or equal byte (signed)
RCC	0x50*	Return on carry clear
RGEQU	0x50*	Return on greater than or equal (unsigned)
BCCH	0x52*	Branch on carry clear halfword
BGEUH	0x52*	Branch on greater than or equal halfword (unsigned)
BCCB	0x53*	Branch on carry clear byte
BGEUB	0x53*	Branch on greater than or equal byte (unsigned)
RGTRU	0x54	Return on greater than (unsigned)
BGUH	0x56	Branch on greater than halfword (unsigned)
BGUB	0x57	Branch on greater than byte (unsigned)
RCS	0x58*	Return on carry set
RLSSU	0x58*	Return on less than (unsigned)
BCSH	0x5A*	Branch on carry set halfword
BLUH	0x5A*	Branch on less than halfword (unsigned)
BCSB	0x5B*	Branch on carry set byte
BLUB	0x5B*	Branch on less than byte (unsigned)

\*Indicates that opcode matches another instruction mnemonic with the same operation.

# INSTRUCTION SET AND ADDRESSING MODES

## Instruction Set Summary by Opcode

Table 5-14. Instruction Set Summary by Opcode (Continued)		
Mnemonic	Opcode	Instruction
RLEQU	0x5C	Return on less than or equal (unsigned)
BLEUH	0x5E	Branch on less than or equal halfword (unsigned)
BLEUB	0x5F	Branch on less than or equal byte (unsigned)
RVC	0x60	Return on overflow clear
BVCH	0x62	Branch on overflow clear, halfword displacement
BVCB	0x63	Branch on overflow clear, byte displacement
RNEQU	0x64	Return on not equal (unsigned)
BNEH	0x66	Branch on not equal halfword (duplicate)
BNEB	0x67	Branch on not equal byte (duplicate)
RVS	0x68	Return on overflow set
BVSH	0x6A	Branch on overflow set, halfword displacement
BVSB	0x6B	Branch on overflow set, byte displacement
REQLU	0x6C	Return on equal (unsigned)
BEH	0x6E	Branch on equal halfword (duplicate)
BEB	0x6F	Branch on equal byte (duplicate)
NOP	0x70	No operation, 1 byte
NOP3	0x72	No operation, 3 bytes
NOP2	0x73	No operation, 2 bytes
RNEQ	0x74	Return on not equal (signed)
BNEH	0x76	Branch on not equal halfword
BNEB	0x77	Branch on not equal
RSB	0x78	Return from subroutine
BRH	0x7A	Branch with halfword (16-bit) displacement
BRB	0x7B	Branch with byte (8-bit) displacement
REQL	0x7C	Return on equal (signed)
BEH	0x7E	Branch on equal halfword
BEB	0x7F	Branch on equal byte
CLRW	0x80	Clear word
CLRH	0x82	Clear halfword
CLRB	0x83	Clear byte
MOVW	0x84	Move word
MOVH	0x86	Move halfword
MOVB	0x87	Move byte
MCOMW	0x88	Move complemented word
MCOMH	0x8A	Move complemented halfword
MCOMB	0x8B	Move complemented byte
MNEGW	0x8C	Move negated word
MNEGH	0x8E	Move negated halfword
MNEGB	0x8F	Move negated byte

**INSTRUCTION SET AND ADDRESSING MODES**  
**Instruction Set Summary by Opcode**

<b>Table 5-14. Instruction Set Summary by Opcode (Continued)</b>		
<b>Mnemonic</b>	<b>Opcode</b>	<b>Instruction</b>
INCW	0x90	Increment word
INCH	0x92	Increment halfword
INCB	0x93	Increment byte
DECW	0x94	Decrement word
DECH	0x96	Decrement halfword
DECB	0x97	Decrement byte
ADDW2	0x9C	Add word
ADDH2	0x9E	Add halfword
ADDB2	0x9F	Add byte
PUSHW	0xA0	Push word
MODW2	0xA4	Modulo word
MODH2	0xA6	Modulo halfword
MODB2	0xA7	Modulo byte
MULW2	0xA8	Multiply word
MULH2	0xAA	Multiply halfword
MULB2	0xAB	Multiply byte
DIVW2	0xAC	Divide word
DIVH2	0xAE	Divide halfword
DIVB2	0xAF	Divide byte
ORW2	0xB0	OR word
ORH2	0xB2	OR halfword
ORB2	0xB3	OR byte
XORW2	0xB4	Exclusive OR word
XORH2	0xB6	Exclusive OR halfword
XORB2	0xB7	Exclusive OR byte
ANDW2	0xB8	AND word
ANDH2	0xBA	AND halfword
ANDB2	0xBB	AND byte
SUBW2	0xBC	Subtract word
SUBH2	0xBE	Subtract halfword
SUBB2	0xBF	Subtract byte
ALSW3	0xC0	Arithmetic left shift word
ARSW3	0xC4	Arithmetic right shift word
ARSH3	0xC6	Arithmetic right shift halfword
ARSB3	0xC7	Arithmetic right shift byte
INSFW	0xC8	Insert field word
INSFH	0xCA	Insert field halfword
INSFB	0xCB	Insert field byte
EXTFW	0xCC	Extract field word
EXTFH	0xCE	Extract field halfword
EXTFB	0xCF	Extract field byte
LLSW3	0xD0	Logical left shift word
LLSH3	0xD2	Logical left shift halfword

**INSTRUCTION SET AND ADDRESSING MODES**  
**Instruction Set Summary by Opcode**

**Table 5-14. Instruction Set Summary by Opcode (Continued)**

<b>Mnemonic</b>	<b>Opcode</b>	<b>Instruction</b>
LLSB3	0xD3	Logical left shift byte
LRSW3	0xD4	Logical right shift word
ROTW	0xD8	Rotate word
ADDW3	0xDC	Add word, 3-address
ADDH3	0xDE	Add halfword, 3-address
ADDB3	0xDF	Add byte, 3-address
PUSHAW	0xE0	Push address word
MODW3	0xE4	Modulo word, 3-address
MODH3	0xE6	Modulo halfword, 3-address
MODB3	0xE7	Modulo byte, 3-address
MULW3	0xE8	Multiply word, 3-address
MULH3	0xEA	Multiply halfword, 3-address
MULB3	0xEB	Multiply byte, 3-address
DIVW3	0xEC	Divide word, 3-address
DIVH3	0xEE	Divide halfword, 3-address
DIVB3	0xEF	Divide byte, 3-address
ORW3	0xF0	OR word, 3-address
ORH3	0xF2	OR halfword, 3-address
ORB3	0xF3	OR byte, 3-address
XORW3	0xF4	Exclusive OR word, 3-address
XORH3	0xF6	Exclusive OR halfword, 3-address
XORB3	0xF7	Exclusive OR byte, 3-address
ANDW3	0xF8	AND word, 3-address
ANDH3	0xFA	AND halfword, 3-address
ANDB3	0xFB	AND byte, 3-address
SUBW3	0xFC	Subtract word, 3-address
SUBH3	0xFE	Subtract halfword, 3-address
SUBB3	0xFF	Subtract byte, 3-address

**5.4.4 Instruction Set Descriptions**

The instruction set is described in detail on the following pages.

**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**             $dst \leftarrow dst + src$

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

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

**Exceptions**            Illegal operand exception occurs if literal or immediate mode is used for *dst*.  
  
                         Integer overflow exception occurs if there is truncation.

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

**ADDB3**  
**ADDH3**  
**ADDW3**

**ADDB3**  
**ADDH3**  
**ADDW3**

### **ADD, 3 Address**

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

## ARITHMETIC LEFT SHIFT

<b>Assembler Syntax</b>	ALSW3 <i>count,src,dst</i> Arithmetic left shift word								
<b>Opcode</b>	0xC0 ALSW3								
<b>Operation</b>	$dst \leftarrow src \ll (\text{count} \ \& \ 0x1F) \text{ bits}$								
<b>Address Modes</b>	<i>count</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate								
<b>Condition Flags</b>	$N \leftarrow 1$ , if $dst < 0$ $Z \leftarrow 1$ , if $dst == 0$ $C \leftarrow 0$ $V \leftarrow 0$ (see Note)								
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .								
<b>Examples</b>	<p>Before: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>8F</td><td>0F</td><td>DF</td><td>FD</td></tr></table></p> <p style="text-align: center;">←increasing bits</p> <p>ALSW3 &amp;2,%r0,%r0</p> <p>After: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>3C</td><td>3F</td><td>7F</td><td>F4</td></tr></table></p>	8F	0F	DF	FD	3C	3F	7F	F4
8F	0F	DF	FD						
3C	3F	7F	F4						
<b>Note</b>	All operands are of type word. However, only the five low-order bits of <i>count</i> 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 type mode changes the type of <i>dst</i> . Zeros replace bits that are shifted out. The sign bit is not changed.								

**ANDB2  
ANDH2  
ANDW2**

**ANDB2  
ANDH2  
ANDW2**

**AND**

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

**ANDB3**  
**ANDH3**  
**ANDW3**

**ANDB3**  
**ANDH3**  
**ANDW3**

### **AND, 3 ADDRESS**

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

**ARSB3**  
**ARSH3**  
**ARSW3**

**ARSB3**  
**ARSH3**  
**ARSW3**

## ARITHMETIC RIGHT SHIFT

<b>Assembler Syntax</b>	ARSB3 <i>count,src,dst</i> ARSH3 <i>count,src,dst</i> ARSW3 <i>count,src,dst</i>	Arithmetic right shift byte Arithmetic right shift halfword Arithmetic right shift word								
<b>Opcodes</b>	0xC7 ARSB3 0xC6 ARSH3 0xC4 ARSW3									
<b>Operation</b>	$dst \leftarrow src \gg (count \ \& \ 0x1f) \text{ bits}$									
<b>Address Modes</b>	<i>count</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate									
<b>Condition Flags</b>	N $\leftarrow$ 1, if <i>dst</i> < 0 Z $\leftarrow$ 1, if <i>dst</i> == 0 C $\leftarrow$ 0 V $\leftarrow$ 0									
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .									
<b>Examples</b>	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> $\leftarrow$ increasing bits  ARSH3 &2,%r0,%r0  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							
<b>Note</b>	All operands are of type word. 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** *disp8*    Branch on carry clear, byte displacement  
                         **BCCH** *disp16*    Branch on carry clear, halfword displacement

**Opcodes**            0x53    **BCCB**  
                         0x52    **BCCH**

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

**Address Modes**        None valid  
                              *disp8* = signed 8-bit value  
                              *disp16* = signed 16-bit value

**Condition Flags**        Unchanged

**Exceptions**            None

**Examples**             **BCCB** 0x9  
                              **BCCH** 0xFF23

**BCSB**  
**BCSH**

**BCSB**  
**BCSH**

**BRANCH ON CARRY SET**

<b>Assembler Syntax</b>	BCSB <i>disp8</i> BCSH <i>disp16</i>	Branch on carry set, byte displacement Branch on carry set, halfword displacement
<b>Opcodes</b>	0x5B BCSB 0x5A BCSH	
<b>Operation</b>	if (C ==1) PC ← PC + SEXT( <i>disp</i> )	
<b>Address Modes</b>	None valid <i>disp8</i> = signed 8-bit value <i>disp16</i> = signed 16-bit value	
<b>Condition Flags</b>	Unchanged	
<b>Exceptions</b>	None	
<b>Examples</b>	BCSB 0xFF BCSH 0x1234	

**BEB**  
**BEH**

**BEB**  
**BEH**

## BRANCH ON EQUAL

**Assembler**      BEB *disp8*      Branch on equal, byte displacement  
**Syntax**          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**          None valid  
**Modes**            *disp8* = signed 8-bit value  
                      *disp16* = signed 16-bit value

**Condition**        Unchanged  
**Flags**

**Exceptions**      None

**Examples**        BEB 0xF1  
                      BEH 0x4221

**BGB**  
**BGH**

**BGB**  
**BGH**

**BRANCH ON GREATER THAN (SIGNED)**

<b>Assembler Syntax</b>	BGB <i>disp8</i> Branch on greater than, byte displacement (signed) BGH <i>disp16</i> Branch on greater than, halfword displacement (signed)
<b>Opcodes</b>	0x47    BGB 0x46    BGH
<b>Operation</b>	if ((N & Z) == 0) PC ← PC + SEXT( <i>disp</i> )
<b>Address Modes</b>	None valid <i>disp8</i> = signed 8-bit value <i>disp16</i> = signed 16-bit value
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Examples</b>	BGB more BGH less

**BGEB**  
**BGEH**

**BGEB**  
**BGEH**

**BRANCH ON GREATER THAN OR EQUAL (SIGNED)**

<b>Assembler Syntax</b>	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)
<b>Opcodes</b>	0x43 BGEB 0x42 BGEH
<b>Operation</b>	if ((N == 0) (Z == 1)) PC ← PC + SEXT( <i>disp</i> )
<b>Address Modes</b>	None valid <i>disp8</i> = signed 8-bit value <i>disp16</i> = signed 16-bit value
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Examples</b>	BGEB again BGEH 0xF102

**BGEUB**  
**BGEUH**

**BGEUB**  
**BGEUH**

**BRANCH ON GREATER THAN OR EQUAL (UNSIGNED)**

**Assembler Syntax**      **BGEUB** *disp8*      Branch on greater than or equal, byte displacement (unsigned)  
                         **BGEUH** *disp16*      Branch on greater than or equal, halfword displacement (unsigned)

**Opcodes**              0x53    **BGEUB**  
                         0x52    **BGEUH**

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

**Address Modes**        None valid  
                         *disp8* = signed 8-bit value  
  
                         *disp16* = signed 16-bit value

**Condition Flags**      Unchanged

**Exceptions**          None

**Examples**             **BGEUB** 0xA1  
                         **BGEUH** ahead

**BGUB**  
**BGUH**

**BGUB**  
**BGUH**

**BRANCH ON GREATER THAN (UNSIGNED)**

<b>Assembler Syntax</b>	BGUB <i>disp8</i> Branch on greater than, byte displacement (unsigned) BGUH <i>disp16</i> Branch on greater than, halfword displacement (unsigned)
<b>Opcodes</b>	0x57    BGUB 0x56    BGUH
<b>Operation</b>	if ((C & Z) == 0) PC ← PC + SEXT( <i>disp</i> )
<b>Address Modes</b>	None valid <i>disp8</i> = signed 8-bit value <i>disp16</i> = signed 16-bit value
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Examples</b>	BGUB 0xDE BGUH 0xF123

**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**             $\text{temp} \leftarrow \text{src2} \ \& \ \text{src1}$

**Address Modes**        *src1*    all modes  
  
                         *src2*    all modes

**Condition Flags**       $N \leftarrow \text{MSB of temp}$   
                          $Z \leftarrow 1, \text{ if temp} == 0$   
                          $C \leftarrow 0$   
                          $V \leftarrow 0$

**Exceptions**            None

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

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

**BLB**  
**BLH**

**BLB**  
**BLH**

**BRANCH ON LESS THAN (SIGNED)**

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

**BLEB**  
**BLEH**

**BLEB**  
**BLEH**

**BRANCH ON LESS THAN OR EQUAL (SIGNED)**

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

**BLEUB**  
**BLEUH**

**BLEUB**  
**BLEUH**

**BRANCH ON LESS THAN OR EQUAL (UNSIGNED)**

<b>Assembler Syntax</b>	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)
<b>Opcodes</b>	0x5F BLEUB 0x5E BLEUH	
<b>Operation</b>	if ((C Z) == 1) PC ← PC + SEXT( <i>disp</i> )	
<b>Address Modes</b>	None valid <i>disp8</i> = signed 8-bit value <i>disp16</i> = signed 16-bit value	
<b>Condition Flags</b>	Unchanged	
<b>Exceptions</b>	None	
<b>Examples</b>	BLEUB 0x14 BLEUH back	

**BLUB**  
**BLUH**

**BLUB**  
**BLUH**

**BRANCH ON LESS THAN (UNSIGNED)**

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

**BNEB**  
**BNEH**

**BNEB**  
**BNEH**

### BRANCH ON NOT EQUAL

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

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

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

**Address**          None valid  
**Modes**            *disp8* = signed 8-bit value  
  
                      *disp16* = signed 16-bit value

**Condition**        Unchanged  
**Flags**

**Exceptions**      None

**Examples**        **BNEB** 0xFE  
                      **BNEH** 0xFF13

## **BPT**

## **BPT**

### **BREAKPOINT TRAP**

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

**BRB**  
**BRH**

**BRB**  
**BRH**

## **BRANCH**

<b>Assembler Syntax</b>	BRB <i>disp8</i> Branch with byte displacement BRH <i>disp16</i> Branch with halfword displacement
<b>Opcodes</b>	0x7B    BRB 0x7A    BRH
<b>Operation</b>	$PC \leftarrow PC + \text{SEXT}(disp)$
<b>Address Modes</b>	None valid <i>disp8</i> = signed 8-bit value <i>disp16</i> = signed 16-bit value
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Examples</b>	BRB 0xA BRH 0xFAA

**BSBB**  
**BSBH**

**BSBB**  
**BSBH**

### BRANCH TO SUBROUTINE

<b>Assembler</b>	BSBB <i>disp8</i>	Branch to subroutine, byte displacement
<b>Syntax</b>	BSBH <i>disp16</i>	Branch to subroutine, halfword displacement

<b>Opcodes</b>	0x37 BSBB
	0x36 BSBH

<b>Operation</b>	$*(SP++) \leftarrow$ address of next instruction
	$PC \leftarrow PC + \text{SEXT}(disp)$

<b>Address</b>	None valid
<b>Modes</b>	<i>disp8</i> = signed 8-bit value
	<i>disp16</i> = signed 16-bit value

<b>Condition</b>	Unchanged
<b>Flags</b>	

<b>Exceptions</b>	None
-------------------	------

<b>Examples</b>	BSBB sub2
	BSBH sub1

**BVCB**  
**BVCH**

**BVCB**  
**BVCH**

### BRANCH ON OVERFLOW CLEAR

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

**Opcodes**          0x63 BVCB  
                     0x62 BVCH

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

**Address**          None valid  
**Modes**            *disp8* = signed 8-bit value  
  
                     *disp16* = signed 16-bit value

**Condition**        Unchanged  
**Flags**

**Exceptions**      None

**Examples**        BVCB 0x7E  
                     BVCH 0x8F21

**BVSB**  
**BVSH**

**BVSB**  
**BVSH**

### BRANCH ON OVERFLOW SET

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

# CALL

# CALL

## CALL PROCEDURE

<b>Assembler Syntax</b>	CALL <i>src,dst</i> Call procedure
<b>Opcode</b>	0x2C    CALL
<b>Operation</b>	tempa    ← & <i>src</i> tempb    ← & <i>dst</i> *(SP+4) ← AP *SP      ← address of next instruction SP        ← SP+8 PC        ← tempb AP        ← tempa
<b>Address Modes</b>	<i>src</i> all modes except literal, register, or immediate  <i>dst</i> all modes except literal, register, or immediate
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	Illegal operand exception occurs if literal, register, expanded-operand type, or immediate mode is used for <i>src</i> or <i>dst</i> .
<b>Examples</b>	CALL -(3*4)(%sp),func1 (see Figure 3-9)
<b>Note</b>	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 first of the word arguments that the calling function pushed on the stack before executing the call.

