SPARC RISC

USER'S GUIDE

CYPRESS SEMICONDUCTOR

Ross Technology Subsidiary

SPARC RISC

USER'S GUIDE

CYPRESS SEMICONDUCTOR

SPARC RISC USER'S GUIDE

ROSS Technology, Inc.
A Cypress Semiconductor Company

ROSS Technology, Inc. • 7748 Hwy. 290 West • Austin • TX 78736 • 512-448-8968
Cypress Semiconductor Corporation • 3901 North First Street • San Jose • CA 95134 • 408-943-2600

Second Edition – February 1990

SPARC is a trademark of Sun Microsystems.

ROSS Technology, Inc. is a subsidiary of Cypress Semiconductor Corporation.

© ROSS Technology, Inc., 1990. The information contained herein is subject to change without notice. ROSS Technology, Inc. assumes no responsibility for the use of any circuitry other than circuitry embodied in a ROSS Technology, Inc. product. Nor does it convey or imply any license under patent or other rights. "ROSS Technology does not authorize its products for use as critical components in life support systems where a malfunction or failure of the product may reasonably be expected to result in significant injury to the user. The inclusion of ROSS Technology products in life support systems applications implies that the manufacturer assumes all risk of such use and in so doing indemnifies ROSS Technology against all damages."

Table of Contents

Foreward ..... **xvii**

Chapter 1: Introduction

1.1 SPARC Overview **1-1**
1.1.1 Partitioning 1-1
1.1.2 The CY7C601 Integer Unit 1-2
1.1.3 CY7C611 Integer Unit for Embedded Control 1-4
1.1.4 CY7C602 Floating-Point Unit 1-4
1.1.5 CY7C157 Cache Data RAM 1-4
1.1.6 CY7C604/CY7C605 Cache Controller and MMU 1-4

1.2 Register Windows **1-6**

1.3 Instruction Set **1-6**
1.3.1 Load and Store Instructions 1-6
1.3.2 Arithmetic/Logical/Shift Instructions 1-6
1.3.3 Control Transfer Instructions 1-6
1.3.4 Read/Write Control Register Instructions 1-6
1.3.5 Floating-Point-Operate and Coprocessor-Operate Instructions 1-7

Chapter 2: CY7C601/CY7C611 Integer Unit

2.1 Description Of Parts **2-2**

2.2 Programming Model **2-2**
2.2.1 Register Windows 2-2
2.2.2 Processor States 2-8
2.2.3 Supervisor/User Modes 2-8
2.2.4 Control/Status Registers 2-8
2.2.5 Data Types 2-12

2.3 Instruction Set **2-15**
2.3.1 Instruction Formats 2-15
2.3.2 Addressing 2-17
2.3.3 Instruction Types 2-19
2.3.4 Op Codes 2-31

2.4 Signal Description **2-43**
2.4.1 Memory Subsystem Interface Signals 2-45
2.4.2 Floating-Point/Coprocessor Interface Signals 2-49
2.4.3 Interrupt and Control Signals 2-51
2.4.4 Power and Clock Signals 2-52

2.5 Pipeline and Instruction Execution Timing 2–52
2.5.1 Stages 2–53
2.5.2 Multicycle Instructions 2–54
2.5.3 Pipeline Freezes 2–58
2.5.4 Traps 2–58

2.6 Bus Operation and Timing 2–58
2.6.1 Instruction Fetch 2–61
2.6.2 Load 2–61
2.6.3 Load with Interlock 2–61
2.6.4 Load Double 2–62
2.6.5 Store 2–63
2.6.6 Store Double 2–64
2.6.7 Atomic Load-Store 2–65
2.6.8 Floating-Point Operations 2–66
2.6.9 Bus Arbitration 2–67
2.6.10 Load with Cache Miss 2–68
2.6.11 Store with Cache Miss 2–69
2.6.12 Memory Exceptions 2–71
2.6.13 Floating-Point Exceptions 2–75
2.6.14 Interrupts 2–75
2.6.15 Reset Condition 2–76
2.6.16 Error Condition 2–76

2.7 Exception Model 2–78
2.7.1 Reset 2–78
2.7.2 Synchronous Traps 2–78
2.7.3 Interrupts 2–80
2.7.4 Floating-Point/Coprocessor Traps 2–81
2.7.5 Trap Operation 2–82

2.8 Coprocessor Interface 2–84
2.8.1 Protocol 2–85
2.8.2 Register Model 2–86
2.8.3 Exceptions 2–86

2.9 CY7C611 Integer Unit for Embedded Control 2–87

Chapter 3: CY7C602 Floating-Point Unit

3.1 CY7C602 Functional Description 3–1

3.2 Floating-Point/Integer Unit Interface 3–4
3.2.1 CY7C602 Instruction Fetch and Execution 3–5
3.2.2 Instruction Pipeline Flush 3–9

3.3 CY7C602 Programming Model 3–12
3.3.1 CY7C602 Registers 3–12

3.3.2 CY7C602 Floating-Point Instructions 3–16
3.3.3 CY7C602 Internal Operation 3–17
3.3.4 CY7C602 IEEE-754 Compliance 3–19
3.3.5 CY7C602 Exception Cases 3–22

3.4 CY7C602 Signal Descriptions 3–23
3.4.1 Integer Unit Interface Signals 3–23
3.4.2 Coprocessor Interface Signals 3–24
3.4.3 System/Memory Interface Signals 3–24

Chapter 4: CY7C604/CY7C605 Cache Controller and MMU

4.1 Memory Management Unit 4–3
4.1.1 Translation Lookaside Buffer (TLB) 4–4
4.1.2 Table Walk 4–8
4.1.3 Page Table Pointer (PTP) 4–9
4.1.4 Page Table Entry (PTE) 4–10
4.1.5 Page Table Pointer Cache (PTPC) 4–11

4.2 MMU Operation Modes 4–13
4.2.1 MMU Flush and Probe Operations 4–14

4.3 CY7C604 / CY7C605 Cache Controllers 4–15
4.3.1 CY7C604/605 Cache Modes 4–15
4.3.2 CY7C604 Cache Controller 4–16
4.3.3 CY7C605 Cache Controller 4–20
4.3.4 CY7C604/CY7C605 Cache Control Signals 4–30
4.3.5 CY7C604/605 Write Buffer 4–31
4.3.6 CY7C604/605 Read Buffer 4–32
4.3.7 CY7C604/605 Cache Flushing Operations 4–32
4.3.8 CY7C604/605 Cacheable/Non-Cacheable Memory Accesses 4–33
4.3.9 CY7C604/605 Mbus Cacheable (MC) Bit 4–33
4.3.10 CY7C604/605 LDSTO (Atomic Load-Store Instruction) cycles 4–34
4.3.11 CY7C604/605 Cache Byte Write Enables 4–34

4.4 CY7C604 / CY7C605 Registers 4–35
4.4.1 CY7C604 System Control Register (SCR) 4–35
4.4.2 CY7C605 System Control Register (SCR) 4–36
4.4.3 CY7C604/605 Context Table Pointer Register (CTPR) 4–37
4.4.4 CY7C604/605 Context Register (CXR) 4–37
4.4.5 CY7C604/605 Reset Register (RR) 4–37
4.4.6 CY7C604/605 Root Pointer Register (RPR) 4–38
4.4.7 CY7C604/605 Instruction access PTP (IPTP) 4–38
4.4.8 CY7C604/605 Data access PTP (DPTP) 4–38
4.4.9 CY7C604/605 Index Tag Register (ITR) 4–38
4.4.10 CY7C604/605 TLB Replacement Control Register (TRCR) 4–39
4.4.11 CY7C604/605 Synchronous Fault Status Register (SFSR) 4–39
4.4.12 CY7C604/605 Synchronous Fault Address Register (SFAR) 4–40

4.4.13 CY7C604/605 Asynchronous Fault Status Register (AFSR) 4–40
4.4.14 CY7C604/605 Asynchronous Fault Address Register (AFAR) 4–40

4.5 CY7C604 / CY7C605 Multichip Configuration 4–41

4.6 CY7C604/605 Diagnostic Support 4–43
4.6.1 CY7C604/605 MMU TLB Entries 4–43
4.6.2 CY7C604/605 Cache Tag Entries 4–44
4.6.3 CY7C604/605 Cache Data Entries 4–44

4.7 CY7C604/605 Reset 4–45
4.7.1 Power-On Reset (POR) 4–45
4.7.2 Watch-Dog Reset (WDR) 4–45
4.7.3 Software Internal Reset (SIR) 4–45
4.7.4 Software External Reset (SER) 4–45
4.7.5 CY7C604/605 Reset in Multichip Configuration 4–46

4.8 CY7C604/605 ASI and Register Mapping 4–46

4.9 Synchronous Faults 4–47
4.9.1 Synchronous Fault Cases 4–50

4.10 CY7C604/605 Pin Definitions 4–55

4.11 Virtual Bus Operation 4–60

4.12 Physical Bus (Mbus) Operation 4–84
4.12.1 Mbus Principles 4–84
4.12.2 Mbus Level 1 Overview 4–84
4.12.3 Mbus Level 2 Overview 4–84
4.12.4 Mbus Signal Summary 4–85
4.12.5 Mbus Address Cycle 4–87
4.12.6 Mbus Data Cycle 4–88
4.12.7 Mbus Transactions 4–88
4.12.8 Mbus Transaction Timing 4–92

Chapter 5: CY7C157 Cache RAM

5.1 Description of Part 5–1

5.2 Operation 5–2

5.3 Bus Timing 5–2

5.4 Signal Descriptions 5–2

Chapter 6: SPARC Instruction Set

6.1 Assembly Language Syntax 6–1
6.1.1 Register Names 6–1

6.1.2 Special Symbol Names 6–2
6.1.3 Values 6–2
6.1.4 Label 6–2
6.1.5 Instruction Mnemonics 6–3

6.2 Definitions 6–4
ADD Add 6–7
ADDcc Add and modify icc 6–8
ADDX Add with Carry 6–9
ADDXcc Add with Carry and modify icc 6–10
AND And 6–11
ANDcc And and modify icc 6–12
ANDN And Not 6–13
ANDNcc And Not and modify icc 6–14
Bicc Integer Conditional Branch 6–15
CALL Call 6–17
CBccc Coprocessor Conditional Branch 6–18
CPop Coprocessor Operate 6–20
FABSs Absolute Value Single 6–21
FADDd Add Double 6–22
FADDs Add Single 6–23
FADDx Add Extended 6–24
FBfcc Floating-Point Conditional Branch 6–25
FCMPd Compare Double 6–27
FCMPEd Compare Double and Exception if Unordered 6–28
FCMPEs Compare Single and Exception if Unordered 6–29
FCMPEx Compare Extended and Exception if Unordered 6–30
FCMPs Compare Single 6–31
FCMPx Compare Extended 6–32
FDIVd Divide Double 6–33
FDIVs Divide Single 6–34
FDIVx Divide Extended 6–35
FdTOi Convert Double to Integer 6–36
FdTOs Convert Double to Single 6–37
FdTOx Convert Double to Extended 6–38
FiTOd Convert Integer to Double 6–39
FiTOs Convert Integer to Single 6–40
FiTOx Convert Integer to Extended 6–41
FMOVs Move 6–42
FMULd Multiply Double 6–43
FMULs Multiply Single 6–44
FMULx Multiply Extended 6–45
FNEGs Negate 6–46
FSQRTd Square Root Double 6–47
FSQRTs Square Root Single 6–48
FSQRTx Square Root Extended 6–49
FsTOd Convert Single to Double 6–50
FsTOi Convert Single to Integer 6–51
FsTOx Convert Single to Extended 6–52
FSUBd Subtract Double 6–53
FSUBs Subtract Single 6–54

FSUBx Subtract Extended 6–55
FxTOd Convert Extended to Double 6–56
FxTOi Convert Extended to Integer 6–57
FxTOs Convert Extended to Single 6–58
IFLUSH Instruction Cache Flush 6–59
JMPL Jump and Link 6–60
LD Load Word 6–61
LDA Load Word from Alternate space 6–62
LDC Load Coprocessor register 6–63
LDCSR Load Coprocessor State Register 6–64
LDD Load Doubleword 6–65
LDDA Load Doubleword from Alternate space 6–66
LDDC Load Doubleword Coprocessor 6–67
LDDF Load Doubleword Floating-Point 6–68
LDF Load Floating-Point register 6–69
LDFSR Load Floating-Point State Register 6–70
LDSB Load Signed Byte 6–71
LDSBA Load Signed Byte from Alternate space 6–72
LDSH Load Signed Halfword 6–73
LDSHA Load Signed Halfword from Alternate space 6–74
LDSTUB Atomic Load/Store Unsigned Byte 6–75
LDSTUBA Atomic Load/Store Unsigned Byte 6–76
LDUB Load Unsigned Byte 6–77
LDUBA Load Unsigned Byte from Alternate space 6–78
LDUH Load Unsigned Halfword 6–79
LDUHA Load Unsigned Halfword from Alternate space 6–80
MULScc Multiply Step and modify icc 6–81
OR Inclusive-Or 6–82
ORcc Inclusive-Or and modify icc 6–83
ORN Inclusive-Or Not 6–84
ORNcc Inclusive-Or Not and modify icc 6–85
RDPSR Read Processor State Register 6–86
RDTBR Read Trap Base Register 6–87
RDWIM Read Window Invalid Mask register 6–88
RDY Read Y register 6–89
RESTORE Restore caller's window 6–90
RETT Return from Trap 6–91
SAVE Save caller's window 6–93
SETHI Set High 22 bits of r register 6–94
SLL Shift Left Logical 6–95
SRA Shift Right Arithmetic 6–96
SRL Shift Right Logical 6–97
ST Store Word 6–98
STA Store Word into Alternate space 6–99
STB Store Byte 6–100
STBA Store Byte into Alternate space 6–101
STC Store Coprocessor register 6–102
STCSR Store Coprocessor State Register 6–103
STD Store Doubleword 6–104
STDA Store Doubleword into Alternate space 6–105
STDC Store Doubleword Coprocessor 6–106

STDCQ Store Doubleword Coprocessor Queue 6–107
STDF Store Doubleword Floating-Point 6–108
STDFQ Store Doubleword Floating-Point Queue 6–109
STF Store Floating-Point register 6–110
STFSR Store Floating-Point State Register 6–111
STH Store Halfword 6–112
STHA Store Halfword into Alternate space 6–113
SUB Subtract 6–114
SUBcc Subtract and modify icc 6–115
SUBX Subtract with Carry 6–116
SUBXcc Subtract with Carry and modify icc 6–117
SWAP Swap r register with memory 6–118
SWAPA Swap r register with memory in Alternate space 6–119
TADDcc Tagged Add and modify icc 6–120
TADDccTV Tagged Add and Trap on Overflow 6–121
Ticc Trap on integer condition codes 6–122
TSUBcc Tagged Subtract and modify icc 6–124
TSUBccTV Tagged Subtract and Trap on Overflow 6–125
UNIMP Unimplemented instruction 6–126
WRPSR Write Processor State Register 6–127
WRTBR Write Trap Base Register 6–128
WRWIM Write Window Invalid Mask register 6–129
WRY Write Y register 6–130
XNOR Exclusive-Nor 6–131
XNORcc Exclusive-Nor and modify icc 6–132
XOR Exclusive-Or 6–133
XORcc Exclusive-Or and modify icc 6–134

Chapter 7: CY7C600 Electrical and Mechanical Characteristics

7.1 CY7C601 Electrical and Mechanical Characteristics 7–1
7.1.1 CY7C601 Maximum Ratings 7–1
7.1.2 CY7C601 Operating Range 7–1
7.1.3 CY7C601 DC Characteristics 7–1
7.1.4 CY7C601 Capacitance 7–1
7.1.5 CY7C601 AC Characteristics 7–2
7.1.6 CY7C601 AC Loads and Waveforms 7–3
7.1.7 CY7C601 AC Waveforms 7–3
7.1.8 CY7C601 PGA Package Dimensions 7–9
7.1.9 CY7C601 PGA Pin Assignments 7–9
7.1.10 CY7C601 QFP Package Dimensions 7–11
7.1.11 CY7C601 QFP Pin Assignments 7–12
7.1.12 CY7C601 Military Specifications 7–13

7.2 CY7C611 Electrical and Mechanical Characteristics 7–14
7.2.1 CY7C611 Maximum Ratings 7–14
7.2.2 CY7C611 Operating Range 7–14
7.2.3 CY7C611 DC Characteristics 7–14
7.2.4 CY7C611 Capacitance 7–14

7.2.5 CY7C611 AC Characteristics 7–15
7.2.6 CY7C611 AC Loads and Waveforms 7–16
7.2.7 CY7C611 AC Waveforms 7–16
7.2.8 CY7C611 PQFP Package Dimensions 7–21
7.2.9 CY7C611 PQFP Pin Assignments 7–22

7.3 CY7C602 Electrical and Mechanical Characteristics 7–23
7.3.1 CY7C602 Maximum Ratings 7–23
7.3.2 CY7C602 Operating Range 7–23
7.3.3 CY7C602 DC Characteristics 7–23
7.3.4 CY7C602 Capacitance 7–23
7.3.5 CY7C602 AC Characteristics 7–24
7.3.6 CY7C602 AC Test Loads and Waveforms 7–25
7.3.7 CY7C602 AC Waveforms 7–25
7.3.8 CYC7602 Pin Assignments 7–27
7.3.9 CY7C602 Package Diagrams 7–28

7.4 CY7C604 Electrical and Mechanical Characteristics 7–29
7.4.1 CY7C604 Maximum Ratings 7–29
7.4.2 CY7C604 Operating Range 7–29
7.4.3 CY7C604 DC Characteristics 7–29
7.4.4 CY7C604 Capacitance 7–29
7.4.5 CY7C604 AC Characteristics 7–30
7.4.6 CY7C604 AC Test Loads and Waveforms 7–31
7.4.7 CY7C604 AC Waveforms 7–31
7.4.8 CY7C604 Pin Configuration 7–36
7.4.9 CY7C604 Package Diagrams 7–38

7.5 CY7C605 Electrical and Mechanical Characteristics 7–39
7.5.1 CY7C605 Maximum Ratings 7–39
7.5.2 CY7C605 Operating Range 7–39
7.5.3 CY7C605 DC Characteristics 7–39
7.5.4 CY7C605 Capacitance 7–39
7.5.5 CY7C605 AC Characteristics 7–40
7.5.6 CY7C605 AC Test Loads and Waveforms 7–41
7.5.7 CY7C605 AC Waveforms 7–41
7.5.8 CY7C605 Pin Configuration 7–46
7.5.9 CY7C605 CPGA Package Diagram 7–48

7.6 CY7C157 Electrical and Mechanical Characteristics 7–49
7.6.1 CY7C157 Maximum Rating 7–49
7.6.2 CY7C157 Operating Range 7–49
7.6.3 CY7C157 DC Characteristics 7–49
7.6.4 CY7C157 Capacitance 7–49
7.6.5 CY7C157 AC Test Loads and Waveforms 7–50
7.6.6 CY7C157 AC Characteristics 7–50
7.6.7 CY7C157 AC Waveforms 7–51
7.6.8 CY7C157 Truth Table 7–52
7.6.9 CY7C157 Pin Timing Cross Reference 7–52
7.6.10 CY7C157 Pin Assignments 7–52
7.6.11 CY7C157 Package Diagrams 7–53

Chapter 8: CY7C600 Ordering Information

8.1 CY7C601 Ordering Information 8-1
8.2 CY7C611 Ordering Information 8-1
8.3 CY7C602 Ordering Information 8-2
8.4 CY7C604 Ordering Information 8-2
8.5 CY7C605 Ordering Information 8-2
8.6 CY7C157 Ordering Information 8-2

Appendix: CY7C600 Uni-Module

A.1 Uni-Module Board Hardware Description A-1
A.1.1 Introduction A-1
A.1.2 Features A-1
A.1.3 Basic Mbus Operation and Timing A-1

Glossary G-1

Index I-1

List of Figures

Chapter 1: Introduction

Figure 1–1. Architectural Partitioning—Uniprocessor System 1–2
Figure 1–2. Architectural Partitioning—Multiprocessors 1–3
Figure 1–3. Embedded Control Configuration 1–3

Chapter 2: CY7C601/CY7C611 Integer Unit

Figure 2–1. Integer Unit Block Diagram 2–1
Figure 2–2. SPARC Register Model 2–2
Figure 2–3. Circular Stack of Overlapping Windows 2–3
Figure 2–4. Overlapping Windows 2–4
Figure 2–5. Registers as Seen by a Procedure 2–5
Figure 2–6. Register Banks for Fast Context Switching 2–7
Figure 2–7. Processor State Register 2–9
Figure 2–8. Window Invalid Mask 2–11
Figure 2–9. Trap Base Register 2–11
Figure 2–10. Processor Data Types 2–13
Figure 2–11. Byte Operand Load and Store 2–14
Figure 2–12. Data Organization in Memory 2–14
Figure 2–13. Extended-Precision Data Organization in Registers 2–15
Figure 2–14. Extended-Precision Data Organization in Memory 2–15
Figure 2–15. Instruction Format Summary 2–16
Figure 2–16. Address Generation 2–18
Figure 2–17. Tagged Data Example 2–22
Figure 2–18. Ticc Trap Address Generation 2–24
Figure 2–19. Delayed Control Transfer 2–27
Figure 2–20. Delayed Control Transfer Couples 2–29
Figure 2–21. CY7C601/CY7C611 External Signals 2–43
Figure 2–22. Processor Instruction Pipeline 2–52
Figure 2–23. Pipeline with All Single-Cycle Instructions 2–53
Figure 2–24. Pipeline with One Double-Cycle Instruction (Load) 2–54
Figure 2–25. Pipeline with One Triple-Cycle Instruction (Store) 2–55
Figure 2–26. Pipeline with Hardware Interlock (Load) 2–56
Figure 2–27. Pipeline During Branch Instruction 2–57
Figure 2–28. Branch with Annulled Delay Instruction 2–57
Figure 2–29. Pipeline Frozen During Bus Arbitration 2–58
Figure 2–30. Pipeline Operation for Taken Trap (Internal) 2–59
Figure 2–31. Data Bus Contents During Data Transfers 2–60
Figure 2–32. Instruction Fetch 2–61
Figure 2–33. Load Single Integer Timing 2–61
Figure 2–34. Load Single with Interlock Timing 2–62
Figure 2–35. Load Double Integer Timing 2–62
Figure 2–36. Store Single Integer Timing 2–63
Figure 2–37. Store Double Integer Timing 2–64
Figure 2–38. Atomic Load-Store Timing 2–65
Figure 2–39. Floating-Point Operation Timing 2–66
Figure 2–40. Bus Arbitration Timing 2–67

Figure 2–41. Load with Cache Miss Timing 2-68
Figure 2–42. Store with Cache Miss Timing 2-69
Figure 2–43. Load with Memory Exception Timing 2-71
Figure 2–44. Store with Memory Exception Timing 2-73
Figure 2–45. Floating-Point Exception Handshake Timing 2-75
Figure 2–46. Asynchronous Interrupt Timing 2-75
Figure 2–47. Power-On Reset Timing 2-76
Figure 2–48. Error/Reset Timing 2-77
Figure 2–49. Best-Case Interrupt Response Timing 2-80
Figure 2–50. Worst-Case Interrupt Response Timing 2-81
Figure 2–51. Coprocessor Register Model 2-86

Chapter 3: CY7C602 Floating-Point Unit

Figure 3–1. CY7C602 Functional Block Diagram 3-2
Figure 3–2. CY7C602 Block Diagram 3-3
Figure 3–3. CY7C602 Address/Instruction Pipe 3-4
Figure 3–4. CY7C601 – CY7C602 Hardware Interface 3-4
Figure 3–5. Instruction Fetch (Cache Hit) 3-6
Figure 3–6. Instruction Fetch (Cache Miss) 3-7
Figure 3–7. Floating-Point Instruction Dispatching 3-8
Figure 3–8. Floating-Point Compare (FCMP) Execution 3-8
Figure 3–9. Floating-Point Instruction Pipeline During A Trap 3-9
Figure 3–10. Effect of FLUSH on LDF Instruction 3-10
Figure 3–11. Effect of FLUSH on STF Instruction 3-10
Figure 3–12. Effect of FLUSH on FPop Instruction 3-10
Figure 3–13. Effect of FLUSH on FCMP Instruction 3-11
Figure 3–14. f Register Organization 3-13
Figure 3–15. f Register Addressing 3-13
Figure 3–16. Floating-Point Status Register 3-14
Figure 3–17. FPU Operation Modes 3-18
Figure 3–18. Floating-Point Exception Handshake 3-18
Figure 3–19. Single-Precision Floating-Point Format 3-20
Figure 3–20. Double-Precision Floating-Point Format 3-20
Figure 3–21. Extended-Precision Floating-Point Format 3-21
Figure 3–22. Extended-Precision Data Organization in Registers 3-21
Figure 3–23. Extended-Precision Data Organization in Memory 3-21

Chapter 4: CY7C604/CY7C605 Cache Controller and MMU

Figure 4–1. Virtual 64-kbyte Cache 4-2
Figure 4–2. Translation Lookaside Buffer (TLB) 4-3
Figure 4–3. Address Comparison 4-4
Figure 4–4. TLB Replacement and Locking 4-7
Figure 4–5. Four-Level Table Walk (4-kbyte Addressing) 4-7
Figure 4–6. Three-Level Table Walk (256-kbyte Addressing) 4-9
Figure 4–7. Page Table Pointer 4-9
Figure 4–8. Page Table Entry Format 4-10
Figure 4–9. Page Table Pointer Cache 4-11
Figure 4–10. Table Walk Algorithm 4-12
Figure 4–11. MMU Flush Address Format 4-14
Figure 4–12. CYC7604 Cache Tag Comparison 4-17

Figure 4–13. CY7C604 Write-Through with No Write Allocate 4–18
Figure 4–14. CY7C604 Copy-Back with Write Allocate 4–18
Figure 4–15. CY7C605 Processor Virtual Cache Tag (PVTAG) Comparison 4–20
Figure 4–16. CY7C605 Cache Tag Entries 4–21
Figure 4–17. CY7C605 Mbus Physical Cache Tag (MPTAG) Comparison 4–22
Figure 4–18. Copy-back Invalid 4–23
Figure 4–19. Copy-back Exclusive Clean 4–24
Figure 4–20. Copy-back Shared Clean 4–25
Figure 4–21. Copy-back Exclusive Modified 4–26
Figure 4–22. Copy-back Shared Modified 4–28
Figure 4–23. Write-Through Invalid 4–28
Figure 4–24. Write-Through Valid 4–29
Figure 4–25. Write Buffers (Write-Through Mode) 4–32
Figure 4–26. Write Buffer (Copy-Back Mode) 4–32
Figure 4–27. Read Buffer (Copy-Back Mode) 4–32
Figure 4–28. CBWE Byte Assignments 4–34
Figure 4–29. CY7C604 System Control Register (SCR) 4–35
Figure 4–30. CY7C605 System Control Register (SCR) 4–36
Figure 4–31. CY7C604/605 Context Table Pointer Register 4–37
Figure 4–32. CY7C604/605 Context Register 4–37
Figure 4–33. CY7C604/605 Reset Register 4–37
Figure 4–34. CY7C604/605 Root Pointer Register 4–38
Figure 4–35. CY7C604/605 Instruction Access PTP Register 4–38
Figure 4–36. CY7C604/605 Data Access PTP Register 4–38
Figure 4–37. CY7C604/605 Index Tag Register 4–38
Figure 4–38. CY7C604/605 TLB Replacement Control Register 4–39
Figure 4–39. CY7C604/605 Synchronous Fault Status Register 4–39
Figure 4–40. CY7C604/605 Synchronous Fault Address Register 4–40
Figure 4–41. CY7C604/605 Asynchronous Fault Status Register 4–40
Figure 4–42. CY7C604/605 Asynchronous Fault Address Register 4–40
Figure 4–43. Two–CMU Multichip Configuration 4–41
Figure 4–44. Examples of Multichip Addressing 4–42
Figure 4–45. TLB Entry Format 4–43
Figure 4–46. CY7C604 Cache Tag Entry Format 4–44
Figure 4–47. CY7C605 Cache Tag Entry Format 4–44
Figure 4–48. CY7C604 and CY7C605 I/O Signals 4–55
Figure 4–49. Mbus Burst Transaction Example 4–86
Figure 4–50. Mbus Address Cycle 4–87
Figure 4–51. Mbus Data Ordering 4–88
Figure 4–52. Mbus Read Transaction 4–88
Figure 4–53. Mbus Write Transaction 4–89
Figure 4–54. Mbus Coherent Read Transaction 4–89
Figure 4–55. Mbus Coherent Read Transaction - MIH asserted 4–90
Figure 4–56. Mbus Coherent Invalidate Transaction 4–90
Figure 4–57. Mbus Coherent Read and Invalidate Transaction 4–91
Figure 4–58. Mbus Coherent Read and Invalidate Transaction - MIH asserted ... 4–91
Figure 4–59. Mbus Coherent Write and Invalidate Transaction 4–92

Chapter 5: CY7C157 Cache RAM

Figure 5–1. CY7C157 Block Diagram 5–1

Chapter 6: SPARC Instruction Set

Figure 6–1. SPARC Instruction Mnemonic Summary 6–3
Figure 6–2. Instruction Description 6–4

List of Tables

Chapter 2: CY7C601/CY7C611 Integer Unit

Table 2–1. Register Addressing 2–3
Table 2–2. Floating-Point Formats 2–12
Table 2–3. Extended-Precision Floating-Point Format 2–15
Table 2–4. op Field Coding 2–17
Table 2–5. op2 Field Coding 2–17
Table 2–6. ASI Assignments 2–19
Table 2–7. Load/Store Instructions 2–20
Table 2–8. Arithmetic/Logical/Shift Instructions 2–21
Table 2–9. Control Transfer Instructions 2–23
Table 2–10. Control Transfer Instruction Characteristics 2–23
Table 2–11. Bicc and Ticc Condition Codes 2–24
Table 2–12. FBfcc Condition Codes 2–24
Table 2–13. CBccc Condition Codes 2–24
Table 2–14. Delayed Control Transfer Instruction Example 2–25
Table 2–15. Effect of Annul Bit Reset (a = 0) 2–26
Table 2–16. Effect of Annul Bit Set (a = 1) 2–26
Table 2–17. Effect of Annul Bit on Delay Instruction 2–27
Table 2–18. Delayed Control Transfer Couple Instruction Sequence 2–28
Table 2–19. Execution of Delayed Control Transfer Couples 2–28
Table 2–20. Read/Write Control Register Instructions 2–29
Table 2–21. Floating-Point-Operate and Coprocessor-Operate Instructions 2–30
Table 2–22. Miscellaneaous Instructions 2–30
Table 2–23. Load/Store Instruction Opcodes 2–31
Table 2–24. Arithmetic/Logical/Shift Instruction Opcodes 2–33
Table 2–25. Control Transfer Instruction Opcodes 2–35
Table 2–26. Bicc and Ticc Condition Codes 2–35
Table 2–27. FBfcc Condition Codes 2–36
Table 2–28. CBccc Condition Codes 2–36
Table 2–29. Read/Write Control Register Instruction Opcodes 2–36
Table 2–30. Floating-Point /Coprocessor Instruction Opcodes 2–37
Table 2–31. Miscellaneous Instruction Opcodes 2–38
Table 2–32. Instruction Opcode Numeric Listing 2–38
Table 2–33. CY7C601 External Signal Summary 2–44
Table 2–34. ASI Assignments 2–46
Table 2–35. SIZE Bit Encoding 2–48
Table 2–36. Internally Generated Opcodes 2–54
Table 2–37. Externally Generated Synchronous Exception Traps 2–78
Table 2–38. Trap Type and Priority Assignments 2–83
Table 2–39. Signal Differences Between CY7C601 and CY7C611 2–87
Table 2–40. CY7C611 Signal Summary 2–88

Chapter 3: CY7C602 Floating-Point Unit

Table 3–1. Load Instruction Execution 3–5
Table 3–2. Store Instruction Execution 3–5
Table 3–3. FPop Execution 3–6

Table 3–4. FHOLD Resource/Operand Dependency Cases 3–12
Table 3–5. Floating-Point Status Register Summary 3–15
Table 3–6. Floating-Point Load and Store Instruction Cycle Count 3–16
Table 3–7. Floating-Point Operate (FPops) Instruction Cycle Count 3–17
Table 3–8. FCC(1:0) Condition Codes 3–23

Chapter 4: CY7C604/CY7C605 Cache Controller and MMU

Table 4–1. Short Translation Bits - ST(1:0) 4–5
Table 4–2. Access-Level Protection Bits—ACC(2:0) 4–5
Table 4–3. Page Table Entry Type 4–10
Table 4–4. MMU Operation Modes 4–13
Table 4–5. TLB Entry Flushing 4–15
Table 4–6. Mbus Snooping Transactions 4–30
Table 4–7. Cache Flush Operations 4–32
Table 4–8. Cacheable/Non-Cacheable Accesses 4–33
Table 4–9. State Table for MC (Memory Cacheable) Bit 4–33
Table 4–10. Byte Write Enables 4–34
Table 4–11. TLB Entry Address Mapping 4–43
Table 4–12. Cache Tag Entry Address Mapping 4–44
Table 4–13. CY7C604/605 Power-On Reset States 4–45
Table 4–14. CY7C604/605 Register Address Mapping 4–46
Table 4–15. Standard ASI Assignments 4–47
Table 4–16. OW Bit States 4–48
Table 4–17. Fault Register Level Field 4–49
Table 4–18. Fault Register Access Type Field 4–49
Table 4–19. Fault Register Fault Type Field 4–49
Table 4–20. Fault Type (FT) for PTE[ET] = 2 4–50
Table 4–21. Fault Register Error Priorities 4–50
Table 4–22. Mbus Signal Summary 4–85
Table 4–23. Bus Status Encoding 4–86

Chapter 6: SPARC Instruction Set

Table 6–1. Instruction Description Notations 6–4
Table 6–2. Instruction Set Summary 6–6

RISC: Fundamentals and Future

by Roger D. Ross, President and CEO of Ross Technology, Inc.