## CFLUSH

## CFLUSH

### CACHE FLUSH

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

**CLRB**  
**CLRH**  
**CLRW**

**CLRB**  
**CLRH**  
**CLRW**

**CLEAR**

**Assembler  
Syntax**

CLRB *dst*    Clear byte  
CLRH *dst*    Clear halfword  
CLRW *dst*    Clear word

**Opcodes**

0x83   CLRB  
0x82   CLRH  
0x80   CLRW

**Operation**

*dst* ← 0

**Address  
Modes**

*dst*   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 mode is used for *dst*.

**Examples**

CLRB \*&0x300  
CLRH %r1  
CLRW (%r0)

**CMPB**  
**CMPH**  
**CMPW**

**CMPB**  
**CMPH**  
**CMPW**

## COMPARE

**Assembler Syntax**      **CMPB** *src1,src2*      Compare byte  
                         **CMPH** *src1,src2*      Compare halfword  
                         **CMPW** *src1,src2*      Compare word

**Opcodes**              0x3F    **CMPB**  
                         0x3E    **CMPH**  
                         0x3C    **CMPW**

**Operation**             $temp \leftarrow src2 - src1$

**Address Modes**        *src1*    all modes

*src2*    all modes

**Condition Flags**       $N \leftarrow 1$ , if *src2* < *src1* (signed)

$Z \leftarrow 1$ , if *src2* == *src1*

$C \leftarrow 1$ , if *src2* < *src1* (unsigned)

$V \leftarrow 0$

**Exceptions**            None

**Examples**              **CMPB** &10,%r0  
                         **CMPH** (%r0),(%r1)  
                         **CMPW** \*\$0x12F7,%r2

**Note**                    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

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

**DIVB2  
DIVH2  
DIVW2**

**DIVB2  
DIVH2  
DIVW2**

**DIVIDE**

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

**DIVB3**  
**DIVH3**  
**DIVW3**

**DIVB3**  
**DIVH3**  
**DIVW3**

## **DIVIDE, 3 ADDRESS**

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

**EXTFB**  
**EXTFH**  
**EXTFW**

**EXTFB**  
**EXTFH**  
**EXTFW**

## EXTRACT FIELD

<b>Assembler Syntax</b>	<b>EXTFB</b> <i>width,offset,src,dst</i> <b>EXTFH</b> <i>width,offset,src,dst</i> <b>EXTFW</b> <i>width,offset,src,dst</i>	Extract field from byte Extract field from halfword Extract field from word				
<b>Opcodes</b>	0xCF <b>EXTFB</b> 0xCE <b>EXTFH</b> 0xCC <b>EXTFW</b>					
<b>Operation</b>	$dst \leftarrow \text{FIELD}(\text{offset}, \text{width}, \text{src})$					
<b>Address Modes</b>	<i>width</i> all modes <i>offset</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate					
<b>Condition Flags</b>	$N \leftarrow$ high-order bit of <i>dst</i> $Z \leftarrow 1$ , if $dst == 0$ $C \leftarrow 0$ $V \leftarrow 0$ (see Note)					
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .					
<b>Examples</b>	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: 0 5px;">00</td><td style="padding: 0 5px;">00</td><td style="padding: 0 5px;">04</td><td style="padding: 0 5px;">56</td></tr></table> ← increasing bits	00	00	04	56	
00	00	04	56			

The field extracted starts at bit 4 of location L1, skipping bits 0 through 3, and extends through bit 14 of L1. These eleven bits are written to bits 0 through 10 of r0; zeros fill the remaining bits of r0.

**Note** Only the low-order five bits of *width* and *offset* are examined. If the sum *width* plus *offset* is greater than 32 (bits), then the field wraps around through bit 0 of the base word. The field specified by *width*, *offset*, and *src* is stored, right adjusted, in *dst*. The remaining bits of *dst* are set to 0. If the field is too large for the size of *dst*, the excess high-order bits are discarded and the V flag is set.

## EXTOP

## EXTOP

### EXTENDED OPCODE

**Assembler Syntax**      **EXTOP** *byte*    Extended opcode

**Opcode**                    0x14    **EXTOP**

**Operation**                /\***EXTOP** executes the following processor operation\*/  
{reserved-opcode exception}

**Address Modes**            None valid  
                              *byte* = 8-bit value

**Condition Flags**            Unchanged

**Exceptions**                Generates reserved opcode exception. See Note.

**Examples**                    **EXTOP** 0x2F

**Note**                        The **EXTOP** opcode is an escape to form additional instructions. The processor does not access *byte* when executing this instruction. Instead, it generates a reserved-opcode exception after decoding the opcode. The operating system's exception handler should access *byte*.

**INCB**  
**INCH**  
**INCW**

**INCB**  
**INCH**  
**INCW**

## **INCREMENT**

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

**INSFB**  
**INSFH**  
**INSFW**

**INSFB**  
**INSFH**  
**INSFW**

## INSERT FIELD

<b>Assembler Syntax</b>	INSFB <i>width,offset,src,dst</i> INSFH <i>width,offset,src,dst</i> INSFW <i>width,offset,src,dst</i>	Insert field from byte Insert field from halfword Insert field from word
<b>Opcodes</b>	0xCB   INSFB 0xCA   INSFH 0xC8   INSFW	
<b>Operation</b>	FIELD( <i>offset,width,dst</i> ) ← <i>src</i>	
<b>Address Modes</b>	<i>width</i> all modes <i>offset</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	N ← bit 31 of <i>dst</i> Z ← 1, if <i>dst</i> == 0 C ← 0 V ← 0 (see Note)	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .	
<b>Examples</b>	Before:   r0 <span style="border: 1px solid black; padding: 2px;">AB</span> <span style="border: 1px solid black; padding: 2px;">CD</span> <span style="border: 1px solid black; padding: 2px;">EF</span> <span style="border: 1px solid black; padding: 2px;">01</span> r1 <span style="border: 1px solid black; padding: 2px;">00</span> <span style="border: 1px solid black; padding: 2px;">00</span> <span style="border: 1px solid black; padding: 2px;">05</span> <span style="border: 1px solid black; padding: 2px;">67</span> ← increasing bits INSFW &11,&8,%r1,%r0 After:   r0 <span style="border: 1px solid black; padding: 2px;">AB</span> <span style="border: 1px solid black; padding: 2px;">C5</span> <span style="border: 1px solid black; padding: 2px;">67</span> <span style="border: 1px solid black; padding: 2px;">01</span>	
	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.	
<b>Note</b>	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 to bit 0 of the destination. Starting with bit 0 of <i>src</i> , ( <i>width</i> +1) bits are placed into <i>dst</i> beginning at the bit designated by <i>offset</i> . If <i>dst</i> is a byte or halfword and ( <i>width</i> + <i>offset</i> ) specifies a field that extends beyond <i>dst</i> , no bits beyond <i>dst</i> are altered but the V flag is set.	

## JMP

## JMP

### JUMP

<b>Assembler Syntax</b>	<code>JMP <i>dst</i>    Jump</code>
<b>Opcode</b>	<code>0x24    JMP</code>
<b>Operation</b>	<code>PC ← &amp;<i>dst</i></code>
<b>Address Modes</b>	<i>dst</i> all modes except literal, register, or immediate
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .
<b>Examples</b>	<code>JMP .L12</code>
<b>Note</b>	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.

## JSB

## JSB

### JUMP TO SUBROUTINE

<b>Assembler Syntax</b>	JSB <i>dst</i> Jump to subroutine
<b>Opcode</b>	0x34    JSB
<b>Operation</b>	*(SP++) ← address of next instruction PC ← & <i>dst</i>
<b>Address Modes</b>	<i>dst</i> all modes except literal, register, or immediate
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	Illegal operand exception occurs if literal, expanded-operand type, or immediate mode is used for <i>dst</i> .
<b>Examples</b>	JSB error
<b>Note</b>	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

<b>Assembler Syntax</b>	LLSB3 <i>count,src,dst</i> LLSH3 <i>count,src,dst</i> LLSW3 <i>count,src,dst</i>	Logical left shift byte Logical left shift halfword Logical left shift word								
<b>Opcodes</b>	0xD3 LLSB3 0xD2 LLSH3 0xD0 LLSW3									
<b>Operation</b>	$dst \leftarrow src \ll (count \& 0x1F) \text{ bits}$									
<b>Address Modes</b>	<i>count</i> all modes <i>src</i> all modes <i>dst</i> all modes except literal or immediate									
<b>Condition Flags</b>	N $\leftarrow$ MSB of <i>dst</i> Z $\leftarrow$ 1, if <i>dst</i> == 0 C $\leftarrow$ 0 V $\leftarrow$ 0, if result must be truncated to fit <i>dst</i> size									
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .									
<b>Examples</b>	Before: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0F</td><td>0F</td><td>DF</td><td>FD</td></tr></table> $\leftarrow$ increasing bits  LLSH3 &2,%r0,%r0  After: r0 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>FF</td><td>FF</td><td>7F</td><td>F4</td></tr></table>	0F	0F	DF	FD	FF	FF	7F	F4	
0F	0F	DF	FD							
FF	FF	7F	F4							
<b>Note</b>	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 low-order bit position (bit 0).									



**MCOMB**  
**MCOMH**  
**MCOMW**

**MCOMB**  
**MCOMH**  
**MCOMW**

### MOVE COMPLEMENTED

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



**MODB2**  
**MODH2**  
**MODW2**

**MODB2**  
**MODH2**  
**MODW2**

## MODULO

<b>Assembler Syntax</b>	MODB2 <i>src,dst</i> MODH2 <i>src,dst</i> MODW2 <i>src,dst</i>	Modulo byte Modulo halfword Modulo word
<b>Opcodes</b>	0xA7 MODB2 0xA6 MODH2 0xA4 MODW2	
<b>Operation</b>	$dst \leftarrow dst \% src$	
<b>Address Modes</b>	<i>src</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	$N \leftarrow 1$ , if $(dst \% src) < 0$ $Z \leftarrow 1$ , if $(dst \% src) == 0$ $C \leftarrow 0$ $V \leftarrow 1$ , if overflow	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .  Integer zero-divide exception occurs if <i>src</i> is equal to 0.  Integer overflow exception occurs if there is truncation.	
<b>Examples</b>	MODB2 &40,%r3 MODH2 4(%r3),%r3 MODW2 %r0,*\$result	

**MODB3**  
**MODH3**  
**MODW3**

**MODB3**  
**MODH3**  
**MODW3**

### **MODULO, 3 ADDRESS**

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

**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 Note

**Exceptions**            Illegal operand exception occurs if literal or immediate mode is used for *dst*.

**Examples**              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.

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

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

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

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

# MOVAW

# MOVAW

## MOVE ADDRESS (WORD)

<b>Assembler Syntax</b>	<code>MOVAW <i>src,dst</i></code> Move address (word)												
<b>Opcode</b>	0x04 MOVAW												
<b>Operation</b>	$dst \leftarrow \&src$												
<b>Address Modes</b>	<i>src</i> all modes except literal, register, or immediate <i>dst</i> all modes except literal or immediate												
<b>Condition Flags</b>	$N \leftarrow \text{MSB of } dst$ $Z \leftarrow 1, \text{ if } dst == 0$ $C \leftarrow 0$ $V \leftarrow 0$												
<b>Exceptions</b>	Illegal operand exception occurs if literal, register, or immediate mode is used for <i>src</i> , or if literal or immediate mode is used for <i>dst</i> .												
<b>Examples</b>	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										
<b>Note</b>	Source operand type is effective address.												

# 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.

**Examples**             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

After:    r0    

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

  
                              r1    

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

  
                              r2    

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

## MOVBLW

## MOVBLW

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**

<b>Assembler Syntax</b>	MULB2 <i>src,dst</i> MULH2 <i>src,dst</i> MULW2 <i>src,dst</i>	Multiply byte Multiply halfword Multiply word
<b>Opcodes</b>	0xAB MULB2 0xAA MULH2 0xA8 MULW2	
<b>Operation</b>	$dst \leftarrow dst * src$	
<b>Address Modes</b>	<i>src</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	$N \leftarrow 1$ , if $(dst * src) < 0$ $Z \leftarrow 1$ , if $(dst * src) == 0$ $C \leftarrow 0$ $V \leftarrow 1$ , if overflow	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> . Integer overflow exception occurs if there is truncation.	
<b>Example</b>	MULBH2 %r2,{sbyte}4(%r6)	

**MULB3**  
**MULH3**  
**MULW3**

**MULB3**  
**MULH3**  
**MULW3**

### **MULTIPLY, 3 ADDRESS**

<b>Assembler Syntax</b>	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
<b>Opcodes</b>	0xEB MULB3 0xEA MULH3 0xE8 MULW3	
<b>Operation</b>	$dst \leftarrow src1 * src2$	
<b>Address Modes</b>	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	$N \leftarrow 1$ , if $(src1 * src2) < 0$ $Z \leftarrow 1$ , if $(src1 * src2) == 0$ $C \leftarrow 0$ $V \leftarrow 1$ , if overflow	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .  Integer overflow exception occurs if there is truncation.	
<b>Examples</b>	MULH3 %r3,*\$0x1004,%r4	

## MVERNO

## MVERNO

### MOVE VERSION NUMBER

**Assembler Syntax**      MVERNO    Move processor version number

**Opcode**                0x3009    MVERNO

**Operation**             r0 ← processor version number

**Address Modes**        None

**Condition Flags**        Unchanged

**Exceptions**            None

**Example**                MVERNO

**Note**                    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 conditions bits to stabilize. The bytes following **NOP2** and **NOP3** are generated by the assembler and are ignored by the processor. They may be any value.

**ORB2**  
**ORH2**  
**ORW2**

**ORB2**  
**ORH2**  
**ORW2**

## **OR**

<b>Assembler Syntax</b>	ORB2 <i>src,dst</i> OR byte ORH2 <i>src,dst</i> OR halfword ORW2 <i>src,dst</i> OR word
<b>Opcodes</b>	0xB3    ORB2 0xB2    ORH2 0xB0    ORW2
<b>Operation</b>	$dst \leftarrow dst src$
<b>Address Modes</b>	<i>src</i> all modes  <i>dst</i> all modes except literal or immediate
<b>Condition Flags</b>	$N \leftarrow$ MSB of <i>dst</i>  $Z \leftarrow 1$ , if $dst == 0$  $C \leftarrow 0$  $V \leftarrow 1$ , if result must be truncated to fit <i>dst</i> size
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .
<b>Examples</b>	ORB2 &12,4(%fp) ORH2 %r0,4(%r0) ORW2 %r3,\$result

**ORB3**  
**ORH3**  
**ORW3**

**ORB3**  
**ORH3**  
**ORW3**

## **OR, 3 ADDRESS**

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

## POPW

## POPW

### POP (WORD)

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

## PUSHAW

## PUSHAW

### PUSH ADDRESS (WORD)

<b>Assembler Syntax</b>	PUSHAW <i>src</i> Push address (word)
<b>Opcode</b>	0xE0    PUSHAW
<b>Operation</b>	$*(SP++) \leftarrow \&src$
<b>Address Modes</b>	<i>src</i> all modes except literal, register, or immediate
<b>Condition Flags</b>	$N \leftarrow$ MSB of address of <i>src</i> $Z \leftarrow 1$ , if $src == 0$ $C \leftarrow 0$ $V \leftarrow 0$
<b>Exceptions</b>	Illegal operand exception occurs if literal, register, expanded-operand type, or immediate mode is used for <i>src</i> .
<b>Example</b>	PUSHAW 0x14(%r6)
<b>Note</b>	Source operand type 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)

<b>Assembler Syntax</b>	PUSHW <i>src</i> Push (word)
<b>Opcode</b>	0xA0    PUSHW
<b>Operation</b>	$*(SP++) \leftarrow src$
<b>Address Modes</b>	<i>src</i> all modes
<b>Condition Flags</b>	$N \leftarrow \text{MSB of } src$ $Z \leftarrow 1, \text{ if } src == 0$ $C \leftarrow 0$ $V \leftarrow 0$
<b>Exceptions</b>	Illegal operand exception occurs if expanded-operand type addressing mode is used.
<b>Example</b>	PUSHW (%r2)

## RCC

## RCC

### RETURN ON CARRY CLEAR

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

**RCS**

**RCS**

**RETURN ON CARRY SET**

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

**REQL**  
**REQLU**

**REQL**  
**REQLU**

**RETURN ON EQUAL**

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

## RESTORE

## RESTORE

### RESTORE REGISTERS

<b>Assembler Syntax</b>	RESTORE %rn    Restore registers
<b>Opcode</b>	0x18    RESTORE
<b>Operation</b>	<pre>tempa ← FP - 28; tempb ← *(FP - 28); tempc ← FP - 24; while (n != FP){ {     register[n] ← (tempc)+;     n+=1; } FP ← tempb; SP ← tempa</pre>
<b>Address Modes</b>	Register mode, where <i>n</i> ranges from 0 through 9
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	See Notes.
<b>Example</b>	RESTORE %r3
<b>Notes</b>	<p>If the operand is not register mode or <i>n</i> is not in the range 0 through 9, the results are indeterminate. Although the results are determinate if <i>n</i> is 0, 1 or 2, the effect is not that of a register restore in a function-calling sequence.</p> <p>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 <i>n</i> specifies the number of registers (9 - <i>n</i>) to be restored for the original function.</p> <p>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.</p> <p>Illegal operand exception occurs if expanded-operand type address mode is used.</p>

# 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 (RET) is the inverse of the call (CALL) instruction. A restore should precede a return (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)

<b>Assembler Syntax</b>	RGEQU    Return on greater than or equal (unsigned)
<b>Opcode</b>	0x50    REGEQU
<b>Operation</b>	if (C==0) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	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)

<b>Assembler Syntax</b>	RGTRU    Return on greater than
<b>Opcode</b>	0x54    RGTRU
<b>Operation</b>	if ((C & Z) == 0) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	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)

<b>Assembler Syntax</b>	RLEQU    Return on less than or equal (unsigned)
<b>Opcode</b>	0x5C    RLEQU
<b>Operation</b>	if ((C Z)==1) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	RLEQU

## RLSS

## RLSS

### RETURN ON LESS THAN (SIGNED)

<b>Assembler Syntax</b>	RLSS    Return on less than (signed)
<b>Opcode</b>	0x48    RLSS
<b>Operation</b>	if ((N==1) & (Z==0)) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	RLSS

## RLSSU

## RLSSU

### RETURN ON LESS THAN (UNSIGNED)

<b>Assembler Syntax</b>	RLSSU    Return on less than (unsigned)
<b>Opcode</b>	0x58    RLSSU
<b>Operation</b>	if (C==1) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	RLSSU

**RNEQ**  
**RNEQU**

**RNEQ**  
**RNEQU**

**RETURN ON NOT EQUAL**

<b>Assembler Syntax</b>	RNEQ    Return on not equal (signed) RNEQU    Return on not equal (unsigned)
<b>Opcode</b>	0x74    RNEQ 0x64    RNEQU
<b>Operation</b>	if (Z==0) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	RNEQ



## RSB

## RSB

### RETURN FROM SUBROUTINE

<b>Assembler Syntax</b>	RSB    Return from subroutine (unconditional)
<b>Opcode</b>	0x78    RSB
<b>Operation</b>	PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	RSB