RISC is the future of computing. Over the next 5 years, a totally new computing standard will emerge based upon RISC (Reduced Instruction Set Computer) architectures. RISC will completely redefine the computer industry's existing price/performance curve, which is based on Complex Instruction Set Computers (CISC), and will be the industrial computing standard that leads us into the 21st century.

Analyzing RISC's potential is much more than simply discussing how many MIPS and MFLOPS will be offered over the next two decades. The technical future of reduced instruction set computers is but one facet of a much bigger drama that is unfolding. First one must understand the technical fundamentals and benefits of RISC as they relate to the more general trends of the entire computer industry, trends that tend to complement RISC. This introduction briefly explains the technical fundamentals of RISC architecture and reviews the broader trends of the computer industry. It will show that RISC architecture has been designed to exploit the computer industry trends and reveal why the future of RISC architecture is fundamentally the future of the entire computer industry.

RISC Described (and CISC exposed)

Today, a tremendous amount of misinformation exists surrounding the fundamentals of RISC architecture. Obviously, the promoters of this misinformation are those who stand to lose the most from its impact: the established manufacturers of proprietary CISC architectures. These manufacturers tell their prospective customers that they can use RISC design techniques on their CISC architectures to get close to RISC's single clock cycle execution feature while maintaining compatibility with their existing binary application software base. There are two subtle but totally misleading concepts in the previous statement. The phrase "RISC design techniques" is blatantly misused, and the phrase "RISC's single clock cycle execution feature" is misleading as well because it falls far short of RISC's true goal. Both of these concepts will be explained and corrected in the ensuing paragraphs.

RISC is quite simply not a set of design techniques. RISC is a new instruction set architecture technique that is distinct and completely different from CISC. It is not backwardly adaptable to CISC, which is now defined by, and indeed captive to, its "prior art" forms. Instruction sets are, after all, the fundamental form of computer architecture. RISC evolved as a solution to the problem of how to derive more power; that is, how to derive more instruction set power out of a computer and its associated compilers. The goal of RISC is not simply to reduce the system's instruction set, it is to intelligently select a set of streamlined instructions that yield maximal data-processing performance within the context of compiled programming techniques. RISC is a way to significantly enhance a system's performance while keeping costs on or below par with CISC. These new instruction set techniques are described below. CISC instruction sets were selected over 20 years ago, and cannot now be changed if CISCs are to maintain compatibility with their existing binary application software base. Consequently, the fallacy of CISC using "so called" RISC technology at the instruction set level is readily apparent. In fact, these instruction set techniques are the real and only difference between RISC and CISC.

RISC has three major instruction set features that distinguish it from CISC. RISC's instruction set attributes include a load/store model of execution, a non-destructive triadic register file that provides a distinct and highly efficient data preservation model, and, lastly, normalized fixed-length instructions. Conversely, CISC uses a memory/register model of execution, an accumulator/register file that engenders a destructive data environment, and variable-length, contextual-field instructions.

RISC's load/store model of execution means that the only instructions that can access main memory are load and store instructions. All other CPU instructions operate on internal registers. By using this model it is possible to decouple loading and storing traffic from data processing operations such as arithmetic or logical instructions, and thereby raise

the operational concurrency of the entire CPU. It also makes it possible to schedule code to fill stall slots that naturally occur due to the latency between the time when a load instruction is issued and the time, typically 2 to 3 clocks later, when the data is returned from memory and is actually ready for use.

However, the true uniqueness in RISC's load/store instruction set philosophy is the recognition that the register file is in actuality a computer's highest-level data cache. This register file data cache differs from other, lower-level data caches in that its use is deterministic and not stochastic. Load instructions are simply a way to fill this cache, and store instructions are merely a way to write back updated data to the lower memory hierarchy. With this in mind, one can argue that load/store operations are not even instructions at all, they are just mechanisms available to software that allow it to administer the register data cache. Consequently, the optimization and direction of this register file data cache can be determined solely by the compiler or assembly language programmer. All of the leading RISC architectures (SPARC, MIPS, Motorola 88K, and Intel 860) have a larger register file than any of the pre-existing commercial CISC architectures. In addition, SPARC has even further evolved beyond the large register file concept by providing a register file extension that is comprised of overlapped register windows. SPARC's overlapped register windows are primarily used to pass parameters during subroutine accesses, thereby further cutting down on load and store traffic and more completely acknowledging the fact that the modern computer's register file has now fully evolved into a deterministic cache subsystem. There is now no way for CISC architectures to directly apply large flat register files to their instruction sets. They could have done so at one time, but now their binary instruction sets are frozen and it is too late. The decision is irrevocable.

RISC's non-destructive, three-register (triadic) architecture model means that information in the CPU is preserved (i.e., maintained in the register data cache) during ongoing data processing. For example, a RISC add instruction would be verbalized as "register A is equal to the result of register B plus register C." All information that was contained in registers B and C is preserved (it is interesting to note that this more natural model is also the one that we use to teach algebra to our children). Data preservation within the register file (i.e., data cache) is a fundamental and obvious requirement to minimize load/store traffic. In contrast the CISC machine's fundamental model is simply stated as "add the contents of register A and register B and place the result in register A." Obviously, the original contents of register A are destroyed, and consequently the name "destructive."

It is also necessary to allow an optimizing compiler to effectively reschedule code to fill pipeline stalls that frequently occur in computational engines. In a computer one can reschedule code so long as it is determined that no data dependencies occur and the original semantic content of the program is maintained. Therefore, a non-destructive register model taken together with a load/store architecture provides a dramatic boost in instruction set architectural performance due to its ability to minimize load/store traffic as well as decouple operations and thereby allow optimizing compilers to efficiently fill stall slots.

Alternatively, CISC machines have a memory/register instruction set architecture. This means that in a CISC architecture one can do an add instruction with an addressing mode that appears to obtain an operand directly from main memory and add it into a register. In reality, this add instruction is forced to do an operand load before it can complete the instruction. However, this load is coupled to the add operation and so the unavoidable stall slot between the load and the add cannot be filled with useful work. Typically 40% to 50% of all instructions dynamically executed in a CISC machine's existing software base utilize and therefore mandate this hidden load of operands.

CISC machines evolved from the accumulator model of execution. In this model the programmer "accumulates" results in a register, thereby destroying the data already existing in that register. The problem with a destructive register model is that it keeps the compiler from performing efficient algorithmic code rescheduling operations that could lead to higher throughput. Data and condition codes in CISC machines is location sensitive because it is constantly being destroyed by new instructions. In addition, this model simultaneously increases a machine's load/store activity when registers must either be saved or restored from main memory by the compiler in its struggle to preserve critical data. Again historically speaking, CISC could have adapted a large triadic register model, but once again it did not, and now it is too late. CISC is a captive of its installed binary software base and established instruction sets.

All true RISC machines utilize fixed-length instructions. Fixed-length instruction sets make possible normalized instruction encoding (i.e., minimize the use of contextual fields) with greatly simplified addressing modes. In addition, operand accesses only occur between registers (i.e., cached data). By making each instruction 32 bits long, instruction decode is much easier and can occur much faster than in CISC architectures. RISC CPUs exploit fine-grain parallelism by decoding all parts of the instruction in parallel. In CISC machines, instruction decode occurs sequentially as the instructions are of variable length and contextual in nature. Hence final instruction decode cannot usually occur until all parts of the instruction are fully analyzed. In CISC machines, depending on the addressing mode and particular instruction used, this can take from 2 to 11 clocks. In RISC machines with 32-bit, fixed-length instructions, this always takes exactly 1 clock.

There are three major effects of RISC's streamlined, or reduced, instruction set architecture techniques. First, due to its instruction set normality, RISC machines have no need for microcode. That is, all instructions can be hardwired in a very efficient manner.

Second, RISC's streamlined instruction set allows for single clock cycle execution. But this is just the tip of the iceburg in that the true goal of RISC is the concurrent execution of many instructions at once. It is in this "superscalar" execution form that RISC's full potential ultimately lies. Although by using of millions of extra transistors CISC could eventually come close to one instruction per clock, superscalability is effectively beyond CISC's practical scope.

Third, because of the concurrency made possible by the instruction set as described previously, RISCs can more aggressively and efficiently exploit the design technique of pipelining. These distinctions explain why RISC can provide a 2 to 5 times performance advantage over CISC given equal technologies of implementation.

Key Historical Trends of the Computer Industry

This section will not attempt to distill the entire history of the computer industry in just a few pages. Rather, it is intended to take a step back and look at some of the more important trends in the industry.

There have been three defacto architectural computing standards in the history of the computer industry: the IBM 360/370, the DEC VAX, and systems based on the Intel 80x86. Most professionals in our industry do not remember that the IBM 360/370 mainframe architecture, originally released in 1964, was in fact the first system to be cloned! This cloning, by companies such as Amdahl and NAS, was a direct realization that the application software was the standard to which the hardware had to comply. This cloning also led to the IBM 370 and PCMs (plug-compatible mainframes) that have held between 50% to 70% of the entire computer industry market for nearly 20 years.

The DEC VAX, a minicomputer or mid-range system, was in reality a way to bring a better level of price/performance to the end user than that offered by mainframes. In the final analysis, price and performance are the drummers to which the entire computer industry marches. By offering a significant advantage in price/performance (i.e., two times the performance or more) over the IBM and PCM mainframes, DEC was able to establish a beachhead in the systems industry that enabled it to become second to only IBM in size.

Computers based upon the 80x86 microprocessor architecture from Intel also offered significantly enhanced price/performance over the mainframe and minicomputer systems that were in existence at the time. As is well known, IBM adopted the 8088 in its original personal computer. This product was brought to market several years after the first personal computers emerged from companies such as Apple. However, distinguishing it from the other market entrants was the fact that the IBM PC was clonable. Cloning again led to the marketshare dominance of this particular computer architecture. Today it is estimated by leading market researchers that approximately 85% of the installed worldwide personal computer base is comprised of IBM and IBM-compatible personal computers. As a result of its use in the IBM personal computer architecture, Intel's 80x86 family today exceeds the sales of all other 16- and 32-bit general-purpose microprocessors combined.

The historical trend toward enhanced system price/performance is to obtain greater performance for absolutely lower costs. In 1990, systems that sell for under $10,000 dominate the entire computer industry, amounting to over 95% of all units shipped and 40% of the total sales dollars of the computer systems industry. In the next ten years this trend should accelerate with systems priced under $7,500 amounting to over 99% of all units shipped and 75% of the total sales dollars of the entire computer systems industry.

With the dramatic increase in the use of low-cost, typically desktop computers, there has been a parallel increase in the use of computer networks. Distributed data processing, also known as networked computing, in which desktop systems are tied to server computers, is now much more common than massive mainframes with several hundred terminals. Interestingly, yesterday's minicomputers and mainframes have become today's servers. However, even these ECL server systems are increasingly giving way to CMOS microprocessor-based systems. These new servers also use industry standard microprocessors, as opposed to designing their own high-cost proprietary CPUs, as a way to offer enhanced price/performance.

Enhanced price/performance has another facet to it: enhanced productivity for the user. Also known as user friendliness, these are quite simply the use of graphics instead of text, and the use of windows and user interfaces rather than simple command lines. These features have made computers much more accessible. However, this user friendliness has not been easy to achieve. First of all, the software behind the user friendliness is large and complex. To run windows and graphics interfaces requires much higher CPU performance than has, until recently, been available in the microprocessor market. Writing software of this complexity has necessitated the use of high-level languages, of which the overwhelming language of choice has been C. Of course each line of C, as with any other high-level language, is comprised of multiple lines of assembly code, so it requires more CPU horsepower to run effectively.

The Future of RISC

The first generation of RISC machines have been what is termed single-instruction launch microarchitectures. Through pipelining it has been possible to significantly overlap the various stages of an instruction's lifecycle, and hence the current

generation of RISC implementations have asymptotically approached a performance rate of 1 clock per instruction (1 CPI). This overlap is required to provide continued execution opportunities instead of suffering through the delays which would otherwise arise due to multiple clock cycle instructions and memory accesses. This does not always work perfectly, however, and consequently the first generation of RISC implementations have an aggregate throughput that is on the order of 1.25 to 1.5 CPI.

The next step in microarchitecture for RISC machines will be the ability to execute two or more instructions simultaneously. This feature is sometimes referred to as "superscalability." RISC implementations will be able to fetch, decode, execute, and finish two or more instructions at the same time. Multiple-instruction launching requires the ability to internally schedule the instructions while simultaneously checking for data dependencies and the availability of computing resources before the instructions are launched. For instance, the ability to launch four integer instructions in the same clock cycle should yield an instruction execution theoretical peak CPI rate of 0.25. The bus bandwidth required to feed both instructions and data into the machine and a high-performance cache architecture and cache refill capability to keep these high-speed channels fully utilized will be very important in multi-launch implementations.

RISC microarchitecture will follow the path of increasing the number of simultaneous execution units and will inevitably evolve into a dataflow type of architecture whereby multiple data operands flow through the machine being used by available execution units. Research on dataflow architectures is currently in advanced stages at leading universities. However, whereas CISC instruction sets have been obsoleted by RISC in the search for higher architectural performance, this will not happen to RISC. RISC instruction sets can and will be preserved in the evolution to dataflow architectures. It will be possible to obtain dramatic performance enhancements in RISC, first through multi-launching, then through dataflow, without making any changes to the fundamental instruction set. These performance improvements will occur under the surface of the instruction set, and will enable a complete continuum of the application software investment. This continuum could last for at least 25 to 30 years, and it will be a truly remarkable period of software base stability.

The performance capability and growth path of RISC architectures have not gone unnoticed. At this point, RISC architectures have clearly hit the mainstream of computing. As of this writing, every major manufacturer of computer systems in the world has somehow endorsed RISC architectures. This list includes IBM, DEC, ICL, Sun, Unisys, NCR, Toshiba, AT&T, Olivetti, and many more. These manufacturers have moved to RISC not because it is a fad, but because they realize that RISC offers fundamentally better price/performance than does CISC. Coincidentally, every major manufacturer of semiconductors has also aligned itself with a RISC architecture in some form or fashion.

RISC architectures are already used in desktop systems from companies such as Sun and HP, in servers from companies such as Solbourne, and in mainframes from companies like ICL. RISC architectures have already proven that they provide from 2 to 5 times the performance of CISC architectures given equal implementation technology (i.e., cost).

Owing to their streamlined, efficient instruction set, RISC architectures result in a fundamentally shorter design cycle for RISC chips as compared to CISC. It is also due to this simplicity that we have seen RISC architectures already fan out into custom CMOS, ECL, gate arrays, and GaAs. The significance of these events is that it is now possible to have a binary software-compatible range of RISC-based computers from the desktop to the mainframe. This has never been achieved in the industry, and this capability is obviously very synergistic with the trend toward networked computing.

Neither of the previous defacto computing standards (IBM 370 and the Intel 80x86) had the benefit of being able to use the application software base available from its competitive predecessors. RISC, however, is able to make use of the existing computing standard software base. That is, by using advanced binary emulation techniques, the entire $15 billion MS-DOS applications software market is now accessible to RISC architectures. So we have the scenario where RISC is able to run its native software several times faster than CISC can run software, and at the same time it can run existing CISC software nearly as fast as the CISC machines can!

The RISC Contenders

There are currently four RISC architectures that are the mainstream contenders in the RISC marketshare race. These architectures are the SPARC architecture from SPARC International, the MIPS Rx000 from MIPS Inc., the MC88000 from Motorola, and the i860 from Intel.

Marketshare for the competing RISC architectures arises from several key factors. These factors are the alliances with key systems manufacturers, the availability of low-cost (under $10,000) desktop systems, a large base of shrinkwrap application software, a wide range of system price options (from under $10,000 to over $1,000,000), competitive semiconductor implementations of the CPUs, multiple sources of the CPUs, and state-of-the-art technology.

At this point in time only SPARC is openly owned and controlled, has independent multiple sources for its chip sets, and has multiple microarchitecture implementations available that all execute the same binary software. Motorola's MC88000 is sole-sourced for commercial applications and second-sourced strictly for military applications by Thompson-CSF. However, Motorola owns and controls the MC88000 microarchitecture. MIPS' architecture is also second-

sourced, but the microarchitecture is solely controlled by MIPS Inc. And Intel's i860 is completely proprietary. Unless MIPS, the MC88000, and the i860 become openly owned and independently second-sourced, it is very unlikely that they will continue to be contenders in the RISC race against SPARC. Hewlett-Packard now realizes the significance of open ownership and its relationship to market success. As a result, they also are now attempting to move their architecture away from a proprietary basis and into the open market.

To date, low-cost systems priced under $10,000 are available that use the SPARC, MIPS and MC88000 architectures. The differentiating factor between these systems is the software base. SPARC's software base is much larger than that for all other RISC architectures combined, and is usable in shrinkwrap form on multiple platforms based on multiple vendor's SPARC chips. This capability was proven by Solbourne Computer in Longmont, Colorado when they created the world's first SPARC-compatible system, thereby making SPARC the only RISC architecture with proven system-level clonability. Motorola is attempting to create a similar capability for the MC88000 through a committee-generated document called the MC88000 BCS (Binary Compatibility Standard). MIPS has no such plans in the works, and has actually seen its base fragment between its own systems, Stardent, DEC, and those of Silicon Graphics. As stated previously, shrinkwrap software led the Intel 80x86 architecture to an overwhelming marketshare lead. Likewise, shrinkwrap software will also be the biggest differentiator in the RISC marketplace and it favors SPARC both from its present large base and also from its growth rate as well.

Summary

The general trends of the computer industry are very complementary to the capabilities of RISC architectures. The computer industry market always thirsts for higher performance at lower prices, and is structuring itself to allow this to happen. RISC, a set of instruction set architecture techniques, offers significant performance advantages over CISC, and requires less transistors to do so. Because of its transistor count frugality, RISC has scaled quickly into very high performance technologies such as ECL and GaAs, and hence is ideally suited to fitting in at all price/performance points existing within the entire computer industry. Most importantly, RISC is affordable on the desktop and is able to efficiently run the huge PC software base that already exists there. In addition, RISC's performance growth path is assured, and is formidable when compared to that for CISC. For all of these reasons, RISC architectures will come to dominate 32-/64-bit computing over the ensuing years.

CYPRESS SEMICONDUCTOR

Chapter 1

Introduction

1

1.1 SPARC Overview

SPARC, an acronym for Scalable Processor ARChitecture, is an open RISC architecture with multiple semiconductor implementations from a number of vendors. SPARC is an architecturally driven standard, with binary compatibility of software between processor versions ensured by enforcing compliance to the architecture standard. The open architecture approach offered by SPARC allows all its participants to make creative contributions in developing their versions of SPARC processor. This results in a vastly greater number of technical contributions than would be possible for a closed architecture held and defined by only one group. This architectural freedom has allowed the SPARC architecture to expand into CMOS gate arrays, full-custom CMOS, bipolar ECL, and GaAs faster than any other RISC architecture. This same freedom allows SPARC vendors to make microarchitectural enhancements to their SPARC implementations while maintaining absolute binary compatibility. The final result of this open architecture approach is that it provides the customer with a wider range of price/performance and technology options that cannot be matched by less innovative and restricted licensing policies. In addition, the various SPARC vendors also participate in standard second-sourcing agreements.

The inclusion of the word "scalable" in the acronym for SPARC emphasizes its importance in the philosophy of the architecture. "Enforced compatibility" has been embraced to ensure migration of the architecture as semiconductor technology improves. Scalability allows SPARC to be re-implemented without complication as semiconductor process technology evolves. This allows SPARC to continually be offered in higher clock speeds and technologies than other RISC architectures, providing rapid performance improvements as process technology continues to be refined. Other RISC processors have complicated their microarchitectures with features that create an unnecessary burden for the hardware designer. These features provide only a minimal performance improvement, but greatly complicate hardware design and cost. The CY7C601 microprocessor does not require multiple-phase clocks, demultiplexing of the processor's address or data buses or many of the other problems that affect hardware complexity and cost. This provides CY7C601 SPARC-based designs with the advantages of excellent performance, low design costs, a high degree of manufacturability, and increased reliability due to its simplicity of design.

The CY7C600 chip set is a 32-bit custom CMOS implementation of the SPARC architecture. Designed by Ross Technology, Inc., a Cypress Semiconductor subsidiary, the chip set is implemented in Cypress's state of the art 0.8-µm CMOS technology. The chip set is in production and is available in clock speeds of 25, 33, and 40 MHz. The CY7C600 family includes the CY7C601 Integer Unit (IU), the CY7C602 Floating-Point Unit (FPU), the CY7C604 Cache controller and MMU (CMU), the CY7C605 Cache controller and MMU for MultiProcessing (CMU-MP), and the CY7C157 Cache RAM (CRAM). The CY7C601, CY7C602, CY7C604 or CY7C605, and two CY7C157s comprise a five-chip CPU, providing up to 29 MIPS of sustained integer performance and over 6 MFLOPS of double-precision floating-point performance at 40 MHz. This CPU includes a SPARC Reference MMU and a 64-kbyte cache, and directly interfaces to a 64-bit physical bus capable of a bandwidth approaching 320 Mbytes per second at 40 MHz. The five-chip CY7C600 CPU requires no glue logic, and provides maximum computing performance with minimal design effort.

1.1.1 Partitioning

The CY7C600 family has been designed to offer a complete solution for high-performance computer and controller applications. The CY7C601 IU and the CY7C602 FPU together comprise the full SPARC instruction set architecture. The CY7C602 replaces two chips that previously made up the FPU, the CY7C608 floating-point controller and the CY7C609 floating-point processor (Texas Instruments' SN74ACT8847). Additional family members include the CY7C604 CMU for uniprocessor applications, the CY7C605 CMU-MP, and the CY7C157 CRAM.

The CY7C611 is a specialized derivative of the CY7C601 integer unit that has been optimized for embedded control applications. It is in production in a cost-effective, 160-pin PQFP package, and is available at a speed of 25 MHz.

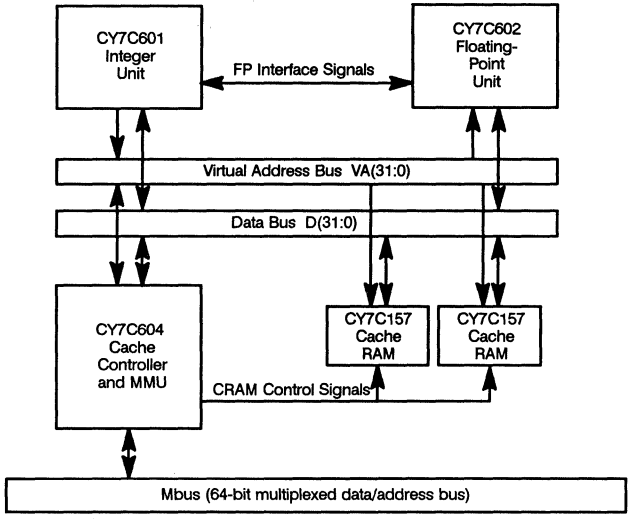


Figure 1–1. Architectural Partitioning—Uniprocessor System

Figure 1–1 and *Figure 1–2* illustrate how CY7C600 family devices connect to each other in both uniprocessor and multiprocessor applications. The CY7C601's second coprocessor interface is not shown in these diagrams. The function of this second coprocessor (CP) is defined by the system designer, but its interface to the CY7C601 is identical to that of the CY7C602 FPU coprocessor.

Figure 1–3 illustrates an embedded control system utilizing the CY7C601 or CY7C611 with an optional CY7C602 FPU and user-designed memory system.

1.1.2 The CY7C601 Integer Unit

The CY7C601 is the primary processing engine in the SPARC architecture, executing all instructions except for specific floating-point and coprocessor operations. The CY7C602 FPU does its floating-point calculations concurrently with the CY7C601 IU. The architecture also allows for concurrent operation through the use of an optional second coprocessor.

Significant features of the CY7C601 include:

- Full binary compatibility with entire SPARC application software base
- Architectural efficiency that sustains 1.25 to 1.5 clocks per instruction
- Large windowed register file
- Tightly coupled floating-point interface
- User/supervisor modes for multitasking
- Semaphore instructions and alternate address spaces for multiprocessing
- Tagged arithmetic instructions to support artificial intelligence software

1.1.2.1 Traps and Exceptions

The CY7C601 supports a full set of traps and exceptions. A table-based set of trap vectors supports 128 hardware and 128 software trap types, both synchronous (error conditions and instructions) and asynchronous (interrupts and reset). The CY7C601 supports a very fast interrupt time of 4 to 7 clocks, depending upon the contents of the instruction pipeline.

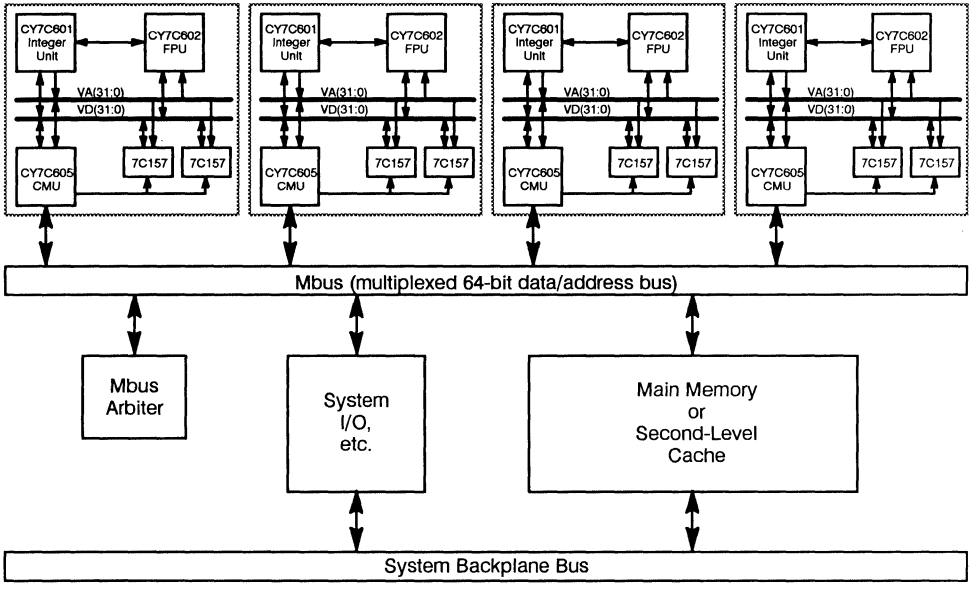


Figure 1–2. Architectural Partitioning—Multiprocessors

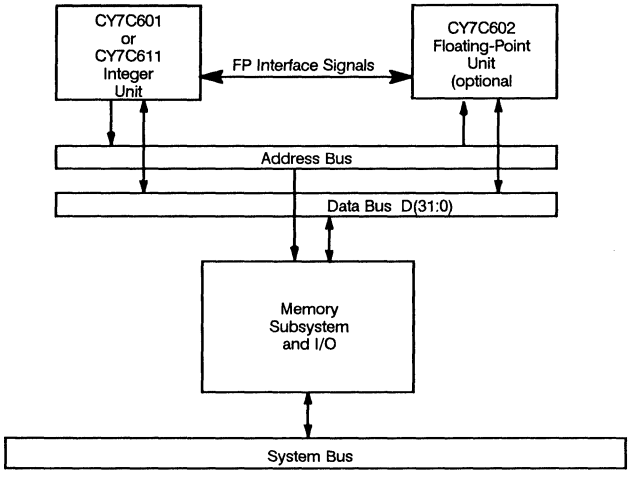


Figure 1–3. Embedded Control Configuration

1.1.2.2 Multitasking

Multitasking is supported with user and supervisor modes. Certain privileged instructions can only be executed while the CY7C601 is in supervisor mode, ensuring that user programs cannot accidentally alter the state of the machine. Supervisor mode is only accessible by using a hardware interrupt or by executing a trap instruction.

1.1.2.3 Multiprocessing

The CY7C601 supports multiprocessing with two instructions for implementing semaphores in memory. Atomic Load/Store Unsigned Byte loads a byte from memory, then sets the memory location to all ones. The SWAP instruction exchanges the contents of a register and a memory location. Both of these instructions are "atomic," meaning uninterruptable.

1.1.3 CY7C611 Integer Unit for Embedded Control

The CY7C611 Integer Unit is a subset of the CY7C601 Integer Unit intended for use in embedded control systems. It is architecturally identical to the CY7C601, and all details concerning the CY7C601 described in Sections 2.1 through 2.8 of Chapter 2 apply to the CY7C611. The CY7C611 is available in a 160-pin plastic QFP and is in production at 25 MHz. The CY7C611 differs from the CY7C601 in that several of the signals available on the CY7C601 that are not required for embedded control systems have been deleted. In addition, the CY7C611 does not have a user-defined coprocessor interface. The CY7C611 does have a floating-point interface, which can also be used to interface to a user-defined coprocessor. Please refer to Section 2.9 for detailed information on the CY7C611.

1.1.4 CY7C602 Floating-Point Unit

The CY7C602 FPU provides high-performance, IEEE STD-754-1985-compatible single- and double-precision floating-point calculations for CY7C600 systems, and is designed to operate concurrently with the CY7C601. All address and control signals for memory accesses by the CY7C602 are supplied by the CY7C601. Floating-point instructions are addressed by the CY7C601, and are simultaneously latched from the data bus by both the CY7C601 and CY7C602. Floating-point instructions are concurrently decoded by the CY7C601 and the CY7C602, but do not begin execution in the CY7C602 until after the instruction is enabled by a signal from the CY7C601. Pending and currently executing FP instructions are placed in an on-chip queue while the CY7C601 continues to execute non-floating-point instructions.

The CY7C602 has a 32 x 32-bit data register file for floating-point operations. The contents of these registers are transferred to and from external memory under control of the CY7C601 using floating-point load/store instructions. Addresses and control signals for data accesses during a floating-point load or store are supplied by the CY7C601, while the CY7C602 supplies or receives data. Although the CY7C602 operates concurrently with the CY7C601, a program containing floating-point computations generates results as if the instructions were being executed sequentially.

1.1.5 CY7C157 Cache Data RAM

The CY7C157 is a 16K x 16-bit high-performance CMOS static RAM designed specifically as a cache memory for CY7C600 systems. It incorporates registered address and write-enable inputs, latched data inputs and outputs, and a self-timed write mechanism—features that have greatly simplified the design of cache memories for the CY7C600 family.

1.1.6 CY7C604/CY7C605 Cache Controller and Memory Management Units

The CY7C604 and CY7C605 are combined cache controller and memory management units designed specifically to support the CY7C601. The CY7C604 and CY7C605 provide control for a 64-kbyte direct-mapped virtual cache and provide a SPARC reference standard MMU for virtual to physical address translation. The CY7C604 and CY7C605 directly interface with the CY7C600 family, requiring no glue logic for a 64-kbyte cache system. The CY7C604 and CY7C605 use two CY7C157 Cache RAMs to implement a 64-kbyte cache system using only three chips. Cache tag memory is provided as an on-chip feature of the CY7C604/CY7C605, thereby reducing hardware complexity for a CY7C604- or CY7C605-based system.