**RVC**

**RVC**

**RETURN ON OVERFLOW CLEAR**

<b>Assembler Syntax</b>	RVC    Return on overflow clear
<b>Opcode</b>	0x60    RVC
<b>Operation</b>	if (V==0) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	RVC

**RVS**

**RVS**

**RETURN ON OVERFLOW SET**

<b>Assembler Syntax</b>	RVS    Return on overflow set
<b>Opcode</b>	0x68    RVS
<b>Operation</b>	if (V==1) PC ← *(--SP)
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	None
<b>Example</b>	RVS

# SAVE

# SAVE

## SAVE REGISTERS

<b>Assembler Syntax</b>	SAVE %r <i>n</i> Save registers
<b>Opcode</b>	0x10 SAVE
<b>Operation</b>	<pre>temp ← SP *(SP++) ← FP while (n != FP){     *(SP++) ← register[n]     n+=1; } SP ←temp + 28; FP ← SP;</pre>
<b>Address Modes</b>	Register mode, where <i>n</i> ranges from 0 through 9
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	See Notes.
<b>Example</b>	SAVE %r3 (see Figure 3-9)
<b>Notes</b>	<p>If the operand is not register mode or <i>n</i> is not in the range 0 to 9, the results are indeterminate. However, if <i>n</i> is 0, 1, or 2, the results are determinate, but SP and FP will not point beyond the register-save area.</p> <p>Temp is a temporary register, and <i>n</i> specifies the number of registers (9 - <i>n</i>) to be saved for the calling function.</p> <p>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 <b>Restore</b> and <b>Return from Procedure</b> 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 3-9 shows the stack after executing 'SAVE %r3'.</p> <p>Illegal operand exception occurs if expanded-operand type addressing mode is used.</p>

## SPOP

## SPOP

### COPROCESSOR OPERATION (no operands)

<b>Assembler Syntax</b>	SPOP <i>word</i> Coprocessor operation
<b>Opcode</b>	0x32 SPOP
<b>Operation</b>	<pre>/* coprocessor operation executes the following processor 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>
<b>Address Modes</b>	None valid, word = 32-bit value
<b>Condition Flags</b>	Determined by the coprocessor status
<b>Exceptions</b>	External memory fault may occur.
<b>Example</b>	SPOP 0XFFFFFFFF

**SPOPRS**  
**SPOPRD**  
**SPOPRT**

**SPOPRS**  
**SPOPRD**  
**SPOPRT**

### **COPROCESSOR OPERATION READ**

<b>Assembler Syntax</b>	SPOPRS <i>word,src</i> SPOPRD <i>word,src</i> SPOPPT <i>word,src</i>	Coprocessor operation read single Coprocessor operation read double Coprocessor operation read triple
<b>Opcode</b>	0x22 SPOPRS 0x02 SPOPRD 0x06 SPOPRT	
<b>Operation</b>	/* coprocessor operation read executes the following processor 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" }	
<b>Address Modes</b>	<i>word</i> none valid, 32-bit value <i>src</i> all modes except register, literal, or immediate	
<b>Condition Flags</b>	Determined by the coprocessor status	
<b>Exceptions</b>	External memory fault may occur.	
<b>Example</b>	SPOPRS 0xF379FFFF,*\$0xFF37 SPOPRD 0xFFFFFFFF,%r3 SPOPRT 0x00000000,(%r4)	

**SPOPS2**  
**SPOPD2**  
**SPOPT2**

**SPOPS2**  
**SPOPD2**  
**SPOPT2**

### **COPROCESSOR OPERATION, 2 ADDRESS**

**Assembler Syntax**

<b>SPOPS2</b>	<i>word,src,dst</i>	Coprocessor operation single, 2-address
<b>SPOPD2</b>	<i>word,src,dst</i>	Coprocessor operation double, 2-address
<b>SPOPT2</b>	<i>word,src,dst</i>	Coprocessor operation triple, 2-address

**Opcode**

0x23	SPOPS2
0x03	SPOPD2
0x07	SPOPT2

**Operation**

```
/* coprocessor operation executes the following
processor 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" }
{ "dst" is written with an access status of
  coprocessor data write" }
```

**Address Modes**

<i>word</i>	none valid, 32-bit value
<i>src</i>	all modes except register, literal, or immediate
<i>dst</i>	all modes except register, literal, or immediate

**Condition Flags**

Determined by the coprocessor status

**Exceptions**

External memory fault may occur.

**Example**

<b>SPOPS2</b>	0xFF,4(%r0)
<b>SPOPD2</b>	0xFFF,%r3
<b>SPOPT2</b>	0xFE,(%r0)

**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 processor 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, 32-bit value  
                         *dst* all modes except register, literal, or immediate

**Condition Flags**        Determined by the coprocessor status.

**Exceptions**            External memory fault may occur.

**Example**                SPOPWS 0x00,%r0  
                         SPOPWD 0x0F,(%r1)  
                         SPOPWT 0x1000,4(%r2)

## STRCPY

## STRCPY

### STRING COPY

<b>Assembler Syntax</b>	STRCPY String copy								
<b>Opcode</b>	0x3035 STRCPY								
<b>Operation</b>	<pre>while ((*r1 = *r0)!=0){     {disable interrupts}     r0++;     r1++;     {enable interrupts} }</pre>								
<b>Address Modes</b>	None								
<b>Condition Flags</b>	Unchanged								
<b>Exceptions</b>	External memory fault may occur in the middle of an iteration.								
<b>Examples</b>	Before: r0 <table border="1"><tr><td>00</td><td>00</td><td>01</td><td>00</td></tr></table> r1 <table border="1"><tr><td>00</td><td>00</td><td>40</td><td>00</td></tr></table> ← increasing bits	00	00	01	00	00	00	40	00
00	00	01	00						
00	00	40	00						

The byte locations starting at 0x100 contain the values 0x01, 0x24, 0xE6, 0x7F, 0x11, and 0x00 (location 0x105).

### STRCPY

After: r0	<table border="1"><tr><td>00</td><td>00</td><td>01</td><td>05</td></tr></table>	00	00	01	05
00	00	01	05		
r1	<table border="1"><tr><td>00</td><td>00</td><td>40</td><td>05</td></tr></table>	00	00	40	05
00	00	40	05		

The byte locations from 0x4000 through 0x4005 now contain the same values as locations 0x100 through 0x105.

## STRCPY

## STRCPY

### 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.



**SUBB2**  
**SUBH2**  
**SUBW2**

**SUBB2**  
**SUBH2**  
**SUBW2**

## **SUBTRACT**

<b>Assembler Syntax</b>	SUBB2 <i>src, dst</i> SUBH2 <i>src, dst</i> SUBW2 <i>src, dst</i>	Subtract byte Subtract halfword Subtract word
<b>Opcodes</b>	0xBF SUBB2 0xBE SUBH2 0xBC SUBW2	
<b>Operation</b>	$dst \leftarrow dst - src$	
<b>Address Modes</b>	<i>src</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	$N \leftarrow 1$ , if $(dst - src) < 0$ $Z \leftarrow 1$ , if $(dst - src) == 0$ $C \leftarrow 1$ , if borrow from sign bit of <i>dst</i> $V \leftarrow 1$ , if overflow	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .  Integer overflow exception occurs if there is truncation.	
<b>Examples</b>	SUBB2 %r6, *\$0x30(%r2) SUBH2 %r0, \$resulth SUBW2 %r3, \$resultw	

**SUBB3**  
**SUBH3**  
**SUBW3**

**SUBB3**  
**SUBH3**  
**SUBW3**

### **SUBTRACT, 3 ADDRESS**

<b>Assembler Syntax</b>	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
<b>Opcodes</b>	0xFF SUBB3 0xFE SUBH3 0xFC SUBW3	
<b>Operation</b>	$dst \leftarrow src2 - src1$	
<b>Address Modes</b>	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	$N \leftarrow 1$ , if $(src2 - src1) < 0$ $Z \leftarrow 1$ , if $(src2 - src1) == 0$ $C \leftarrow 1$ , if carry out of sign bit of <i>dst</i> $V \leftarrow 1$ , if overflow	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .  Integer overflow exception occurs if there is truncation.	
<b>Examples</b>	SUBB3 %r3,*\$0x1005,%r2 SUBH3 %r1,%r3,%r0 SUBW3 \$N1,\$N2,\$result	

**SWAPBI  
SWAPHI  
SWAPWI**

**SWAPBI  
SWAPHI  
SWAPWI**

**SWAP (INTERLOCKED)**

<b>Assembler Syntax</b>	SWAPBI <i>dst</i> SWAPHI <i>dst</i> SWAPWI <i>dst</i>	Swap byte (interlocked) Swap halfword (interlocked) Swap word (interlocked)
<b>Opcodes</b>	0x1F SWAPBI 0x1E SWAPHI 0x1C SWAPWI	
<b>Operation</b>	{set interlock} $tempa \leftarrow dst$ $dst \leftarrow r0$ $r0 \leftarrow tempa$	
<b>Address Modes</b>	<i>dst</i> all modes except register, literal, or immediate	
<b>Condition Flags</b>	$N \leftarrow$ MSB of $r0$ $Z \leftarrow 1$ , if $r0 == 0$ $C \leftarrow 0$ $V \leftarrow 0$	
<b>Exceptions</b>	Illegal operand exception occurs if register, literal, expanded-operand type, or immediate mode is used for <i>dst</i> .	
<b>Examples</b>	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	
<b>Note</b>	Final value of $r0$ sets the condition codes. The SAS code is read interlocked (7) for both the read and write bus transactions.	

**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**             $temp \leftarrow src - 0$

**Address Modes**        *src*    all modes

**Condition Flags**       $N \leftarrow 1$ , if *src* < 0 (signed)  
                          $Z \leftarrow 1$ , if *src* == 0  
                          $C \leftarrow 0$   
                          $V \leftarrow 0$

**Exceptions**            None

**Examples**             TSTH 14(%r2)

**Note**                    This instruction only sets condition codes. Its action is the same as a compare instruction, where the first operand is zero, such as

CMPB &0,*src2*

However, test is faster because it is one byte shorter.

**XORB2**  
**XORH2**  
**XORW2**

**XORB2**  
**XORH2**  
**XORW2**

## EXCLUSIVE OR

<b>Assembler Syntax</b>	XORB2 <i>src,dst</i> XORH2 <i>src,dst</i> XORW2 <i>src,dst</i>	Exclusive OR byte Exclusive OR halfword Exclusive OR word
<b>Opcodes</b>	0xB7 XORB2 0xB6 XORH2 0xB4 XORW2	
<b>Operation</b>	$dst \leftarrow dst \wedge src$	
<b>Address Modes</b>	<i>src</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	$N \leftarrow \text{MSB of } dst$ $Z \leftarrow 1, \text{ if } dst == 0$ $C \leftarrow 0$ $V \leftarrow 1, \text{ if result must be truncated to fit } dst \text{ size}$	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .	
<b>Examples</b>	XORB2 &40,4(%r4) XORH2 %r1,\$result XORW2 4(%r1),\$result	

**XORB3**  
**XORH3**  
**XORW3**

**XORB3**  
**XORH3**  
**XORW3**

**EXCLUSIVE OR, 3 ADDRESS**

<b>Assembler Syntax</b>	XORB3 <i>src1,src2,dst</i> XORH3 <i>src1,src2,dst</i> XORW3 <i>src1,src2,dst</i>	Exclusive OR byte, 3 address Exclusive OR halfword, 3 address Exclusive OR word, 3 address
<b>Opcodes</b>	0xF7 XORB3 0xF6 XORH3 0xF4 XORW3	
<b>Operation</b>	$dst \leftarrow src1 \wedge src2$	
<b>Address Modes</b>	<i>src1</i> all modes <i>src2</i> all modes <i>dst</i> all modes except literal or immediate	
<b>Condition Flags</b>	N $\leftarrow$ MSB of <i>dst</i> Z $\leftarrow$ 1, if <i>dst</i> == 0 C $\leftarrow$ 0 V $\leftarrow$ 1, if result must be truncated to fit <i>dst</i> size	
<b>Exceptions</b>	Illegal operand exception occurs if literal or immediate mode is used for <i>dst</i> .	
<b>Examples</b>	XORB3 &4,*12(%r3),*\$0x400 XORH3 %r1,4(%r1),%r0 XORW3 %r0,%r1,%r3	



**Chapter 6**

**Operating System  
Considerations**

## CHAPTER 6. OPERATING SYSTEM CONSIDERATIONS

### CONTENTS

<ul style="list-style-type: none"> <li>6. OPERATING SYSTEM CONSIDERATIONS ..... 6-1</li> <li>6.1 FEATURES OF THE OPERATING SYSTEM ..... 6-1 <ul style="list-style-type: none"> <li>6.1.1 Memory Management Considerations for Virtual Memory Systems ..... 6-3</li> </ul> </li> <li>6.2 STRUCTURE OF A PROCESS ..... 6-4 <ul style="list-style-type: none"> <li>6.2.1 Execution Privilege ..... 6-4</li> <li>6.2.2 Execution Stack ..... 6-5</li> <li>6.2.3 Process Control Block ..... 6-6 <ul style="list-style-type: none"> <li>Initial Context for a Process ... 6-9</li> <li>Saved Context for a Process ... 6-10</li> <li>Memory Specifications ..... 6-10</li> </ul> </li> <li>6.2.4 Processor Status Word ..... 6-11</li> </ul> </li> <li>6.3 SYSTEM CALL ..... 6-11 <ul style="list-style-type: none"> <li>6.3.1 Gate Mechanism ..... 6-14 <ul style="list-style-type: none"> <li>Pointer Table ..... 6-14</li> <li>Handling-Routine Tables ..... 6-14</li> </ul> </li> <li>6.3.2 GATE Instruction ..... 6-15 <ul style="list-style-type: none"> <li>First Entry Point ..... 6-15</li> <li>Second Entry Point - The Gate Mechanism ..... 6-16</li> </ul> </li> <li>6.3.3 Return-From-Gate Instruction ..... 6-18</li> </ul> </li> <li>6.4 Process Switching ..... 6-18 <ul style="list-style-type: none"> <li>6.4.1 Context Switching Strategy .... 6-19 <ul style="list-style-type: none"> <li>R Bit ..... 6-19</li> <li>I Bit ..... 6-19</li> </ul> </li> <li>6.4.2 Call Process Instruction ..... 6-22</li> <li>6.4.3 Return-to-Process Instruction ..... 6-24</li> </ul> </li> <li>6.5 INTERRUPTS ..... 6-25 <ul style="list-style-type: none"> <li>6.5.1 Interrupt-Handler Model ..... 6-25</li> <li>6.5.2 Interrupt Mechanism ..... 6-26 <ul style="list-style-type: none"> <li>Full-Interrupt Handler's PCB ..... 6-27</li> <li>Interrupt Stack and ISP ..... 6-28</li> <li>Interrupt-Vector Table ..... 6-29</li> </ul> </li> <li>6.5.3 On-Interrupt Microsequence ... 6-30</li> <li>6.5.4 Returning From an Interrupt .. 6-31 <ul style="list-style-type: none"> <li>Full Interrupts ..... 6-31</li> <li>Quick Interrupts ..... 6-31</li> </ul> </li> </ul> </li> <li>6.6 EXCEPTIONS ..... 6-32 <ul style="list-style-type: none"> <li>6.6.1 Levels of Exception Severity ... 6-32</li> <li>6.6.2 Exception Handler ..... 6-32</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>6.6.3 Exception Microsequences ..... 6-33 <ul style="list-style-type: none"> <li>Normal Exceptions ..... 6-33</li> <li>Stack Exceptions ..... 6-36</li> <li>Process Exceptions ..... 6-37</li> <li>Reset Exceptions ..... 6-37</li> </ul> </li> <li>6.7 MEMORY MANAGEMENT FOR VIRTUAL MEMORY SYSTEMS ..... 6-38 <ul style="list-style-type: none"> <li>6.7.1 Initializing the Memory Management Unit ..... 6-43 <ul style="list-style-type: none"> <li>Defining Virtual Memory ..... 6-43</li> <li>Peripheral Mode ..... 6-43</li> </ul> </li> <li>6.7.2 MMU Interactions ..... 6-43 <ul style="list-style-type: none"> <li>MMU Exceptions ..... 6-44</li> <li>Flushing ..... 6-44</li> </ul> </li> <li>6.7.3 Efficient Mapping Strategies ... 6-44</li> <li>6.7.4 Object Traps ..... 6-45</li> <li>6.7.5 Indirect Segment Descriptors .. 6-45</li> <li>6.7.6 Using the Cacheable Bit ..... 6-45</li> <li>6.7.7 Using the Page-Write Fault .... 6-45</li> <li>6.7.8 Access Protection ..... 6-46</li> <li>6.7.9 Using the Software Bits ..... 6-46</li> </ul> </li> <li>6.8 OPERATING SYSTEM INSTRUCTIONS ..... 6-46 <ul style="list-style-type: none"> <li>6.8.1 Notation ..... 6-46</li> <li>6.8.2 Privileged Instructions ..... 6-48 <ul style="list-style-type: none"> <li>Call Process (CALLPS) ..... 6-49</li> <li>Disable Virtual Pin and Jump (DISVJMP) ..... 6-51</li> <li>Enable Virtual Pin and Jump (ENBVJMP) ..... 6-52</li> <li>Interrupt Acknowledge (INTACK) ..... 6-53</li> <li>Return to Process (RETPS) ... 6-54</li> <li>Wait (WAIT) ..... 6-56</li> </ul> </li> <li>6.8.3 Nonprivileged Instructions ..... 6-57 <ul style="list-style-type: none"> <li>Gate (GATE) ..... 6-58</li> <li>Move Translated Word (MOVTRW) ..... 6-61</li> <li>Return from Gate (RETG) .... 6-63</li> </ul> </li> <li>6.8.4 Microsequences ..... 6-65 <ul style="list-style-type: none"> <li>On-Normal Exception ..... 6-66</li> <li>On-Stack Exception ..... 6-68</li> <li>On-Process Exception ..... 6-69</li> <li>On-Reset Exception ..... 6-70</li> <li>On-Interrupt ..... 6-71</li> <li>XSWITCH Microsequence .... 6-74</li> </ul> </li> </ul> </li> </ul>
---	---

## **6. OPERATING SYSTEM CONSIDERATIONS**

The *WE 32100* 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 which will allow the system designer to make full use of the power of the *WE 32100* Microprocessor as an execution vehicle for today's efficient process-oriented operating systems.

The typical operating system for the *WE 32100* 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 either the call-process (CALLPS) or return-to-process (RETPS) privileged 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 32101* Memory Management Unit (MMU), an essential part of a virtual memory-based operating system using the *WE 32100* 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 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 the level 3 activities.
- User (level 3) - The least privileged level; most user programs can run in this level.

## OPERATING SYSTEM CONSIDERATIONS

### Features of the Operating System

Table 6-1 lists the powerful *WE* 32100 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 **6.8 Operating System Instructions**.

Instruction	Mnemonic	Opcode	Bytes	Cycles	Conditions*
<b>Nonprivileged Instructions:</b> Move translated word	MOVTRW	0x0C	9	**	Case 7
Gate	GATE	0x3061	2	134—136	Case 8
Return from gate	RETG	0x3045	2	69—71	Case 9
<b>Privileged Instructions:</b> Enable virtual pin and jump	ENBVJMP	0x300D	2	**	Unchanged
Disable virtual pin and jump	DISVJMP	0x3013	2	**	
Call process	CALLPS	0x30AC	2	†	Case 8
Return to process	RETPS	0x30C8	2	‡	Case 9
Wait for interrupt	WAIT	0x2F	1	**	Unchanged
Interrupt acknowledge	INTACK	0x302F	5—6	**	

\* Refer to Table 5-11 for condition flag code assignments.

\*\* Not Applicable

† 224—226 cycles with R-bit clear block moves; 302—304 cycles with R-bit clear and no block moves.

‡ 189—191 cycles with R-bit clear block moves; 277—279 cycles with R-bit clear and no block moves.

## OPERATING SYSTEM CONSIDERATIONS

### Memory Management Considerations for Virtual Memory Systems

Other features of the microprocessor's architecture that are provided for operating system design are summarized as follows:

- 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 **6.3.3 Return-From-Gate Instruction**.
- The microprocessor provides flexibility in transferring execution control between privilege levels. Control is transferred through the gate mechanism, as discussed in **6.3 System Call**.
- A scheduler may explicitly switch processes (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 processor supports a layered exception-handling structure that uses different mechanisms (process switching or gate mechanism), depending on the severity of the exception.
- The processor 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.
- 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 processor supports memory management, permitting the user to believe the system has 4 Gbytes of memory. However, the operating system must provide the information required by a memory management unit to translate virtual addresses (i.e., memory descriptors) or 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 address into physical addresses and implement the protection of data for each process. A virtual memory operating system is supported by a variety of features.

- Support of contiguous and paged segments. Segments, or blocks of memory, are defined by memory descriptors. The WE 32101 Memory Management Unit uses segment descriptors to define contiguous segments (i.e., a block of memory defined 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).
- *Present* bits to indicate whether or not a segment is currently in main memory.
- *Referenced* and *modified* bits to aid implementation of a least recently used (LRU) algorithm in the operating system.

## OPERATING SYSTEM CONSIDERATIONS

### Structure of a Process

- 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 6.7.5 **Indirect Segment Descriptors**).
- Access fields contained in segment descriptors are 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.
- An *object-trap* feature provides a mechanism in which I/O devices or external processors appear as normal segments from the user-software point of view.
- Segment marking as cacheable or not cacheable using a cacheable bit. This can be an aid in the use of an external data cache in the system main memory (see 6.7.6 **Using the Cacheable Bit**).
- A unique exception (page-write) that can be issued on any attempt to write a given page (see 6.7.7 **Using the Page-Write Fault**).

## 6.2 STRUCTURE OF A PROCESS

Each process executing in the *WE* 32100 Microprocessor consists of the following elements:

- A processor status word (PSW) - the privileged CPU register contains status information on both the instruction just executed and the current process.
- A process control block (PCB) - a process data structure in external memory that contains the hardware context of a process when the process is not executing. 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.
- A process control block pointer (PCBP) - the privileged CPU register that 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 processor 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. However, a controlled return requires a transfer to a less privileged execution level. See 6.3 **System Call** 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 processor 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 which is 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 processor is not in kernel mode, it generates a normal exception (privileged register) when an instruction tries to write to one of them. 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 32100* Microprocessor has only one execution stack per process. Other processors 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 32100* 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.

## OPERATING SYSTEM CONSIDERATIONS

### Process Control Block

Before executing a transfer to a more privileged level through a *system call or gate*, the processor 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 processor handles 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 processor 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 processor 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 normal exception.

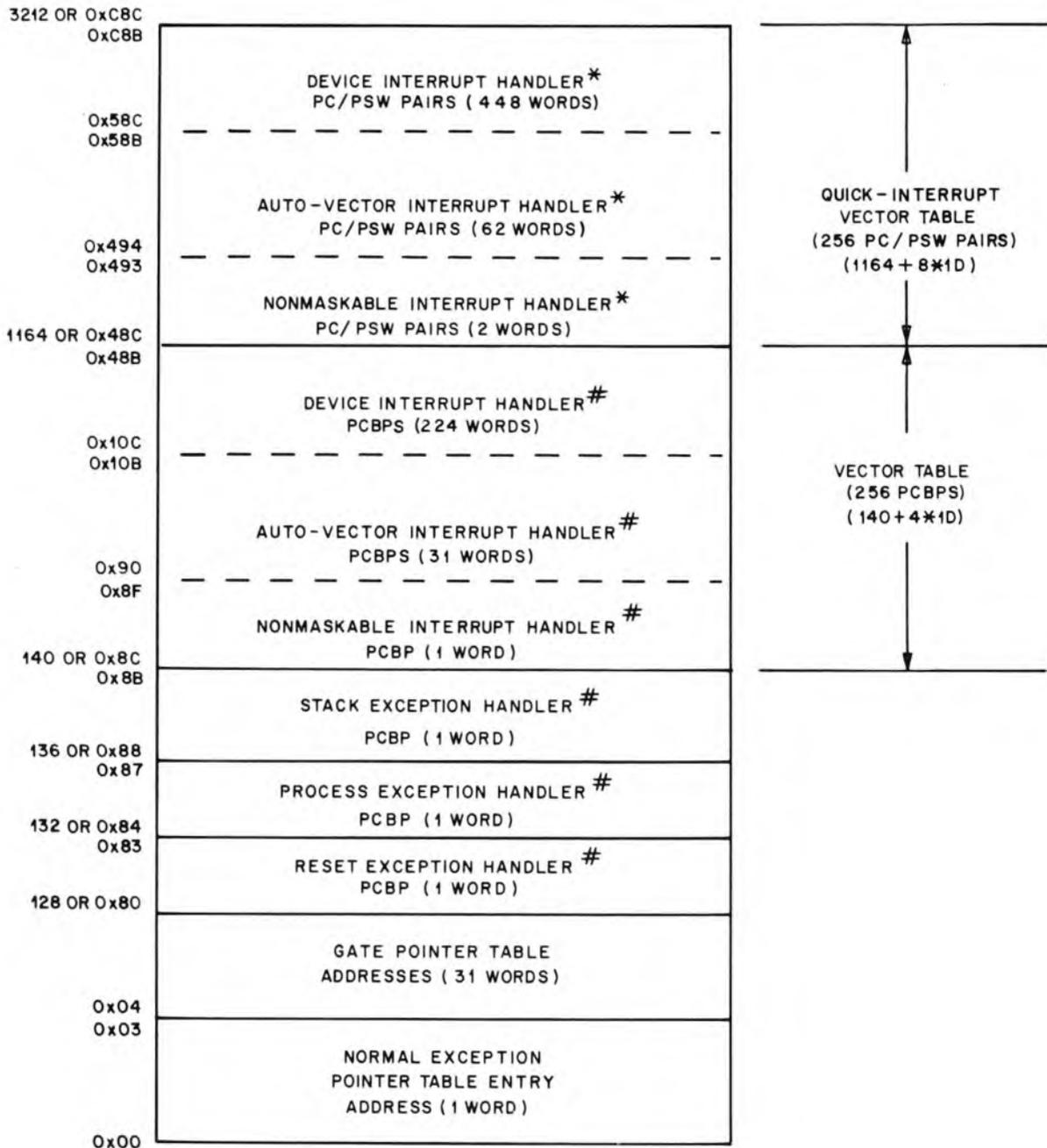
### 6.2.3 Process Control Block

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. The PCBP is taken from the top of the interrupt stack when RETPS is executed.

# OPERATING SYSTEM CONSIDERATIONS

## Process Control Block



Note:

\*Implicit Gate

#Implicit Process Switch

**Figure 6-1. Memory Map**

## OPERATING SYSTEM CONSIDERATIONS

### Process Control Block

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. The PCB, illustrated on Figure 6-2, 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 (1).
- 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 clear (0).

**Note:** If the I bit in the PSW of a process is initially set (1), execution starts from its initial-context values. If the bit is clear (0), execution resumes from an intermediate context. See **6.4.1 Context Switching Strategy** for more information on the I bit.

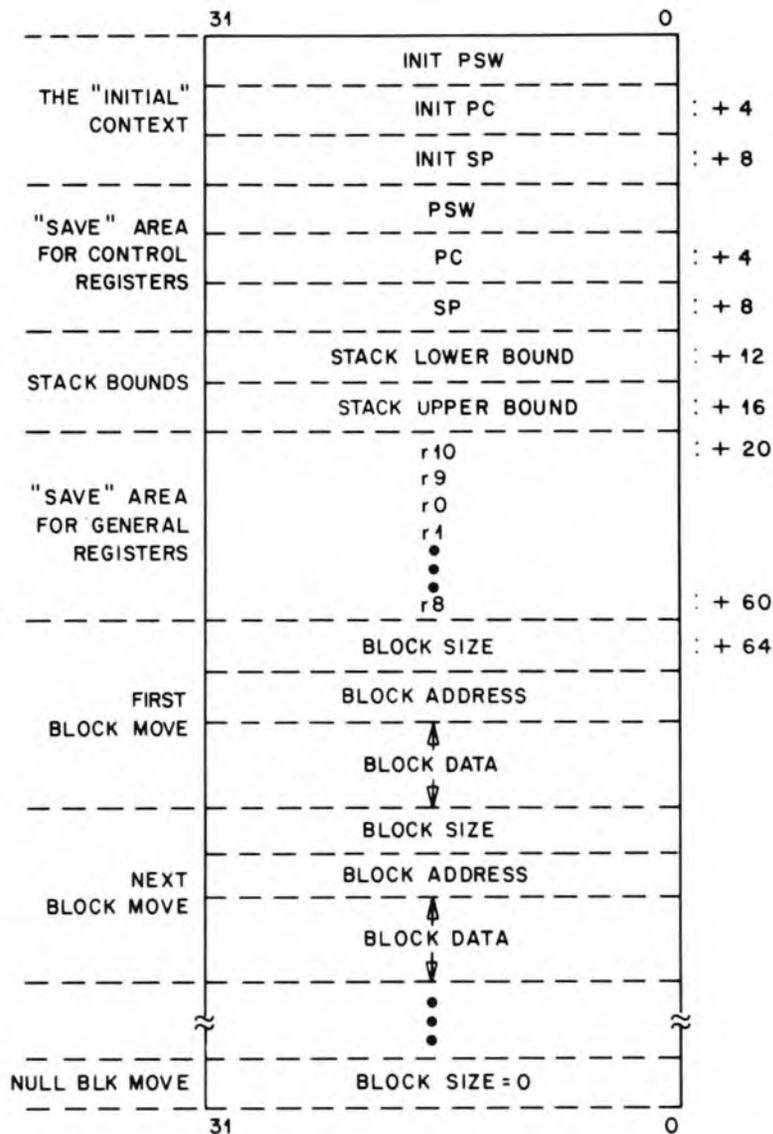
- Stack bounds. The upper and lower stack bounds define the area allocated to the execution stack for this process.
- General register save area. This area is reserved for saving the contents of register r0 through r10. Registers r9 and r10 are the frame pointer (FP) and argument pointer (AP), respectively. 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.

**Note:** The R bit of the PSW must be set (1) 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 **6.4.1 Context Switching Strategy** 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 (DISVJMP), enable or disable the processor's virtual address pin ( $\overline{\text{VAD}}$ ) to tell the MMU it is generating virtual (enable) or physical (disable) addresses. Before the instruction ENBVJMP or DISVJMP is executed, the virtual or physical, respectively, address of next instruction to be executed must be stored in r0.

## OPERATING SYSTEM CONSIDERATIONS

### Initial Context for a Process



**Figure 6-2. A Typical Process Control Block**

### Initial Context for a Process

The initial context of the executing process is set up as follows:

- The PCBP, stored in memory, points to the initial-context area of its PCB.
- The initial PSW occupies the first PCB location, and its I bit should be set (1) 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 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 **6.2.4 Processor Status Word** and **6.4.1 Context Switching Strategy** for more details about the R and I bits.

## OPERATING SYSTEM CONSIDERATIONS

### Saved Context for a Process

- The second PCB word, the initial PC, is the address of the first instruction that process executes.
- The third PCB word contains the initial SP (the address of the first location on the execution stack).
- 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 processor uses the current PCBP to save the context of Process A (the executing process) in the current PCB. Using offsets from PCBP to access the correct PCB location for process A, the processor stores PC, PSW, SP, and if the PSW R bit is set to 1, the general registers. 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 (1), the processor, 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 where the processor 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 processor 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 processor 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 processor 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 **6.6 Exceptions**. 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 processor implicitly alters some fields during normal execution at other levels. In particular, most instructions change the condition flags.

### 6.3 SYSTEM CALL

The system-call (*gate*) mechanism provides a means of controlled entry into a function by installing a new PSW and PC value. If the new PSW has a different privilege level than the current PSW, a transition to a different execution level occurs.

On simpler processors, 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 **6.3.1 Gate Mechanisms**.

**OPERATING SYSTEM CONSIDERATIONS**  
**System Call**

<b>Table 6-2. Processor Status Word Fields</b>			
<b>Bit(s)</b>	<b>Field</b>	<b>Contents</b>	<b>Description</b>
0-1	ET	Exception Type	This read-only field indicates the type of exception generated during operations and is interpreted as: <b>Code Description</b> 00 On Reset Exception 01 On Process Exception 10 On Stack Exception 11 On Normal Exception
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 (1) at the beginning of every instruction and cleared (0) 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: <b>Code Description</b> 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.

<b>Table 6-2. Processor Status Word Fields (Continued)</b>			
Bit(s)	Field	Contents	Description
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 microprocessor; level 1111, the highest interrupt priority level, indicates that no interrupts (except a nonmaskable interrupt) can interrupt the microprocessor.
17	TE	Trace Enable	This bit enables the trace function. When TE is set (1), 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 using the conditional branch instructions and indicate the following when set (1): 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 (0) 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 (1), the cache is not used. Changes to the state of the CD bit via instructions with the PSW as an explicit destination may corrupt the contents of the instruction cache. Therefore, only microsequence instructions should be used to change the CD bit state.
24	QIE	Quick-Interrupt Enable	The QIE enables and disables the quick-interrupt facility. If QIE is set (1), an interrupt is handled via the quick-interrupt sequence.
25	CFD	Cache Flush Disable	When this bit is set (1), it disables cache flushing (emptying of the instruction cache contents) during the XSWITCH_TWO microsequence.
26-31		Unused	These bits are not used and are always cleared (0).

## OPERATING SYSTEM CONSIDERATIONS

### Gate Mechanism

#### 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 processor 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.

### 6.3.2 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 only to execute 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 6.6 Exceptions). 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. Only the five least significant bits of r0 are used.
- 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.

An example of indexing for the GATE instruction is shown on Figure 6-3.

The on-normal exception microsequence is modeled after a GATE. On a normal exception, the CPU supplies all appropriate information needed to execute a GATE-like sequence.

The GATE instruction executes the following tasks in sequence.

#### First Entry Point

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 SP is outside either boundary, a stack exception (stack) 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 processor generates a stack exception (stack).
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.)

## OPERATING SYSTEM CONSIDERATIONS

### Second Entry Point

#### Second Entry Point - The Gate Mechanism

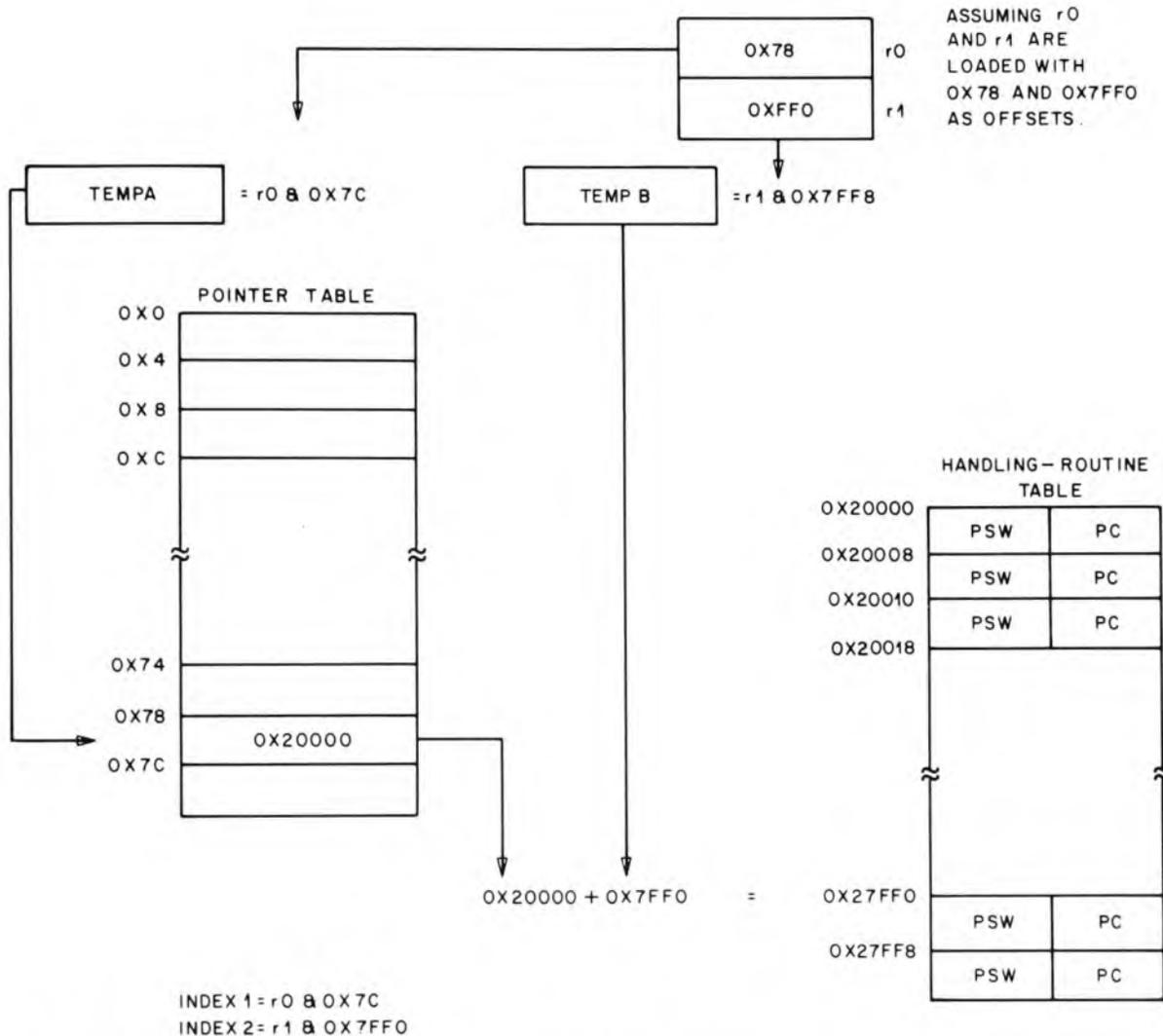
1. GATE again forces kernel execution level for memory accesses.
2. GATE uses tempa as a pointer to read the starting address of a handling-routine table from the pointer table and write it to tempa. It then adds tempa and tempb (the offset into the handling-routine table) and stores the result, index2, in tempb. 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 OE, NZVC, TE, CM, R, and I. It then sets PSW fields ISC, TM, and ET to 7, 0, and 3, respectively.
4. GATE uses index2 to locate and load the new PC.
5. 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 processor 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



Note: 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 6-3. Pointer and Handling Table Indexing**

## OPERATING SYSTEM CONSIDERATIONS

### Return from Gate Instruction

#### 6.3.3 Return-From-Gate Instruction

The return-from-gate (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. Sets the trace mask (TM) bit in PSW to zero.
3. 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 microprocessor issues a normal exception (illegal level change).
4. Writes the PSW fields OE, NZVC, TE, CM, PM, R, and I using the values in tempa (the saved PSW).
5. Loads PC from tempb.
6. Adjusts SP to the location below the saved PSW and PC (thus completing a pop of the PSW and PC from the stack).
7. 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 32100 Microprocessor invokes a process switch by automatically saving or restoring a process' context. However, a PCB only defines hardware context (as described in **6.2.3 Process Control Block**), 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, the call process (CALLPS) instruction performs both the save of the previous context and load of the new process context. The processor does not accept interrupts until the CALLPS 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, then the interrupt-handling scheme causes the saved PCB context to be overwritten. This cannot happen with the WE 32100 Microprocessor.