The CY7C604 is optimized for uniprocessor systems, providing cache locking and cache expandability to 256 kilobytes using additional CY7C604s. The cache locking feature of the CY7C604 allows deterministic response from the cache system, an important feature for real-time systems. The SPARC reference MMU, supported on both the CY7C604 and the CY7C605, provides translation of a 4-Gbyte virtual address space to a 64-Gbyte physical address space. Both the

CY7C604 and the CY7C605 provide a 64-entry fully associative TLB (Translation Lookaside Buffer), used in translating virtual addresses to physical addresses. TLB entries may be locked, excluding critical TLB entries from replacement and thereby preventing unnecessary table walks. Table walking (required to obtain additional virtual to physical address translations not stored in the TLB) for the CY7C604 and CY7C605 is implemented in hardware, providing a substantial time savings over software table walk routines.

The SPARC MMU section of the CY7C604/CY7C605 is designed for the efficient support of multitasking operating systems. CY7C604/CY7C605 TLB and cache tag entries allow a maximum of 4096 different context tags to identify tasks within an operating system. The SPARC MMU implemented in the CY7C604/CY7C605 provides extensive memory access level protection (user/supervisor and read/write/execute), including an execute-only memory access level. The ability to mark memory accesses as execute-only provides a security feature that can be used to protect proprietary features of a software system from unauthorized scrutiny. The CY7C604 and CY7C605 MMU also support multilevel address mapping, allowing software to select a region of 4 kbytes, 256 kbytes, 16 Mbytes, or 4 Gbytes to be addressed by a single TLB entry. This feature allows efficient utilization of TLB entries, which in turn reduces the number of table walks caused by system software.

The CY7C605 is an extension of the CY7C604 designed for use in multiprocessor systems. The CY7C605 provides a dual cache tag memory, which allows the CY7C605 to perform bus snooping while it simultaneously supports cache accesses by the CY7C601. The CY7C605 implements a cache coherency protocol based on the IEEE Futurebus, which has been recognized as a superior protocol for maintaining consistency of shared data in a multiprocessing system. The CY7C605 supports direct data intervention, which is the capability of a CY7C605-based cache to directly supply modified data to another requesting cache without first requiring main memory to be updated. This feature provides a significant performance advantage over cache systems that must update main memory in order to supply modified data to another cache. In addition to direct data intervention, the CY7C605 also supports memory reflection. Memory reflection allows a memory system to automatically update itself during a direct data intervention operation. This feature allows a multiprocessing system to update both a requesting cache and main memory in a single bus operation.

Both the CY7C604 and the CY7C605 are specifically designed to support secondary cache systems. The use of common secondary caching provides the advantage of increased cache performance for each processing node of a multiprocessor system without the expense of large caches for each node. This approach also provides a direct upgrade path to the next generation of high-integration SPARC processors. The CY7C605 is designed to be pin compatible with the CY7C604. This feature allows a system to be upgraded from uniprocessor to multiprocessor by modifying the operating system and replacing the CY7C604 with the CY7C605.

The CY7C604 and CY7C605 support the SPARC Mbus standard bus interface. The Mbus is a peer level, high-speed, 64-bit, multiplexed address and data bus which supports a full peer-level protocol (i.e., multiple bus masters). The CY7C604/605 Mbus supports data transfers in transaction sizes of 1, 2, 4, 8, or 32 bytes. These data transfers are performed in either burst or non-burst mode, depending upon size. Data transactions larger than eight bytes (one doubleword) are transferred in burst mode, which consists of an address phase followed by four data phases. Non-burst transactions consist of an address phase followed by one data phase, and are used for data transactions of eight or less bytes. Bus mastership is granted and controlled by an external bus arbiter. The bus arbiter sets bus priorities, and grants access to a bus master.

Mbus is divided into two levels of implementation: level 1 and level 2. Level 1, implemented on the CY7C604, is the uniprocessor version of Mbus. Level 1 is a subset of level 2, which is the multiprocessor version of Mbus. The CY7C605 supports level 2 Mbus. Level 2 Mbus includes the IEEE Futurebus (MOSEI) cache coherency protocol, which has been recognized in the industry as a superior method of supporting multiprocessing systems. Level 2 Mbus defines five cache states for describing cache line status. Transactions on the Mbus are monitored or "snooped" by the CY7C605 and other bus agents on the level 2 Mbus to maintain ownership and modified status for each cache line. Transactions on the level 2 Mbus are made with respect to the cache line ownership and modified status to ensure consistency for shared data images.

The level 2 Mbus supports direct data intervention, which allows a cache system with the up-to-date version of a cache line to directly supply the data to another cache system without having to first update main memory. Direct data intervention provides a significant performance improvement over systems that do not support this feature. In addition, the CY7C605 provides support for memory systems with reflective memory controllers. A memory system with reflective memory control can recognize a cache-to-cache data transaction and automatically update itself without delaying the system. Another system concept supported by the CY7C605 is secondary caching. Secondary caching provides a performance advantage over systems directly using main memory, and provides an economic advantage over systems using large caches for each processing node.

1.2 Register Windows

The CY7C601 contains a large, 32-bit-wide, triple-port register file that is divided into multiple windows which are controlled by internal hardware. Each window contains 24 working registers and has access to 8 global registers. Combined with the CY7C601's register-to-register architecture, this file operates effectively as a compiler-directed, copy-back data cache, considerably reducing data bus traffic. Load instructions enter data into this cache, and store instructions "copy back" information when it needs to be replaced into main memory.

The register file is managed as a circular stack, with the first and last windows overlapping each other. Each window overlaps the previous window and succeeding window by 8 registers, making the window mechanism ideal for passing parameters in procedure calls. Results left in the overlapping registers by a calling routine automatically become available operands for the called routine as the window moves, and vice versa. This parameter passing technique eliminates the need for the loads and stores to memory required by machines using a stack during procedure calls.

1.3 Instruction Set

SPARC defines 55 basic integer instructions, 14 basic floating-point instructions, and two coprocessor-operate instruction formats. CY7C600 instructions fall into five basic categories: load/store, arithmetic/logical/shift, control transfer, read/write control register, and floating-point-operate/coprocessor-operate.

1.3.1 Load and Store Instructions

Load and store instructions are the only way to access memory or external registers. Addresses are calculated using the contents of two registers or one register and a constant. The destination may be either an integer unit, floating-point unit, or coprocessor register, which either supplies or receives the data. In order to greatly speed up memory accesses, halfword, word, and doubleword data must be aligned on their corresponding boundaries. If they are not, a trap is generated when an access is attempted.

1.3.1.1 Address Space Identifier

Whenever an address is sent to the address bus, the processor also generates 8 bits of address space identifier (ASI). The ASI pins identify to the external system which of the 256 possible address spaces is to be accessed. For most CY7C601 operations, one of four standard ASI values are asserted. These four ASI values indicate whether the processor is in user or supervisor mode, and whether the access is an instruction or data reference.

The address space identifier is intended for use by the system operating software. Consequently, the instructions that specify a particular ASI value (load/store alternate) are privileged and can only be executed in the supervisor mode. Many of the ASI bit patterns are assigned for accessing various features of the CY7C604/CY7C605. A large block of address spaces are reserved for the designer to implement as desired.

1.3.2 Arithmetic/Logical/Shift Instructions

These instructions compute a result using two source operands and place the result in a destination register. In addition to standard arithmetic operations, the CY7C601 includes tagged arithmetic operations. Tagged arithmetic instructions assume that the least-significant two bits of the operands are tags, and set a condition code bit if they are not zero. Tagged instructions are used with artificial intelligence languages such as LISP to indicate the data type of the operands. The use of tagged arithmetic instructions allows languages such as LISP and Prolog to run significantly faster than on RISC machines without this type of instruction.

1.3.3 Control Transfer Instructions

Control transfer instructions include jumps, calls, branches, and traps. Transfer of control to the new address is usually delayed until after execution of the next instruction immediately following the jump, call or branch, etc., so that the transfer doesn't create a hole or bubble in the instruction pipeline. It is the compiler's or the assembly language programmer's job to attempt to place a useful instruction in this delay slot.

1.3.4 Read/Write Control Register Instructions

These include instructions to read and write the contents of various CY7C601 control registers. The source (read) or destination (write) is implied by the instruction name.

1.3.5 Floating-Point-Operate and Coprocessor-Operate Instructions

This category includes floating-point calculations, floating-point register operations, and instructions involving computations or other operations in the second coprocessor.

Floating-point-operate instructions execute concurrently with CY7C601 instructions and possibly with other floating-point instructions. Concurrent execution is also possible with the coprocessor-operate instructions if they are so implemented.

Coprocessor-operate instructions are defined by the coprocessor itself. In the CY7C601, they are specified by the CPop instruction. The SPARC architecture will accommodate 1024 coprocessor-operate instructions.

Floating-point and coprocessor loads and stores are not operate instructions; they belong to the "load and store" category discussed in Section 1.4.1.

CY7C601/CY7C611 Integer Unit

This section describes the workings of the CY7C601 Integer processing Unit (IU), the main computing engine in the SPARC architecture. Descriptions and explanations given for the CY7C601 also apply to the CY7C611 integer unit, except for those differences noted in Section 2.9.

The CY7C600-family IUs are based on the SPARC 32-bit RISC architecture, which defines a processor capable of execution at a rate approaching one instruction per clock cycle. The CY7C601/611 supports a tightly-coupled Floating-Point coprocessor Unit (FPU) and a second, system-specific coprocessor, all three of which may operate concurrently. The CY7C601/611 executes all instructions except floating-point-operate and coprocessor-operate instructions.

A block diagram of the CY7C601/611 is shown in *Figure 2-1*. The processor is organized around the ALU and the shift unit. These are both two-operand units, accepting 32-bit information from either source 1 or source 2 of the register file, the program counters, or the instruction decoder. ALU or shift unit results may be passed to the register file, address bus, program counters, control registers, or back to themselves.

One of the characteristics of the SPARC load/store architecture is that neither the ALU nor the shift unit directly pass results to the instruction/data bus. Memory data moves in and out of the register file through alignment units to and from the instruction/data bus. Instructions are taken directly from the bus and fed to a four-stage instruction pipeline.

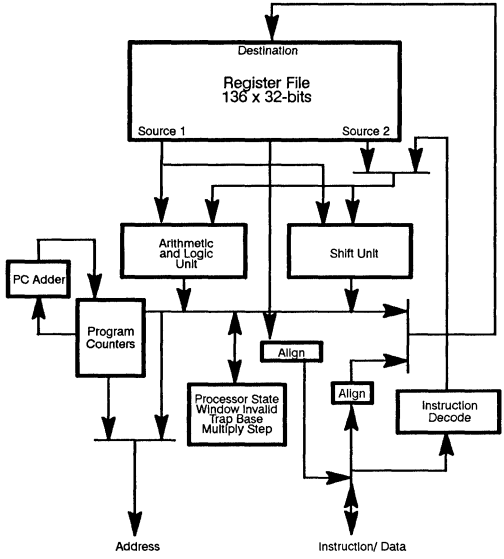


Figure 2-1. Integer Unit Block Diagram

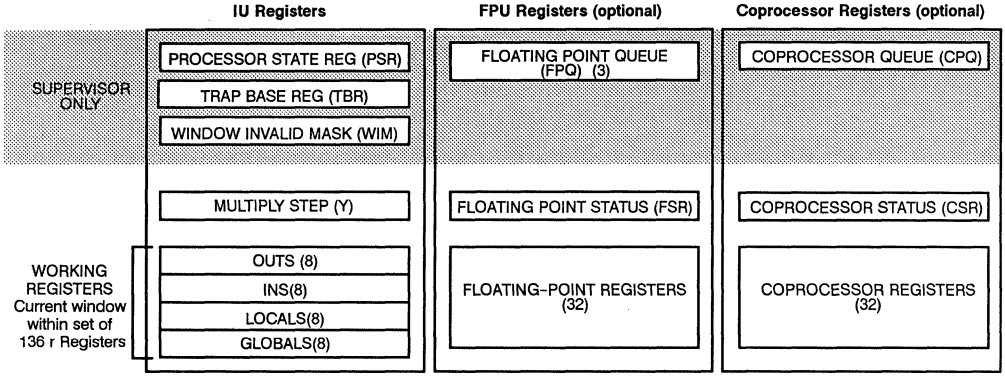


Figure 2–2. SPARC Register Model

The SPARC architecture uses a "windowed" register file model in which the file is divided up into groups of registers called windows. This windowed register model simplifies compiler design, speeds procedure calls, and efficiently supports A/I programming languages such as Prolog, LISP and Smalltalk.

A unique pair of coprocessor interfaces and a common connection to the system data and virtual address buses form the physical interface between the IU, the FPU, and a coprocessor. The coprocessor interfaces provide the synchronization and error handling that enable all three processors to operate concurrently. A common interface to the virtual address bus and data bus permits the IU to provide all addresses for floating-point and coprocessor load and store instructions.

2.1 Description Of Parts

The standard version of the integer unit, the CY7C601, contains a 136 x 32 register file divided into eight overlapping windows. It is supplied in 207-pin PGA and 208-pin QFP packages, which allows 32-bit address and data buses, an eight-bit ASI bus, a number of control lines, and floating-point-coprocessor and second coprocessor interfaces.

The CY7C611 embedded control IU is internally the same as the CY7C601, but it is externally optimized for board-space-sensitive controller applications. By eliminating some external pins, the CY7C611 fits into a 160-pin PQFP package. In the smaller package, the address bus is modified to 24 bits, the ASI bus to 3 bits, and the second coprocessor interface and five control lines are omitted. See Section 2.9 for further information.

2.2 Programming Model

This section describes the CY7C601/611's register model, register window mechanism, processor states, supervisor/user modes, control/status registers, and data types. The concepts and properties explained here are central to an understanding of the CY7C601/611's operation.

The register set shown in *Figure 2–2* is a snapshot of the registers the CY7C601/611 sees at any given moment. The working registers constitute the current window on the register file. Registers within the shaded area are accessible only in the supervisor mode.

Working registers are used for normal operations and are called *r* registers in the CY7C601/611, *f* registers in the FPU, and *c* registers in the coprocessor. The various control/status registers keep track of and/or control the state of each processor. See Section 3.3.1 for an explanation of the FPU's register set.

2.2.1 Register Windows

The 136 *r* registers of the CY7C601/611 are 32-bits wide and are divided into a set of 128 window registers and a set of eight global registers. The 128 window registers are grouped into eight sets of 24 *r* registers called windows.

Table 2–1. Register Addressing

Register numbers	Name
r[24] to r[31]	ins
r[16] to r[23]	locals
r[8] to r[15]	outs
r[0] to r[7]	globals

The SPARC architecture supports a maximum of 32 windows. The currently active window (the window visible to the programmer) is identified by the Current Window Pointer (CWP), a 5-bit field in the Processor State Register (PSR) (see Section 2.2.4.2).

At any given time, a program can address 32 active registers: 24 window registers and the eight *globals*. By software convention, the window registers are divided into 8 *ins*, 8 *locals*, and 8 *outs*. Registers are addressed as shown in *Table 2–1*.

The current window pointer (CWP) acts as an index pointer within the stack of 128 window registers. Changing the current window pointer by one offsets *r* register addressing by 16. Since 24 *r* registers can be addressed by a single CWP value, incrementing or decrementing the CWP results in an eight register overlap between windows. This overlap of window registers is used to pass parameters from one window to the next.

2

2.2.1.1 Windowing

The register file is implemented as a circular stack, with the highest numbered window joined to the lowest. In the CY7C601, window 7 adjoins window 0 (see *Figure 2–3*).

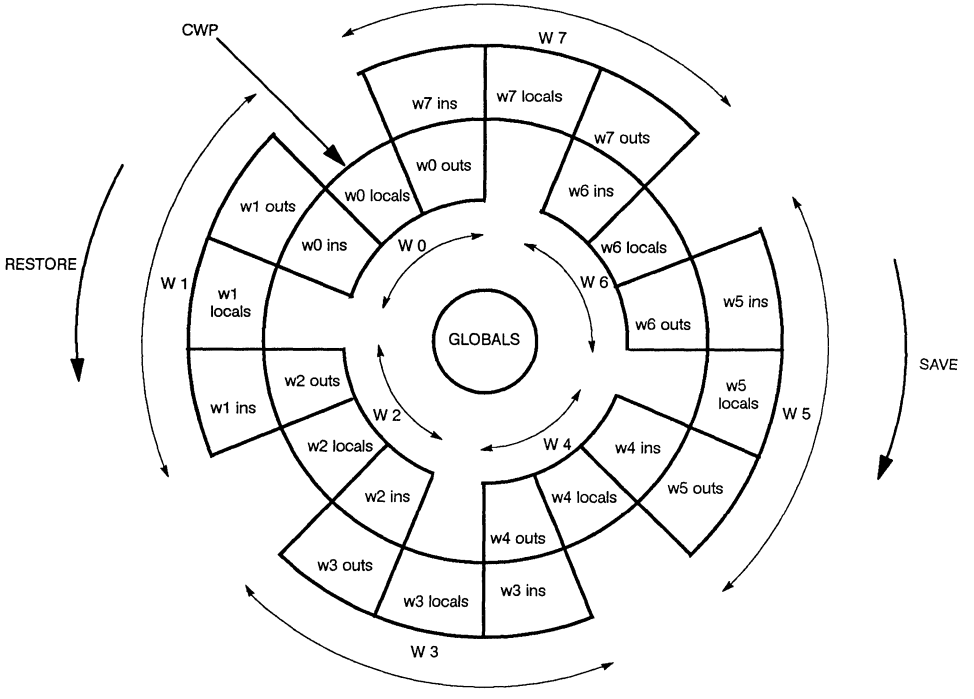


Figure 2–3. Circular Stack of Overlapping Windows

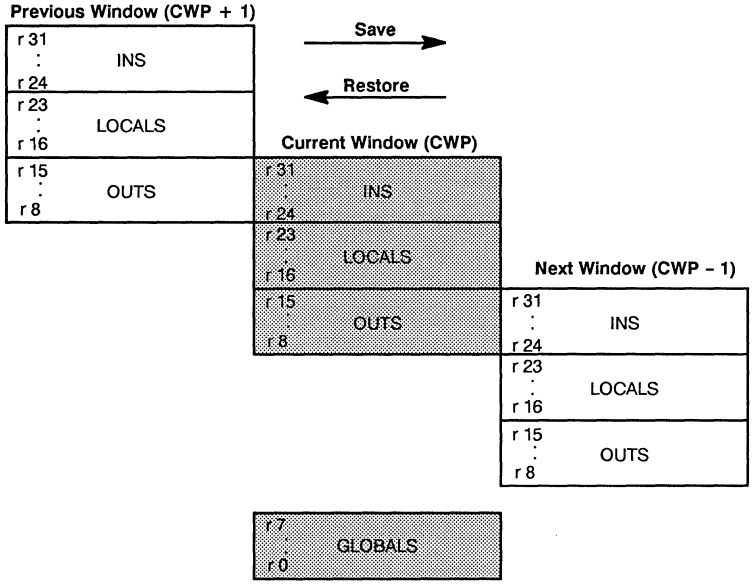


Figure 2–4. Overlapping Windows

Note that each window shares its *ins* and *outs* with adjacent windows (refer to *Figure 2–4*). *Outs* from a previous window (CWP + 1) are the *ins* of the current window, and the *outs* of the current window are the *ins* of the next window (CWP - 1). While only adjacent windows share *ins* and *outs*, *globals* are shared by all windows. A window's *locals*, on the other hand, are not shared at all, belonging only to that window.

After power-on reset, the state of the current window pointer and the WIM register (see Section 2.2.4.3) are undefined. The power-on reset trap routine must initialize the CWP and WIM register for correct operation.

2.2.1.1.1 Parameter Passing

Register window overlap provides an efficient means of passing parameters during procedure calls and returns. One method of implementing a procedure call that takes advantage of the overlap is to have the calling procedure move the parameters to be passed into its *outs* registers, then execute a CALL instruction. A SAVE instruction then decrements the CWP to activate the next window. The calling procedure's *outs* become the called procedure's *ins*, making the passed parameters directly accessible.

When a called procedure is ready to return results to the procedure that called it, those results are moved into its *ins* registers and it then executes a return, usually with a JMPL instruction. A RESTORE instruction increments the CWP to activate the previous window. The called procedure's *ins* are still the calling procedure's *outs*; thus the results are available to the calling procedure. Note that the terms *ins* and *outs* are defined relative to calling, not returning.

If the calling procedure must pass more parameters than can be accommodated by the *outs* and *globals*, the additional parameters must be passed on the memory stack. One method of handling the stack pointer is to dedicate an *out* register in the current window to hold the stack pointer (see *Figure 2–5*). After a call, this pointer (which is now in an *ins* register) can be used as the frame pointer for the called procedure. The SAVE instruction, in addition to decrementing the CWP, also performs an ADD using registers from the current window and placing the result in a register in the next window. This feature can be used to set a new stack pointer for the called procedure from the old pointer in the calling procedure. RESTORE also performs an ADD, using registers in the current window and placing the result in the previous window.

	r31	(i7) return address
	r30	(FP) frame pointer
	r29	(i5) incoming param reg 5
in	r28	(i4) incoming param reg 4
	r27	(i3) incoming param reg 3
	r26	(i2) incoming param reg 2
	r25	(i1) incoming param reg 1
	r24	(i0) incoming param reg 0
	r23	(l7) local 7
	r22	(l6) local 6
	r21	(l5) local 5
local	r20	(l4) local 4
	r19	(l3) local 3
	r18	(l2) local 2
	r17	(l1) local 1
	r16	(l0) local 0
	r15	(o7) temp
	r14	(SP) stack pointer
	r13	(o5) outgoing param reg 5
out	r12	(o4) outgoing param reg 4
	r11	(o3) outgoing param reg 3
	r10	(o2) outgoing param reg 2
	r9	(o1) outgoing param reg 1
	r8	(o0) outgoing param reg 0
	r7	(g7) global 7
	r6	(g6) global 6
	r5	(g5) global 5
global	r4	(g4) global 4
	r3	(g3) global 3
	r2	(g2) global 2
	r1	(g1) global 1
	r0	(g0) 0
	f31	floating-point value
floating point	:	:
	f0	floating-point value

Figure 2–5. Registers as Seen by a Procedure

2.2.1.1.2 Window Overflow and Underflow

No matter how many windows a register file has, it is possible that at some point the program will try to use more than are available. Since the register file is a circular stack, something must be done to prevent overwriting the oldest window as the stack wraps around.

The CY7C601/611 handles this by allowing bits in the Window Invalid Mask (WIM) register to be set, which are used to mark windows that will trigger an underflow or overflow trap (see Section 2.2.4.3). If a SAVE instruction points the CWP to a marked window, a window overflow trap is generated. This means that in the CY7C601, only seven of the eight windows are available for calls, because the last window must be saved for the trap handler. However, since a typical overflow trap handler would transparently save one or more of the oldest windows to memory, the program sees an apparently infinite number of windows.

The CY7C601/611 automatically decrements the CWP upon encountering a trap. This happens without generating another window overflow trap, regardless of the state of the WIM register. By setting at least one window as masked by the WIM register, the system is assured of at least one window for use by the trap handler.

A RESTORE instruction will cause a window underflow trap if it attempts to restore to a window invalidated by the WIM register. Execution of a RETurn from Trap (RETT) instruction under the same circumstances will also generate an underflow trap. SAVE, RESTORE, and RETT always check the WIM register before completing their actions.

As an example, in *Figure 2–3*, if the procedure using the window labeled w0 executes a CALL and SAVE sequence, a window overflow trap will occur (assuming WIM bit 7 is set). The overflow trap handler may safely use only the *locals* of w7, because w7's *ins* are w0's *outs* and w7's *outs* are w6's *ins*.

Active window = 0	CWP = 0
Previous window = 1	CWP+1 = 1
Next window = 7	CWP-1 = 7
Trap window = 7	WIM = $10000000_{(base\ 2)}$

The overflow trap handler is responsible for saving one or more of the least recently used windows to the memory stack. Simulations of register file management methods show that saving and restoring one window at a time is the simplest and most effective algorithm for handling overflow and underflow. The stack pointer to the window-save area must be aligned to a word boundary in valid memory and, for efficiency, should be doubleword aligned. This is because it is faster to load and store doublewords than to load and store words.

A linear sequence of doubleword loads and stores is also used to speed up context switches. In a context switch, only the windows containing valid data are saved, and on average this is about half the number of CY7C601/611 windows, minus one for the reserved trap window.

2.2.1.1.3 Alternate Register Window Usage

Although the windowing layout is particularly well suited to procedure calls and returns, hardware does not force their use for that purpose alone. Except for the eight-register overlap and the partial fixing of the function of several registers by the instruction set (see Section 2.2.1.2), register windows can be viewed and manipulated as needed to fit the application at hand.

For example, the register set can be treated as a flat register file. Access to any particular register in any window is obtained by writing its window value into the current window pointer located in the processor state register. Moreover, windows naturally segment registers into blocks that could be dedicated to specific purposes and accessed through the CWP. Register saving and parameter passing could be done with a standard push/pop stack in memory, although this would substantially increase bus traffic.

For real-time and embedded controller systems, where fast context switching may be more important than procedure calling, the register file can easily be divided into banks of registers separated by trap handling windows set up by the WIM register (see Section 2.2.4.3). Switching from one register bank to another is accomplished by writing to the CWP field of the processor state register. *Figure 2–6* shows the CY7C601/611 register file divided into four banks, each with its own trap handler window of eight local registers. *Globals* are accessible by all processes.

2.2.1.2 Special Registers

In general, the window registers seen at any given time can be used in any manner desired, while keeping in mind that windows overlap at both ends. However, the instruction set does fix the use of r[0] and partially fixes the use of r[15].

Global register r[0] always returns the value 0 when read, making the most frequently used constant easily available at all times. In addition, when addressed as a destination operand, r[0] discards the value written to it.

The CALL instruction writes its own address into register r[15] (*out* register 7) of the calling procedure's window. If a SAVE instruction then activates a new window, r[15] of the old window becomes r[31] (*in* register 7) of the new window and serves as the return address to the calling procedure. However, if the register is needed for some other purpose, the return address can be saved to a stack or simply overwritten.

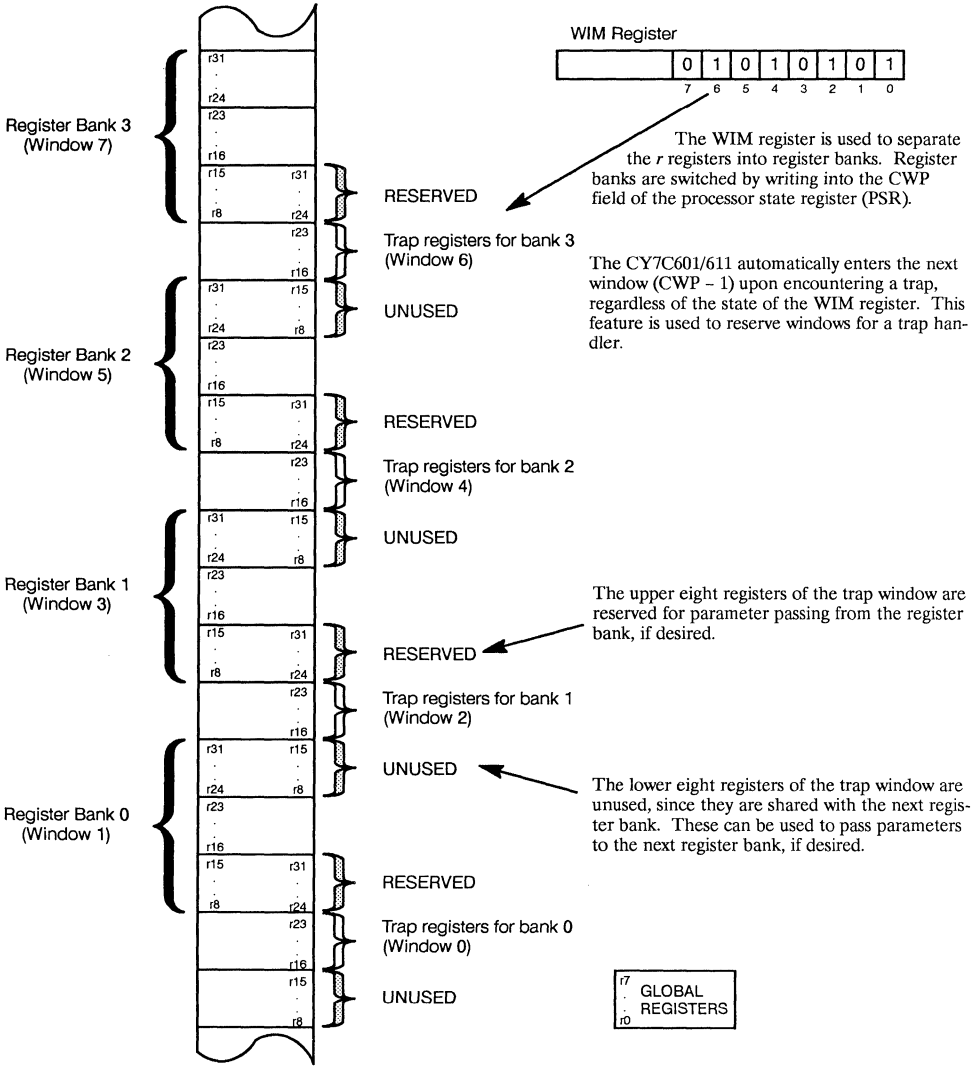


Figure 2–6. Register Banks for Fast Context Switching

Two other registers are also used by hardware to save information during a trap. Registers r[17] and r[18] (*locals* 1 and 2) of the trap window (not the trapping procedure's window) are used to save the contents of the program counters (PC and nPC) at the time the trap is taken. Because the trap window *locals* are all a trap handler is allowed to use (unless it saves to the system stack), this limits the trap handler's usable registers to six.

2.2.2 Processor States

The CY7C601/611 is always in one of three possible states: execute mode, reset mode, or error mode. Execute mode is the normal operating mode.

The processor enters error mode (at which point it halts and asserts $\overline{\text{ERROR}}$) if a synchronous trap is generated while traps are disabled (see Section 2.7). The CY7C601/611 remains in error mode until the $\overline{\text{RESET}}$ signal is asserted, whereupon it enters reset mode. The external system is responsible for asserting $\overline{\text{RESET}}$ whenever the error mode signal, $\overline{\text{ERROR}}$, is detected.

Reset mode is entered whenever the $\overline{\text{RESET}}$ signal is asserted (see Section 2.4). The processor remains in that mode until $\overline{\text{RESET}}$ is deasserted. Upon deassertion, the processor enters execute mode, where the first instruction address to be executed is address 0 in the supervisor instruction address space (see Sections 2.2.3 and 2.3.2.6).

The CY7C601/611 fetches instructions in the execute mode. If the instruction belongs to the floating-point unit or second coprocessor, execution is directed to the appropriate coprocessor. Otherwise, the instruction is executed by the integer unit.

2.2.3 Supervisor/User Modes

In support of multitasking, the CY7C601/611 employs a supervisor/user model of operation. The processor is in supervisor mode when the S bit in the Processor State Register (PSR) is set, and in user mode when S is reset (see Section 2.2.4.2). The state of this bit determines which address space is accessed with the ASI bits (see Section 2.3.2.6) and whether or not privileged instructions may be used. Privileged instructions restrict control register access to supervisor software, preventing user programs from accidentally altering the state of the machine.

In non-multitasking situations, such as embedded systems, user (application) code would probably run in supervisor mode to gain access to the PSR's CWP field and other control registers. The only way a program running in user mode may enter supervisor mode is to encounter a software or hardware trap. A return to user mode is accomplished by executing a Return from Trap (RETT) instruction, which restores the state of the S bit to what it was before the trap was taken. A commonly used trap return is the JMPL, RETT delayed control transfer couple (refer to Section 2.3.3.4.4). This restores both the PC and nPC (see Section 2.2.4.1) and the previous state of the S bit.

2.2.4 Control/Status Registers

CY7C601/611 control/status registers are all 32 bits wide. The two program counters can only be read to and written to indirectly using such instructions as a CALL, JMPL, software trap (Ticc), and Return from Trap (RETT). The Processor State Register (PSR), Window Invalid Mask (WIM), Trap Base Register (TBR), and multiply-step register (Y), are all read/write registers. Read/write instructions that access the PSR, WIM, and TBR are privileged and thus may only be used in supervisor mode.

Two of these registers, the PSR and TBR, have both read-only status fields and programmable read/write mode fields. In *Figure 2–7* and *Figure 2–9*, the read-only status fields appear in lower case italic (for example, *impl*) while the writable mode fields appear in UPPER CASE (for example, PIL).

2.2.4.1 Program Counters (PC and nPC)

The Program Counter (PC) contains the address of the instruction currently being executed by the CY7C601/611, and the next Program Counter (nPC) holds the address (PC + 4) of the next instruction to be executed (assuming there is no control transfer and a trap does not occur). The nPC is necessary to implement delayed control transfers, wherein the instruction that immediately follows a control transfer may be executed before control is transferred to the target address (see Section 2.3.3.4). Having both the PC and nPC available to the trap handler allows a trap handler to choose between retrying the instruction causing the trap (after the trap condition has been eliminated) or resuming program execution after the trap causing instruction.

2.2.4.2 *Processor State Register (PSR)*

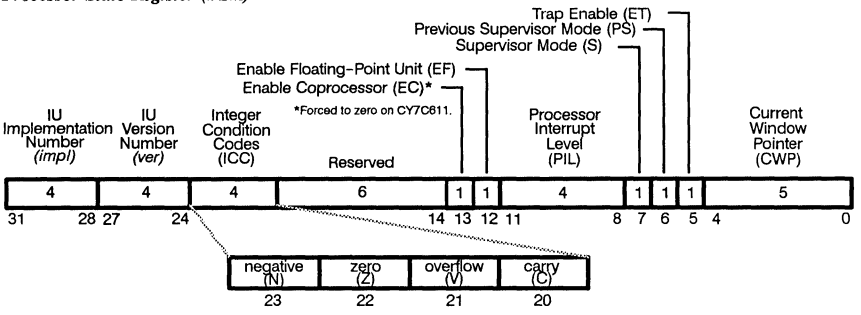


Figure 2–7. Processor State Register

This is the CY7C601/611's key status and control register, containing fields that report the status of processor operations or control processor operations. Instructions that modify its fields include SAVE, RESTORE, Ticc, RETT, and any instruction that modifies the condition code field (*icc*). Any hardware or software action that generates a trap will modify the S, PS, and ET fields. The PSR may be read or written directly using the privileged instructions RDPSR and WRPSR. The PSR is made up of the following fields:

impl—Implementation

Bits 28 through 31 contain the processor's implementation number. The implementation number for the CY7C601 and CY7C611 is 0001. WRPSR does not modify this field.

ver—Version

Bits 24 through 27 contain the CY7C601/611's version number. WRPSR does not modify this field. The current version number for the CY7C601 is 0001, and the current version number for the CY7C611 is 0011.

icc—Integer Condition Codes

Bits 20 through 23 hold the integer unit's condition codes. These bits are modified by arithmetic and logical instructions whose names end with the letters *cc* (for example, ANDcc), and can be overwritten by the WRPSR instruction. The Bicc and Ticc instructions base their control transfer on these bits, which are defined as follows:

N—Negative

Bit 23 indicates whether the ALU result was negative for the last icc-modifying instruction.

0 = not negative

1 = negative

Z—Zero

Bit 22 indicates whether the ALU result was zero for the last icc-modifying instruction.

0 = result was nonzero

1 = result was zero

V—Overflow

Bit 21 indicates whether an arithmetic overflow occurred during the last *icc*-modifying instruction. The overflow bit is also set if a tagged operation (TADDcc, TSUBcc, etc.) is performed on non-tagged operands (refer to Section 2.3.3.2.3). Logical instructions that modify the icc field always set the overflow bit to 0.

0 = arithmetic overflow did not occur

1 = arithmetic overflow did occur

C—Carry

Bit 20 indicates whether an arithmetic carry out of result bit 31 occurred from the last icc-modifying addition or if a borrow into bit 31 resulted from the last icc-modifying subtraction. Logical instructions that modify the icc field always set the carry bit to 0.

0 = a carry/borrow did not occur

1 = a carry/borrow did occur

Reserved

Bits 14 through 19 are reserved. A WRPSR should write only 0s to this field.

EC—Coprocessor Enabled

This bit determines whether the optional second coprocessor is enabled or disabled.

0 = disabled

1 = enabled

If the coprocessor is either disabled or enabled but not present, a CPop, CBccc, or coprocessor load/store instruction will cause a coprocessor-disabled trap. When the CP is disabled, it retains that state until it is re-enabled or reset. Even when disabled, the coprocessor can continue to execute instructions if it contains a queue. Note that the CY7C611 does not support a coprocessor interface, and on the CY7C611 the EC bit is permanently set to zero.

EF—Floating-Point Unit Enabled

Bit 12 determines whether the FPU is enabled or disabled.

0 = disabled

1 = enabled

If the FPU is either disabled or enabled but not present, an FPop, FBfcc, or floating-point load/store instruction will cause a floating-point-disabled trap. When disabled, the FPU retains that state until it is re-enabled or reset. Even when disabled, it can continue to execute any instructions in its queue.

PIL—Processor Interrupt Level

Bits 8 through 11 identify the processor's external interrupt priority level. The processor will only accept external interrupts whose interrupt level is greater than the value in PIL. Bit 11 of the PIL is the MSB and bit 8 is the LSB.

S—Supervisor

Bit 7 determines whether the processor is in supervisor or user mode. Because WRPSR is privileged and only available in the supervisor mode, supervisor mode can only be entered by a software or hardware trap.

0 = user mode

1 = supervisor mode

PS—Previous Supervisor

Bit 6 holds the value that was in the S bit at the time the most recent trap was taken.

ET—Enable Traps

Bit 5 determines whether traps are enabled. If traps are disabled, all asynchronous traps are ignored. If a synchronous or floating-point/coprocessor trap occurs while traps are disabled, the CY7C601/611 halts and enters the error mode (see Section 2.7).

0 = traps disabled

1 = traps enabled

CWP—Current Window Pointer

Bits 0 through 4 contain a pointer to the currently active register file window. CWP is decremented by traps and the SAVE instruction, and is incremented by RESTORE and RETT instructions.

The Floating-Point Enabled (EF) bit can be used by the programmer to control FPU use when running multiple processes. By disabling the EF bit while running a process that doesn't require the FPU, software would not have to save and restore the FPU's registers across context switches. If the FPU is not present, as signaled by the input pin, $\overline{FP}$, the EF bit can be used to provoke floating-point instruction set emulation by generating a floating-point-disabled trap if execution of a floating-point instruction is attempted. This technique may be used with the coprocessor as well.

If it is necessary for the software to manually disable traps, care must be taken when changing the ET bit from enabled (ET = 1) to disabled (ET = 0), since the RDPSR, WRPSR instruction sequence is interruptible. One way to handle that is to write all interrupt trap handlers so that before they return program control to the supervisor software that was interrupted, they restore the PSR to the value it had before the interrupt was taken. This will guarantee a correct result when the interrupted RDPSR, WRPSR sequence continues. The only PSR bit that cannot be restored is the PS bit, which is overwritten when the trap is taken.

An alternative to the RDPSR–WRPSR sequence is to generate a "trap instruction" trap with a Ticc instruction. A taken trap automatically sets ET to 0, disabling further traps.

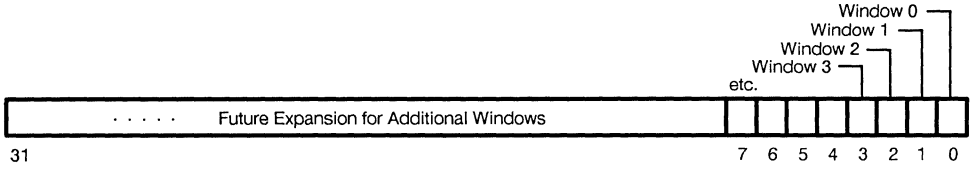


Figure 2–8. Window Invalid Mask

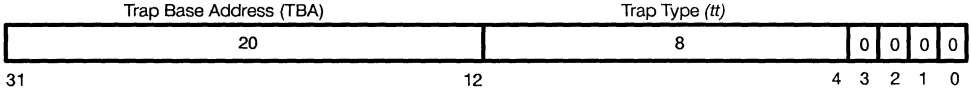


Figure 2–9. Trap Base Register

2.2.4.3 Window Invalid Mask Register (WIM)

This register designates which window(s) will cause generation of an underflow or overflow trap when pointed to by the CWP as the result of a SAVE, RESTORE, or RETT instruction.

Each bit in the WIM register (see *Figure 2–8*) corresponds to a window; if a bit is set to 1, the window corresponding to that bit is marked as invalid. If a SAVE, RESTORE, or RETT instruction would cause the CWP to point to a window whose WIM bit equals 1, a window overflow (SAVE) or window underflow (RESTORE, RETT) trap is generated. The trap handler uses the *local* registers of the invalidated window.

A WIM bit is usually set by the operating system software to identify the boundary between the oldest and newest window. The overflow or underflow trap prevents previous windows from being overwritten or restores previous windows from memory. WIM can also be used to mark off register banks for fast context switching (see Section 2.2.1.1.3).

WIM is read by the RDWIM instruction, and written by the WRWIM instruction. Bits corresponding to unimplemented windows read as zeros and are unaffected by writes.

NOTE: The WIM register is NOT cleared during reset. It must be initialized by software.

2.2.4.4 Trap Base Register (TBR)

When a trap occurs, the program counter (PC) is loaded with the contents of the trap base register. The TBR contains two fields that together constitute a pointer into the trap table, which in turn contains the trap handler address (see *Figure 2–9*). RDTBR can read the entire register; however, the WRTBR instruction can write only to the Trap Base Address field. Only hardware can write to the Trap Type field, and bits 0 through 3 are zeros and are unaffected by a write. The Trap Type field can be directly manipulated using the Ticc instruction. For more information on trap operation, see Section 2.7.

TBA—Trap Base Address

Bits 12 through 31 contain the most-significant 20 bits of the trap table address. This field applies to all trap types except reset, which forces address 0. The TBA is software controlled.

tt—Trap Type

Bits 4 through 11 comprise the Trap Type field, an eight-bit value that provides an offset into the trap table based on the type of trap being taken (see Section 2.7.5.3). This field retains its value until the next trap is taken.

2.2.4.5 Y Register

The Y register is used by the multiply step instruction (MULScc) to create 64-bit products. This register is read and written using the non-privileged RDY and WRY instructions.

Table 2–2. Floating-Point Formats

Single-Precision Floating-Point Format

s = sign (1) e = biased exponent (8) f = fraction (23)		
normalized number (0 < e < 255):		$(-1)^S * 2^{e-127} * 1.f$
subnormal (e=0):	f ≠ 0	$(-1)^S * 2^{-126} * 0.f$
zero (e=0):	f ≠ 0	$(-1)^S * 0$
signaling NaN:	f ≠ 0	s=u; e=255 (max); f=.0uuu-uu (at least one bit must be nonzero)
quiet NaN:	f ≠ 0	s=u; e=255 (max); f=.1uuu-uu
infinity:		s=0 or 1, depending upon sign; e=255 (max); f=.00-00 (all zeros)

Double-Precision Floating-Point Format

s = sign (1) e = biased exponent (11) f = fraction (52)		
normalized number (0 < e < 2047):		$(-1)^S * 2^{e-1023} * 1.f$
subnormal (e=0):	f ≠ 0	$(-1)^S * 2^{-1022} * 0.f$
zero (e=0):	f ≠ 0	$(-1)^S * 0$
signaling NaN:	f ≠ 0	s=u; e=2047 (max); f=.0uuu-uu (at least one bit must be nonzero)
quiet NaN:	f ≠ 0	s=u; e=2047 (max); f=.1uuu-uu
infinity:		s=0 or 1, depending upon sign; e=2047 (max); f=.00-00 (all zeros)

2.2.5 Data Types

The CY7C601/611 supports ten data types (eleven with extended-precision floating-point, see Section 2.2.5.3). Integer types include byte, unsigned byte, halfword, unsigned halfword, word, unsigned word, doubleword, and tagged data. ANSI/IEEE 754-1985 floating-point types include single- and double-precision. A byte is 8 bits wide, halfwords are 16 bits, words and single-precision floating-point are 32 bits, doublewords and double-precision floating-point are 64 bits. *Table 2–2* shows the formats for single-precision and double-precision floating-point numbers.

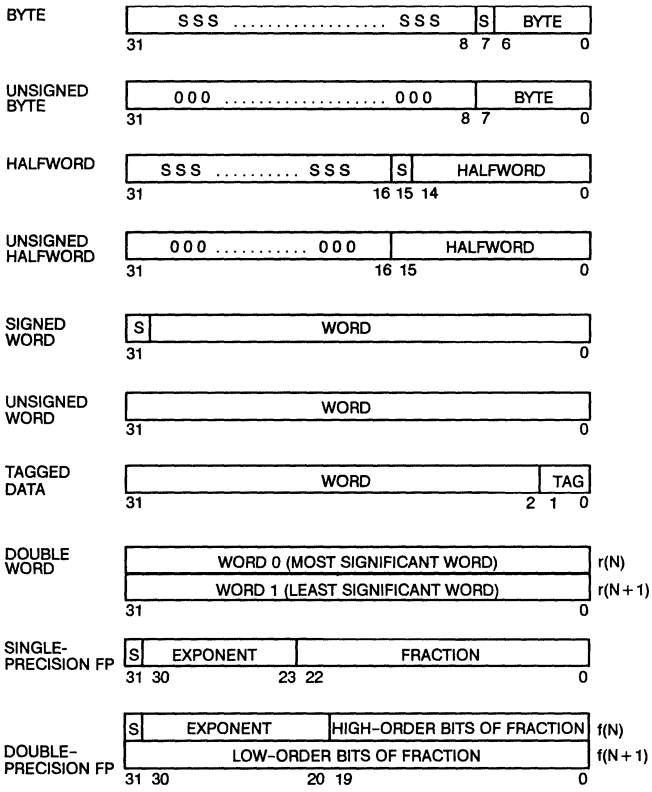


Figure 2–10. Processor Data Types

2.2.5.1 Data Organization In Registers

The organization of the ten data types when loaded into registers is shown in *Figure 2–10*.

When moving memory data to or from the registers, byte operands are always loaded to or extracted from the lower eight bits of a register. On a load, bits 8 through 31 are sign-extended for a byte or zero-extended for an unsigned byte. Halfwords are always loaded to or extracted from the lower 16 bits of a register. Bits 16 through 31 are sign-extended for a halfword or zero-extended for an unsigned halfword during a load. All 32 bits of a signed or unsigned word are loaded from or stored to memory. Stores of byte and halfword data are not sign-extended. Tagged data is handled as an unsigned word. Doubleword operands load to and store from two contiguous registers, r[n] and r[n + 1], with r[n] containing the most significant word. *Figure 2–11* illustrates the relationship between the way data is stored in memory and the way it is loaded into registers.

For single-precision, floating-point operands, bit 31 contains the sign bit, bits 23 through 30 contain the eight bits of exponent, and bits 0 through 22 contain the 23-bit fraction. Double-precision operands require a register pair, with the upper-order register (r[n]) containing the sign bit, 11-bit exponent, and the high-order bits of the fraction. The lower-order register (r[n + 1]) contains the low-order bits of the fraction. Total fraction size is 52 bits.

When loading doublewords or double-precision operands from memory to the working registers (either *r* or *f*), the destination register must be at an even address or the hardware will force such an address. For example, an attempted load double to register r[9] would be forced to r[8], so that the most significant word would be loaded in r[8] and the least significant word in r[9]. A load double to r[0] would result in the loss of the most significant word.

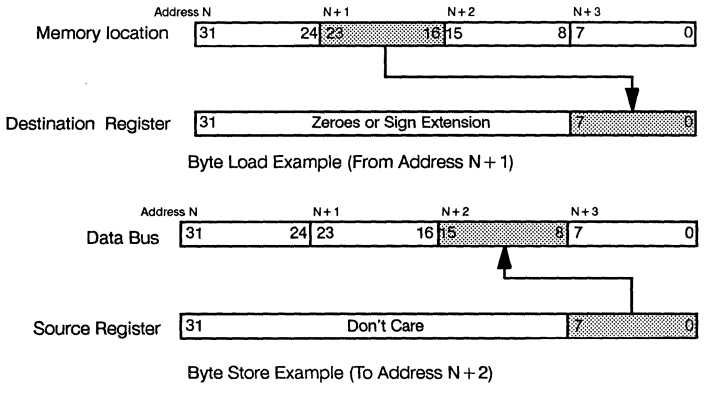


Figure 2–11. Byte Operand Load and Store

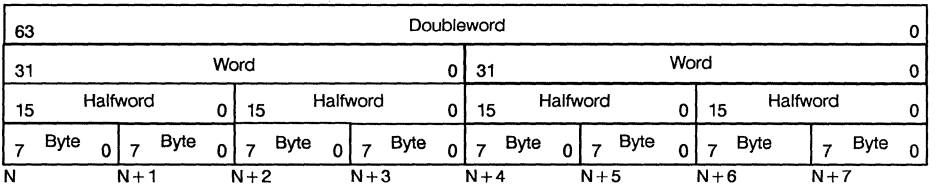


Figure 2–12. Data Organization in Memory

2.2.5.2 Data Organization In Memory

Organization and addressing of data in memory follows the "Big-Endian" convention wherein lower addresses contain the higher-order bytes (see *Figure 2–12*). For a stored word, address N corresponds to the most significant byte of the word, and address N + 3 corresponds to the least significant byte. The address of a halfword, word, or doubleword is also the address of its most significant byte. A halfword datum must be located on a halfword boundary (address bit <0> = 0), which is evenly divisible by 2. Similarly, a word must be located on a word boundary (address bits <1:0> = 0) evenly divisible by 4, and a doubleword must be located on a doubleword boundary (address bits <2:0> = 0) evenly divisible by 8. Attempting to access misaligned data will generate a memory_address_not_aligned trap.

2.2.5.3 Extended Precision

The SPARC architecture supports another data type, an ANSI/IEEE 754-1985 extended-precision floating-point type with a width of 128 bits (see *Table 2–3*). For the present, however, the CY7C602 FPU does not implement extended-precision Floating-Point-operate (FPop) instructions, so they must be emulated in software. An extended-precision format FPop will generate a floating-point-exception trap if execution is attempted.

When loaded to the working registers, extended-precision operands require a register quadruple (see Figure 2–13). The upper-order register (r[N]) contains the sign bit, a 15-bit exponent, and a 16-bit reserved field. The next register (r[N + 1]) contains the one-bit integer part and 31 high-order bits of the fraction. The next register (r[N + 2]) holds the 32 low-order bits of the fraction. Total fraction size is 63 bits. The fourth extended-precision register (r[N + 3]) is reserved. As with double-precision operands, when loading an extended-precision operand, the destination register must be at an even address or the hardware will force an even address.

The memory address of an extended-precision datum is also the address of its most significant byte (see *Figure 2–14*). An extended-precision datum must be located on an extended-precision boundary (address bits <3:0> = 0), which is evenly divisible by 16.

Table 2–3. Extended-Precision Floating-Point Format

s = sign (1) e = biased exponent (15) j = integer part (1) f–msb f–lsb = f = fraction (63)	
normalized number (0 < e < 32767; j = 1): subnormal number (e = 0; j = 0) (f ≠ 0): zero (s = 0; e = 0) (f ≠ 0) (j ≠ 0):	(–1) s * 2 e–16383 * j.f (–1) s * 2 –16383 * j.f (–1) s * 0

signaling NaN: f ≠ 0
 	
quiet NaN: f ≠ 0
 	
infinity:
	s = u; e = 32767 (max); j = u;
	f = .0 uuu uu (at least one bit must be nonzero)
	s = u; e = 32767 (max); j = u;
	f = .1 uuu uu
	s = 0 or 1, depending upon sign;
	e = 32767 (max); j = u;
	f = .000 00 (all zeroes)

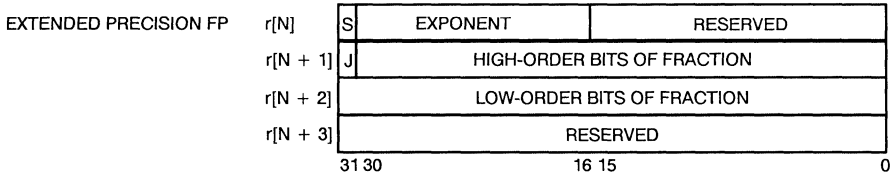


Figure 2–13. Extended-Precision Data Organization in Registers

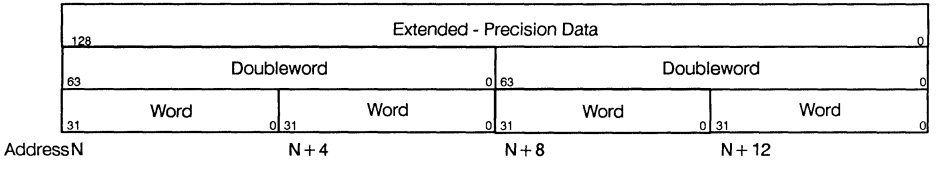


Figure 2–14. Extended-Precision Data Organization in Memory

2.3 Instruction Set

This section describes the CY7C601/611 instruction set as defined by the SPARC architecture. Included are subsections on instruction formats, addressing, instruction types, and an op code summary. Chapter 6, SPARC Instruction Set, contains a description of the assembly language syntax and a complete set of instruction definitions.

2.3.1 Instruction Formats

There are only three basic instruction formats plus three subformats. Format 1 is used for the CALL instruction, format 2 for the SETHI and Branch instructions, and format 3 for the remaining integer and floating-point/coprocessor instructions. *Figure 2–15* shows each format with its fields, bit positions, and the instructions that use that format. All instructions are one word long and aligned on word boundaries in memory. For most instructions, operands are located in source registers (represented by *rs1* and *rs2*). The remaining instructions use one source register plus a displacement or immediate operand contained within the instruction itself.

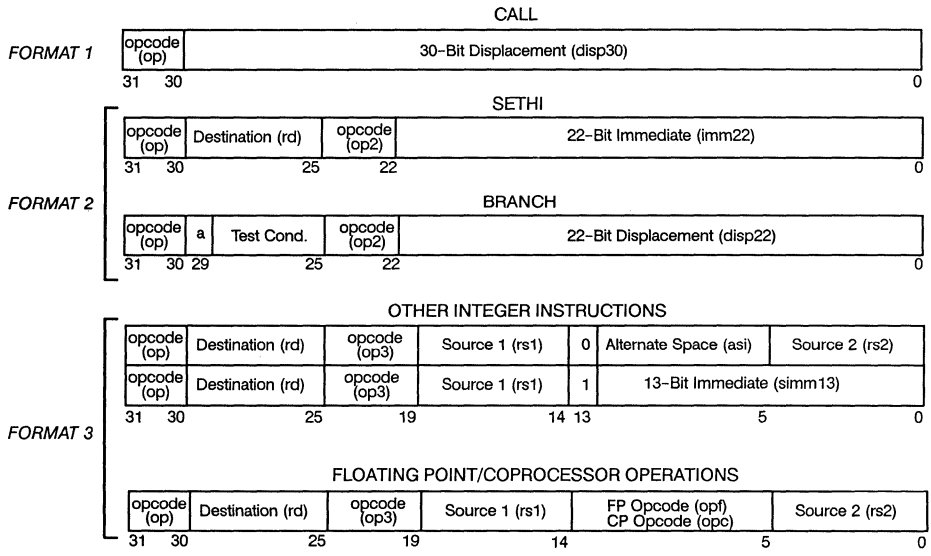


Figure 2–15. Instruction Format Summary

a The *a* (annul) bit is used in branch instructions to control the execution of the delay instruction that immediately follows a control transfer instruction (see Section 2.3.3.4.3).

asi The address space identifier is an eight-bit field used in load/store alternate instructions. See Section 2.3.2.6.

cond This field identifies the condition code used for a branch instruction.

disp22 This field contains the 22-bit displacement value used for PC-relative addressing for a taken branch. It is sign extended to full-word size when used.

disp30 This field contains the 30-bit displacement used for the PC-relative addressing of a CALL instruction.

i The *i* (immediate) bit determines whether the second ALU operand (for non-FPop instructions) will be r[rs2] (i = 0), or a sign-extended *simm13* (i = 1).

imm22 This field contains the 22-bit constant used by the SETHI instruction.

op The *op* field selects the instruction format as shown in *Table 2–4*.

op2 The *op2* field (*Table 2–5*) contains the instruction opcode for format 2 instructions (op = 0).

op3 The 6-bit *op3* field contains the instruction opcode for a format 3 instruction (op = 2 or 3).

opc The 9-bit *opc* identifies a coprocessor-operate (CPop) instruction. The relationship between the *opc* field and CPop instructions is described in Section 2.3.3.6.

opf The 9-bit *opf* identifies a floating-point-operate (FPop) instruction. The relationship between the *opf* field and FPop instructions is described in Section 2.3.3.6.

rd The *r* register (or *r* register pair) or *f* register (or *f* register pair) specified in the *rd* field serves as the source during store instructions. For all other instructions, the identified register (register pair) serves as the destination. Note that r[0] as a source supplies the value 0, and as a destination causes the result to be discarded. Note that *rd* must be a *r* register for integer instructions and must be a *f* register for floating-point instructions.

rs1 The 5-bit *rs1* field identifies the register containing the first source operand. The source is a *r* register for integer instructions, a *f* register for floating-point instructions, or a *c* register for coprocessor instructions.

rs2 The 5-bit *rs2* field identifies the register containing the second source operand. The source is a *r* register for integer instructions, a *f* register for floating-point instructions, or a *c* register for coprocessor instructions.

simm13 This field holds the 13-bit immediate value used as the second ALU operand when i = 1. It is sign-extended to full-word size when used.

Table 2–4. op field Coding

op Value	Instruction
00	Bicc, FBfcc,CBccc, SETHI
01	Call
10 or 11	Other

Table 2–5. op2 Field Coding

op2 Value	Instruction
000	UNIMPlemented
010	Bicc
100	SETHI
110	FBfcc
111	CBccc

Unused (reserved) bit patterns which are used in the *op, op2, op3,* or *i* (wrong bit used) fields of instructions will cause an illegal_instruction trap. Fields that are not used for a particular instruction are ignored and so will not cause a trap, regardless of the bit pattern placed in that field. Unused or reserved bit patterns used in the *opf* or *opc* fields of a floating-point or coprocessor instruction cause an fp exception or a cp exception.

2.3.2 Addressing

Because it uses a load/store architecture, the CY7C601/611 needs only four address modes. Memory address generation is done only for load and store instructions and is byte oriented. Program counter–relative addressing is generated only for calls and branches and is word-boundary oriented because it is addressing instructions. Register-indirect addressing applies to jumps, returns, and traps and is also word-boundary oriented. Address generation is illustrated in *Figure 2–16.*

2.3.2.1 Two Register

Two-register addressing uses the *rs1* and *rs2* fields (instruction format 3) to specify two source registers whose 32-bit contents are added together to create a memory address. This is a load/store (or register-indirect) addressing mode.

2.3.2.2 Register Plus 13-Bit Immediate

This addressing mode is used where an immediate value is required as one of the sources. The address is generated by adding the 32-bit source register specified by *rs1* (format 3) to a 13-bit, sign-extended immediate value contained in the instruction. This is a load/store (or register-indirect) addressing mode.

2.3.2.3 13-Bit Immediate

Immediate addressing is a special case of register-plus-immediate addressing. In this case, the *rs1*-specified register is r[0] (whose value is 0), which means the address is generated using only the 13-bit immediate value. Use of this special case allows absolute addressing of the upper and lower 4 kbytes of a memory (or instruction) space with the 13-bit immediate value. Immediate addressing is the simplest method of addressing because no registers need be set up beforehand.

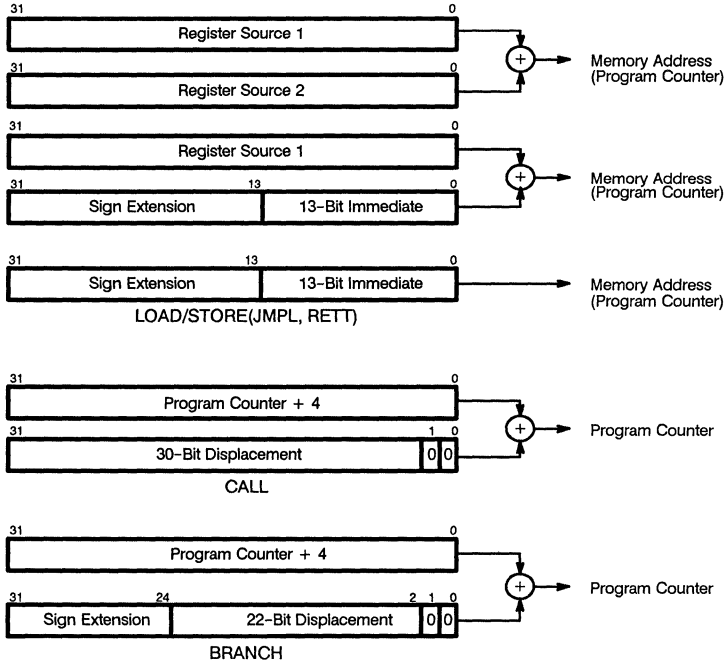


Figure 2–16. Address Generation

2.3.2.4 CALL

Address generation for the CALL instruction is program counter–relative, that is, the target address is based on the program counter. Because the CY7C601/611 is a delayed-control-transfer machine (see Section 2.3.3.4), before the address is calculated, the PC is replaced by the nPC, so the calculation is actually done with PC + 4 (see *Figure 2–16*).

An address is generated by adding this PC + 4 value to the 30-bit word displacement contained in the CALL instruction. The displacement is formed by appending two zeros to the 30-bit value from the instruction. This allows control transfers to any word-boundary location in the virtual memory instruction space. The result of the address generation becomes the new nPC.

2.3.2.5 Branch

Branch instructions also use PC-relative addressing, but in this case, the value added to PC + 4 is a sign-extended 22-bit word displacement. Again, the displacement is formed by appending two zeros to the 22-bit value contained in the branch instruction and then sign extending out to 32 bits. This allows a branching range of 8 Mbytes on word boundaries. The generated address becomes the new nPC.

Table 2–6. ASI Assignments

CY7C601 Address Space Identifier (ASI)	CY7C611 Address Space Identifier (ASI)	Address Space
00001000 (08 H)	000 (0 H)	User Instruction
00001010 (0A H)	010 (2H)	User Data
00001001 (09 H)	001 (1 H)	Supervisor Instruction
00001011 (0B H)	011 (3 H)	Supervisor Data

2.3.2.6 ASI

In addition to the 32 bits of address output by the processor, an additional eight bits of Address Space Identifier (ASI) is also sent to system memory during a memory access. These ASI bits control access to 256 32-bit address spaces, which may or may not overlap depending upon the designer's implementation. The SPARC architecture defines four ASI values for user instructions, user data, supervisor instructions, and supervisor data (see *Table 2–6*). These four ASI values all map to the same 32-bit address space, and are used to implement access-level protection. ASI values are commonly used to identify user/supervisor accesses, to identify special protected memory accesses such as boot PROM, and to access resources such as CY7C604/CY7C605 control registers, TLB entries, cache tag entries, etc..

The ASI value is supplied by the CY7C601/611 for each instruction fetch and each data access encountered. The CY7C600 family assigns a number of these ASI values to the CY7C604/CY7C605 and a number are reserved for future assignment. Nevertheless, nearly 80 are left unassigned for use by the system. Refer to *Table 4–15* for ASI assignments reserved for the CY7C604/CY7C605.

2.3.3 Instruction Types

CY7C601/611 instructions fall into six functional categories: load/store, arithmetic/logical/shift, control transfer, read/write control register, floating-point-operate/coprocessor-operate, and miscellaneous. For complete information on each instruction, see Chapter 6.

2.3.3.1 Load/Store

Load and store instructions (see *Table 2–7*) move bytes, halfwords, words, and doublewords between the byte-addressable main memory and a register in either the IU, FPU, or CP. They are the only instructions that access data memory. For floating-point and coprocessor loads and stores, the CY7C601/611 generates the memory address and the FPU or CP receives or supplies the data.

The CY7C601/611 implements a hardware-interlocked delay when an instruction immediately following a load tries to read the register being loaded. The data will be supplied, but only after a one-cycle delay.

Load and store instructions use two-register, register-plus-immediate, and immediate addressing modes. In addition to the 32-bit address, the CY7C601/611 also generates an eight-bit address space identifier.

2.3.3.1.1 ASI

The Address Space Identifier (ASI) is used by the external system to ascertain which of the 256 available address spaces to access for the load or store being executed. Access to these alternate spaces can be gained directly by using the "load from alternate space" and "store to alternate space" instructions. These instructions use two-register addressing and the *asi* field in instruction format 3. The address space specified in the *asi* field overrides the automatic ASI assignment made by the processor, giving access to such resources as system control registers that are invisible to the user. Because the ASI is intended for use by the system operating software, the alternate space instructions are privileged and can only be executed in supervisor mode.