### 6.4.1 Context Switching Strategy

The process-switch mechanism uses two PSW parameter bits, R 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 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 processor uses the R 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 explained 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, it signifies that 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 CALLPS (call process instruction) or simulated CALLPS is performed, the processor saves 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 or requires block moves. The R bit setting never changes, even though a process switches in and out many times.

#### 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.

Consider two processes: process A, the current or old process, and process B, the incoming or new process. The function of the I bit is explained as follows:

- On leaving process A, the microprocessor always writes the PC, SP, and PSW values starting at the location pointed to by process A's PCBP and then saves process A's PCBP on the interrupt stack. On entry to process B, the microprocessor 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, full interrupts, and exceptions that perform a process switch.

## OPERATING SYSTEM CONSIDERATIONS

### I Bit

- If the I bit is set (1) 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 set to zero. 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 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 instruction.

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.

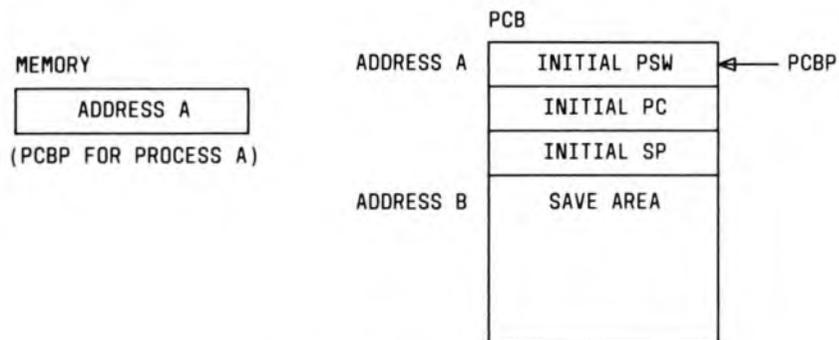
When process A is called initially by the CALLPS instruction (an explicit process switch), the processor loads the PCBP register with the starting address (address A) of the process A PCB (see part A of Figure 6-4). It then loads the PSW, the program counter, and the stack pointer with their initial context. Next, if the I bit in the PSW is set (1), the processor clears the I bit and increments the PCBP register by twelve bytes to the saved-context area (address B) of the process A PCB (see part B of Figure 6-4). This will cause any later process switch to save PSW, PC, and SP values in the intermediate context area instead of overwriting the initial-context values. The process A initial-context area and its PCBP stored in memory are not affected on this process switch.

Part A of Figure 6-5 shows the effect on the PCBP and the process A PCB if a process switch occurs before process A is finished. Here, the processor 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 stores the PCBP on the interrupt stack. This time, the PSW I bit will be clear and the PC points to the next process A instruction.

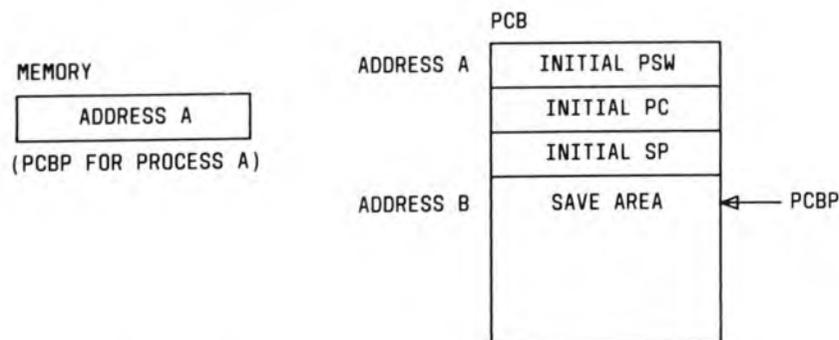
When the processor restores process A (see part B of Figure 6-5), the processor retrieves the PCBP from the interrupt stack. Remember that the PCBP points to the saved-context

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 processor 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 (0), then 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 on Figures 6-4 and 6-5 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 RETPS (return-to-process) instruction. Also, a process that had an initial I bit value of zero is restarted from an intermediate context on any subsequent CALLPS instruction after it was first switched to. A process starts from its initial context (initial I bit value is set) whenever a CALLPS instruction is executed.



**A. Context at Start of Switch to Process A**

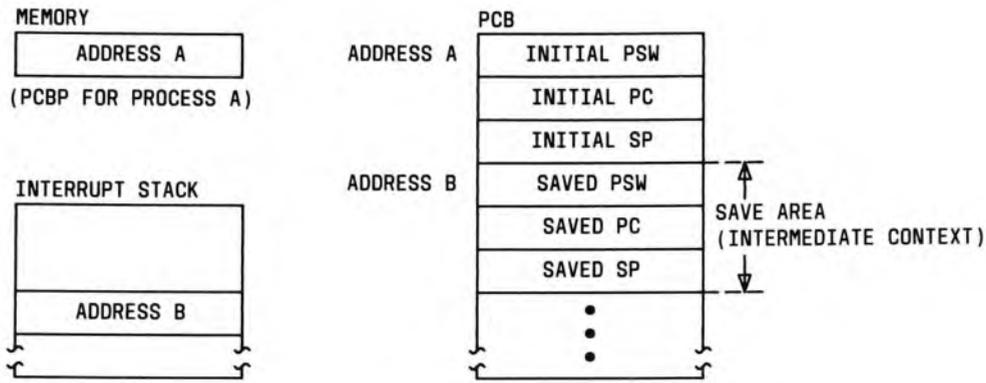


**B. Context After Switch to Process A**

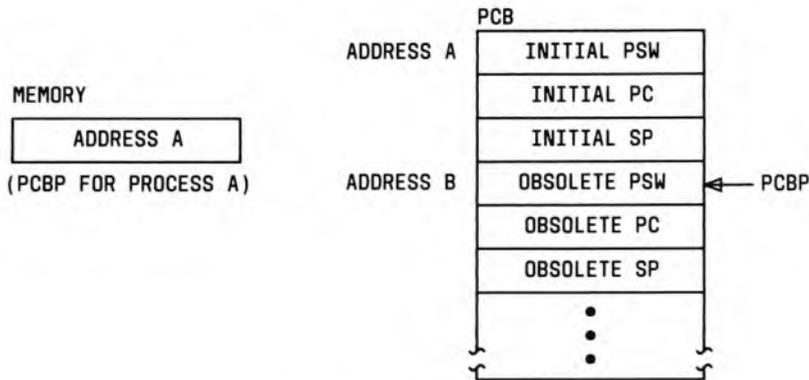
**Figure 6-4. A PCB on an Initial Process Switch to a Process**

# OPERATING SYSTEM CONSIDERATIONS

## Call Process Instruction



### A. Context After Switch to Some Other Process



### B. Context After Process A Is Switched Back to and Restored

Figure 6-5. A PCB on a Process Switch During Execution of a Process

## 6.4.2 Call Process Instruction

The call process (CALLPS) instruction, mentioned in the discussion of the R and I bits, is the process analog of the call procedure (CALL) and save registers (SAVE) instructions that carry out a function call. To execute CALLPS, the processor 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 **6.5.2 Interrupt Mechanism**). If a memory exception occurs when accessing this stack, the processor 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 context. (All writes are made to the saved-context area of process PCB because the I bit of an executing process PSW is always clear.) If a memory exception occurs on a PCB access, the processor issues a process exception (old PCB).

XSWITCH\_ONE does the following:

- a. Using tempa as a pointer to the process B PCB, it copies the R bit from the new PSW into the R bit of the current PSW. (The R bit will be used later.)
  - b. It 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.
  - c. It saves PC (address of the next instruction) and SP in the process A PCB.
  - d. It writes r0 through r10 to the general register area of the process A PCB if the R bit of the process B PSW is set. Otherwise, these registers are not saved.
  - e. It 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 processor issues a process exception (new PCB).

XSWITCH\_TWO does the following:

- a. It loads PCBP from tempa (which contains process B's PCBP value).
  - b. It reads in the new PSW and sets its TM bit to 0. Next, it loads the new PC and SP. PC now contains the address of the first instruction for Process B.
  - c. It 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 Process B's PCB.
  - d. It returns control to CALLPS.
6. Writes 7, 0, 3 to the ISC, TM, and ET fields, respectively, of the PSW.

## OPERATING SYSTEM CONSIDERATIONS

### Return-to-Process Instruction

7. Calls the function `XSWITCH_THREE()` for block moves.

`XSWITCH_THREE` does the following:

- a. It tests the R bit in the PSW.
  - If the R bit is set, it loads the block-move information 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, r1 with the size of the block (number of words to be moved), and r2 with the destination of the move. Then it executes a move block (`MOVBLW`) instruction.
  - If the R bit is clear (0), no block moves are performed.
- b. It returns control to `CALLPS`.

8. Releases kernel execution level on memory accesses and Process B begins executing.

#### 6.4.3 Return-to-Process Instruction

The `RETPS` instruction restores a process from its interrupted state and may be executed only when the processor is in kernel mode. An attempt to execute the `RETPS` instruction while not in kernel mode will generate a normal exception (privileged instruction). `RETPS` is the process analog of a function return that uses the restore registers (`RESTORE`) and return-from-procedure (`RET`) instructions. Again, the R and I bits in the PSW determine the context-switching strategy.

The `CALLPS` and `RETPS` instructions act similarly, except the `RETPS` does not save the context of the exiting process. For this discussion, process A is the returned-to-process. `RETPS` performs the following sequential steps:

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 processor issues a reset exception (interrupt-stack).
2. Loads the PSW R bit with R bit from `tempa`.
3. Calls `XSWITCH_TWO()` to restore the Process A context. If a memory exception occurs when accessing its PCB, 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. It loads PCBP from `tempa`.
  - b. It loads PSW from the PCB, writes a 0 to the TM bit, and then loads PC and SP. Because this is a return process, the I bit is clear and all control registers are loaded from the saved-context area of its PCB.
  - c. It returns control to `RETPS`.
4. Writes 7, 0, 3 to the ISC, TM, and ET fields, respectively, of PSW.

5. If R bit is set (1), calls XSWITCH\_THREE() to perform any block moves.  
XSWITCH\_THREE does the following:
  - a. It tests the R bit in the PSW.
    - If the R bit is set (1), it does the block moves in the block-move areas of the process A PCB. For each block to be moved, r0 gets the starting address of a block-move area in the PCB, r1 gets the size of the block (number of words to be moved), and r2 gets the destination of the move. Then the function executes a move block instruction (MOVBLW).
    - If the R bit is clear (0), no block moves are performed.
  - b. It returns control to RETPS.
6. If the R bit is set (1), RETPS loads r0—r10 from the general register save area of process A PCB.
7. Releases kernel execution level on memory accesses and process A resumes executing.

## 6.5 INTERRUPTS

When an external device requests an interrupt, a processor 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 modeled after a gate (system call) or process switch. 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 32100* Microprocessor uses either the process switch or gate switch 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 switch model, an interrupt (called a quick interrupt in this case) causes an implicit gate to a handler function. When full interrupts are used, the processor interrupt mechanism meets the isolation and execution-level requirements because each interrupt handler is a separate process with its own execution stack. The processor 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.

## OPERATING SYSTEM CONSIDERATIONS

### Interrupt Mechanism

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 whether or not there will be an interrupt.
- Determining how an interrupt request will be acknowledged and what the interrupt-ID value is.
- Saving the old context and bringing in a new context.

The first part involves checking the  $\overline{\text{NMINT}}$  and  $\text{IPL}[3-0]$  pins, and the  $\text{IPL}$  field of the PSW. The next part involves the  $\overline{\text{NMINT}}$ ,  $\overline{\text{AVEC}}$ ,  $\text{IPL}[3-0]$  and  $\overline{\text{INTOPT}}$  pins, and an *interrupt acknowledge* or *auto-vector interrupt acknowledge* bus cycle. The final part involves the  $\text{QIE}$  field of the PSW and a quick-interrupt (gate-like) sequence or a full-interrupt (process-switch) sequence.

The following algorithm describes the interrupt behavior. The notation used is:

- $\text{INT}==1$  if there is to be an interrupt
- $\text{ID}$  is the value of the interrupt-ID in the on-interrupt microsequence
- $\text{NMI}$ ,  $\overline{\text{INTOPT}}$ , and  $\overline{\text{AVEC}}$  represent the complements of the values of the nonmaskable interrupt ( $\overline{\text{NMINT}}$ ), interrupt option ( $\overline{\text{INTOPT}}$ ), and auto-vector ( $\overline{\text{AVEC}}$ ) pins, respectively.

```
I=0;
if( $\overline{\text{NMINT}}==0$ ) {
    INT=1;
    ID=0;
}
else if((requested_interrupt_level) > (PSW <IPL>)) {
    INT=1;
    if( $\overline{\text{AVEC}}==0$ )
        ID=(INTOPT concatenated with interrupt request level);
    else ID=(value fetched in interrupt acknowledge cycle);
}
if(INT==1) {
    call on-interrupt microsequence;
}
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  $PSW < IPL > \neq 15$ , no interrupts will be acknowledged (except the nonmaskable interrupt).

After acknowledging an interrupt the processor performs its on-interrupt microsequence (an implicit process or gate switch). Its actions are similar to a CALLPS instruction for a full interrupt and a gate (GATE) instruction for a quick interrupt, but with a few differences.

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 **6.4.1 Context Switching Strategy**).

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 **6.4.3 Return-to-Process Instruction**).

When a quick interrupt activates an interrupt handler, the current PC and PSW values are stored on the execution stack. A simulated gate is then performed to load the PC and PSW registers with the initial information for the interrupt handler. A quick-interrupt gate does not perform any stack bounds check; therefore, quick interrupts should not occur in processes where the stack may be bad (e.g., a user process with a stack that is unreliable). Also, a quick-interrupt gate sets the PSW interrupt priority level (IPL) field to 15, thus disabling all interrupts except a nonmaskable interrupt.

Only a nonmaskable interrupt may interrupt the current quick-interrupt handler. When this happens, the PC and PSW values of the interrupted handler are stored on the execution stack and another simulated gate is performed. Thus, the interrupted handler can resume execution from its interrupted state.

To return from a quick interrupt, an interrupt handler should restore the IPL field in the PSW and then execute an RETG instruction (see **6.3.3 Return-From-Gate 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

## **OPERATING SYSTEM CONSIDERATIONS**

### **Interrupt Stack and ISP**

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 should contain 1. If the interrupt handler wants to use the general registers, the PSW R bit should be 1.

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.

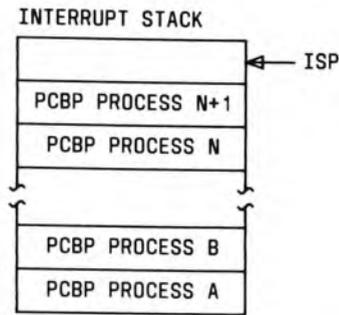
An interrupt handler's MMU map specification, if maintained in the PCB block-move areas, is used when loading an initial context or restoring an intermediate context. Therefore, the user must ensure that the operating system restores the map data to its initial state before a return-from-interrupt. This can be done by maintaining appropriate R bit 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 processor to track the nesting of interrupt handlers and active processes and is never used as an execution stack.

The processor 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. An attempt to write this register other than in kernel level causes a normal exception (privileged register).

At any level of full-interrupt handling, the interrupt stack contains the PCBPs for all lower priority interrupt handlers that were interrupted while executing. The entry at the bottom of the stack is the PCBP for the process that was interrupted by the first interrupt handler (see Figure 6-6).



PROCESS B INTERRUPTED PROCESS A.  
PROCESS N+1 IS LAST PROCESS INTERRUPTED.

Figure 6-6. An Interrupt Stack

Because an RETPS restores the process that was interrupted, the process at the bottom of the stack is eventually restored. However, any interrupt handler whose PCBP is on this stack may force a return to a different process. If any interrupt handler does this, be sure that it overwrites the normal-process PCBP at the bottom of this stack with the PCBP of the desired process.

### Interrupt-Vector Table

The user must provide interrupt-vector tables for full and quick interrupts, depending on how interrupts are to be handled (process switches and/or gates). Figure 6-1 shows the memory locations where interrupt PCBPs and PC/PSW pairs must be stored. If the nonmaskable and auto-vector interrupts are not used, those locations can be used to store the PCBPs for device-interrupt handlers. The full-interrupt-vector table starts at location 140 (8C hex) to store the PCBP (up to 256 PCBPs) 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 processor 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.

## OPERATING SYSTEM CONSIDERATIONS

### On-Interrupt Microsequence

#### 6.5.3 On-Interrupt Microsequence

The on-interrupt microsequence is a sequence of actions built into the *WE 32100* Microprocessor that responds to interrupts. The on-interrupt microsequence handles both full and quick interrupts. For full interrupts, the processor performs an implicit process switch. For quick interrupts, the processor performs a GATE-like PSW/PC switch. Here, process A is the interrupted process and process B is the interrupt handler. (See 6.4.2 **Call Process Instruction** for descriptions 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 processor 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 to 1).
4. Performs steps 5 through 11 for a full interrupt.
5. Forms an index  $140+4*\textit{tempa}$ , 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 processor 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 processor generates a process exception (old-PCB).
8. Calls XSWITCH\_TWO () to load the process B 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 will be cleared and PCBP adjusted to the saved-context area of process B PCB.
9. Writes 7, 0, 3 to the PSW's ISC, TM, and ET fields, 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 index,  $1164+\textit{tempa}*8$ , which is written to *tempa*. This index is used to locate the PSW and PC of the appropriate interrupt handler.
14. Releases kernel level on memory accesses.
15. Pushes the PSW and PC of process A onto the execution stack.
16. Forces kernel level on memory accesses.

17. Sets the PSW with a value indexed by tempa, and PC with a value indexed by 4+tempa. Some fields in the PSW are unchanged. Additionally, the IPL field is set to 15 to mask any subsequent interrupts. If a memory exception occurs, a normal exception (gate vector) is generated.
18. Releases kernel level on memory accesses. For quick interrupts this is the last step of the on-interrupt microsequence.

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**

The interrupted process can be restored after either a field or quick 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 instruction. Unlike the call process, RETPS 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 RETPS 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 execution stack). To return from a quick-interrupt handler, the handler must execute a return-from-gate instruction. Also, before returning from a quick interrupt, the IPL field of the PSW should be set to the previous state of the interrupted process.

# OPERATING SYSTEM CONSIDERATIONS

## Exceptions

### 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 processor during instruction execution. The fault handler for such events may restart the instruction that caused the fault.
- External faults - error conditions detected outside the processor and conveyed to it over its fault input. The processor recognizes the fault during instruction execution and the appropriate fault handler may then restart the execution.
- Traps - internal error conditions detected by the processor at the end of an instruction. After the trap is handled, execution may resume with the next instruction.

The exception mechanism for the *WE 32100* 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 processor recognizes four levels of exception severity, with zero (0) 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, ET is 3 and ISC 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, and on-reset exception microsequences do not use the ET and ISC values, but preserve them for an incoming exception handler. The on-normal exception microsequence uses 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 PCBPs or the address of the handling-routine table (for normal exceptions).

### 6.6.3 Exception Microsequences

The processor'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 processor starts performing a higher level microsequence. Thus, the processor can ripple up levels of exception severity.

Any exception during an on-reset sequence (the severest exception level) causes the processor to restart the on-reset sequence. Trying to recover from the exception, the processor 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.

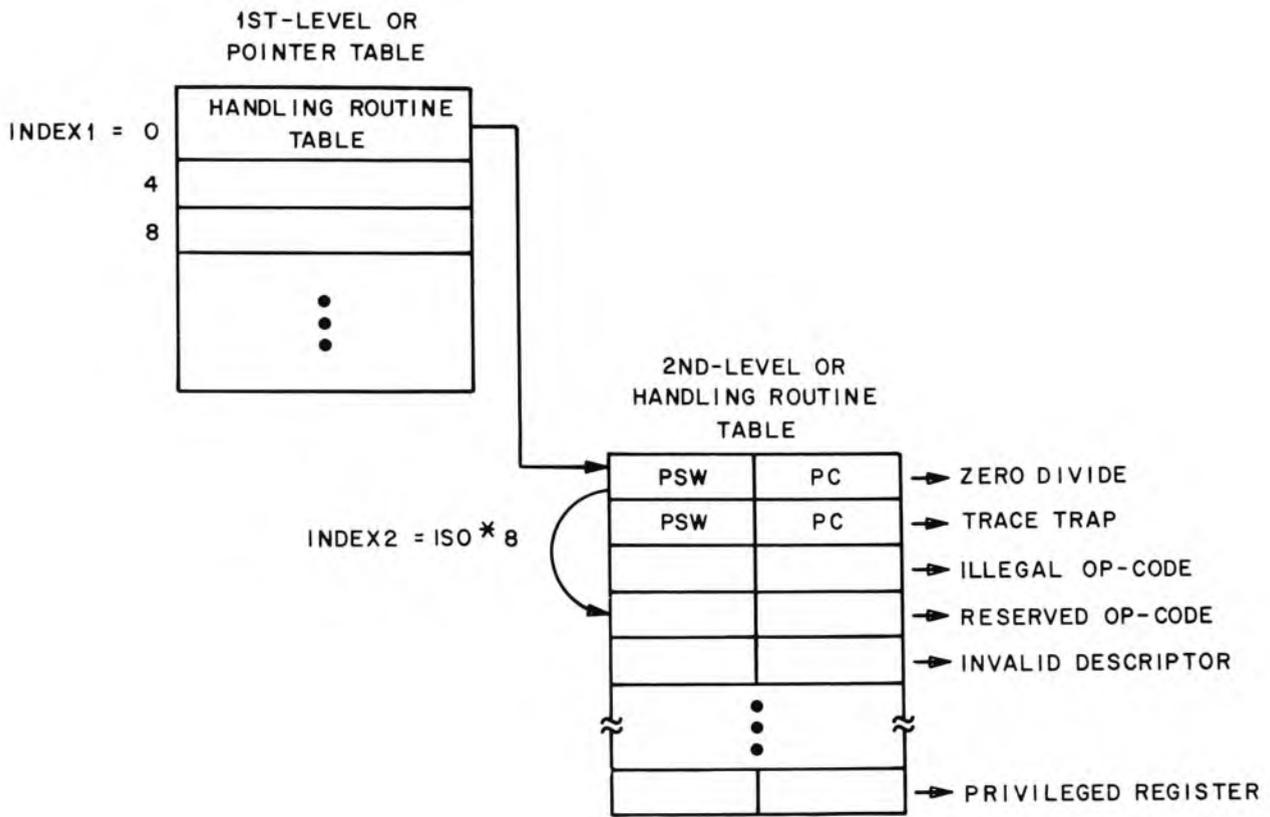
### Normal Exceptions

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.

When a normal exception occurs, the processor executes the on-normal exception microsequence. After some set up operations, the microsequence enters the gate instruction at its second entry point (see **6.3.2 Gate Instruction**). 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.

# OPERATING SYSTEM CONSIDERATIONS

## Normal Exceptions



**Figure 6-7. On-Normal Exception Indexing**

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 **6.3.1 Gate Mechanism** 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, shifted three bits toward the most significant bit (MSB). As shown on Figure 6-7 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 will contain 7, 1, 3, respectively.

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.

Table 6-4. Normal Exceptions (ET=3)		
ISC	Exception	Cause
0	Integer zero divide (Internal fault)	An attempt to divide by zero. This exception is always enabled. <sup>1</sup>
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. <sup>1</sup>
5	External memory (External fault)	A exception when accessing external memory.
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. <sup>1</sup>
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. <sup>2)</sup>
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 the ISP, PCBP, or PSW when not in kernel level. <sup>1</sup>

<sup>1</sup> This exception sets the condition flags as if the instruction was successfully completed.

<sup>2</sup> Before the overflow trap occurs, the processor 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 processor first detects a normal exception and then detects the stack exception while executing the implicit GATE (system call). In effect, the processor automatically ripples up to a stack exception from a normal 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 processor 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 systemwide. Thus, no context information is lost.

The on-stack exception microsequence saves the process A 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 and 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:

- Growing the stack of the process.
- Bringing in a missing page of the stack (in demand-paging systems).

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 the process A 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 processor executes its on-process exception microsequence, an implicit process switch. Because the error condition signifies that the process A PCB cannot be accessed, its context cannot be saved. The microsequence stores the process A PCBP on the interrupt stack and loads the process B PCBP from location 132 (0x84). The ISC field from the process A PSW is copied before the process B 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 processor could not save the process A hardware context, process B normally kills Process A. However, it can identify an old (good) process from its PCBP on the interrupt stack. If the exception is a new PCB exception, the process A 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 previously, then the process C 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.

### **Reset Exceptions**

A reset exception implies an error condition in accessing critical system data and requires restarting of the system. On a reset exception, the processor 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.

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

### Memory Management for Virtual Memory Systems

On a reset exception, the processor 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.

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 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

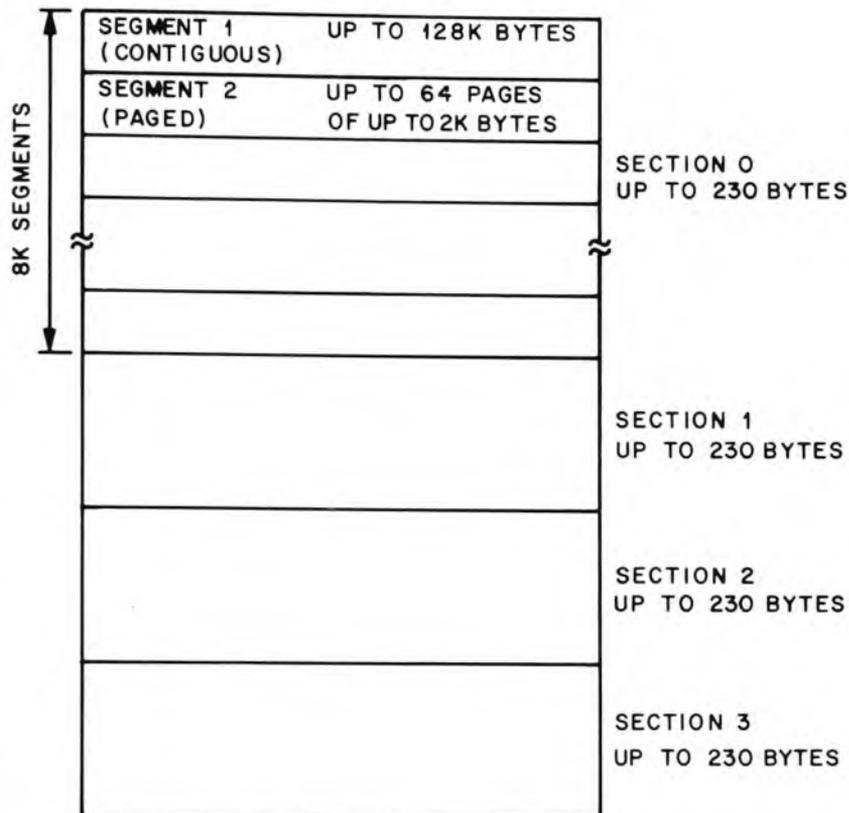
When a virtual memory system is used for a WE 32100 Microprocessor based system, a memory management unit (MMU) is required. The main function of an MMU is to translate virtual addresses into physical addresses. The MMU has the additional responsibility of providing protection for the system memory space.

The following is a brief description of the features of the WE 32101 MMU. For a complete description, refer to the *WE 32101 Memory Management Unit Information Manual*.

The virtual address space is divided into a number of sections by the MMU. Each section is in turn subdivided into segments. Segments may either be *contiguous* or *paged* and are mapped into the physical address space by the MMU.

## OPERATING SYSTEM CONSIDERATIONS

### Memory Management for Virtual Memory Systems



**Figure 6-8. Virtual Memory Space for a Process**

The *WE 32101* Memory Management Unit was developed to complement the *WE 32100* Microprocessor for creation of a virtual memory system. This section describes the features of the MMU that are important for system design. A complete technical summary of the MMU is provided in the *WE 32101* Memory Management Unit Data Sheet.

The *WE 32101* Memory Management Unit divides the virtual address space into four sections and provides both contiguous and paged segments for the system. A contiguous segment can be as large as 128 Kbytes and a paged segment can contain up to sixty-four 2-Kbyte pages. Figure 6-8 is an example of how virtual address space can be allocated.

The MMU divides virtual addresses into three fields for contiguous segments and four for paged segments. A virtual address referencing a contiguous segment is divided into three fields: a section ID (SID) field, a segment select (SSL) field, and a segment offset (SOT) field. The SID field specifies the section of virtual address space, the SSL field specifies the segment within the section, and the SOT field specifies the byte within the segment. The format of these virtual addresses is shown on Figure 6-9.

## OPERATING SYSTEM CONSIDERATIONS

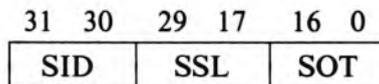
### Memory Management for Virtual Memory Systems

For paged segments, the SOT field is subdivided into a page select (PSL) field and a page offset (POT) field. The PSL field specifies the page within the segment and the POT field specifies the byte within the page. The format of these virtual addresses is shown on Figure 6-10.

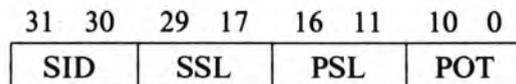
The MMU performs address translation using descriptors that contain the information necessary for segment and page mapping. The MMU has two types of descriptors: segment descriptors (SD) for mapping contiguous and paged segments and page descriptors (PD) for mapping pages within paged segments. An SD contains a segment base address that is added to an offset (from the virtual address SOT) to form the physical address. The PD contains a page base address that is concatenated with a page offset (from the virtual address POT) 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.

The SDs for each of the four sections of virtual memory are located in physical memory in segment descriptor tables (SDTs). There is one SDT associated with each section. The PDs for each paged segment are located in physical memory in page descriptor tables (PDTs), and there is one PDT associated with each paged segment. Contiguous segments are represented by an SDT entry, while paged segments are represented by both an SDT entry and an entire PDT (the SDT entry contains the physical base address of the PDT).



**Figure 6-9. Virtual Address Fields For A Contiguous Segment**



**Figure 6-10. Virtual Address Fields For A Paged Segment**

Figure 6-11 is a model showing how a virtual address is translated to a physical address for a contiguous segment. 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 starting physical address of the contiguous segment is contained in the indexed SD. This address is added to the SOT field to form the required physical address.

Figure 6-12 shows the paged segment model. This translation is identical to the contiguous segment address translation up to the point where the SD is indexed. For paged segments, the address in the SD is used as the base address of a 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.

# OPERATING SYSTEM CONSIDERATIONS

## Memory Management for Virtual Memory Systems

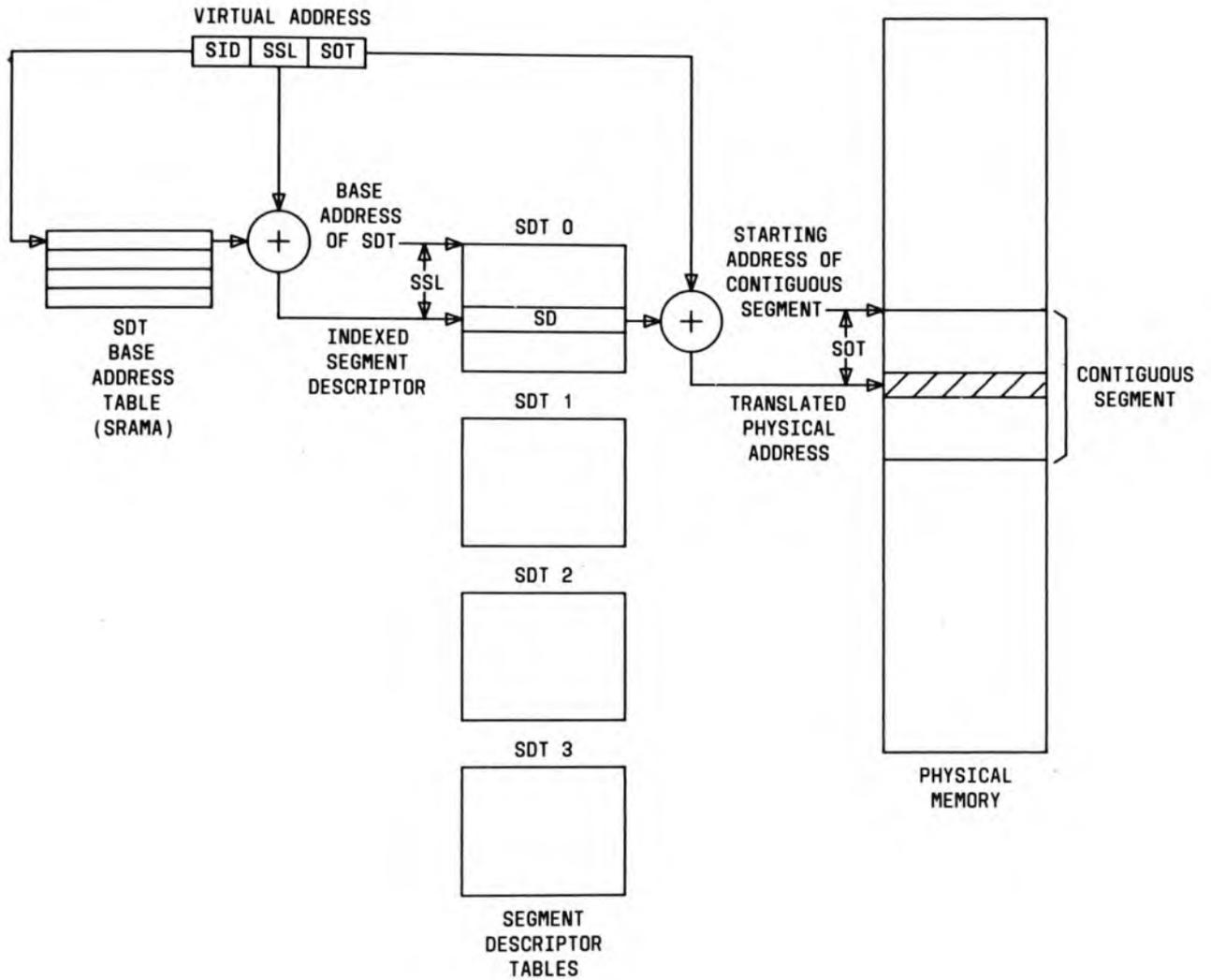


Figure 6-11. Virtual to Physical Translation for Contiguous Segments

# OPERATING SYSTEM CONSIDERATIONS

## Memory Management for Virtual Memory Systems

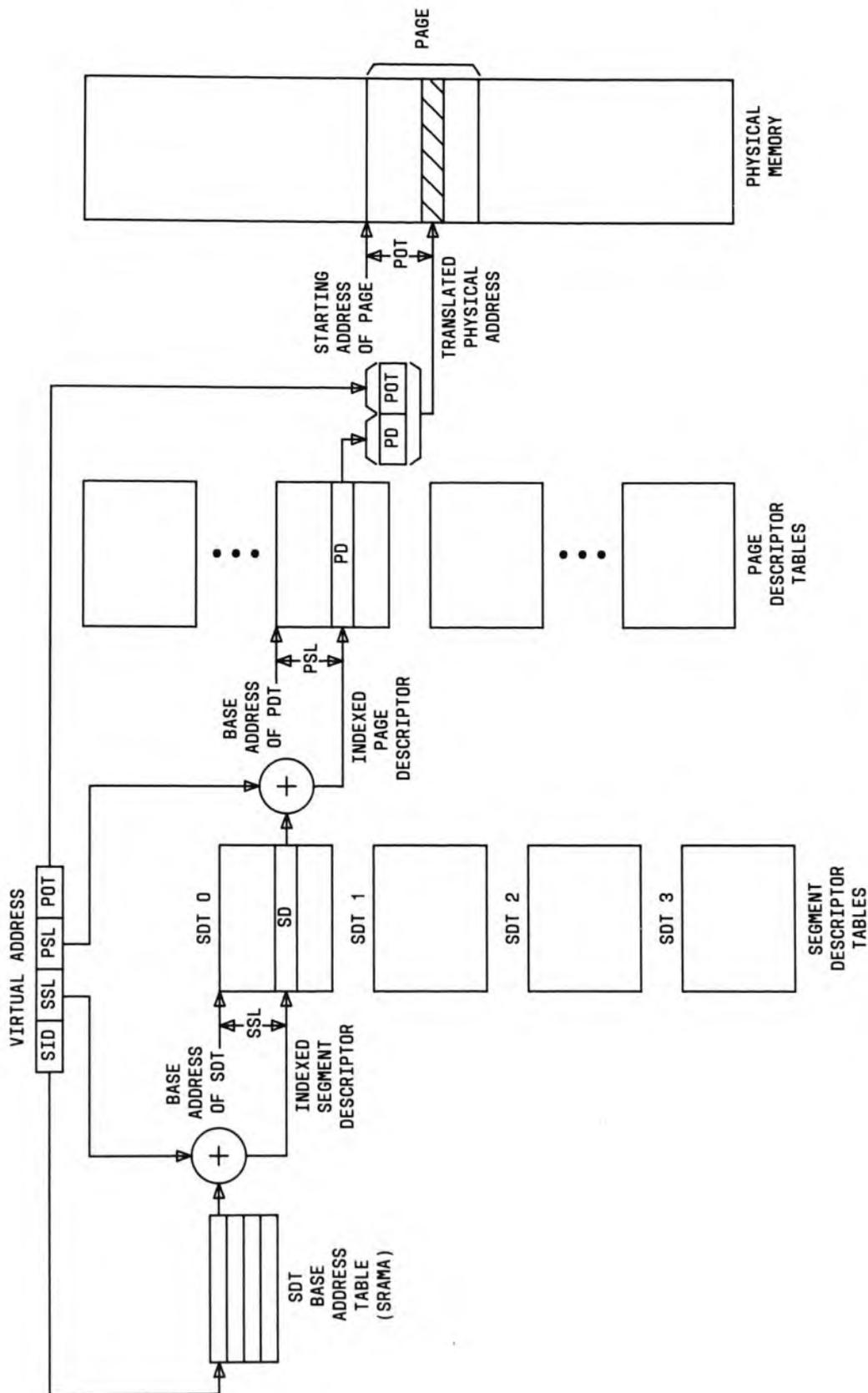


Figure 6-12. Virtual to Physical Translation for Paged Segments

### **6.7.1 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.

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. The way these tables are set up determines which segments in virtual memory are to be contiguous or paged and where the 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 some of the internal elements of the MMU. The elements that require initialization are the section RAMs and the configuration register (CR). 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 (e.g., double-page-hit).

### **6.7.2 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.

## **OPERATING SYSTEM CONSIDERATIONS**

### **MMU Exceptions**

#### **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 that is 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 (LRU) 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 the cache entry for that page or segment, if there is such a cache entry. 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.

### **6.7.3 Efficient Mapping Strategies**

The memory mapping defined by the operating system may have an enormous effect 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 contiguous segments so that few entries will be needed in the descriptor tables and descriptor caches. If physical memory is scarce, however, use of several large contiguous blocks could result in long waits to move the blocks in and out, thus wasting the physical memory where another large block cannot fit.

If only part of a segment need be in memory at a time, paged segments make more efficient use of memory.

#### **6.7.4 Object Traps**

Through object traps, the operating system can invoke a process or procedure whenever virtual addresses in a given segment are generated. The MMU can then save the virtual address that caused the trap. This facility can be used to make I/O devices or external processors appear as normal segments from the user-software point of view.

#### **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 on 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. As an example, 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. In this scenario, the cacheable bit is used to indicate the contents of the associated segment that are not cacheable.

#### **6.7.7 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



*--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 6-4 through 6-7 for the ISC values.

The following symbols are used to identify processor 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)
R $n$	Register $n$ , $rn, n = 0$ to 8 (assembler syntax % $rn$ )
SP	Stack pointer, r12 (assembler syntax %sp)

## OPERATING SYSTEM CONSIDERATIONS

### Privileged Instructions

#### 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).

Instruction	Mnemonic
Call process	CALLPS
Disable virtual pin and jump	DISVJMP
Enable virtual pin and jump	ENBVJMP
Interrupt acknowledge	INTACK
Return-to-process	RETPS
Wait	WAIT

The DISVJMP and ENBVJMP instructions disable or enable the processor's virtual address pin and then jump to an address. ENBVJMP enables an MMU, signaling that the processor 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 **6.4.2 Call Process Instruction** and **6.4.3 Return-to-Process Instruction**, respectively. 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

### 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. It:

- Saves the context (register contents) of the current process in the current PCB (if R bit of new process is set).
- Pushes the current PCBP value onto the interrupt stack.
- Puts the new PCBP value (from register **r0**) into the PCBP register.
- Sets the PSW, PC, and SP registers from the new PCB.
- Performs block moves (if any) for the new process (if R bit of PSW is set).
- 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)
```

```
/* 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
```