Table 2–7. Load/Store Instructions

Name		Operation		Cycles
LDSB	(LDSBA*)	Load Signed Byte	(from Alternate Space)	2
LDSH	(LDSHA*)	Load Signed Halfword	(from Alternate Space)	2
LDUB	(LDUBA*)	Load Unsigned Byte	(from Alternate Space)	2
LDUH	(LDUHA*)	Load Unsigned Halfword	(from Alternate Space)	2
LD	(LDA*)	Load Word	(from Alternate Space)	2
LDD	(LDDA*)	Load Doubleword	(from Alternate Space)	3
LDF		Load Floating-Point		2
LDDF		Load Double Floating-Point		3
LDFSR		Load Floating-Point Status		2
LDC		Load Coprocessor		2
LDDC		Load Double Coprocessor		3
LDCSR		Load Coprocessor Status Register		2
STB	(STBA*)	Store Byte	(into Alternate Space)	3
STH	(STHA*)	Store Halfword	(into Alternate Space)	3
ST	(STA*)	Store Word	(into Alternate Space)	3
STD	(STDA*)	Store Doubleword	(into Alternate Space)	4
STF		Store Floating-Point		3
STDF		Store Double Floating-Point		4
STFSR		Store Floating-Point Status Register		3
STDFQ*		Store Double Floating-Point Queue		4
STC		Store Coprocessor		3
STDC		Store Double Coprocessor		4
STCSR		Store Coprocessor State Register		3
STDCQ*		Store Double Coprocessor Queue		4
LDSTUB	(LDSTUBA*)	Atomic Load-Store Unsigned Byte	(in Alternate Space)	4
SWAP	(SWAPA*)	Swap *r* Register with Memory	(in Alternate Space)	4

* denotes supervisor instruction

2.3.3.1.2 Multiprocessing Instructions

In addition to alternate address spaces, the CY7C601/611 provides two uninterruptible instructions, SWAP and LDSTUB (atomic load and store unsigned byte), to support tightly coupled multiprocessing.

The SWAP instruction exchanges the contents of an *r* register with a word from a memory location without allowing asynchronous traps or other memory accesses during the exchange.

The LDSTUB instruction reads a byte from memory into an *r* register and then overwrites the memory byte to all ones. As with SWAP, LDSTUB prevents asynchronous traps and other memory accesses during its execution. LDSTUB is used to construct semaphores.

Multiple processors attempting to simultaneously execute SWAP or LDSTUB to the same memory location are guaranteed that the competing instructions will execute in serial order.

2.3.3.2 Arithmetic/Logical/Shift

This class of instructions performs a computation on two source operands and writes the result into a destination register (r[rd]). One of the source operands is always a register, r[rs1], and the other depends on the state of the instruction's "i" (immediate) bit. If i = 0, the second operand is register r[rs2]. If i = 1, the operand is the 13-bit, sign-extended constant in the instruction's *simm13* field. SETHI is a special case because it is a single-operand instruction.

Table 2–8. Arithmetic/Logical/Shift Instructions

Name		Operation		Cycles
ADD	(ADDcc)	Add	(and modify icc)	1
ADDX	(ADDXcc)	Add with Carry	(and modify icc)	1
TADDcc	(TADDccTV)	Tagged Add and modiify icc	(and Trap on oVerflow)	1
SUB	(SUBcc)	Subtract	(and modify icc)	1
SUBX	(SUBXcc)	Subtract with Carry	(and modify icc)	1
TSUBcc	(TSUBccTV)	Tagged Subtract and modify icc	(and Trap on oVerflow)	1
MULScc		Multiply Step and modify icc		1
AND	(ANDcc)	And	(and modify icc)	1
ANDN	(ANDNcc)	And Not	(and modify icc)	1
OR	(ORcc)	Inclusive Or	(and modify icc)	1
ORN	(ORNcc)	Inclusive Or Not	(and modify icc)	1
XOR	(XORcc)	Exclusive Or	(and modify icc)	1
XNOR	(XNORcc)	Exclusive Nor	(and modify icc)	1
SLL		Shift Left Logical		1
SRL		Shift Right Logical		1
SRA		Shift Right Arithmetic		1
SETHI		Set High 22 Bits of *r* Register		1

For most arithmetic and logical instructions, there is both a version that modifies the integer condition codes and one that doesn't (see *Table 2–8*).

Shift instructions shift left or right by a distance specified in either a register or an immediate value in the instruction.

The multiply step instruction, MULScc, is used to generate the signed or unsigned 64-bit product of two 32-bit integers. For more information on MULScc, refer to its definition in Chapter 6.

2.3.3.2.1 Register r[0]

Because register r[0] reads as a 0 and discards any result written to it as a destination, it can be used with some instructions to create syntactically familiar pseudoinstructions. For example, an integer COMPARE instruction is created using the SUBcc (subtract and set condition codes) with r[0] as its destination. A TEST instruction uses SUBcc with r[0] as both the destination and one of the sources. A register-to-register MOVE is accomplished using an ADD or OR instruction with r[0] as one of the source registers. A negation is done with SUB and r[0] as one source. If the assembler being used supports psuedoinstructions, it translates the psuedoinstruction into the equivalent instruction in the native assembly language. Refer to your assembly language manual for details.

2.3.3.2.2 SETHI

SETHI is a special instruction that can be combined with another arithmetic instruction (such as an OR immediate) to construct a 32-bit constant. SETHI loads a 22-bit immediate value into the upper 22 bits of the destination register and clears the lower 10 bits. The arithmetic immediate instruction which follows is used to load the lower 10 bits. Note that the 13-bit immediate value gives a 3 bit overlap with the 22-bit SETHI value. SETHI can also be combined with a load or store instruction to construct a 32-bit memory address.

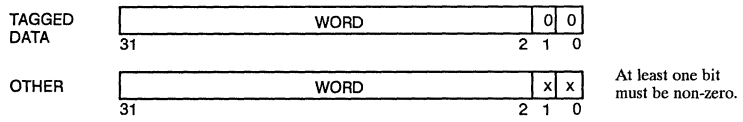


Figure 2–17. Tagged Data Example

2.3.3.2.3 Tagged Arithmetic

The tagged arithmetic instructions are useful for languages that employ tags, such as LISP, Smalltalk, or Prolog. For efficient support of such languages, the SPARC architecture defines tagged data as a data type. Tagged data are assumed to be 30 bits wide with the tag bits (the least two significant bits) set to zero (see *Figure 2–17*). A tagged add (TADDcc) or subtract (TSUBcc) will set the overflow bit if either of the operands has a nonzero tag or if a normal overflow occurs.

Tagged add or subtract instructions are normally followed by a conditional branch. If the overflow bit is set during a tagged add or subtract operation, control is commonly transferred to a routine that checks the operand types. In order to expedite this software construct, the SPARC architecture provides two trap on overflow instructions: TADDccTV and TSUBccTV, which automatically trap if the overflow bit is set during their execution.

2.3.3.3 Control Transfer

Control transfer instructions are those that change the values of the PC and nPC. These include conditional branches (Bicc, FBfcc, CBccc), a call (CALL), a jump (JMPL), conditional traps (Ticc), and a return from trap (RETT). Also included are the SAVE and RESTORE instructions, which don't transfer control but are used to save or restore windows during a call to a new procedure or a return to a calling procedure (see *Table 2–9*).

In the CY7C601, control transfer is usually delayed so that the instruction immediately following the control-transfer instruction (called the delay instruction) can be executed before control transfers to the target address. The delay instruction is always fetched. However, the annul or *a* bit in conditional branch instructions can cause the instruction to be annulled (i.e., prevent execution) if the branch is not taken (or always annulled in the case of BA, FBA, and CBA). If a branch is taken, the delay instruction is always executed (except for BA, FBA, and CBA, see Section 2.3.3.4.3). *Table 2–10* shows the characteristics of each control transfer type.

Program Counter Relative

PC-relative addressing computes the target address by adding a displacement to the program counter. See Section 2.3.2.

Register-Indirect

Register-indirect addressing computes the target address as either r[rs1] + r[rs2] if i = 0, or r[rs1] + *simm13* if i = 1. See Section 2.3.2.

Delayed

A control-transfer instruction is delayed if it transfers control to the target address after a one-instruction delay. See Section 2.3.3.4.

Annul Bit

In an instruction with an annul bit, the delay instruction that follows may be annulled. See Section 2.3.3.4.3.

2.3.3.3.1 Branching and the Condition Codes

The condition code bits in the *icc*, *fcc*, and *ccc* fields, are located (respectively) in the PSR (Processor State Register), FSR (Floating-point State Register), and CSR (Coprocessor State Register). The integer condition code bits are modified by arithmetic and logical instructions whose names end with the letters *cc*, or they may be written directly with WRPSR. The floating-point condition codes are modified by the floating-point compare instructions, FCMP and FCMPE, or directly with the STFSR instruction. Modification of the coprocessor condition codes is done directly with STCSR or by operations defined by the particular coprocessor implementation.

Except for BA (Branch Always) and BN (Branch Never), a Bicc instruction evaluates the integer condition codes as specified in the *cond* field. If the tested condition evaluates as true, the branch is taken, causing a PC-relative delayed transfer to the address [(PC + 4) + sign extnd(disp22)]. If the evaluation result is false, the branch is not taken. For BA and BN, there is no evaluation; the result is simply forced to true for BA and false for BN.

Table 2–9. Control Transfer Instructions

Name	Operation	Cycles
SAVE	SAVE caller's window	1
RESTORE	RESTORE caller's window	1
Bicc	Branch on integer condition codes	1*
FBfcc	Branch on floating-point condition codes	1*
CBccc	Branch on coprocessor condition codes	1*
CALL	Call	1*
JMPL	JuMP and Link	2*
RETT	RETurn from Trap	2*
Ticc	Trap on integer condition codes	1 (4 if taken)

* assumes delay slot is filled with a useful instruction

Table 2–10. Control Transfer Instruction Characteristics

Instructions	Addressing Mode	Delayed	Annul Bit
Conditional Branch	Program Counter Relative	yes	yes
Call	Program Counter Relative	yes	yes
Jump	Register Indirect	yes	no
Return	Register Indirect	yes	no
Trap	Register Indirect	no	no

If the branch is not taken, then the annul bit is checked. If the "a" bit is set, the delay instruction is annulled. If "a" is not set, the delay instruction is executed. If the branch is taken, the annul bit is ignored and the delay instruction is executed. For more information on delayed control transfer and the annul bit, see Section 2.3.3.4.

BN, of course, never branches, and therefore executes like a NOP (but is not recommended as a NOP instruction). However, as far as the annul bit is concerned, BN acts like a normal branch instruction, annulling the delay instruction if a = 1 and executing it if a = 0.

BA, on the other hand, always branches, so the annul bit would normally be ignored. But for BA, FBA, and CBA, the effect of the annul bit is changed. See Section 2.3.3.4.3 for details.

As illustrated in *Table 2–11*, Bicc and Ticc instructions test for the same conditions and use the same *cond* field codes during their evaluations.

An FBfcc instruction operates in the same way as a Bicc, except it tests the FCC < 1:0 > signals output by the CY7C602 floating-point unit (see *Table 2–12*). The FCC < 1:0 > signals are floating-point condition codes which are set by executing a floating-point compare instruction. A CBccc instruction behaves in the same manner as a FBfcc, except it tests the CCC < 1:0 > signals supplied by the coprocessor (see *Table 2–13*). Both FBN and CBN behave in the same way as BN.

2.3.3.3.2 Trap Instructions

The "Trap on integer condition codes" (Ticc) instruction evaluates the condition codes specified by its *cond* (condition) field. If the result is true, a trap is immediately taken (no delay instruction). If the condition codes evaluate to false, Ticc executes as a NOP.

Once the Ticc is taken, it identifies which software trap type caused it by writing its trap number + 128 (the offset for trap instructions) into the *tt* field of the Trap Base Register (TBR), as illustrated in *Figure 2–18*. The trap number is the least significant seven bits of either "r[rs1] + r[rs2]" if the *i* field is zero, or "r[rs1] + sign extnd(simm13)" if the *i* field is one. The processor then disables traps (ET = 0), saves the state of S into PS, decrements the CWP, saves PC and nPC into the *locals* r[17] and r[18] (respectively) of the new window, enters supervisor mode (S = 1), and writes the trap base register to the PC and TBR + 4 to nPC.

Table 2–11. Bicc and Ticc Condition Codes

Cond.	Test	Cond.	Test
0000	Never	1000	Always
0001	Equal to	1001	Not equal to
0010	Less than or equal	1010	Greater than
0011	Less than	1011	Greater than or equal to
0100	Less than or equal to, unsigned	1100	Greater than, unsigned
0101	Carry set (less than, unsigned)	1101	Carry clear (greater than or equal to, unsigned)
0110	Negative	1110	Positive
0111	Overflow set	1111	Overflow clear

Table 2–12. FBfcc Condition Codes

Cond.	Test	Cond.	Test
0000	Never	1000	Always
0001	Not equal to	1001	Equal to
0010	Less than or greater than	1010	Unordered or equal to
0011	Unordered or less than	1011	Greater than or equal to
0100	Less than	1100	Unordered or greater than or equal to
0101	Unordered or greater than	1101	Less than or equal to
0110	Greater than	1110	Unordered or less than or equal to
0111	Unordered	1111	Ordered

Table 2–13. CBccc Condition Codes

Opcode	Cond.	CCC[1:0] Test	Opcode	Cond.	CCC[1:0] Test
CBN	0000	Never	CBA	1000	Always
CB123	0001	1 or 2 or 3	CB0	1001	0
CB12	0010	1 or 2	CB03	1010	0 or 3
CB13	0011	1 or 3	CB02	1011	0 or 2
CB1	0100	1	CB023	1100	0 or 2 or 3
CB23	0101	2 or 3	CB01	1101	0 or 1
CB2	0110	2	CB013	1110	0 or 1 or 3
CB3	0111	3	CB012	1111	0 or 1 or 2

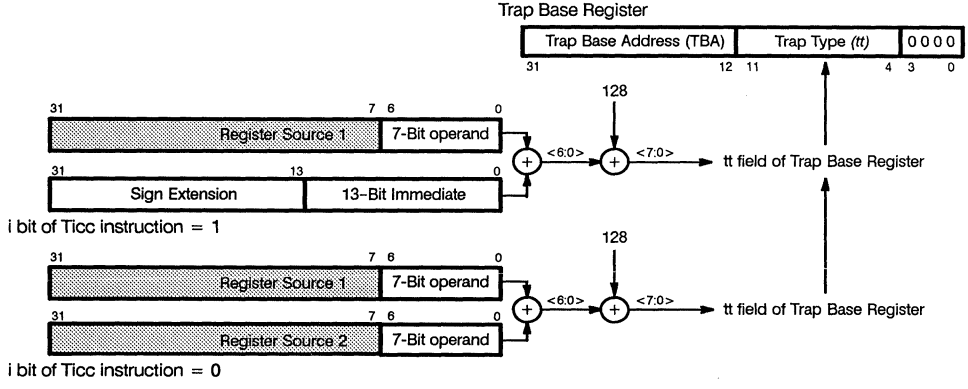


Figure 2–18. Ticc Trap Address Generation

Ticc can be used to implement kernel calls, breakpointing, and tracing. It can also be used for run-time checks, such as out-of-range array indices, integer overflow, etc.

Return from a trap is accomplished using the delayed control transfer couple, JMPL, RETT. RETT first increments the CWP by one, calculates the return address (using register-indirect addressing), and then checks for a number of trap conditions before it allows a return. An illegal_instruction trap is generated if traps are enabled (ET = 1) when RETT is executed. If ET = 0, RETT checks for other trap conditions and will generate a reset trap and enter error mode for the following conditions: S = 0, the new CWP would cause a window underflow, or the return address is not word aligned. If none of these conditions exist, RETT enables traps (ET = 1), restores the previous supervisor state to the S bit, and writes the target address into the nPC.

2.3.3.3.3 Calls and Returns

Calling a subroutine or procedure can be done in one of two ways. A CALL instruction computes its target address using a PC-relative displacement of 30-bits. The JuMP and Link (JMPL) instruction uses register-indirect addressing (the sum of two registers or the sum of a register and a 13-bit signed immediate value) to compute its target address. Either instruction allows control transfer to any arbitrary instruction address.

Control transfer to a procedure that requires its own register window is done with either a CALL or JMPL instruction and a SAVE instruction. A procedure that does not need a new window, a so-called "leaf" routine, is invoked with only the CALL or JMPL.

The CALL instruction stores its return address (the current PC) into *outs* register r[15]. When the new window is activated, this becomes *ins* register r[31] (see *Figure 2–4*). The JMPL instruction stores its return address (the contents of PC, which is the Link) into the *r* register specified in the destination field, *rd*.

The primary purpose of the SAVE instruction is to "save" the caller's window by decrementing the Current Window Pointer (CWP) by one, thereby activating the next window and making the current window into the previous window. SAVE also performs a normal ADD, using source registers from the caller's window, but writing the result into a destination register in the new window. This can be used to set a new stack pointer from the previous one (see Section 2.2.1.1.1).

Return from a procedure requiring its own window is done with a RESTORE and a JMPL instruction. A leaf procedure returns by executing a JMPL only. The target address for the return is normally that of the instruction following the CALL's or JMPL's delay instruction; that is, the return address + 8. The RESTORE instruction restores the caller's window by incrementing the CWP by one, causing the previous window to become the current window. As with SAVE, RESTORE performs an ADD using source registers from the called (new) window and writing the result into the calling (previous) window.

Both SAVE and RESTORE compare the new CWP against the Window Invalid Mask (WIM) to check for window overflow or underflow. They may also be used to atomically change the CWP while establishing a new memory stack pointer in an *r* register.

2.3.3.4 Delayed Control Transfer

Traditional architectures usually execute the target instruction of a control transfer immediately after the control transfer instruction. However, in a pipelined RISC architecture, this type of transfer would require flushing the instruction that follows the control transfer instruction. To avoid creating a hole or bubble in the pipeline, the CY7C601/611 delays execution of the target instruction until the instruction following the control transfer instruction is executed. The instruction in this delay slot is called the delay instruction.

Table 2–14. Delayed Control Transfer Instruction Example

PC	nPC	Instruction
8	12	Non-control transfer
12	16	Control transfer (target = 40)
16	40	Non-control transfer (delay instruction) (Transfers control to 40)
40	44	...

Table 2–15. Effect of Annul Bit Reset ($a = 0$)

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc ($a = 0$) 40	Not Taken
16	20	Delay slot instruction	Executed
20	24	...	Executed

Table 2–16. Effect of Annul Bit Set ($a = 1$)

PC	nPC	Instruction	Action
8	12	Non-control transfer	Executed
12	16	Bicc ($a = 1$) 40	Not Taken
16	20	Delay slot inst. (annulled)	Not Executed
20	24	...	Executed

2.3.3.4.1 PC and nPC

The Program Counter (PC) contains the address of the instruction currently being executed by the CY7C601/611, and the next Program Counter (nPC) holds the address (PC + 4) of the next instruction to be executed (assuming a control transfer or a trap does not occur).

Most instructions end by copying the contents of the nPC into the PC and then they either increment nPC by four or write a computed control transfer target address into nPC. At this point, the PC points to the instruction that is about to begin execution and the nPC points to the instruction that will be executed after that, i.e. the second instruction after the currently executing instruction. It is the existence of the nPC that allows the execution of the delay instruction before transfer of control to the target instruction.

2.3.3.4.2 Delay Instruction

The instruction pointed to by the nPC when the PC is pointing to a delayed-control-transfer instruction is called the delay instruction. Normally, this is the next sequential instruction in the code stream. However, if the instruction that preceded the delayed control transfer was itself a delayed control transfer, the target of the preceding control transfer becomes the delay instruction (that's where the nPC will point). For more on delayed control transfer couples, see Section 2.3.3.4.4.

Table 2–14 shows the order of execution for a simple (not back-to-back) delayed control transfer. The order of execution is 8, 12, 16, 40. If the delayed-control-transfer instruction were not taken, the order would be 8, 12, 16, 20.

2.3.3.4.3 Annul Bit

The *a* (annul) bit is only available on conditional branch instructions (Bicc, FBfcc, and CBccc), where it changes the behavior of the delay instruction. If *a* is set on a conditional branch instruction (except BA, FBA, and CBA) and the branch is *not* taken, the delay instruction is annulled (not executed). An annulled instruction has no effect on the state of the CY7C601/611 nor can a trap occur during an annulled instruction. If the branch is taken, the *a* bit is ignored and the delay instruction is executed. *Table 2–15* and *Table 2–16* show the effect of the annul bit when it is reset or set.

The "branch always" instructions (BA, FBA, and CBA) are a special case. If the *a* bit is set in these instructions, the delay instruction is annulled, even though the branch is taken. Effectively, this gives a "traditional" non-delayed branch. When a = 0 in a "branch always" instruction, it behaves the same as any other conditional branch; the delay instruction is executed. *Figure 2–19* displays the effect the *a* bit has on any branch for either the set or reset state. *Table 2–17* summarizes the effect the annul bit has on the execution of delay instructions.

Table 2–17. Effect of Annul Bit on Delay Instruction

a bit	Type of branch	Delay instruction executed?
a = 1	Always	No
	Conditional, taken	Yes
	Conditional, not taken	No
a = 0	Always	Yes
	Conditional, taken	Yes
	Conditional, not taken	Yes

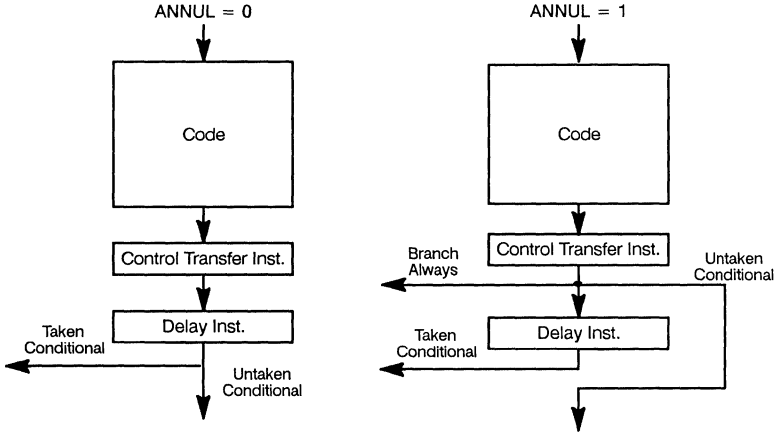


Figure 2–19. Delayed Control Transfer

2.3.3.4.4 Delayed Control Transfer Couples

The occurrence of two back-to-back, delayed control transfer instructions is called a delayed control transfer couple, which the processor handles differently from a simple control transfer. An instruction sequence containing a delayed control transfer couple is shown in *Table 2–18*, and the order of execution for the six different cases of back-to-back, delayed control transfer instructions is shown in *Table 2–19*.

The delay slot instruction for a delayed control transfer instruction is the instruction fetched after the delayed control transfer instruction. For most cases, this instruction is located immediately in the code listing after the delayed control transfer instruction. However, in the case of a delayed control transfer couple, the target instruction of the first delayed control transfer instruction is the delay slot instruction for the second delayed control transfer instruction, since that target instruction is the next instruction to be fetched. The delay slot instruction for the second delayed control transfer instruction is the next instruction loaded into the instruction pipeline after the second delayed control transfer instruction.

In the following tables, "delayed control transfer instruction" is abbreviated to "DCTI". A "Non-DCTI" may be either a non-control transfer instruction or a control transfer that is not delayed (i.e., a Ticc). Where the annul bit is not indicated, it may be either 0 or 1.

Case 1 of *Table 2–19* includes the "JMPL, RETT" couple, which is the normal method of returning from a trap handler. The JMPL, RETT couple ensures correct values of PC and nPC are restored upon exiting the trap routine, even in the case of a trap caused by a delay slot instruction (see Section 2.3.3.4.2). The case of a trap caused by a delay slot instruction is one where the nPC will not be PC + 4, thus requiring both PC and nPC to be restored. The JMPL, RETT couple allows the choice of re-executing the trapped instruction or executing the instruction following the trap occurrence. Refer to the RETT entry in Chapter 6 for further information.

Table 2–18. Delayed Control Transfer Couple Instruction Sequence

Address	Instruction	Target
8:	Non DCTI	
12:	DCTI	40
16:	DCTI	60
20:	Non DCTI	
24:	...	
...	...	
40:	Non DCTI	
44:	...	
...	...	
60:	Non DCTI	
64:	...	
...	...	

Table 2–19. Execution of Delayed Control Transfer Couples

Case	DCTI at Location 12	DCTI at Location 16	Order of Execution
1	DCTI Unconditional	DCTI Taken	12,16,40,60,64,...
2	DCTI Unconditional	B*cc(a = 0) Untaken	12,16,40,44,...
3	DCTI Unconditional	B*cc(a = 1) Untaken	12,16,44,48,...(40 annulled)
4	DCTI Unconditional	B*A(a = 1)	12,16,60,64,...(40 annulled)
5	B*A(a = 1)	any CTI	12,40,44,...(16 annulled)
6	B*cc	DCTI	Not Supported

Definitions:

B*A----------BA,FBA, or CBA
B*cc----------Bicc,FBicc, or CBicc (except B*A)
DCTI Uncond.---CALL,JMPL,RETT, or B*A(a = 0)
DCTI Taken-----CALL,JMPL,RETT,B*cc taken, or B*A(a = 0)

Cases 1–5 described in *Table 2–19* are illustrated in *Figure 2–20*. In case 1, the first DCTI is fetched at address 12 and the target address is calculated while the delay slot instruction is fetched. The delay slot instruction for the first DCTI (located at address 16) is another DCTI, which also has a delay slot. The target address of the first DCTI has been calculated by the time the first delay slot instruction has been fetched, and the target instruction is fetched at address 40. The target instruction is the instruction located in the instruction pipeline after the second DCTI, and therefore it is the delay slot instruction for the second DCTI. The target instruction for the second DCTI (address 60) is fetched after the delay slot instruction for the second DCTI (which is also the target address for the first DCTI) has been fetched.

Case 2 differs from case 1 in that the second DCTI is conditional, and is not taken. In case 2, the instruction at address 40 (target for DCTI #1) is the delay slot instruction for the second DCTI. Since the second DCTI does not cause a branch, the instruction fetch continues to address 44.

Case 3 is an interesting case in which the target instruction of the first DCTI is annulled by the second DCTI. This causes the instruction at address 40 to be annulled. Since the second DCTI is an untaken conditional branch, instruction fetch continues after the annulled target instruction (address 44).

Case 4 illustrates a DCTI followed by a branch always instruction with the annul bit set. This causes the target instruction of the first DCTI (address 40) to be annulled, and program control is transferred to the target of the second DCTI at address 60.

Case 5 illustrates the case where the second DCTI is annulled by the annul bit of the first DCTI. The second DCTI, since it is annulled, has no effect on instruction fetch. This case is identical to the case of any other annulled delay slot instruction.

When the first instruction of a delayed control transfer couple is a conditional branch, control transfer is undefined (case 6). If such a couple is executed, the location where execution continues is within the same address space but is otherwise undefined. Execution of this sequence does not change any other aspect of the processor state.

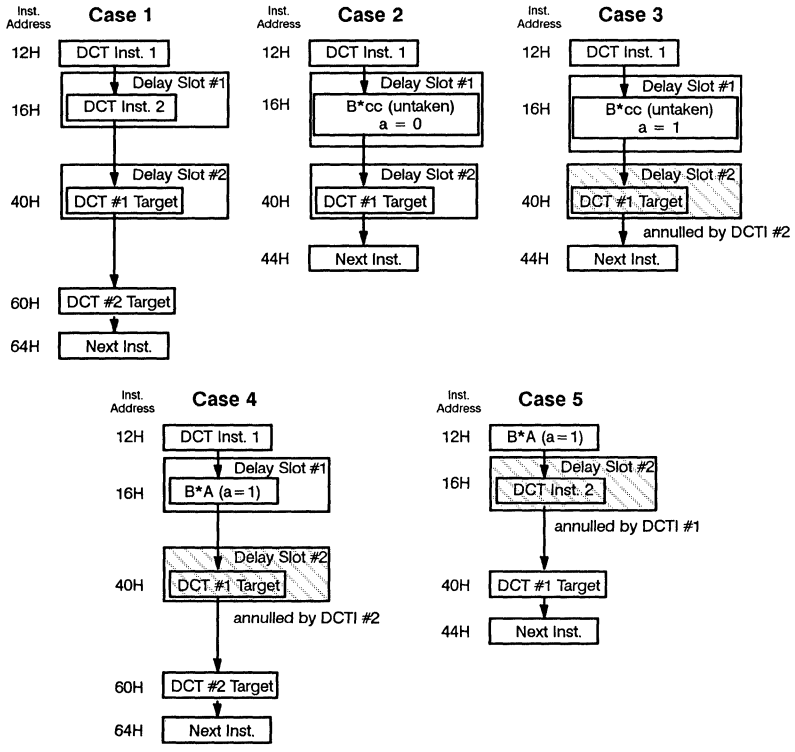


Figure 2–20. Delayed Control Transfer Couples

Table 2–20. Read/Write Control Register Instructions

Name	Operation	Cycles
RDY	Read Y Register	1
RDPSR*	Read Processor State Register	1
RDWIM*	Read Window Invalid Mask	1
RDTBR*	Read Trap Base Register	1
WRY	Write Y Register	1
WRPSR*	Write Processor State Register	1
WRWIM*	Write Window Invalid Mask	1
WRTBR*	Write Trap Base Register	1

* denotes supervisor instruction

Table 2–21. Floating-Point-Operate and Coprocessor-Operate Instructions

Name	Operation	Cycles
FPop	Floating-Point Operations	1 to launch
CPop	Coprocessor Operations	1 to launch

Table 2–22. Miscellaneaous Instructions

Name	Operation	Cycles
UNIMP	Unimplemented Instruction	1
IFLUSH	Instruction Cache Flush	1

2.3.3.5 Read/Write Control Registers

This class of instruction reads or writes the contents of the various control registers (see *Table 2–20*). The source (read) or destination (write) is implied by the instruction name. Read/write instructions are provided for the PSR, WIM, TBR, FSR, CSR, and the Y register. Reads and writes to the PSR, WIM, and TBR are privileged and are available in supervisor mode only.

2.3.3.6 Floating-Point-Operate and Coprocessor-Operate

Floating-point calculations are accomplished with floating-point-operate instructions (FPops), which are register-to-register instructions that compute some result as a function of one or two source operands (see *Table 2–21*). The result is always placed in a destination register (i.e., source operands are not overwritten). The source and destination registers are *f* registers from the CY7C602's register file. See Section 3.3.1 for more information. If no CY7C602 is present, or if the EF bit of the PSR is not set, executing a floating-point instruction will generate a fp disabled trap.

Coprocessor-operate instructions (CPops) are executed by the attached coprocessor. Coprocessor instructions use the *c registers* located in the coprocessor's register file as source and destination registers. If there is no attached coprocessor, attempted execution of a coprocessor instruction generates a cp disabled trap.

Floating-point and coprocessor load/store instructions are not operate instructions; they fall under the CY7C601/611's load/store instruction category (see Section 2.3.3.1).

Except for *op* and *op3*, which specify the particular floating-point-operate or coprocessor-operate instruction to be executed, the instruction fields of an FPop or CPop are interpreted by the CY7C602 or coprocessor. Floating-point-operate instructions execute concurrently with CY7C601/611 instructions. CPops can also execute concurrently with both CY7C601 and FPop instructions if they are designed to do so.

Because the CY7C601/611 and CY7C602 can execute instructions concurrently, when a floating-point exception occurs, the PC does contain the address of an FPop instruction, but not the one that caused the exception. However, the front entry of the floating-point queue contains the offending instruction and its address.

If the coprocessor executes instructions concurrently with the CY7C601, the architecture will support a coprocessor queue that functions in the same fashion as the floating-point queue.

2.3.3.7 Miscellaneous

Instructions in this category handle special circumstances within the integer unit (see *Table 2–22*). Execution of the UNIMP instruction causes an illegal instruction trap, so its execution is normally avoided except as part of a checking routine. Details of one possible use for UNIMP are given in its definition in Chapter 6.

The IFLUSH instruction is used to flush a word from an internal (to the CY7C601/611) instruction cache. Current integer unit implementations (CY7C601/611) do not incorporate an internal instruction cache, so IFLUSH would normally execute as a NOP. However, if there is an external instruction cache, IFLUSH causes an illegal instruction trap if the $\overline{\text{IFT}}$ signal is LOW (see Section 2.4).

2.3.4 Op Codes

This section contains tables that give a complete list of the instruction opcodes, both by functional groups and in ascending numeric order.

2.3.4.1 Load/Store Instructions

Table 2-23. Load/Store Instruction Opcodes

	Opcodes with Format						
Mnemonic	31 30	29 25	24 19	18 14	13	12 5	4 0
LD	1 1	rd	0 0 0 0 0 0	rs1	i = 0	asi	rs2
					i = 1	simm13	
LDA	1 1	rd	0 1 0 0 0 0	rs1	i = 0	asi	rs2
LDC	1 1	rd	1 1 0 0 0 0	rs1	i = 0	ignored	rs2
					i = 1	simm13	
LDCSR	1 1	rd	1 1 0 0 0 1	rs1	i = 0	ignored	rs2
					i = 1	simm13	
LDD	1 1	rd	0 0 0 0 1 1	rs1	i = 0	asi	rs2
					i = 1	simm13	
LDDA	1 1	rd	0 1 0 0 1 1	rs1	i = 0	asi	rs2
LDDC	1 1	rd	1 1 0 0 1 1	rs1	i = 0	ignored	rs2
					i = 1	simm13	
LDDF	1 1	rd	1 0 0 0 1 1	rs1	i = 0	ignored	rs2
					i = 1	simm13	
LDF	1 1	rd	1 0 0 0 0 0	rs1	i = 0	ignored	rs2
					i = 1	simm13	
LDFSR	1 1	rd	1 0 0 0 0 1	rs1	i = 0	ignored	rs2
					i = 1	simm13	
LDSB	1 1	rd	0 0 1 0 0 1	rs1	i = 0	asi	rs2
					i = 1	simm13	
LDSBA	1 1	rd	0 1 1 0 0 1	rs1	i = 0	asi	rs2
LDSH	1 1	rd	0 0 1 0 1 0	rs1	i = 0	asi	rs2
					i = 1	simm13	
LDSHA	1 1	rd	0 1 1 0 1 0	rs1	i = 0	asi	rs2
LDSTUB	1 1	rd	0 0 1 1 0 1	rs1	i = 0	asi	rs2
					i = 1	simm13	
LDSTUBA	1 1	rd	0 1 1 1 0 1	rs1	i = 0	asi	rs2
LDUB	1 1	rd	0 0 0 0 0 1	rs1	i = 0	asi	rs2
					i = 1	simm13	
LDUBA	1 1	rd	0 1 0 0 0 1	rs1	i = 0	asi	rs2
LDUH	1 1	rd	0 0 0 0 1 0	rs1	i = 0	asi	rs2
					i = 1	simm13	
LDUHA	1 1	rd	0 1 0 0 1 0	rs1	i = 0	asi	rs2

Table 2–23. Load/Store Instruction Opcodes (continued)

Mnemonic	Opcodes with Format 31 30	29 25	24 19	18 14	13	12 5	4 0
ST	1 1	rd	000100	rs1	i = 0	asi	rs2
					i = 1	simm13	
STA	1 1	rd	010100	rs1	i = 0	asi	rs2
STB	1 1	rd	000101	rs1	i = 0	asi	rs2
					i = 1	simm13	
STBA	1 1	rd	010101	rs1	i = 0	asi	rs2
STC	1 1	rd	110100	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STCSR	1 1	rd	110101	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STD	1 1	rd	000111	rs1	i = 0	asi	rs2
					i = 1	simm13	
STDA	1 1	rd	010111	rs1	i = 0	asi	rs2
STDC	1 1	rd	110111	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STDCQ	1 1	rd	110110	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STDF	1 1	rd	100111	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STDFQ	1 1	rd	100110	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STF	1 1	rd	100100	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STFSR	1 1	rd	100101	rs1	i = 0	ignored	rs2
					i = 1	simm13	
STH	1 1	rd	000110	rs1	i = 0	asi	rs2
					i = 1	simm13	
STHA	1 1	rd	010110	rs1	i = 0	asi	rs2
SWAP	1 1	rd	001111	rs1	i = 0	asi	rs2
					i = 1	simm13	
SWAPA	1 1	rd	011111	rs1	i = 0	asi	rs2

2.3.4.2 *Arithmetic/Logical/Shift Instructions*

Table 2–24. Arithmetic/Logical/Shift Instruction Opcodes

	Opcodes with Format							
Mnemonic	31 30	29 25	24 19	18 14	13	12 5	4 0	
ADD	1 0	rd	000000	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ADDcc	1 0	rd	010000	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ADDX	1 0	rd	001000	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ADDXcc	1 0	rd	011000	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
AND	1 0	rd	000001	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ANDcc	1 0	rd	010001	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ANDN	1 0	rd	000101	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ANDNcc	1 0	rd	010101	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
MULScc	1 0	rd	100100	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
OR	1 0	rd	000010	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ORcc	1 0	rd	010010	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ORN	1 0	rd	000110	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
ORNcc	1 0	rd	010110	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
SLL	1 0	rd	100101	rs1	i = 0	ignored	rs2	
					i = 1	shcnt		
SRA	1 0	rd	100111	rs1	i = 0	ignored	rs2	
					i = 1	shcnt		
SRL	1 0	rd	100110	rs1	i = 0	ignored	rs2	
					i = 1	shcnt		
SUB	1 0	rd	000100	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
SUBcc	1 0	rd	010100	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
SUBX	1 0	rd	001100	rs1	i = 0	ignored	rs2	
					i = 1	simm13		
SUBXcc	1 0	rd	011100	rs1	i = 0	ignored	rs2	
					i = 1	simm13		

Table 2–24. Arithmetic/Logical/Shift Instruction Opcodes (continued)

	Opcodes with Format																															
Mnemonic	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TADDcc	1	0	rd					1	0	0	0	0	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
TADDccTV	1	0	rd					1	0	0	0	1	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
TSUBcc	1	0	rd					1	0	0	0	0	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
TSUBccTV	1	0	rd					1	0	0	0	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
XNOR	1	0	rd					0	0	0	1	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
XNORcc	1	0	rd					0	1	0	1	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
XOR	1	0	rd					0	0	0	0	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
XORcc	1	0	rd					0	1	0	0	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
	31	30	29	28	27	26	25	24	23	22	21																					0
SETHI	0	0	rd					1	0	0	imm22																					

2.3.4.3 *Control Transfer Instructions*

Table 2–25. Control Transfer Instruction Opcodes

Mnemonic	Opcodes with Format																															
	31	30	29			25	24			19	18			14	13	12				5	4			0
JMPL	1	0	rd				1 1 1 0 0 0				rs1				i = 0	ignored					rs2			
															i = 1	simm13								
RESTORE	1	0	rd				1 1 1 1 0 1				rs1				i = 0	ignored					rs2			
															i = 1	simm13								
RETT	1	0	ignored				1 1 1 0 0 1				rs1				i = 0	ignored					rs2			
															i = 1	simm13								
SAVE	1	0	rd				1 1 1 1 0 0				rs1				i = 0	ignored					rs2			
															i = 1	simm13								
	31	30	29	28	25		24	22	21															0
Bicc	0	0	a	cond			0 1 0		disp22															
CBccc	0	0	a	cond			1 1 1		disp22															
FBfcc	0	0	a	cond			1 1 0		disp22															
	31	30	29	28	25		24			19	18			14	13	12				5	4			0
Ticc	1	0	I*	cond			1 1 1 0 1 0				rs1				i = 0	ignored					rs2			
															i = 1	simm13								
CALL	0	1	disp30																					

*I = ignored.

Table 2–26. Bicc and Ticc Condition Codes

Cond.	Test
0000	Never
0001	Equal to
0010	Less than or equal to
0011	Less than
0100	Less than or equal to, unsigned
0101	Carry set (less than, unsigned)
0110	Negative
0111	Overflow set
1000	Always
1001	Not equal to
1010	Greater than
1011	Greater than or equal to
1100	Greater than, unsigned
1101	Carry clear (greater than or equal, unsigned)
1110	Positive
1111	Overflow clear

Table 2–27. FBfcc Condition Codes

Cond.	Test
0000	Never
0001	Not equal
0010	Less than or greater than
0011	Unordered or less than
0100	Less than
0101	Unordered or greater than
0110	Greater than
0111	Unordered
1000	Always
1001	Equal
1010	Unordered or equal
1011	Greater than or equal
1100	Unordered or greater than or equal
1101	Less than or equal
1110	Unordered or less than or equal
1111	Ordered

Table 2–28. CBccc Condition Codes

Opcode	Cond.	CCC[1:0] Test
CBN	0000	Never
CB123	0001	1 or 2 or 3
CB12	0010	1 or 2
CB13	0011	1 or 3
CB1	0100	1
CB23	0101	2 or 3
CB2	0110	2
CB3	0111	3
CBA	1000	Always
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

2.3.4.4 Read/Write Control Register Instructions

Table 2–29. Read/Write Control Register Instruction Opcodes

	Opcodes with Format																															
Mnemonic	31	30	29				25	24					19	18				14	13	12												0
RDPSR	1	0	rd					1 0 1 0 0 1						ignored					I*	ignored												
RDTBR	1	0	rd					1 0 1 0 1 1						ignored					I*	ignored												
RDWIM	1	0	rd					1 0 1 0 1 0						ignored					I*	ignored												
RDY	1	0	rd					1 0 1 0 0 0						ignored					I*	ignored												
	31	30	29				25	24					19	18				14	13	12					5	4					0
WRPSR	1	0	ignored					1 1 0 0 0 1						rs1					i = 0	ignored						rs2					
																			i = 1	simm13											
WRTBR	1	0	ignored					1 1 0 0 1 1						rs1					i = 0	ignored						rs2					
																			i = 1	simm13											
WRWIM	1	0	ignored					1 1 0 0 1 0						rs1					i = 0	ignored						rs2					
																			i = 1	simm13											
WRY	1	0	ignored					1 1 0 0 0 0						rs1					i = 0	ignored						rs2					
																			i = 1	simm13											

*I = ignored.

2.3.4.5 *Floating-Point/Coprocessor Instructions*

Table 2–30. Floating-Point /Coprocessor Instruction Opcodes

| | Opcodes with Format | | |

2.3.4.6 *Miscellaneous Instructions*

Table 2–31. Miscellaneous Instruction Opcodes

Mnemonic	Opcodes with Format																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IFLUSH	1	0	ignored					1	1	1	0	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
UNIMP	0	0	ignored					0	0	0	const22																					

2.3.4.7 *Opcodes In Ascending Numeric Order*

Table 2–32. Instruction Opcode Numeric Listing

Mnemonic	Opcodes with Format																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UNIMP	0	0	ignored					0	0	0	const22																					
Bicc	0	0	a	cond				0	1	0	disp22																					
SETHI	0	0	rd					1	0	0	imm22																					
FBfcc	0	0	a	cond				1	1	0	disp22																					
CBccc	0	0	a	cond				1	1	1	disp22																					
CALL	0	1	disp30																													
ADD	1	0	rd					0	0	0	0	0	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
AND	1	0	rd					0	0	0	0	0	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
OR	1	0	rd					0	0	0	0	1	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
XOR	1	0	rd					0	0	0	0	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
SUB	1	0	rd					0	0	0	1	0	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
ANDN	1	0	rd					0	0	0	1	0	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
ORN	1	0	rd					0	0	0	1	1	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
XNOR	1	0	rd					0	0	0	1	1	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
ADDX	1	0	rd					0	0	1	0	0	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
SUBX	1	0	rd					0	0	1	1	0	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
ADDcc	1	0	rd					0	1	0	0	0	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
ANDcc	1	0	rd					0	1	0	0	0	1	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												
ORcc	1	0	rd					0	1	0	0	1	0	rs1					i = 0	ignored								rs2				
																			i = 1	simm13												

Table 2–32. Instruction Opcode Numeric Listing (continued)

| Mnemonic | Opcodes with Format | | |

Table 2–32. Instruction Opcode Numeric Listing (continued)

Mnemonic	Opcodes with Format 31 30	29 25	24 22	21 19	18 14	13 12 5	4 0
FPOP1	1 0	rd	1 1 0 1 0 0		rs1	OPF	rs2
FMOVs	1 0	rd	1 1 0 1 0 0		ignored	0 0 0 0 0 0 0 0 1	rs2
FNEGs	1 0	rd	1 1 0 1 0 0		ignored	0 0 0 0 0 0 1 0 1	rs2
FABSs	1 0	rd	1 1 0 1 0 0		ignored	0 0 0 0 0 1 0 0 1	rs2
FSQRTs	1 0	rd	1 1 0 1 0 0		ignored	0 0 0 1 0 1 0 0 1	rs2
FSQRTd	1 0	rd	1 1 0 1 0 0		ignored	0 0 0 1 0 1 0 1 0	rs2
FSQRTx	1 0	rd	1 1 0 1 0 0		ignored	0 0 0 1 0 1 0 1 1	rs2
FADDs	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 0 0 0 1	rs2
FADDd	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 0 0 1 0	rs2
FADDx	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 0 0 1 1	rs2
FSUBs	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 0 1 0 1	rs2
FSUBd	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 0 1 1 0	rs2
FSUBx	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 0 1 1 1	rs2
FMULs	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 1 0 0 1	rs2
FMULd	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 1 0 1 0	rs2
FMULx	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 1 0 1 1	rs2
FDIVs	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 1 1 0 1	rs2
FDIVd	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 1 1 1 0	rs2
FDIVx	1 0	rd	1 1 0 1 0 0		rs1	0 0 1 0 0 1 1 1 1	rs2
FiTOs	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 0 1 0 0	rs2
FdTOs	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 0 1 1 0	rs2
FxTOs	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 0 1 1 1	rs2
FiTOd	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 1 0 0 0	rs2
FsTOd	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 1 0 0 1	rs2
FxTOd	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 1 0 1 1	rs2
FiTOx	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 1 1 0 0	rs2
FsTOx	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 1 1 0 1	rs2
FdTOx	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 0 1 1 1 0	rs2
FsTOi	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 1 0 0 0 1	rs2
FdTOi	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 1 0 0 1 0	rs2
FxTOi	1 0	rd	1 1 0 1 0 0		ignored	0 1 1 0 1 0 0 1 1	rs2
FPOP2	1 0	rd	1 1 0 1 0 1		rs1	OPF	rs2
FCMPs	1 0	ignored	1 1 0 1 0 1		rs1	0 0 1 0 1 0 0 0 1	rs2
FCMPd	1 0	ignored	1 1 0 1 0 1		rs1	0 0 1 0 1 0 0 1 0	rs2
FCMPx	1 0	ignored	1 1 0 1 0 1		rs1	0 0 1 0 1 0 0 1 1	rs2
FCMPEs	1 0	ignored	1 1 0 1 0 1		rs1	0 0 1 0 1 0 1 0 1	rs2
FCMPEd	1 0	ignored	1 1 0 1 0 1		rs1	0 0 1 0 1 0 1 1 0	rs2
FCMPEx	1 0	ignored	1 1 0 1 0 1		rs1	0 0 1 0 1 0 1 1 1	rs2
CPOP1	1 0	rd	1 1 0 1 1 0		rs1	OPC	rs2
CPOP2	1 0	rd	1 1 0 1 1 1		rs1	OPC	rs2
JMPL	1 0	rd	1 1 1 0 0 0		rs1	i = 0 ignored	rs2
						i = 1 simm13	

Table 2–32. Instruction Opcode Numeric Listing (continued)

| Mnemonic | Opcodes with Format | | |

Table 2–32. Instruction Opcode Numeric Listing (continued)

	Opcodes with Format						
Mnemonic	31 30	29 25	24 22	21 19	18 14	13	12 5	4 0
LDSBA	1 1	rd	0 1 1 0 0 1		rs1	i = 0	asi	rs2
LDSHA	1 1	rd	0 1 1 0 1 0		rs1	i = 0	asi	rs2
LDSTUBA	1 1	rd	0 1 1 1 0 1		rs1	i = 0	asi	rs2
SWAPA	1 1	rd	0 1 1 1 1 1		rs1	i = 0	asi	rs2
LDF	1 1	rd	1 0 0 0 0 0		rs1	i = 0	ignored	rs2
						i = 1	simm13	
LDFSR	1 1	rd	1 0 0 0 0 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	
LDDF	1 1	rd	1 0 0 0 1 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STF	1 1	rd	1 0 0 1 0 0		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STFSR	1 1	rd	1 0 0 1 0 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STDFQ	1 1	rd	1 0 0 1 1 0		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STDF	1 1	rd	1 0 0 1 1 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	
LDC	1 1	rd	1 1 0 0 0 0		rs1	i = 0	ignored	rs2
						i = 1	simm13	
LDCSR	1 1	rd	1 1 0 0 0 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	
LDDC	1 1	rd	1 1 0 0 1 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STC	1 1	rd	1 1 0 1 0 0		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STCSR	1 1	rd	1 1 0 1 0 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STDCQ	1 1	rd	1 1 0 1 1 0		rs1	i = 0	ignored	rs2
						i = 1	simm13	
STDC	1 1	rd	1 1 0 1 1 1		rs1	i = 0	ignored	rs2
						i = 1	simm13	

2.4 Signal Description

This section provides a description of the CY7C601's (and CY7C611's) external signals. Functionally, the IU's external signals can be divided into four categories: memory subsystem interface, floating-point/coprocessor interface, interrupt and control signals, and power and clock signals.

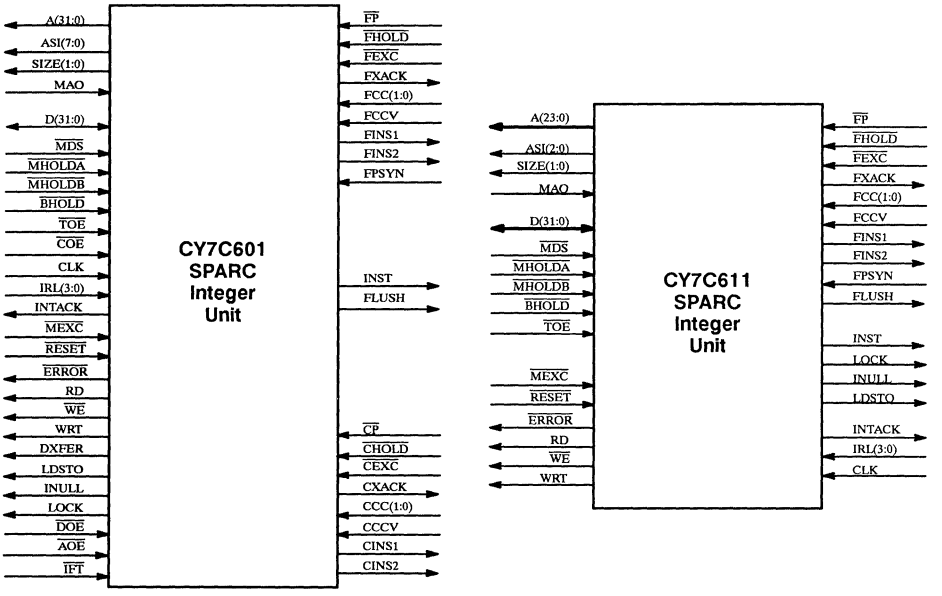


Figure 2–21. CY7C601/CY7C611 External Signals

Signals that are active LOW are marked with an overscore; all others are active HIGH. *Figure 2–21* summarizes the signals described in this section. *Table 2–33* provides a summary of the external signals for the CY7C601. The external signal summary for the CY7C611 is listed in *Table 2–40* in Section 2.9.

Note: In the descriptions below, and in this manual in general, when a signal is **asserted** it is active, and when it is **deasserted** it is inactive. When a signal is HIGH, it is a logical 1; when it is LOW, it is a logical 0. This is true regardless of whether it is asserted or deasserted.

Table 2–33. CY7C601 External Signal Summary

Memory Subsystem Interface Signals:			
Pin Name	**Description**	**Signal Type**	**Active**
A<31:0>	Address Bus	Three-State Output	
$\overline{AOE}$	Address Output Enable	Input	LOW
ASI<7:0>	Address Space Identifier	Three-State Output	
$\overline{COE}$	Control Output Enable	Input	LOW
$\overline{BHOLD}$	Bus Hold	Input	LOW
D<31:0>	Data Bus	Three-State BiDir.	
$\overline{DOE}$	Data Output Enable	Input	LOW
DXFER	Data Transfer	Three-State Output	HIGH
$\overline{IFT}$	Instruction Cache Flush Trap	Input	LOW
INULL	Integer Unit Nullify Cycle	Three-State Output	HIGH
LDSTO	Atomic Load-Store	Three-State Output	HIGH
LOCK	Bus Lock	Three-State Output	HIGH
MAO	Memory Address Output	Input	HIGH
$\overline{MDS}$	Memory Data Strobe	Input	LOW
$\overline{MEXC}$	Memory Exception	Input	LOW
$\overline{MHOLDA}$	Memory Bus Hold A	Input	LOW
$\overline{MHOLDB}$	Memory Bus Hold B	Input	LOW
RD	Read Access	Three-State Output	HIGH
SIZE<1:0>	Bus Transaction Size	Three-State Output	
$\overline{WE}$	Write Enable	Three-State Output	LOW
WRT	Advanced Write	Three-State Output	HIGH
Floating-Point / Coprocessor Interface Signals:			
Pin Name	**Description**	**Signal Type**	**Active**
CCC<1:0>	Coprocessor Condition Codes	Input	
CCCV	Coprocessor Condition Codes Valid	Input	HIGH
$\overline{CEXC}$	Coprocessor Exception	Input	LOW
$\overline{CHOLD}$	Coprocessor Hold	Input	LOW
CINS1	Coprocessor Instruction in Buffer 1	Three-State Output	HIGH
CINS2	Coprocessor Instruction in Buffer 2	Three-State Output	HIGH
$\overline{CP}$	Coprocessor Unit Present	Input	LOW
CXACK	Coprocessor Exception Acknowledge	Three-State Output	HIGH
FCC<1:0>	Floating-Point Condition Codes	Input	
FCCV	Floating-Point Condition Codes Valid	Input	HIGH
$\overline{FEXC}$	Floating-Point Exception	Input	LOW
$\overline{FHOLD}$	Floating-Point Hold	Input	LOW
FINS1	Floating-Point Instruction in Buffer 1	Three-State Output	HIGH
FINS2	Floating-Point Instruction in Buffer 2	Three-State Output	HIGH
FLUSH	Floating-Point/Coprocessor Instruction Flush	Three-State Output	HIGH
$\overline{FP}$	Floating-Point Unit Present	Input	LOW
FXACK	Floating-Point Exception Acknowledge	Three-State Output	HIGH
INST	Instruction Fetch	Three-State Output	HIGH

Table 2–33. CY7C601 External Signal Summary (continued)

Interrupt and Control Signals:			
Pin Name	**Description**	**Signal Type**	**Active**
IRL<3:0>	Interrupt Request Level	Input	
INTACK	Interrupt Acknowledge	Three-State Output	HIGH
$\overline{\text{RESET}}$	Reset	Input	LOW
$\overline{\text{ERROR}}$	Error State	Three-State Output	LOW
FPSYN	Floating-Point Synonym Mode	Input	HIGH
$\overline{\text{TOE}}$	Test Mode Output Enable	Input	LOW
Power and Clock Signals:			
Pin Name	**Description**	**Signal Type**	
CLK	Clock	Input	
VCCI	Main internal VCC	Input	
VCCO	Output driver VCC	Input	
VCCT	Input circuit VCC	Input	
VSSI	Main internal VSS	Input	
VSSO	Output driver VSS	Input	
VSST	Input circuit VSS	Input	

The following sections describe the external signals for the CY7C601 and CY7C611. Signals that are modified for the CY7C611 are listed in brackets, such as *[A<23:0>]*. Signals not available on the CY7C611 are denoted as *[Not available on CY7C611]*.

2.4.1 Memory Subsystem Interface Signals

Memory interface signals consist of the address lines (40 bits), bidirectional data lines (32 bits), transaction size lines (2 bits), and various control signals.

2.4.1.1 A<31:0>—Address Bus (output) [A<23:0>]

The 32-bit address bus carries instruction or data addresses during a fetch or load/store operation. Addresses are sent out unlatched and must be latched external to the CY7C601/611. Assertion of the MAO signal during a cache miss (which is signaled by pulling one of the $\overline{\text{MHOLD}}$ lines low) will force the Integer Unit to place the previous (missed) address on the address bus. The address bus is three-stated when the $\overline{\text{AOE}}$ or $\overline{\text{TOE}}$ signal is deasserted (HIGH).

2.4.1.2 $\overline{\text{AOE}}$—Address Output Enable (input) [Not available on CY7C611]

Assertion of this signal enables the output drivers for the address bus, A<31:0>, and the ASI bus, ASI<7:0>, and is the normal condition. Deassertion of $\overline{\text{AOE}}$ three-states the output drivers and should only be done when the bus is granted to another bus master (i.e., when either $\overline{\text{BHOLD}}$ or $\overline{\text{MHOLDA/B}}$ is asserted).

2.4.1.3 ASI<7:0>—Address Space Identifier (output) [ASI<2:0>]

These 8 bits constitute the Address Space Identifier (ASI), which identifies the memory address space to which the instruction or data access is being directed. The ASI bits are sent out unlatched—simultaneously with the memory address—and must be latched externally. Assertion of the MAO signal during a cache miss (which is signaled by pulling one of the $\overline{\text{MHOLD}}$ lines low) will force the integer unit to place the previous address space identifier on the ASI<7:0> pins. The ASI pins are three-stated when the $\overline{\text{AOE}}$ or $\overline{\text{TOE}}$ signal is deasserted (HIGH). Encoding of the ASI bits is shown in *Table 2–34*. Additional ASI assignments for the SPARC architecture are listed in *Table 4 - 15*.

Table 2–34. ASI Assignments

CY7C601 Address Space Identifier (ASI)	CY7C611 Address Space Identifier (ASI)	Address Space
00001000 (08 H)	000 (0 H)	User Instruction
00001010 (0A H)	010 (2H)	User Data
00001001 (09 H)	001 (1 H)	Supervisor Instruction
00001011 (0B H)	011 (3 H)	Supervisor Data

2.4.1.4 $\overline{BHOLD}$—Bus Hold (input)

$\overline{BHOLD}$ is asserted when an external bus master wants control of the data bus. Assertion of this signal will freeze the processor pipeline, so after deassertion of $\overline{BHOLD}$, external logic must guarantee that the data at all inputs to the CY7C601/611 is the same as it was before $\overline{BHOLD}$ was asserted. This signal is tested on the falling edge (midpoint) of a cycle and must be valid and stable at the processor for the duration of the specified set-up time prior to the falling edge of CLK. All HOLD signals are latched in the CY7C601/611 (transparent latch with clock high) before they are used. Because $\overline{MDS}$ and $\overline{MEXC}$ signals are not recognized while this input is active, $\overline{BHOLD}$ should only be used for bus access requests by an external device. $\overline{BHOLD}$ should not be asserted when LOCK is asserted.

2.4.1.5 $\overline{COE}$—Control Output Enable (input) [Not available on CY7C611]

Assertion of this signal enables the output drivers for SIZE < 1:0 >, RD, $\overline{WE}$, WRT, LOCK, LDSTO, and DXFER outputs, and is the normal condition. Deassertion of $\overline{COE}$ three-states these output drivers and should only be done when the bus is granted to another bus master (i.e., when either $\overline{BHOLD}$ or $\overline{MHOLDA/B}$ is asserted).

2.4.1.6 D<31:0>—Data Bus (bidirectional)

These pins form a 32-bit bidirectional data bus that serves as the interface between the integer unit and memory. The data bus is only driven by the CY7C601/611 during the execution of integer store instructions and the store cycle of atomic-load-store instructions. Similarly, the CY7C602 FPU drives the data bus only during the execution of floating-point store instructions.

Store data is sent out unlatched and must be latched externally before it is used. Once latched, store data is valid during the second data cycle of a store single access, the second and third data cycle of a store double access, and the third data cycle of an atomic-load-store access.

Alignment for load and store instructions is performed by the processor. Doublewords are aligned on 8-byte boundaries, words on 4-byte boundaries, and halfwords on 2-byte boundaries. If a doubleword, word, or halfword load or store instruction generates an improperly aligned address, a memory address not aligned trap will occur. Instructions and operands are always expected to reside in a 32-bit wide memory. D < 31 > corresponds to the most significant bit of the most significant byte of a 32-bit word going to or from memory.

2.4.1.7 $\overline{DOE}$—Data Output Enable (input) [Not available on CY7C611]

Assertion of this signal enables the output drivers for the data bus, D < 31:0 >, and is the normal condition. Deassertion of $\overline{DOE}$ three-states the data bus output drivers and should only be done when the bus is granted to another bus master (i.e., when either $\overline{BHOLD}$ or $\overline{MHOLDA/B}$ is asserted).

2.4.1.8 DXFER—Data Transfer (output) [Not available on CY7C611]

DXFER is used to differentiate between the addresses being sent out for instruction fetches and the addresses of data fetches. DXFER is asserted by the processor during the address cycles of all bus data transfer cycles, including both cycles of store single and all three cycles of store double and atomic load-store. DXFER is sent out unlatched and must be latched externally before it is used.

2.4.1.9 $\overline{IFT}$—Instruction Cache Flush Trap (input) [Not available on CY7C611]

The state of this pin determines whether or not execution of the IFLUSH instruction generates a trap. If $\overline{IFT} = 0$, then execution of IFLUSH causes an illegal instruction trap. If $\overline{IFT} = 1$, then IFLUSH executes like a NOP with no side effects.

2.4.1.10 *INULL—Integer Unit Nullify Cycle (output)*

The processor asserts INULL to indicate that the current memory access is being nullified. It is asserted in the same cycle in which the address being nullified is active (though no longer on the address bus, the address is held in the external address latches). INULL is used to prevent a cache miss (in systems with cache memory) and to disable memory exception generation for the current memory access. This means that $\overline{\text{MDS}}$ and MEXC should not be asserted for a memory access in which INULL = 1. INULL is a latched output and should not be latched externally. If a floating-point unit or coprocessor is present in the system, INULL should be ORed with the FNULL and CNULL signals to generate a final NULL signal.

INULL is asserted under the following conditions:

1. During the second data cycle of any store instruction (including Atomic Load-Store) to nullify the second occurrence of the store address.
2. On all traps, to nullify the third instruction fetch after the trapped instruction. For reset, it nullifies the error-producing address.
3. On a load in which the hardware interlock is activated.
4. JMPL and RETT instructions.

2.4.1.11 *LDSTO—Atomic Load-Store (output)*

This signal is used to identify an atomic load-store to the system and is asserted by the integer unit during all the data cycles (the load cycle and both store cycles) of atomic load-store instructions. LDSTO is sent out unlatched and must be latched externally before it is used.

2.4.1.12 *LOCK—Bus Lock (output)*

LOCK is asserted by the processor when it needs to retain control of the bus (address and data) for multiple cycle transactions (Load Double, Store Single and Double, Atomic Load-Store). The bus will not be granted to another bus master as long as LOCK is asserted. Note that $\overline{\text{BHOLD}}$ should not be asserted in the processor clock cycle which follows a cycle in which LOCK is asserted. LOCK is sent out unlatched and must be latched externally before it is used.

2.4.1.13 *MAO—Memory Address Output (input)*

This signal is asserted during an $\overline{\text{MHOLD}}$ condition to force the previous (missed) memory access parameters back on their various buses and control lines. The miss parameters are those that were valid on the rising edge of the clock, one cycle before the cycle in which $\overline{\text{MHOLD}}$ was asserted. A logic HIGH value at this pin during a cache miss causes the integer unit to put A<31:0>, ASI<7:0>, SIZE<1:0>, RD, $\overline{\text{WE}}$, WRT, LDSTO, LOCK, and DXFER values corresponding to the missed memory address on the bus.

Normally, MAO is kept at a LOW level, thereby selecting the access parameters for the current memory address. MAO should not be used for a cache miss during a store cycle, because it would select the wrong value for $\overline{\text{WE}}$.

MAO must be driven LOW while $\overline{\text{RESET}}$ is LOW.

2.4.1.14 *$\overline{\text{MDS}}$—Memory Data Strobe (input)*

$\overline{\text{MDS}}$ is asserted by the memory system to enable the clock to the integer unit's instruction register (during an instruction fetch) or to the load result register (during a data fetch) while the pipeline is frozen with an $\overline{\text{MHOLDA/B}}$. In a system with cache, $\overline{\text{MDS}}$ is used to signal the processor when the missed data (cache miss) is ready on the data bus. In a system with slow memories, $\overline{\text{MDS}}$ tells the processor when the read data is available on the bus. During a cache line replacement, $\overline{\text{MDS}}$ may be asserted anywhere within the $\overline{\text{MHOLD}}$ cycle and deasserted before $\overline{\text{MHOLD}}$ is released. For example, if a cache miss occurs on word 2 of a 4-word cache line, $\overline{\text{MDS}}$ should only be driven active while word 2 is being replaced in the cache.

$\overline{\text{MDS}}$ is also used to strobe in the $\overline{\text{MEXC}}$ memory exception signal. $\overline{\text{MDS}}$ may only be asserted when the pipeline is frozen with $\overline{\text{MHOLDA/B}}$. The CY7C601/611 samples $\overline{\text{MDS}}$ with an on-chip transparent latch before it is used.