## CALLPS

## CALLPS

## CALLPS

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  
Modes**

None

**Condition  
Flags**

Set by new PSW

**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

<b>Assembler Syntax</b>	DISVJMP
<b>Opcode</b>	0x3013
<b>Description</b>	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).
<b>Operands</b>	<b>r0</b> is an implicit source operand (it should contain the new physical PC value).
<b>Operation</b>	if(!kernel-level) normal-exception (privileged-opcode) {Reset virtual address pin ( $\overline{\text{VAD}}$ ) to 1} PC = r0 {flush instruction cache}
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	normal exception (privileged opcode)
<b>Example</b>	{load physical address of next instruction into r0} DISVJMP
<b>Notes</b>	Opcode occupies 16 bits.

## ENBVJMP

## ENBVJMP

### Enable Virtual Pin and Jump

<b>Assembler Syntax</b>	ENBVJMP
<b>Opcode</b>	0x300D
<b>Description</b>	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).
<b>Operands</b>	<b>r0</b> is an implicit source operand (it should contain the new virtual PC value).
<b>Operation</b>	if(!kernel-level) normal-exception (privileged-opcode) {Set virtual address pin ( $\overline{VAD}$ ) to 0} PC = r0 {flush instruction cache}
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	normal exception (privileged opcode)
<b>Example</b>	{load virtual address of next instruction into r0} ENBVJMP
<b>Notes</b>	Opcode occupies 16 bits.

## INTACK

## INTACK

### Interrupt Acknowledge

<b>Assembler Syntax</b>	INTACK interrupt acknowledge
<b>Opcode</b>	0x302F INTACK
<b>Operation</b>	under "interrupt acknowledge" status $r0 \leftarrow (\text{Interrupt} - \text{ID}) \ll 2$
<b>Address Modes</b>	None
<b>Condition Flags</b>	Unchanged
<b>Exceptions</b>	privileged-opcode exception
<b>Examples</b>	INTACK
<b>Notes</b>	This instruction is privileged.

If  $\text{NMINT}==0$  and  $\text{AVEC}==0$ , 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  $\text{NMI}==0$ , an "auto-vector-interrupt acknowledge" access is performed (with all 1s on the address bus) and 0 is stored in r0. If  $\text{NMI}==1$  and  $\text{AVEC}==0$ , and "auto-vector-interrupt acknowledge" access is performed, and the "requesting level" (inverted and put on address bus and returned as "interrupt-ID") is indeterminate.

## RETPTS

## RETPTS

### Return to Process

<b>Assembler Syntax</b>	RETPTS
<b>Opcode</b>	0x30C8
<b>Description</b>	<p>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. It:</p> <ul style="list-style-type: none"><li>• Pops the saved (old) PCBP value from the interrupt stack.</li><li>• Puts the old PCBP value into the PCBP register.</li><li>• Sets the PSW, PC, and SP registers from the saved values in the old PCB.</li><li>• Performs block moves (if any) for the old process (if the R bit of the PSW is set).</li><li>• Puts the saved register values from the old PCB into the CPU registers (if the R bit in PSW is set).</li><li>• Exits, going to the old process.</li></ul>
<b>Operands</b>	None
<b>Operation</b>	<pre>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 bit from new PSW to current PSW so block moves and register restores will occur if needed. */ PSW&lt;R&gt; = *tempa&lt;R&gt;  /* Any memory exception in the following microsequences will cause a process exception (new PCB).*/</pre>

## RETPS

## RETPS

```
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
```

```
XSWITCH_THREE()
```

```
/* if R bit set, move saved register values from new PCB into CPU  
registers. */  
if(PSW<R>) {  
    FP = *(PCBP + 24)  
    r0 = *(PCBP + 28)  
    |  
    r8 = *(PCBP + 60)  
    AP = *(PCBP + 20)  
}
```

```
{unforce kernel level on memory accesses}  
{end of operation}
```

<b>Address</b>	None
<b>Modes</b>	
<b>Condition</b>	Set by new PSW
<b>Flags</b>	
<b>Exceptions</b>	normal exception (privileged opcode) process exception (old PCB and new PCB) reset exception (interrupt stack)
<b>Example</b>	RETPS
<b>Notes</b>	Opcode occupies 16 bits. There is no check of the interrupt stack. Any exception in accessing this stack causes a reset.

## WAIT

## WAIT

### Wait

### Assembler Syntax

WAIT

### Opcode

0x2F

### Description

This instruction halts the CPU, stopping instruction, fetching, and execution until an interrupt or external reset occurs.

### Operands

None

### Operation

if(!kernel-level)  
    normal-exception (privilege-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.

### 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

GATE and RETG were discussed previously in **6.3.2 Gate Instruction** and **6.3.3 Return-from-Gate Instruction**, respectively.

MOVTRW tells an enabled MMU to intercept the virtual address sent by the processor translate it, and return the physical address to the destination. If no MMU is enabled the system treats the MT access as a read, then this instruction acts as a normal MOVW (i.e., the source is copied into the destination).

## 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. It:

- 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.
- 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.
- 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.
- 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.
- 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).
- The PC is set to the address of the GATE-handling routine.
- GATE exits, going to the new PC.

### 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).

## GATE

## GATE

```
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
```

```
/* 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 values as old PSW */
tempb <IPL> = PSW <IPL>
tempb <R> = PSW <R>
/* set new PSW ISC/TM/ET to 7/1/3, respectively */
tempb <ISC> = 7
tempb <TM> = 1
tempb <ET> = 3
```

## GATE

## GATE

```
/* 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 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 and an MT access status is performed, and it is expected that the MMU will translate the address and return the corresponding physical address.

**Operands**                *src* - contains virtual address to be translated  
*dst* - contains the physical address after translation

**Operation**                {under MT status}  
*dst* = &*src*

**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 32101* 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.

## MOVTRW

## MOVTRW

When MOVTRW is executed in physical mode with the WE 32101 Memory Management Unit present, the MMU will behave as if a read operation in physical mode is taking place.

In systems without an MMU, some other device must respond to the MT access.

The source operand is an *address of* operand. The destination operand is of the type word. If *&src* is not a word address, a normal exception (external memory) will occur.

During an MOVTRW instruction, the status pins identify the memory access as being MT.

## RETG

## RETG

### Return from Gate

**Assembler** RETG

**Syntax**

**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 values as current PSW. \*/

tempa <IPL> = PSW <IPL>

tempa <CFD> = PSW <CFD>

tempa <QIE> = PSW <QIE>

tempa <CD> = PSW <CD>

tempa <R> = PSW <R>

/\* set new PSW ISC/TM/ET to 7/0/3, respectively \*/

tempa <ISC> = 7

tempa <TM> = 0 /\* avoids RETG trace trap \*/

tempa <ET> = 3

/\* 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}

**RETG****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

**RETG**

#### **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 actions. This sequence is identical to that of GATE except that zero (instead of **r0**) is used as the offset into the first-level table (index1), and the ISC value (instead of **r1**) is used as the offset into the second-level table (index2).

A RETG instruction can be used to return from a normal exception.

**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).

```
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 */
*(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).

## ON-NORMAL EXCEPTION

## ON-NORMAL EXCEPTION

```
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>
/* 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 */
PC = *(tempa + 4) /* get new PC */
PSW = tempb

/* finish push of old PC and PSW */
SP = SP + 8

{release kernel level on memory accesses}
{end of operation}
```

### Condition Flags

Set by new PSW

### Exceptions

stack exception (stack-bound and stack)  
process exception (gate PCB)  
reset exception (gate vector)

### Notes

The value of the ISC field of the PSW is the identity of the normal exception. See Table 6-4 for a list of normal exceptions. The ISC field of the saved PSW contains this code.

Some exceptions set the condition flags as if the instruction that caused the exception was successfully completed.

## ON-STACK EXCEPTION

## ON-STACK EXCEPTION

<b>Description</b>	<p>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.</p> <p>A RETPS instruction can be used to return from the stack exception handler process.</p>
<b>Operation</b>	<pre>/* 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). */ PSW&lt;ET&gt; = 2 /* stack exception */ PSW&lt;ISC&gt; = code for cause of exception 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&lt;ISC&gt; = 7 PSW&lt;TM&gt; = 0 /* prevent trace trap */ PSW&lt;ET&gt; = 3  {release kernel level on memory accesses} {end of operation}</pre>
<b>Condition Flags</b>	Set by the new PSW
<b>Exceptions</b>	process exception (old PCB and new PCB) reset exception (interrupt stack and system data)
<b>Notes</b>	The ISC field of the saved PSW contains the code that caused the stack exception.

## 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}
```

**Condition  
Flags** Set by new PSW

**Exceptions** reset exception (system data, interrupt stack, and new PCB)

**Notes** The ISC field of the PSW presented to the exception handling process will contain the code corresponding to the condition that caused the process exception.

## 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

{force kernel level on memory accesses}

/* force physical mode */
{Set VAD pin to one}

/* 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).

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}
```

### 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 will contain the code corresponding to the condition that caused the reset exception.

## 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 <QIE>).

For *full* (QIE==0) interrupts, 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* (QIE==1) interrupts, the on-interrupt microsequence implements a GATE-like PSW/PC switch, pushing the old PSW and PC onto 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. However, quick interrupt does not perform any stack bounds check, so it should not be used with an untrusted user process, which may have a bad value in SP. Unlike GATE, quick interrupt does update the PSW <IPL> field to act.

If an interrupt request is granted and auto-vectoring is requested (via the  $\overline{\text{AVEC}}$  pin), an auto-vector 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 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 (via the  $\overline{\text{NMINT}}$  pin), an auto-vector 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 ID.

#### Operation

{Get interrupt-ID value via interrupt acknowledge bus cycle}

tempa = interrupt-ID

if(memory-exception)

    stack-exception(interrupt-ID-fetch)

/\* test for full or quick interrupt \*/

if(PSW <QIE> == 1)

    goto QINT /\* quick interrupt \*/

/\* it is a full interrupt \*/

## ON-INTERRUPT

## ON-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).*/
XSWITCH_ONE()

/* Any memory exception in the XSWITCH_TWO or
XSWITCH_THREE microsequences 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

XSWITCH_THREE()

{release kernel level on memory accesses}
{end of operation}

QINT: /* it is a quick interrupt */

/* Put address of new PC and PSW pair in quick interrupt table into
tempa. */ tempa = 1164 + tempa * 8 /* 48C+tempa*8 hex */

/* When writing to the execution stack in the following two operations,
a memory exception causes a stack exception (stack).

Save old PC and PSW on execution stack.
Note: No stack bounds check. */
*SP = PC /* address of next instruction */
/* set PSW ISC/TM/ET to 1/0/2, respectively */
```

## ON-INTERRUPT

## ON-INTERRUPT

```
PSW<ISC> = 1
PSW<TM> = 0
PSW<ET> = 2
/* push PSW */
*(SP + 4) = PSW

/* A memory exception from here until 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}
tempb = *tempa
/* adjust previous execution level in new PSW */
tempb<PM> = PSW<CM>
/* set new PSW IPL to 15 */
tempb<IPL> = 15
/* new PSW R/ISC/TM/ET values same as old values */
tempb<R> = PSW<R>
tempb<ISC> = PSW<ISC>
tempb<TM> = PSW<TM>
tempb<ET> = PSW<ET>
/* put new PC/PSW values into PC/PSW registers */
PC = *(tempa + 4)
PSW = tempb

/* finish push of old PC and PSW */
SP = SP + 8
PSW<ISC> = 7
PSW<TM> = 1
PSW<ET> = 3

{release kernel level on memory accesses}
{end of operation}
```

**Condition  
Flags**

Set by new PSW

**Exceptions**

normal exception (gate vector)  
stack exception (stack and interrupt-ID fetch)  
process exception (old PCB and new PCB)  
reset exception (system data and interrupt stack)

**Notes**

The interrupt-ID fetch is 8 bits, and is zero-extended to 32 bits in tempa.

## 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, saving the current registers in the current PCB. 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.

The action taken when a memory exception is encountered in the XSWITCH microsequences is determined by the calling sequence.

**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 from new PSW to current PSW */
PSW<R> = *tempa<R>

/* 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>) {
    *(PCBP + 20) = AP
    *(PCBP + 24) = FP
    *(PCBP + 28) = r0
    |
    |
    *(PCBP + 60) = r8
    FP = PCBP + 52
}

return
```

## 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<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>) {  
    /* 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

```
/* do block moves, if PSW<R>==1 */
XSWITCH_THREE:
    if(PSW<R>) {
        /* 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
```

## XSWITCH

## **Glossary**



- Absolute deferred mode** - An address mode that uses an address embedded in the operand to locate a pointer to data.
- Absolute mode** - An address mode that uses an address embedded in the operand to locate data.
- Argument pointer (AP)** - User register that points to the beginning location in the stack where a set of arguments for a function has been pushed.
- 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.
- Cache** - A high-speed memory filled at a lower speed from main memory. Used to reduce memory access time.
- Cache disable (CD)** - A field in the PSW that enables and disables the instruction cache.
- Cache flush disable (CFD)** - A field in the processor status word (PSW) that enables and disables instruction cache flushing (emptying of the cache's contents) when a new process is loaded via the XSWITCH\_TWO microsequence.
- Condition codes (NZVC)** - The flags in this 4-bit field reflect the resulting status of the most recent instruction execution which affects them. The four flags are negative (N), zero (Z), overflow (V), and carry (C).
- Coprocessor** - A support processor that operates synchronously with the CPU to provide greater throughput in arithmetic or I/O functions.
- Current execution level (CM)** - A 2-bit field in the PSW that represents the current execution level. The four execution levels are kernel, executive, supervisor, and user.
- Dedicated registers** - Seven registers (r9—r15) that have specific, predetermined functions.
- Displacement mode** - An address mode that uses a register and an offset, both embedded in the operand, added together to form the address of data.
- Displacement deferred mode** - An address mode that uses a register and an offset, both embedded in the operand, added together to form the address of a pointer to data.
- Enable overflow trap (OE)** - A field in the PSW that enables overflow traps.
- Exception type (ET)** - A 2-bit field in the PSW that indicates exceptions generated during operations. The four types of exceptions are normal, stack, process, and reset.
- 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.
- Expanded-operand-type mode** - An address mode that changes the type of

## GLOSSARY

the instruction for an operand and those that follow it in the instruction. It does not affect immediate operands.

**Faults** - Error conditions that are detected outside the microprocessor and conveyed to the microprocessor over its fault input FAULT.

**Fetch unit** - The elements in this unit handle the instruction stream and perform memory-based operand accesses.

**Frame pointer (FP)** - User register that points to the beginning location in the stack of a function's local variables.

**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).

**General purpose registers** - Nine registers (r0—r8) that may be used for high-speed accumulation, for addressing, or for temporary data storage.

**Halfword** - 16-bit quantity that may appear at any address in memory that is divisible by 2.

**Immediate mode** - An address mode where the operand contains actual data to be used by instruction.

**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.

**Internal state code (ISC)** - A 4-bit field in the PSW that distinguishes between exceptions of the same exception type.

**Interrupt** - A means by which external devices may request service by the

microprocessor.

**Interrupt priority level (IPL)** - A 4-bit field in the PSW that represents the current interrupt priority level.

**Interrupt stack pointer (ISP)** - User register that contains the 32-bit memory address of the top of the interrupt stack.