2.4.1.15 *$\overline{\text{MEXC}}$—Memory Exception (input)*

Assertion of this signal by the memory system initiates an instruction access exception or data access exception trap and indicates to the CY7C601/611 that the memory system was unable to supply a valid instruction or data. If $\overline{\text{MEXC}}$ is as-

serted during an instruction fetch cycle, it generates an instruction access exception trap. If asserted during a data cycle, it generates a data access exception trap.

$\overline{\text{MEXC}}$ is used as a qualifier for the $\overline{\text{MDS}}$ signal, and must be asserted when both $\overline{\text{MHOLDA/B}}$ and $\overline{\text{MDS}}$ are already asserted. If $\overline{\text{MDS}}$ is applied without $\overline{\text{MEXC}}$, the CY7C601/611 accepts the contents of the data bus as valid. If $\overline{\text{MEXC}}$ accompanies MDS, an exception is generated and the data bus content is ignored.

$\overline{\text{MEXC}}$ is latched in the processor on the rising edge of CLK and is used in the following cycle. $\overline{\text{MEXC}}$ must be deasserted in the same clock cycle in which $\overline{\text{MHOLDA/B}}$ is deasserted.

2.4.1.16 $\overline{\text{MHOLD}}$(A/B)—Memory Holds (inputs)

$\overline{\text{MHOLDA}}$ is used to freeze the clock to both the integer and floating-point units during a cache miss (for systems with cache memory) or when accessing a slow memory. The processor pipeline is frozen while $\overline{\text{MHOLDA}}$ is asserted and the CY7C601/611 outputs revert to and maintain the value they had at the rising edge of the clock in the cycle in which $\overline{\text{MHOLDA}}$ was asserted. This signal is tested on the falling edge (midpoint) of a cycle and must be valid and stable at the processor for the duration of the specified set-up time prior to the falling edge of CLK.

$\overline{\text{MHOLDB}}$ behaves in the same fashion as $\overline{\text{MHOLDA}}$, and either can be used to stop the processor during a cache miss or memory exception. The pipeline is actually frozen by a "final" hold signal that is the logical OR of all hold signals ($\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, and BHOLD). All HOLD signals are latched in the CY7C601/611 (transparent latch with clock high) before they are used.

Note that $\overline{\text{MHOLD}}$ must be driven HIGH while $\overline{\text{RESET}}$ is LOW.

2.4.1.17 RD—Read Access (output)

RD is sent out during the address portion of an access to specify whether the current memory access is a read (RD = 1) or a write (RD = 0) operation. RD is set to "0" only during the address cycles of store instructions. For atomic load-store instructions, RD is "1" during the load address cycle and "0" during the two store address cycles. It is sent out unlatched by the Integer Unit and must be latched externally before it is used.

RD is used in conjunction with SIZE < 1:0 >, ASI < 7:0 >, and LDSTO to determine the type and to check the read/write access rights of bus transactions. It may also be used to turn off the output drivers of data RAMs during a store operation.

2.4.1.18 SIZE < 1:0 > —Bus Transaction Size (outputs)

The coding on these pins specifies the size of the data being transferred during an instruction or data fetch. The value of the size bits during a given cycle relates only to the memory address which appears on pins A < 31:0 > simultaneously with the size outputs. It does not apply to data which may be on the data bus during that same cycle.

Size bits are sent out unlatched and must be latched external to the CY7C601/611 before they are used. SIZE < 1:0 > remains valid during the data address cycles of loads, stores, load doubles, store doubles, and atomic load-stores. Encoding of the size bits is shown in *Table 2–35*. For example, during an instruction fetch, SIZE < 1:0 > is set to "10", because all instructions are 32 bits long. For doubleword instructions, SIZE < 1:0 > is "11" for all data address cycles.

Table 2–35. SIZE Bit Encoding

SIZE < 1 >	SIZE < 0 >	Data Transfer Type
0	0	Byte
0	1	Halfword
1	0	Word
1	1	Word (Load/Store Double)

2.4.1.19 $\overline{\text{WE}}$—Write Enable (output)

$\overline{\text{WE}}$ is asserted by the integer unit during the cycle in which the store data is on the data bus. For a store single instruction, this is during the second store address cycle; the second and third store address cycles of store double instructions, and

the third load-store address cycle of atomic load-store instructions. It is sent out unlatched and must be latched externally before it is used. To avoid writing to memory during memory exceptions, $\overline{WE}$ must be externally qualified by the $\overline{MHOLDA/B}$ signals.

2.4.1.20 WRT—Advanced Write (output)

WRT is an early write signal, asserted by the processor during the first store address cycle of integer single or double store instructions, the first store address cycle of floating-point single or double store instructions, and the second load-store address cycle of atomic load-store instructions. WRT is sent out unlatched and must be latched externally before it is used.

2.4.2 Floating-Point/Coprocessor Interface Signals

The IU incorporates a dedicated group of pins that act as direct-connect interfaces between the integer unit and both the floating-point unit and the coprocessor. Using these connections, no external circuits are required to interface the IU to the FPU and coprocessor. The interfaces consist of the following signals:

2.4.2.1 CCC<1:0>—Coprocessor Condition Codes (input) [Not available on CY7C611]

These lines represent the current condition code bits from the Coprocessor State Register (CSR), qualified by the CCCV signal. When CCCV = 1, these bits are valid. During the execution of a CBccc instruction, the processor uses CCC<1:0> to determine whether or not to take the branch. These bits are latched by the processor before they are used.

2.4.2.2 CCCV—Coprocessor Condition Codes Valid (input) [Not available on CY7C611]

This signal is a specialized hold used to synchronize coprocessor compare instructions with coprocessor branch instructions. It is asserted (the normal condition) whenever the CCC<1:0> bits are valid. A coprocessor would deassert CCCV (CCCV = 0) as soon as a coprocessor compare instruction enters the coprocessor queue, unless an exception is detected (see Section 2.8). Deasserting CCCV freezes the integer unit pipeline, preventing any further compares from entering the pipeline. CCCV is reasserted when the compare is completed and the coprocessor condition codes are valid, thus ensuring that the condition codes match the proper compare instruction. CCCV is latched in the CY7C601 before it is used.

2.4.2.3 $\overline{CEXC}$—Coprocessor Exception (input) [Not available on CY7C611]

$\overline{CEXC}$ is used to signal the integer unit that a coprocessor exception has occurred. $\overline{CEXC}$ must remain asserted until the CY7C601 takes the trap and acknowledges the FPU exception via the CXACK signal. Although coprocessor exceptions can occur at any time, they are taken by the CY7C601 only during the execution of a subsequent CPop, a CBfcc instruction, or a coprocessor load or store instruction. A coprocessor implementation should deassert $\overline{CHOLD}$ if it detects an exception while $\overline{CHOLD}$ is asserted. In such a case, $\overline{CEXC}$ should be asserted one cycle before $\overline{CHOLD}$ is deasserted. $\overline{CEXC}$ is latched in the CY7C601 before it is used.

2.4.2.4 $\overline{CHOLD}$—Coprocessor Hold (input) [Not available on CY7C611]

This signal is asserted by the coprocessor if a situation arises in which it cannot continue execution. The coprocessor checks all dependencies in the decode stage of the instruction and asserts $\overline{CHOLD}$ (if necessary) in the next cycle. If the integer unit receives a $\overline{CHOLD}$, it freezes the instruction pipeline in the same cycle. Once the conditions causing the $\overline{CHOLD}$ are resolved, the coprocessor deasserts $\overline{CHOLD}$, releasing the instruction pipeline. $\overline{CHOLD}$ is latched in the CY7C601 before it is used.

The conditions under which the coprocessor asserts $\overline{CHOLD}$ are implementation dependent.

2.4.2.5 CINS1—Coprocessor Instruction in Buffer 1 (output) [Not available on CY7C611]

CINS1 is asserted by the integer unit during the decode stage of the coprocessor instruction that is in the D1 buffer of the coprocessor chip. The coprocessor uses this signal to begin decoding and execution of the D1 instruction, and to latch it into its execute-stage register. CINS1 and CINS2 are never asserted in the same cycle.

2.4.2.6 *CINS2—Coprocessor Instruction in Buffer 2 (output) (Not available on CY7C611)*

CINS2 is asserted by the Integer Unit during the decode stage of the coprocessor instruction that is in the D2 buffer of the coprocessor chip. The Coprocessor uses this signal to begin decoding and execution of the D2 instruction, and to latch it into its execute-stage register. CINS1 and CINS2 are never asserted in the same cycle.

2.4.2.7 $\overline{CP}$*—Coprocessor Unit Present (input) [Not available on CY7C611]*

When pulled low, $\overline{CP}$ indicates that a coprocessor is available to the system. It is normally pulled up to VDD through a resistor, and then grounded by connection to the coprocessor. The integer unit will generate a cp disabled trap if $\overline{CP} = 1$ during the execution of an CPop, CBfcc, or coprocessor load or store instruction.

2.4.2.8 *CXACK—Coprocessor Exception Acknowledge (output) [Not available on CY7C611]*

CXACK is asserted by the integer unit to inform the coprocessor that a trap has been taken for the currently asserted $\overline{CEXC}$ signal. Receipt of the asserted CXACK causes the coprocessor to deassert $\overline{CEXC}$, which in turn causes the to deassert CXACK. CXACK is a latched output and should not be latched externally.

2.4.2.9 *FCC < 1:0 > —Floating-Point Condition Codes (input)*

These lines represent the current condition code bits from the FPU's Floating-point State Register (FSR), qualified by the FCCV signal. When FCCV = 1, these bits are valid. During the execution of an FBfcc instruction, the processor uses FCC < 1:0 > to determine whether or not to take the branch. These bits are latched by the processor before they are used.

2.4.2.10 *FCCV—Floating-Point Condition Codes Valid (input)*

This signal is a specialized hold used to synchronize FPU compare instructions with floating-point branch instructions. It is asserted (the normal condition) whenever the FCC < 1:0 > bits are valid. The CY7C602 deasserts FCCV (FCCV = 0) as soon as a floating-point compare instruction enters the floating-point queue, unless an exception is detected (see Section 3.2.1.2.1). Deasserting FCCV freezes the integer unit pipeline, preventing any further compares from entering the pipeline. FCCV is reasserted when the compare is completed and the floating-point condition codes are valid, thus ensuring that the condition codes match the proper compare instruction. FCCV is latched in the CY7C601/611 before it is used.

2.4.2.11 $\overline{FEXC}$*—Floating-Point Exception (input)*

$\overline{FEXC}$ is used to signal the integer unit that a floating-point exception has occurred. $\overline{FEXC}$ must remain asserted until the CY7C601/611 takes the trap and acknowledges the FPU exception via the FXACK signal. Although floating-point exceptions can occur at any time, they are taken by the CY7C601/611 only during the execution of a subsequent FPop, an FBfcc instruction, or a floating-point load or store instruction. The CY7C602 deasserts $\overline{FHOLD}$ if it detects an exception while $\overline{FHOLD}$ is asserted. In such a case, $\overline{FEXC}$ is asserted one cycle before $\overline{FHOLD}$ is deasserted. FEXC is latched in the CY7C601/611 before it is used.

2.4.2.12 $\overline{FHOLD}$*—Floating-Point Hold (input)*

This signal is asserted by the CY7C602 if a situation arises in which the FPU cannot continue execution. The FPU checks all dependencies in the decode stage of the instruction and asserts $\overline{FHOLD}$ (if necessary) in the next cycle. If the integer unit receives an $\overline{FHOLD}$, it freezes the instruction pipeline in the same cycle. Once the conditions causing the $\overline{FHOLD}$ are resolved, the FPU deasserts $\overline{FHOLD}$, releasing the instruction pipeline. $\overline{FHOLD}$ is latched in the CY7C601/611 before it is used.

An $\overline{FHOLD}$ is asserted if (1) the FPU encounters an STFSR instruction with one or more FPops pending in the queue, (2) if either a resource or operand dependency exists between the FPop being decoded and any FPops already being executed, or (3) if the floating-point queue is full.

2.4.2.13 *FINS1—Floating-Point Instruction In Buffer 1 (output)*

FINS1 is asserted by the integer unit during the decode stage of the floating-point instruction that is in the D1 buffer of the floating-point unit (see Section 3.2). The FPU uses this signal to begin decoding and execution of the D1 instruc-

tion, and to latch it into its execute-stage register. FINS1 and FINS2 are never asserted in the same cycle and both are ignored if (1) FLUSH is asserted, (2) any HOLD is asserted, or (3) if FCCV or CCCV is deasserted.

2.4.2.14 FINS2—Floating-Point Instruction In Buffer 2 (output)

FINS2 is asserted by the integer unit during the decode stage of the floating-point instruction that is in the D2 buffer of the floating-point unit (see Section 3.1). The FPU uses this signal to begin decoding and execution of the D2 instruction, and to latch it into its execute-stage register. FINS1 and FINS2 are never asserted in the same cycle and both are ignored if (1) FLUSH is asserted, (2) any HOLD is asserted, or (3) if FCCV or CCCV is deasserted.

2.4.2.15 FLUSH—Floating-Point/Coprocessor Instruction Flush (output)

This signal is asserted by the integer unit whenever it takes a trap. FLUSH is used by the FPU (or coprocessor) to flush the instructions in its instruction buffers. These instructions, as well as the instructions annulled in the CY7C601/611's pipeline, are restarted after the trap handler is finished. If the trap was not caused by a floating-point (or coprocessor) exception, instructions already in the floating-point (or coprocessor) queue may continue their execution. If the trap was caused by a floating-point (or coprocessor) exception, the fp (or cp) queue must be emptied before the FPU (coprocessor) can resume execution.

2

2.4.2.16 $\overline{FP}$—Floating-point Unit Present (input)

When pulled low, $\overline{FP}$ indicates that a floating-point unit is available to the system. It is normally pulled up to VDD through a resistor, and then grounded by connection to the FPU. The integer unit will generate an fp disabled trap if $\overline{FP} = 1$ during the execution of an FPop, FBfcc, or floating-point load or store instruction.

2.4.2.17 FXACK—Floating-Point Exception Acknowledge (output)

FXACK is asserted by the integer unit to inform the floating-point unit that a trap has been taken for the currently asserted $\overline{FEXC}$ signal. Receipt of the asserted FXACK causes the FPU to deassert $\overline{FEXC}$, which in turn causes the CY7C601/611 to deassert FXACK. FXACK is a latched output and should not be latched externally.

2.4.2.18 INST—Instruction Fetch (output)

The INST signal is asserted by the integer unit whenever a new instruction is being fetched. It is used by the floating-point unit or coprocessor to latch the instruction currently on the data bus into an FPU or coprocessor instruction buffer. SPARC-compatible floating-point units and coprocessors have two instruction buffers (D1 and D2) to save the last two fetched instructions (see Section 3.2). When INST is asserted, a new instruction enters buffer D1 and the instruction that was in D1 moves to buffer D2. INST is a latched output and should not be latched externally.

2.4.3 Interrupt and Control Signals

The following signals are used by the integer unit to control and to receive input from external events.

2.4.3.1 $\overline{ERROR}$—Error State (output)

This signal is asserted when the integer unit enters the error mode state. This happens if a synchronous trap occurs while traps are disabled (the PSR's ET bit = 0). Before it enters the error mode state, the CY7C601/611 saves the PC and nPC and sets the trap type (tt) for the trap causing the error mode into the TBR. It then asserts the $\overline{ERROR}$ signal and halts. The only way to restart a processor which is in the error mode state is to trigger a reset by asserting the $\overline{RESET}$ signal.

2.4.3.2 FPSYN—Floating-point Synonym Mode (input)

This is a mode pin which will be used to allow execution of additional instructions in future designs. For the CY7C601/611, it should be kept grounded.

2.4.3.3 INTACK—Interrupt Acknowledge (output)

INTACK is a latched output that is asserted by the integer unit when an external interrupt is *taken*, not when it is sampled and latched.

2.4.3.4 IRL<3:0>—Interrupt Request Level (input)

The state of these pins defines the External Interrupt Level (IRL). IRL<3:0> =0000 indicates that no external interrupts are pending and is the normal state of the IRL pins. IRL<3:0> = 1111 signifies a nonmaskable interrupt. All other interrupt levels are maskable by the Processor Interrupt Level (PIL) field of the Processor State Register (PSR). The integer unit uses two on-chip synchronizing latches to sample these signals, and a given level must remain valid for two consecutive cycles to be recognized. External interrupts should be latched and prioritized by external logic before they are passed to the CY7C601/611. Logic must also keep an interrupt valid until it is taken and acknowledged. External interrupts can be acknowledged by system software or by the CY7C601/611's INTerrupt ACKnowledge (INTACK) signal.

2.4.3.5 $\overline{\text{RESET}}$—Integer Unit Reset (input)

Assertion of this pin will reset the integer unit. $\overline{\text{RESET}}$ must be asserted for a minimum of eight processor clock cycles. After $\overline{\text{RESET}}$ is deasserted, the integer unit starts fetching from address 0. $\overline{\text{RESET}}$ is latched by the CY7C601/611 before it is used.

2.4.3.6 $\overline{\text{TOE}}$—Test Mode Output Enable (input)

When *deasserted*, this signal will three-state all integer unit output drivers. Thus, in normal operation, this pin should always be asserted (tied to ground). Deassertion of $\overline{\text{TOE}}$ isolates the CY7C601/611 from the system for debugging purposes.

2.4.4 Power and Clock Signals

The signals listed below provide clocking and power to the integer unit.

2.4.4.1 CLK—Clock (input)

CLK is a 50%-duty-cycle clock used for clocking the integer unit's pipeline registers. The rising edge of CLK defines the beginning of each pipeline stage and a processor cycle is equal to a full clock cycle.

2.4.4.2 VCCO, VCCI, VCCT—Power (inputs)

These pins provide + 5V power to various sections of the processor. Power is supplied on three different buses to provide clean, stable power to each section: output drivers, main internal circuitry, and the input circuits. VCCO pins supply the output driver bus; VCCI pins supply main internal circuitry bus; and VCCT pins supply the input circuit bus. See Section 7.1 for pin identification.

2.4.4.3 VSSO, VSSI, VSST—Ground (inputs)

These pins provide ground return for the power signals. Ground is supplied on three different buses to match the power signals to each section: VSSO pins for the output driver bus; VSSI pins for the main internal circuitry bus; and VSST pins for the input circuit bus. See Section 7.1 for pin identification.

2.5 Pipeline and Instruction Execution Timing

One of the major contributing factors to the CY7C601/611's very high performance is an instruction execution rate approaching one instruction per clock cycle. To achieve that rate of execution, the CY7C601/611 employs a four-stage instruction pipeline that permits parallel execution of multiple instructions.

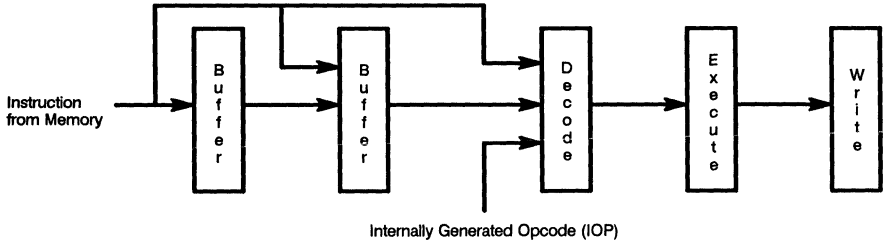


Figure 2–22. Processor Instruction Pipeline

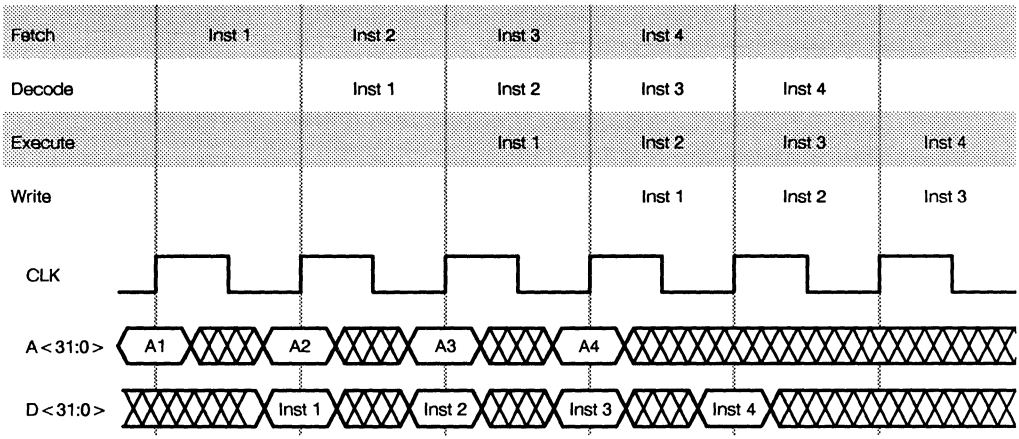


Figure 2–23. Pipeline with All Single-Cycle Instructions

2.5.1 Stages

Instruction execution is broken into four stages corresponding to the stages of the pipeline:

1. **Fetch**—The processor outputs the instruction address to fetch the instruction.
2. **Decode**—The instruction is placed in the instruction register and decoded. The processor reads the operands from the register file and computes the next instruction address.
3. **Execute**—The processor executes the instruction and saves the results in temporary registers. Pending traps are prioritized and internal traps taken during this stage.
4. **Write**—If no trap is taken, the processor writes the result to the destination register.

All four stages operate in parallel, working on up to four different instructions at a time. A basic "single-cycle" instruction enters the pipeline and completes in four cycles. By the time it reaches the write stage, three more instructions have entered and are moving through the pipeline behind it. So, after the first four cycles, a single-cycle instruction exits the pipeline and a single-cycle instruction enters the pipeline on every cycle (see *Figure 2–23*).

Of course, a "single-cycle" instruction actually takes four cycles to complete, but they are called single cycle because with this type of instruction the processor can complete one instruction per cycle after the initial four-cycle delay.

2.5.1.1 Internal Opcodes

Instructions that require extra cycles automatically insert internal opcodes (IOPs) into the decode stage as they move into the execute stage. These internal opcodes are unique to the instruction that generates them. They move all the way through the pipeline, performing functions specific to the instruction that created them. For example, in *Figure 2–24*, the data load in cycle four can be thought of as the fetch for the IOP that starts in cycle three; together they make a complete four-cycle instruction that balances out the pipeline. JMPL and RETT also generate an IOP, but have no external data cycle.

Multicycle instructions may generate up to three IOPs to complete execution. *Table 2–36* lists the instructions that require IOPs and the number generated.

Because instructions continue to be fetched even though IOPs occupy the decode stage, a two-stage prefetch buffer is used to hold instructions until they can move into the decode stage (see *Figure 2–22*). This enables the processor to fully utilize the data bus bandwidth and still keep the pipeline full. Only two buffers are required because a maximum of two cycles are available for instruction fetching for any multicycle instruction.

Table 2–36. Internally Generated Opcodes

Instruction	Number of Internal Opcodes
Single Loads	1
Double Loads	2
Single Stores	2
Double Stores	3
Atomic Load-Store	3
Jump	1
Return from Trap	1

2.5.2 Multicycle Instructions

Multicycle instructions are those that take more than four cycles (one bus cycle plus the three pipeline cycles) to complete. A double-cycle instruction takes five cycles (two bus cycles), a triple-cycle instruction takes six cycles (three bus cycles), and so on.

In most cases, the extra cycles required by multicycle instructions result from data bus usage (e.g., a data load or store to memory) that prevents the processor from fetching the next instruction during those cycles. In *Figure 2–24*, the fetch of instruction Inst 3 is delayed by one cycle for the data load, and in *Figure 2–25*, the store sequence delays the Inst 3 fetch by two cycles.

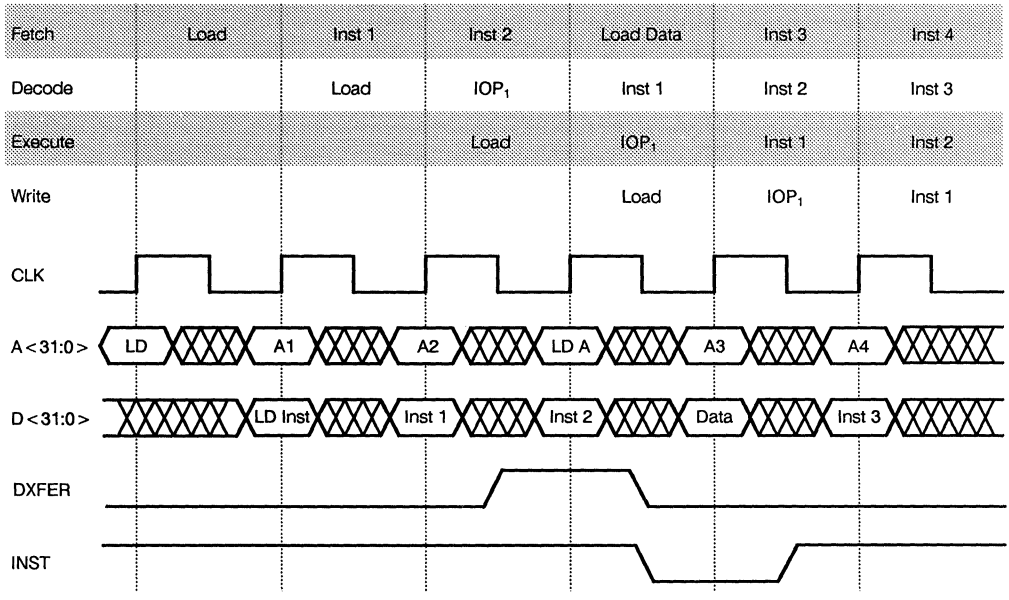


Figure 2–24. Pipeline with One Double-Cycle Instruction (Load)

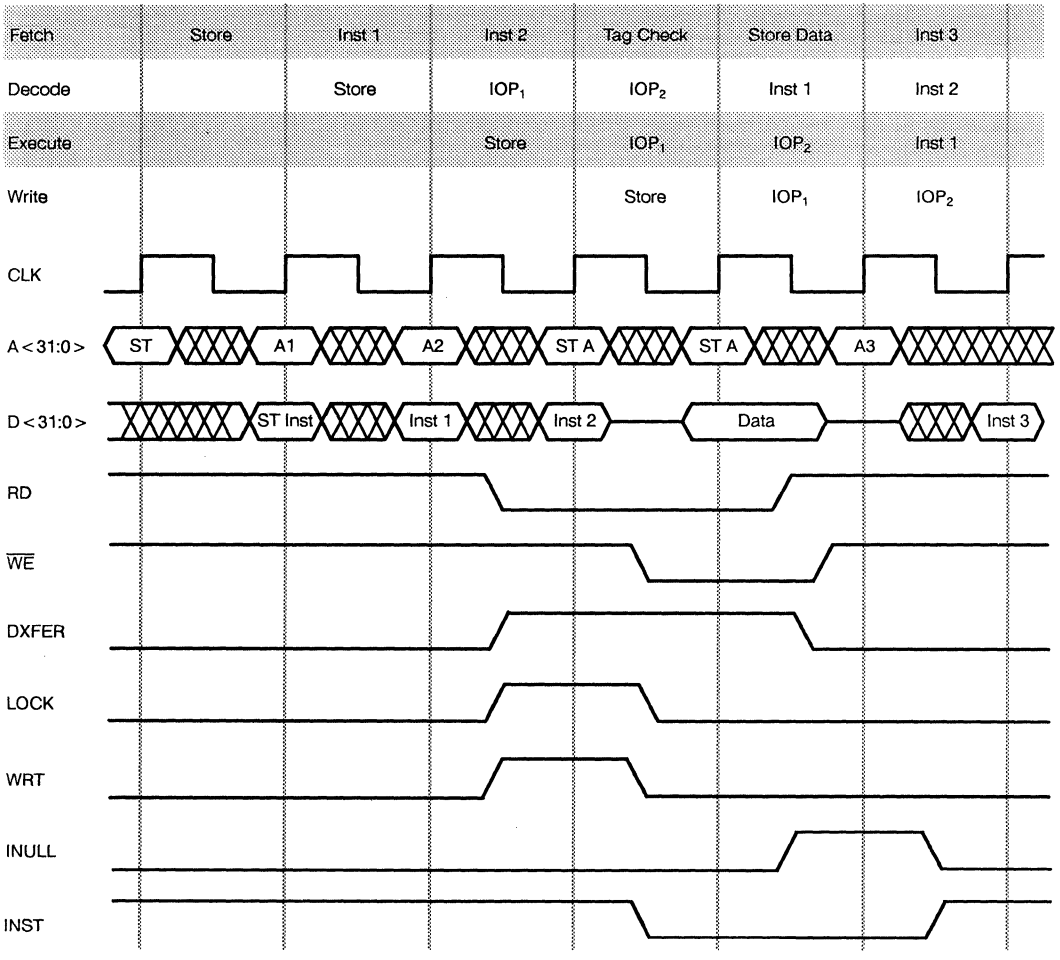


Figure 2–25. Pipeline with One Triple-Cycle Instruction (Store)

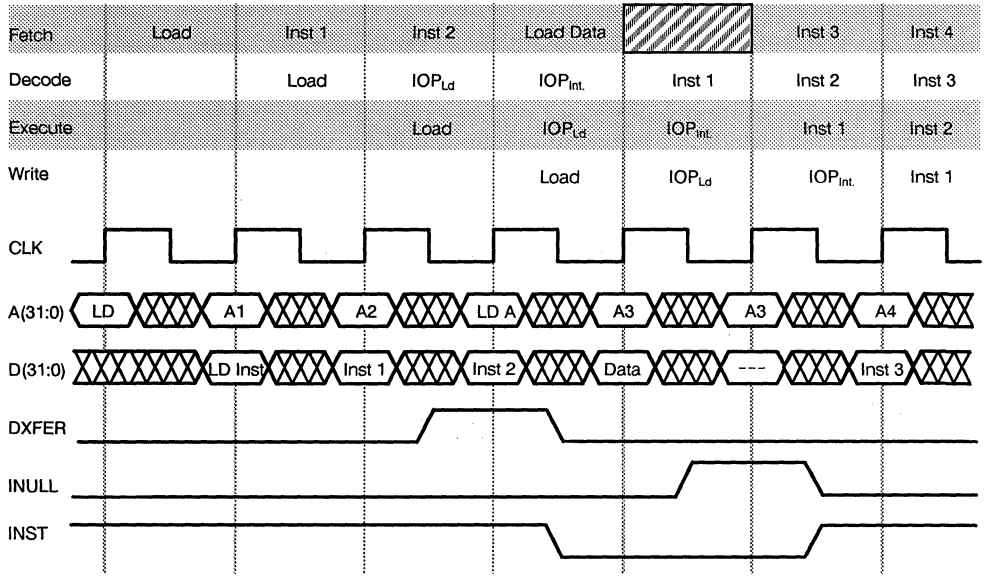


Figure 2–26. Pipeline with Hardware Interlock (Load)

2.5.2.1 Register Interlocks

The pipeline holds several instructions at any given time, so it is possible that an instruction may try to use the contents of a particular register which is in the process of being updated by a previous instruction. Special bypass paths in the pipeline of the CY7C601/611 make the correct data available to subsequent instructions for all internal register to register operations, but cannot solve the problem of loads to the registers from external memory. For this case, interlock hardware prevents an instruction following a load instruction from reading the register being loaded until the load is complete (see *Figure 2–26*). This also applies to a a CALL instruction with a delay slot instruction using r[15] and a JMPL with a delay slot instruction using the same register specified as the r[rd] of the JMPL. To maximize performance, compilers and assembly language programmers should avoid loads followed immediately by instructions using the loaded register's contents.

2.5.2.2 Branching

The CY7C601/611's delayed-control-transfer mechanism allows branches (taken or untaken) to occur without creating a bubble in the pipeline (see *Figure 2–27*). Special parallel hardware enables the processor to evaluate the condition codes and calculate the effective branch address during the decode stage rather than the execute stage, so that only one delay instruction is required between the branch and the target instruction (or the next instruction, if the branch is not taken). See Section 2.3.3.3.1 for a discussion on branching.

If the compiler or programmer cannot place an appropriate instruction in the delay instruction slot, the delay instruction can be annulled by setting the branch instruction's *a* bit. The result is shown in *Figure 2–28*.