**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.

**Operand descriptor** - First byte of an operand defining which address mode and register the operand uses.

**Pipelining** - Overlapping the execution of instructions to increase the microprocessor's performance.

**Prefetch** - A technique where the CPU fetches an instruction prior to the completion of previous instructions.

- Previous execution level (PM)** - A 2-bit field in the PSW that represents the previous execution level. The four execution levels are kernel, executive, supervisor, and user.
- 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. This context consists of the initial and current contents of control registers (PSW, PC, and SP), the last contents of registers r0 through r10, boundaries for an execution stack, and memory specifications for the process.
- 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.
- Quick interrupt** - An interrupt whose handling routine pushes the old PSW and PC on the stack and fetches a new PSW and PC that correspond to the interrupt's handler. For this reason the quick interrupt handling routine requires less time than a full interrupt which implements a process switch to the interrupt's handler. Interrupts are handled via the quick-interrupt sequence if the QIE bit in the PSW is set (1).
- Quick-interrupt enable (QIE)** - A field in the PSW that enables and disables the quick-interrupt facility.
- 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.
- Register deferred mode** - An address mode that uses a register name, embedded in an operand, which contains a pointer to data to be used by the instruction.
- Register mode** - An address mode that uses a register name, embedded in an operand, which contains data to be used by the instruction.
- Register-initial context (RI)** - A 2-bit field in the PSW that controls the microprocessor context switching strategy.
- Reset exceptions** - A class of exceptional conditions that is triggered by an error condition in accessing critical system data.
- Short offset mode** - An address mode that uses an offset embedded in an operand. The offset is added to the frame pointer or argument pointer to form the address of 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

## GLOSSARY

memory address of the next item to be stored on (pushed onto) the stack or the last item retrieved (popped) from the stack.

**Trace enable (TE)** - A field in the PSW that enables the trace function.

**Trace mask (TM)** - A field in the PSW that enables masking of a trace trap.

**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.

**User registers** - The sixteen 32-bit registers (r0—r15) that are available to the user. The user registers consist of nine general purpose registers (r0—r8) and seven dedicated registers (r9—r15).

**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	PCB - Process control block
BPT - Breakpoint trap	PCBP - Process control block pointer
C - Condition flag bit carry	PD - Page descriptors
CALLPS - Call process	PDT - Page descriptor table
CD - Cache disable	PM - Previous execution level
CFD - Cache flush disable	POT - Page offset field
CM - Current execution level	PPC - Prefetch counter
CMOS - Complimentary metal-oxide semiconductor	PSL - Page select field
CPU - Central processing unit	PSW - Processor status word
CR - Configuration register	QIE - Quick interrupt enable
DMA - Direct memory access	RAM - Random access read/write memory
EPROM - Erasable programmable ROM	RI - Register-initial
ET - Exception type	ROM - Read-only memory
FLTAR - Fault address register	rrrr - Register field
FLTCR - Fault code register	RSB - Return from subroutine
FP - Frame pointer	SD - Segment descriptors
I/O - Input/output	SDP - Software demand paging
IPL - Interrupt priority level	SDT - Segment descriptor table
ISC - Internal state code	SGP - Software generation programs
ISP - Interrupt stack pointer	SID - Section ID field
LSB - Least significant bit	SOT - Segment offset field
mmmm - Mode field	SP - Stack pointer
MMU - Memory management unit	SSL - Segment select field
MSB - Most significant bit	TE - Trace enable
N - Condition flag bit negative	TM - Trace mask
NOP - No operation	TT - Trace trap
OE - Overflow enable	TTL - Transistor-transistor logic
PC - Program counter	V - Condition flag bit overflow
	Z - Condition flag bit zero



**Index**



**A****Absolute**

- address modes, 5-7
- deferred mode, 5-13
- mode, 5-13

**Access abort, 3-7****Access protection, 6-46****Access status**

- code, 3-5
- signals, 3-4

**Address**

- and data bus, 2-1
- fault, 4-20
- modes, 5-3
- mode syntax, 5-6
- signals, 3-2
- strobe, 3-3
- translation, 6-38
- virtual, 6-38

**Addressing modes, Chapt. 5**

- absolute, 5-7
- absolute deferred, 5-13
- deferred displacement, 5-11
- displacement, 5-5, 5-7
- expanded operand type, 5-5
- immediate, 5-7
- negative literal, 5-6
- operands, 5-16
- positive literal, 5-5
- register, 5-5, 5-7
- register deferred, 5-5
- short offset, 5-5
- special, 5-7
- syntax, 5-6

**Alignment**

- fault bus activity, 4-46
- instructions and data, 5-17

**Analysis pod, 1-5****AP. See argument pointer.****Arbitration signals, 3-3, 4-38****Architecture, 1-2, 6-1**

- pipelining, 2-1, 4-47

**Argument pointer (AP), 2-5,**

- short offset mode, 5-5

**Arithmetic instructions, 5-19****Assembler syntax, 5-29****Assembly language**

- operations, 5-31
- symbols, 4-3, 5-31
- syntax, 5-15

**Asserted signal, 3-1, 4-3****Asynchronous read, 4-5****Asynchronous write, 4-8****Autovector interrupt, 3-2, 4-35, 6-26****B****Bit**

- cacheable (\$), 6-45
- software, 6-46

**Bit field**

- base word, 2-7
- defined, 2-7,
- extraction of, 2-9
- instructions, 5-1
- offset, 2-7,
- width, 2-7,

**Block diagram, 2-1****Blockfetch, 3-4**

- operation, 4-15

**Borrow, 2-14****Branch instructions, 5-22****Bus**

- address, 2-1, 4-2
- arbitration, 4-38, 4-39
- arbitrator, 3-6
- exception signals, 3-7
- exceptions, 4-20
  - retry and relinquish, 4-24
- operation, Chapt. 4
- request, 3-7, 4-41
  - acknowledge, 3-7

**Byte**

- data, 2-6, 5-1
- deferred displacement mode, 5-5
- displacement mode, 5-5
- instructions, 5-16

**C****Carry, 2-14****Cache**

# INDEX

- disable, 6-13
- flush disable, 6-13
- instruction cache flush, 6-13
- instruction cache hit, 4-45
- memory, 6-45
- memory management unit descriptor, flushing, 6-44
- Call process instruction, 6-18, 6-49
- Call-save sequence, 5-26
- Clock
  - input, 4-1
  - signals, 3-2
  - state, 4-2
- CM. See current execution level.
- Condition codes, 6-13
- Condition flags, 2-13, 5-29,
- Conventional registers, 2-3
- Context switching, strategy, 6-19
- Control signals, 3-3
- Control-register save area, 6-8
- Coprocessor, 4-48
  - broadcast, 4-48
  - data write, 4-55
  - done, 3-3
  - instructions, 5-27
  - interface, 4-48
  - operand fetch, 4-53
  - status fetch, 4-54
- Current execution level (CM), 6-12
- Cycle, 5-16
  - initiate, 3-3

## D

### Data

- bus shadow, 3-7
- dependencies, 5-17
- in memory, 2-10
- ready, 3-4
- signals, 3-2
- size, 3-5
- storage, 2-10
- strobe, 3-4
- transfer acknowledge, 3-4
- transfer instructions, 5-17
- types, 2-6, 5-1

- Deferred address modes
  - defined, 5-9
- Deferred displacement mode, 5-11
- Descriptor byte format, 5-4
- Development
  - microprocessor, 1-1
  - system support signals, 3-9
- Direct memory access (DMA), 4-41
- Displacement modes, 5-7, 5-10
- DMA. See direct memory access.
- Double word (blockfetch), 3-4

## E

- Enable overflow trap, 6-13
- Entry point, 6-15
- Evaluation board, 1-6
- Exception
  - breakpoint trap, 6-35
  - conditions, 6-32
  - defined 6-32
  - external memory, 6-35
  - gate vector, 6-38
  - handler, 6-32
  - illegal level change, 6-35
  - illegal opcode, 6-35
  - integer overflow, 6-35
  - integer zero divide, 6-35
  - interrupt-stack fault, 6-38
  - invalid descriptor, 6-35
  - memory management unit, 6-44
  - new-PCB fault, 6-37
  - normal, 6-33
  - old-PCB fault, 6-37
  - on-normal, 6-33, 6-66
  - on-process, 6-37, 6-69
  - on-reset, 6-37, 6-70
  - on-stack, 6-36, 6-68
  - privileged-opcode, 6-35
  - privileged-register, 6-35
  - process, 6-37
  - reserved-data-type, 6-35
  - reserved opcode, 6-35
  - reset, 6-37
  - severity, levels of, 6-32
  - stack, 6-36

- system-data, 6-38
  - trace-trap, 6-35
  - type, 6-12
  - Execution
    - modes, 3-6
    - modes, levels, 6-1, 6-4, 6-12
    - privilege, 6-4
    - stack, 6-5
  - Executive mode (level 1), 6-1
  - Expanded-operand type mode, 5-5, 5-14
  - Explicit process switch, 6-3, 6-18
  - Extending data, 2-10
- F**
- Fault, 3-8, 4-21
    - blockfetch, 4-27
    - defined, 4-20
    - exception mechanism, 6-32
    - external, 6-32
    - gate-PCB, 6-37
    - gate vector, 6-16, 6-38
    - internal, 6-32
    - interrupt-stack, 6-38
    - memory, 4-20, 6-32, 6-44
    - new-PCB, 6-37
    - old-PCB, 6-37
    - stack, 6-36
  - Fetch unit, 2-1
  - Fields
    - PSW, 6-12
  - First entry point, 6-15
  - Floating-point data types, 2-8
  - Flushing
    - instruction cache, 6-13
    - memory management unit
      - descriptor cache, 6-44
  - FP. See frame pointer.
  - Frame pointer (FP), 2-4,
    - short offset mode, 5-5
  - Full interrupts, 4-32, 6-31
  - Full-interrupt handler's PCB, 6-27
  - Functional group instructions, 5-16
- G**
- Gate, 6-11
    - instruction, 6-15, 6-58
    - mechanism, 6-14, 6-16
  - PCB fault, 6-37
    - return from, 6-18, 6-63
    - vector fault, 6-16, 6-38
  - General-purpose registers, 2-3, 5-1
  - General-register save area, 6-7
- H**
- Halfword
    - boundary, 2-10
    - data, 2-6, 5-1
    - deferred displacement mode, 5-5
    - displacement mode, 5-5
    - immediate mode, 5-6
  - Handling-routine tables, 6-14
  - High impedance, 3-9
  - High-level language support group, 3-4
- I**
- I bit, 6-19
  - Immediate modes, 5-7, 5-12
  - Implicit process switch, 6-3, 6-25, 6-30,
    - 6-36 thru 6-38
  - Indexing
    - on-normal exception, 6-34
    - pointer and handling table, 6-17
  - Indirect segment descriptors, 6-45
  - Initial context for a process, 6-9, 6-19
  - Initialize
    - memory management unit, 6-43
  - Instruction
    - and data alignment, 5-17
    - format, Chapt. 2, 2-12, 5-3
    - pipelining, 5-17
    - queue status, 3-9
    - timing, 5-16
  - Instruction set, 1-3, 2-11, Chapt. 5
    - functional groups, 5-16
    - listings, 5-29
    - operating system, 6-2, 6-46
    - storage, 2-12
    - summary by mnemonic, 5-32

## INDEX

- summary by opcode, 5-36
- Instruction cache
  - hit, 4-44
  - on-chip, 5-16
- Instructions
  - branch, 5-22
  - jump, 5-22
  - procedure-call, 5-26
- Interface signals, 3-33
- Interlocked operation, read, 4-12
- Internal reset, 4-42
- Internal State Code (ISC), 6-12, 6-33 thru 6-38
- Interrupt
  - acknowledge, 4-32, 6-26
  - autovector, 4-35, 6-26
  - handler model, 6-25
  - handler's PCB, 6-27
  - mechanism, 6-26
  - nonmaskable, 4-35
  - on-interrupt microsequence, 6-30, 6-71
  - option, 3-2
  - priority level, 3-3, 6-13
  - quick, 6-31
  - request and acknowledge codes, 4-34
  - returning from, 6-31
  - signals, 3-2
  - stack 6-28
  - stack pointer (ISP), 2-5, 6-28
  - structure, 6-25
  - vector table, 6-29
- ISC. See internal state code.

## J

- Jump instructions, 5-22

## K

- Kernel mode (level 0), 6-1

## L

- Language support group
  - high-level, 2-4

- Least significant bit (LSB), 2-6
- Level
  - priority of interrupts, 3-3
- Levels of exception severity, 6-32
- Listing
  - instruction set, 5-29
- Logical instructions, 5-20
- LSB. See least significant bit.

## M

- Map, memory, 6-7
- Mapping strategy, 6-44
- Memory
  - data in, 2-10
  - instructions in, 2-12
  - management, 6-38
  - management, virtual, 6-3, 6-38
  - map, 6-7
  - options, 6-40
  - PCB specifications, 6-10
  - translation, 6-40
  - virtual, 6-39
- Memory management unit (MMU), 6-38
  - exceptions, 6-44
  - initialized, 6-43
  - interactions, 6-43
  - mapping strategies, 6-44
  - peripheral mode, 6-43
  - translation
    - contiguous segment, 6-40
    - paged segment, 6-40
- Microsequence, 6-2, 6-65
  - defined, 6-2
  - on-interrupt, 6-30, 6-71
  - on-normal, 6-33, 6-66
  - on-process, 6-37, 6-69
  - on-reset, 6-37, 6-70
  - on-stack, 6-36, 6-68
  - XSWITCH, 6-23 thru 6-25, 6-74
- Miscellaneous instructions, 5-27
- mxxx field (address mode field), 5-3
- MMU. See memory management unit.
- Mnemonic, 2-12, 5-3
  - instructions by, 5-32

## Mode

- absolute, 5-13
- absolute deferred, 5-13
- expanded-operand, 5-14
- immediate, 5-12

Modes, addressing. See addressing modes.

Most significant bit (MSB), 2-6,

MSB. See most significant bit.

## N

- Negated signal, 3-1, 4-3,
- Negative literal mode, 5-6
- New-PCB fault, 6-37
- Nonmaskable interrupt, 3-3, 4-35
- Nonprivileged instructions, 6-57
- Normal exceptions, 6-33
- Notation, 5-29
  - operation, 6-46

## O

- Object traps, 6-45
- Old-PCB fault, 6-37
- On-chip instruction cache, 5-16
- On-interrupt microsequence, 6-30, 6-71
- On-normal exception, 6-33, 6-66
- On-process exception, 6-37, 6-69
- On-reset exception, 6-37, 6-70
- On-stack exception, 6-36, 6-68
- Opcodes,
  - instruction set, 5-36
- Operand, 5-14
  - descriptor, 5-14
  - format, 2-13, 5-4
  - in instruction format, 5-14
  - syntax, 5-6, 5-15
  - See also addressing modes.
- Operating system
  - features, 6-1
  - instructions, 6-2, 6-46
  - support, 1-3, 2-5, Chapt. 6
- Operation, Chapt. 4
  - read, 4-2
  - write, 4-2
- opnd. See operand.
- Outputs

- during DMA, 4-41

- during reset, 4-43

Overflow, 2-14

## P

PC. See program counter.

PCB. See process control block.

PCBP. See process control block pointer.

Peripheral mode, 6-42

Physical

- address, 6-38

- memory, 6-14, 6-38

Pin assignments, 3-9

Pipelining, 4-47

- defined, 2-2

- instruction, 5-17

PM. See previous execution level.

Pointer

- defined, 5-9

Pointer table, 6-14

Positive literal mode, 5-5

Previous execution level (PM), 6-12

Privileged

- execution modes, 6-1, 6-4

- instructions, 6-2, 6-48

- register, 2-2, 5-2

Procedure transfer, 5-22

Process

- defined, 6-1

- exceptions, 6-37

- structure, 6-4

- switching, 6-1, 6-18

- explicit, 6-3, 6-18

- implicit 6-3, 6-25, 6-30, 6-36 thru 6-38

Process control block (PCB), 6-4, 6-6

Process control block pointer (PCBP), 2-5, 6-5

- locations, 6-7

- register, 5-1, 6-5

Processor status word (PSW), 2-5, 6-11

- fields, 6-12

- register, 2-2, 5-2

Program control instructions, 5-22

Program counter (PC), 2-4, 5-2

PSW. See processor status word.

# INDEX

## Q

Quick interrupt, 6-25, 6-31  
enable, 6-13

## R

R bit, 6-19  
Read operation, 4-2  
Read/Write (R/W), 3-5  
Registers, Chapt. 2, 5-1  
    assembler syntax, 5-2  
    conventional, 2-3  
    data storage, 2-11  
    deferred mode, 5-5, 5-8  
    initial context, 6-12  
    modes, 5-5, 5-7, 5-8  
    set, 5-2  
    syntax, 5-1  
    user, 2-2, 5-1  
Relinquish and retry, 4-24  
    of blockfetch, 4-32  
    request, 3-8  
    request acknowledge, 3-8  
Reserved  
    data type exception, 6-35  
    opcode, exception, 6-35  
Reset, 4-42  
    acknowledge, 3-8  
    exceptions, 6-37, 6-70  
    internal, 4-42  
    request, 3-8  
    sequence, 4-44  
    signal, 4-43  
    states, 4-43  
Retry, 3-8, 4-24  
Return  
    from gate, 6-18, 6-63  
    from interrupt, 6-31  
    to process, 6-24, 6-54  
rrrr field (register field), 5-3  
R/W. See Read/Write.

## S

Save-context area, 6-7

Saved context for a process, 6-10, 6-19  
Second entry point, 6-16  
Segment descriptors  
    indirect, 6-45  
Short offset address modes, 5-5  
Sign extension, 2-9  
Signals, Chapt. 3  
    address and data, 3-2  
    access status, 3-4  
    arbitration, 3-6  
    bus exception, 3-7  
    clock, 3-2  
    descriptions, 3-1  
    development system support, 3-9  
    interface and control, 3-3  
    interrupt, 3-2  
    sampling points, 4-1  
SP. See stack pointer.  
Special address modes, 5-7  
Stack, 2-4  
    bounds, 6-6, 6-36  
    exception handler, 6-6, 6-36  
    exceptions, 6-36, 6-68  
    execution, 6-5  
    fault, 6-36  
    instructions, 5-27  
    interrupt, 6-6, 6-28  
    pointer (SP), 2-7,  
Stack-bound, 6-9, 6-36  
    exception, 6-36  
    fault, 6-36  
Start of instruction, 3-9  
Status codes, 3-5  
Stop, 3-9  
Storage  
    data, 2-10  
    of instructions, 2-12  
    register data, 2-11  
Structure of a process, 6-4  
Subroutine transfer, 5-22  
Supervisor mode (level 2), 6-1  
Support  
    high-level language, 2-4  
    operating system, 2-5  
    products, 1-4  
Symbols,

assembly language, 5-31

### Synchronous

read, 4-3

ready, 3-4

write, 4-8

### Syntax

address mode, 5-6

assembler, 5-2, 5-29

assembly language, 5-15

operand, 5-6, 5-15

register, 5-1

System reset, 4-42

## T

Trace enable, 6-13

Trace mask, 6-12

### Transfer

procedure, 5-22

subroutine, 5-22

Translation, virtual, 6-40

### Trap, 6-32

enable overflow, 6-13

object, 6-45

## U

UNIX System, 1-4

### User

mode (level 3), 6-1

registers, 2-2, 5-1

## V

### Virtual

address, 3-6, 6-39

address space, 6-39

memory, 6-39, 6-43

memory, division of, 6-39

memory space, 6-39

## W

### Word, 2-6

address modes, 5-5

data, 2-6

boundary, 2-8

deferred displacement mode, 5-5

displacement mode, 5-5

storage, 2-13

Write operation, 4-2

## X

XSWITCH function, 6-23 thru 6-25, 6-74

XSWITCH\_ONE, 6-74

XSWITCH\_TWO, 6-75

XSWITCH\_THREE, 6-76

## Z

Zero, 2-14

extension, 2-9