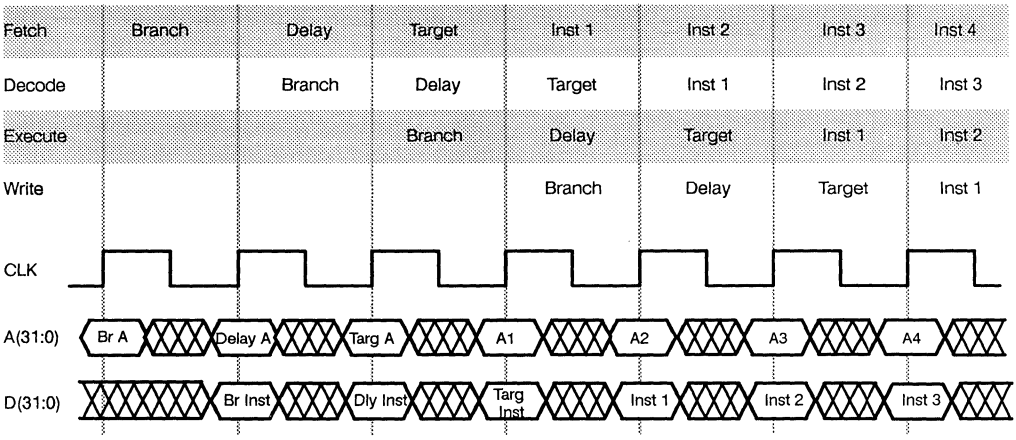


Figure 2–27. Pipeline During Branch Instruction

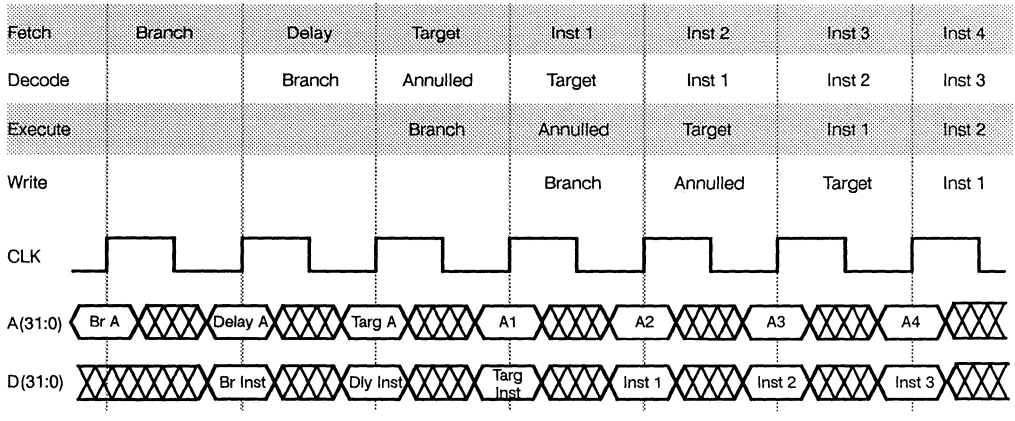


Figure 2–28. Branch with Annulled Delay Instruction

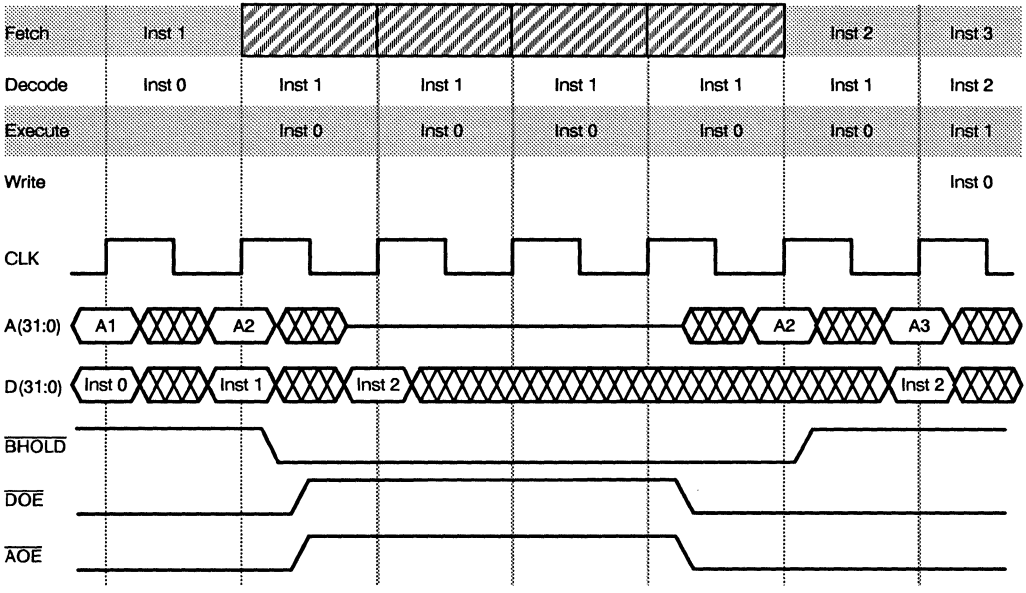


Figure 2–29. Pipeline Frozen During Bus Arbitration

2.5.3 Pipeline Freezes

Whenever the processor receives an externally generated hold input, such as $\overline{\text{MHOLDA/B}}$ or $\overline{\text{BHOLD}}$, the instruction pipeline is frozen. How long it is frozen depends on the type of hold and the external hardware generating the hold. *Figure 2–29* shows the pipeline frozen by a $\overline{\text{BHOLD}}$ as the result of bus arbitration initiated by another bus master in the system.

2.5.4 Traps

Figure 2–30 shows the pipeline operation when an internally generated trap is taken. Instructions in the pipeline after detection of the trap are annulled and the first instruction of the trap target routine is executed in the fourth cycle following detection.

2.6 Bus Operation and Timing

This section covers standard and non-standard bus operations. Standard operations include instruction fetch, load integer, load double integer, load floating-point, load double floating-point, store integer, store double integer, store floating-point, store double floating-point, atomic load-store unsigned byte, and floating-point operations (FPops). Non-standard operations include bus arbitration, cache misses, exceptions, and the reset and error conditions. Coprocessor loads, coprocessor stores, and coprocessor operations are identical in timing to their floating-point counterpart, and are not repeated as a separate case in this section.

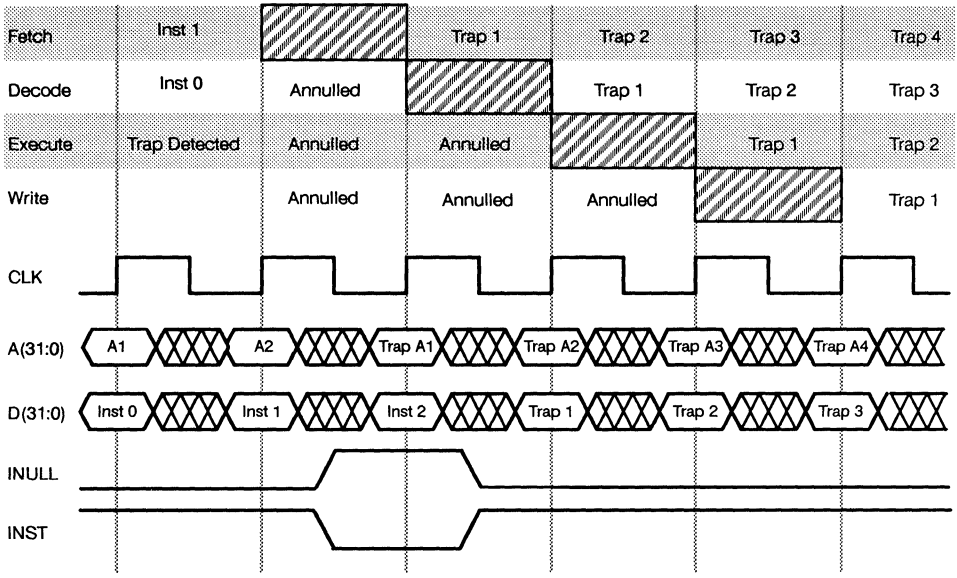


Figure 2–30. Pipeline Operation for Taken Trap (Internal)

Each of the following sections describes a type of bus transaction along with appropriate timing diagrams. The timing diagrams show multiple instructions being fetched for the pipeline. Instruction addresses are sent out in the cycle before the instruction fetch. Instruction fetch cycles begin with the instruction address latched by the memory at the beginning of the fetch cycle and end with the instruction supplied by the memory. Instruction decode begins with the latching of the instruction at rising clock edge of the cycle after the fetch cycle. If the instruction is multicycle, or execution requires an interlock, IOPs are inserted into the pipeline at the decode stage and propagate through the pipeline like a fetched instruction.

The cross-hatched areas shown in the traces are periods in which the signal is not guaranteed to be asserted or deasserted; in other words, undefined.

In general, signals are valid at the beginning of a cycle, i.e., on the rising edge of the clock. In support of the CY7C601/611's high-speed operation, many signals are sent out unlatched. Refer to Section 2.4 for further details on CY7C601/611 signals.

The processor automatically aligns byte (and halfword) transfers as previously shown in *Figure 2–11*. *Figure 2–31* shows the relationship between the data transferred during byte, halfword, and word operations and the pins of the data bus. For byte and halfword data transfers, the CY7C601/611 repeats the byte or halfword on each eight-bit or 16-bit section of the bus. In other words, the undefined portions of the bus illustrated in *Figure 2–31* are actually a repeat of the data driven onto the bus. However, this feature is not specified in the SPARC Architecture Reference, and may not be supported on other SPARC processors.

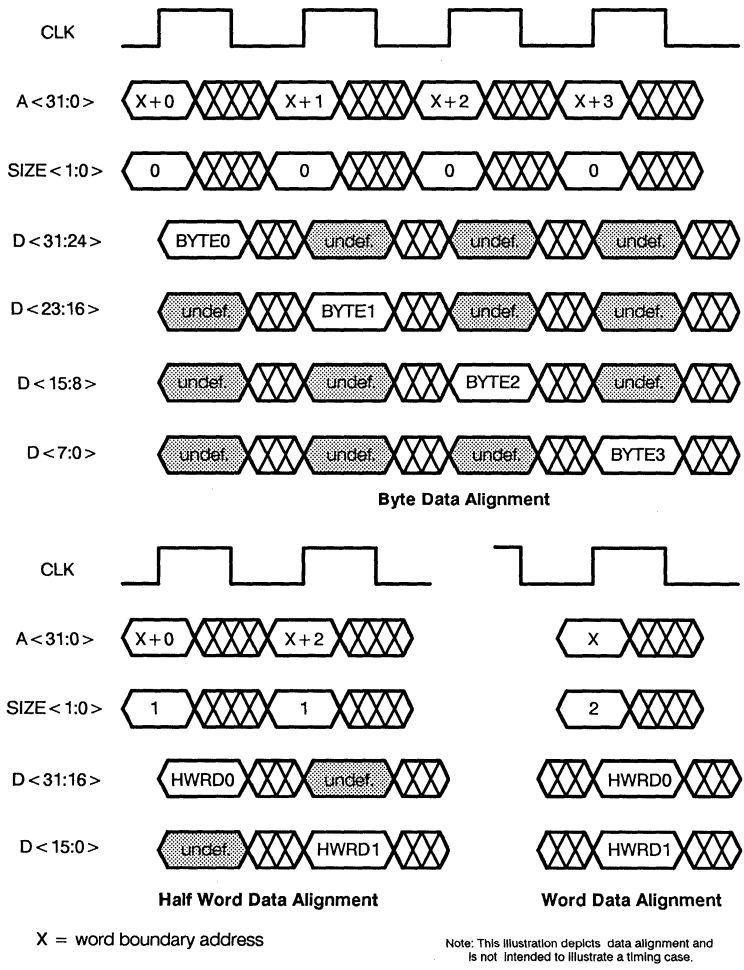


Figure 2–31. Data Bus Contents During Data Transfers

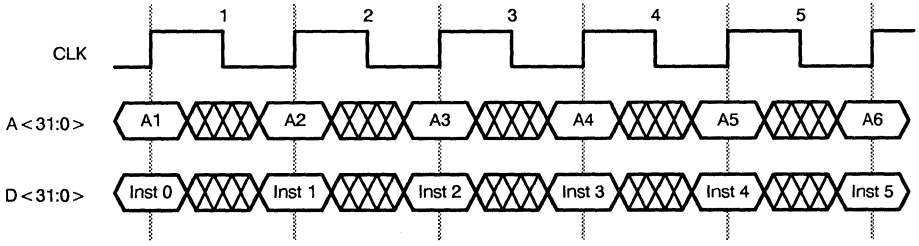


Figure 2–32. Instruction Fetch

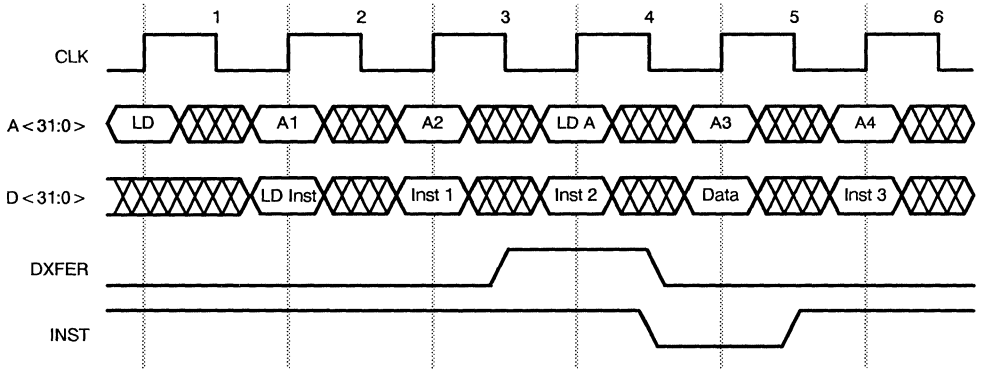


Figure 2–33. Load Single Integer Timing

2.6.1 Instruction Fetch

The instruction fetch cycle is that cycle in which both the instruction address and the data (the instruction itself) are active on their respective buses (see *Figure 2–32*). The instruction address on A < 31:0 > is actually sent out in the previous cycle, but is held into the fetch cycle. It should be latched externally. The instruction is returned on the data bus at the very end of the fetch cycle and is held into the decode cycle. It is latched into the on-chip instruction register at the beginning of the decode cycle.

2.6.2 Load

Figure 2–33 shows the timing for a load single integer instruction. Because the bus is used for a data fetch in the fifth cycle, this is a double-cycle instruction. Note that DXFER is active in the cycle in which the load data address is sent out, while INST is inactive in the cycle in which the load data is on the data bus.

2.6.3 Load with Interlock

In a load with interlock situation, the instruction following the load tries to use the contents of the load's destination register before the load data is available. This requires the insertion of an IOP into the decode stage of the pipeline (see Section 2.5.1.1) in the fourth cycle, which must be matched by a null bus cycle in the fetch stage to balance the pipeline (see *Figure 2–34*).

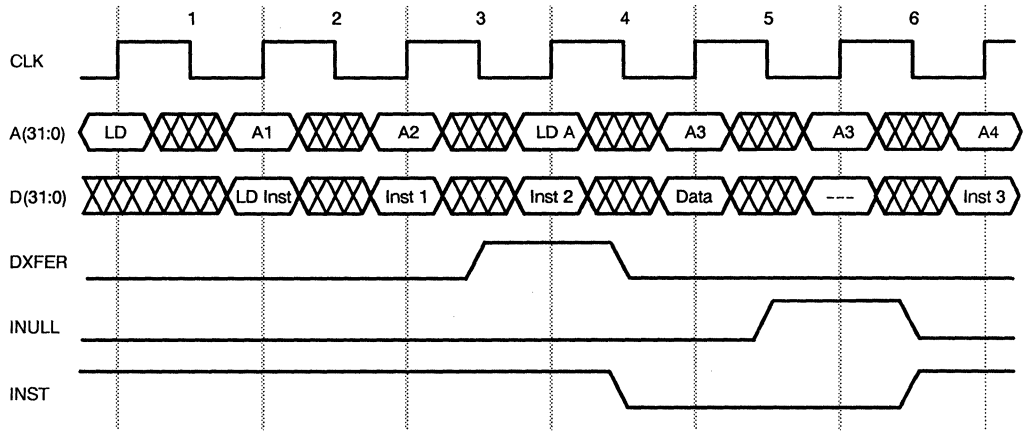


Figure 2–34. Load Single with Interlock Timing

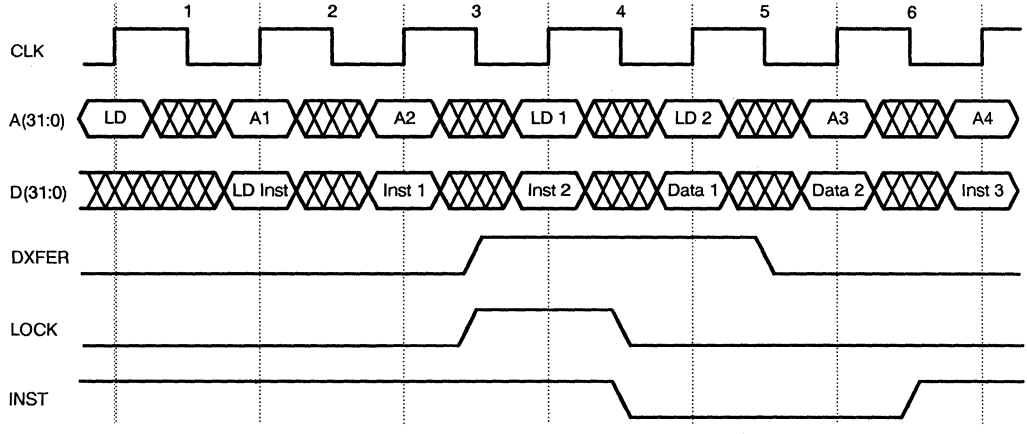


Figure 2–35. Load Double Integer Timing

2.6.4 Load Double

The timing for a load double integer is shown in *Figure 2–35*. The timing is essentially the same as a load single except for the additional data fetch in the fifth cycle. That makes load double a triple-cycle instruction. The most-significant word is fetched in cycle four and the least-significant word in cycle five. Note that the size bits are set to 11 during the address portion of both loads and that the bus is locked to allow the completion of both loads without interruption.

Load single and load double floating-point instructions look identical to their integer counterparts except that the FINS1/FINS2 signal is active for floating-point operations.

2.6.5 Store

Store transactions involve more bus activity than loads, as shown in the store single integer timing in *Figure 2–36*. Store single is a triple-cycle instruction because it includes an extra tag check cycle in which to check an external cache for the store address. This extra cycle also gives the processor and the memory system time to three-state the data bus and turn it around for the store. The store address is sent out again in the fifth cycle to complete the data transfer. Note that the store data is generated by the processor off the falling edge of CLK and is therefore only available at the very end of the first data cycle (see Section 7.1).

Note also that INULL is active during the second application of the store address. If there is a cache miss on the tag check cycle, INULL prevents an additional miss the second time the address is sent out in the store cycle. Because it is a triple-cycle instruction, LOCK is asserted to retain control of the buses.

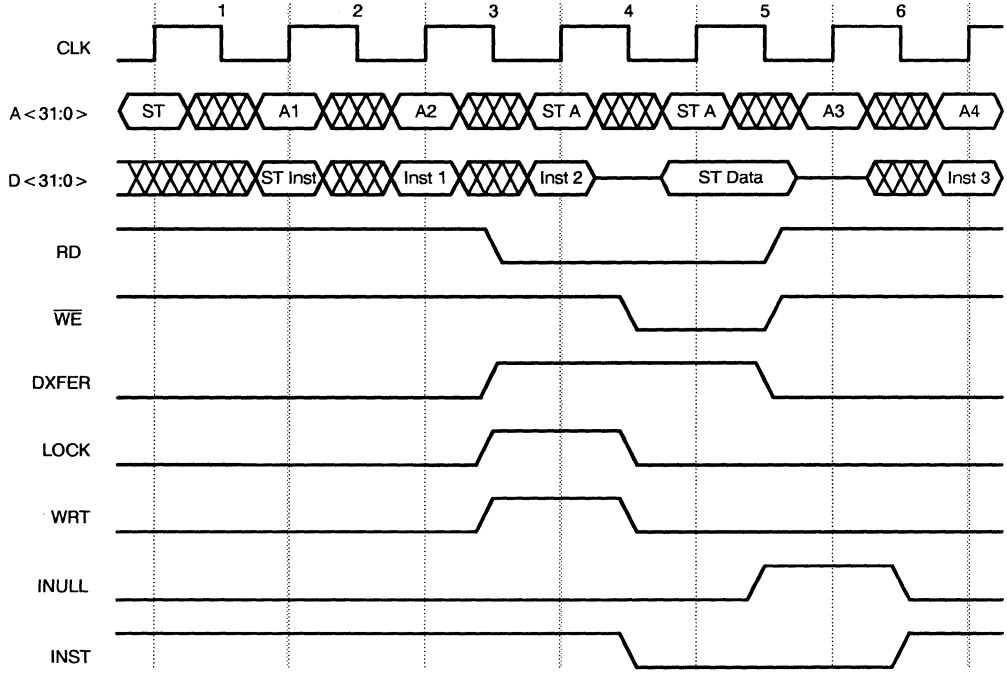


Figure 2–36. Store Single Integer Timing

2.6.6 Store Double

The timing for a store double integer is shown in *Figure 2–37*. The timing is essentially the same as store single except for the additional store cycle in the sixth cycle, making it a four-cycle instruction. The most-significant word is stored in cycle five and the least-significant word in cycle six. Note that the size bits are set to 11 during the address portion of all three data cycles and that the bus is locked to allow the completion of both stores without interruption. INULL is not active for the address of the least-significant store because there cannot be a miss on this cycle if there wasn't one on the tag check cycle, unless the cache line is less than two words.

Store single and store double floating-point instructions look identical to their integer counterparts except that the FINS1/FINS2 signal is active for floating-point operations.

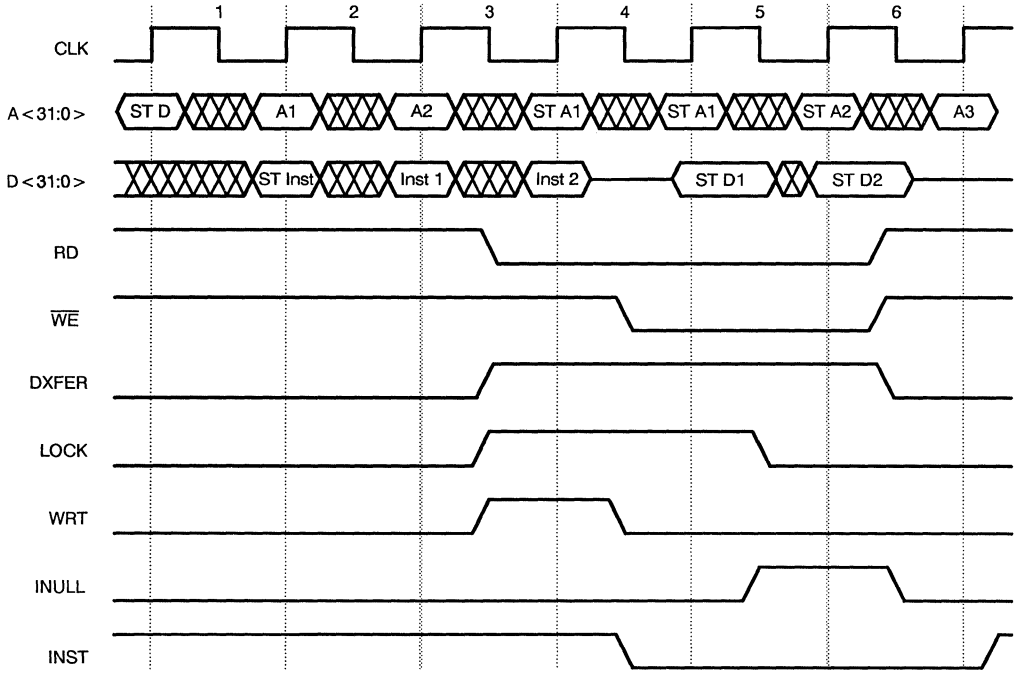


Figure 2–37. Store Double Integer Timing

2.6.7 Atomic Load-Store

Atomic transactions consist of two or more steps which are indivisible; once the sequence begins in the instruction pipeline, it cannot be interrupted. Because atomic operations are four-cycle instructions, the CY7C601/611 asserts LOCK for as long as necessary to make sure that no interruption occurs on the bus. *Figure 2–38* applies to the atomic operations load-store unsigned byte (LDSTUB, LDSTUBA) and word swap (SWAP, SWAPA). Note that, as with any store, INULL is active on the second occurrence of the store address.

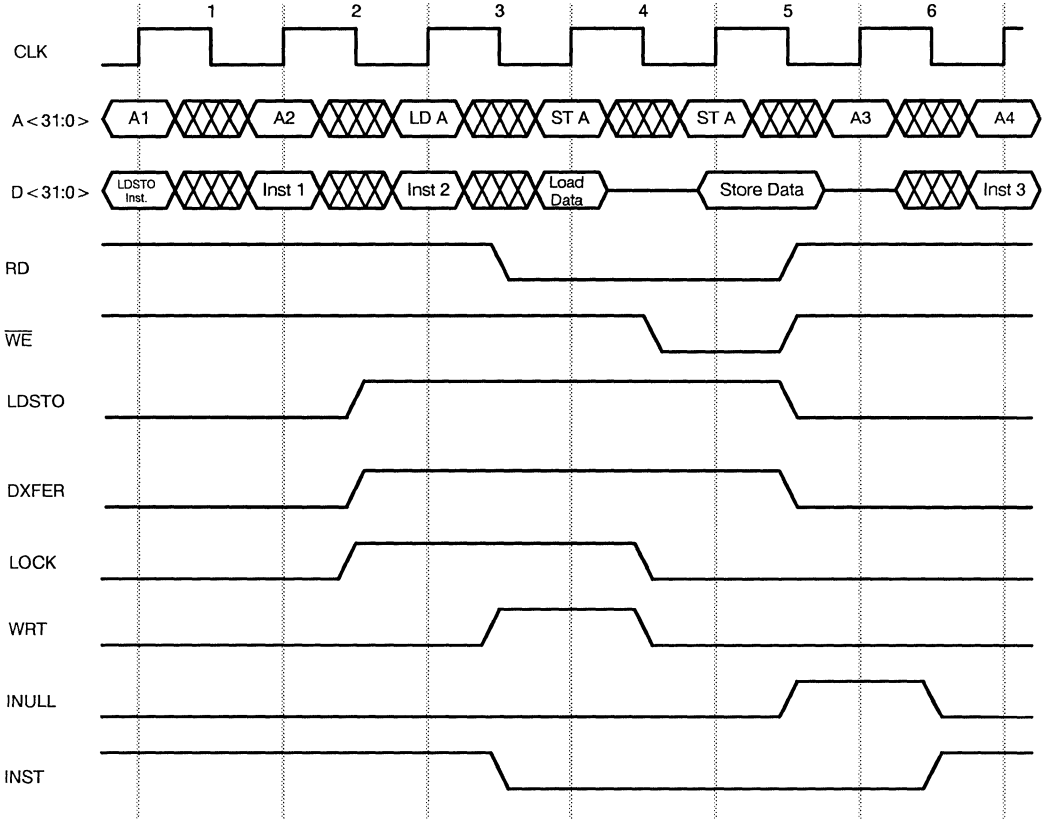


Figure 2–38. Atomic Load-Store Timing

2.6.8 Floating-Point Operations

The timing for floating-point operations and integer operations is the same except for the addition of the FINS1 and FINS2 signals in floating-point operations. In this example, Instruction 1 is a floating–point operation (see *Figure 2–39*). FINS1/2 tell the floating-point unit to move an instruction out of its decode buffer and begin execution. The FPU also makes use of the INST signal to latch instructions into its decode buffers.

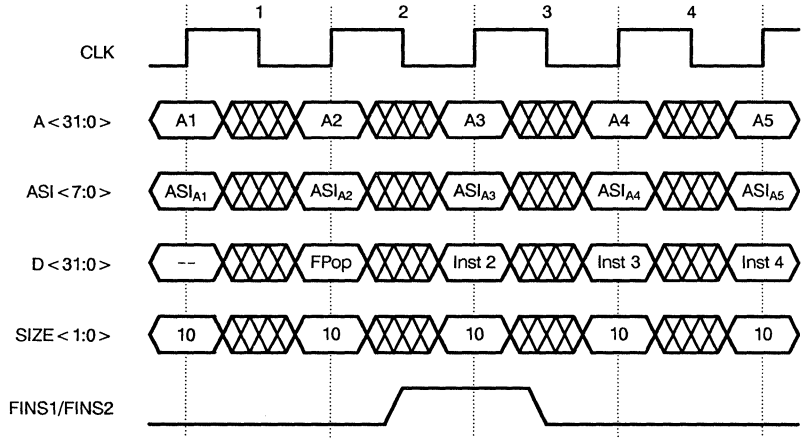


Figure 2–39. Floating-Point Operation Timing

2.6.9 Bus Arbitration

The CY7C601/611 does not have on-chip bus arbitration circuitry because it is designed to operate as a bus slave. Therefore, external circuitry must arbitrate between external bus requests and the CY7C601/611. When the CY7C601/611 needs to retain the buses it asserts the LOCK signal. The arbitration circuitry should assert $\overline{\text{BHOLD}}$ when it needs to keep the CY7C601/611 off the buses. When $\overline{\text{BHOLD}}$ is asserted, the processor's instruction pipeline is frozen until it is deasserted. The arbitration circuitry should also deassert the $\overline{\text{DOE}}$, $\overline{\text{AOE}}$, and $\overline{\text{COE}}$ signals to three-state the CY7C601's address bus, data bus and control signal output drivers so they may be driven by an external source (see *Figure 2–40*).

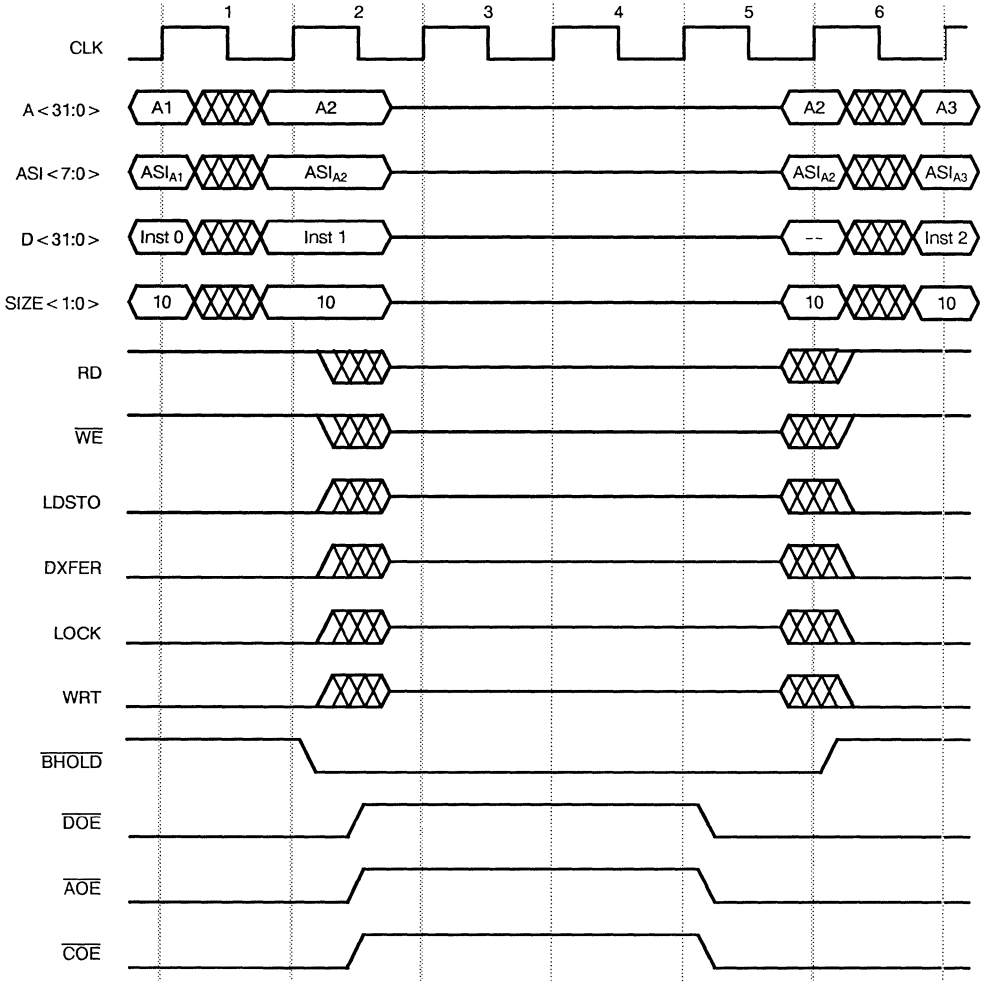


Figure 2–40. Bus Arbitration Timing

2.6.10 Load with Cache Miss

Figure 2–41 gives the timing for a load with cache miss. Cache logic must stop the processor by asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ in the next cycle. However, the processor stops with the address of the next instruction on the address bus rather than the instruction that caused the miss. In order to retrieve the proper load data, the memory system needs the missed address on the bus. To do this the memory system must send an MAO signal, forcing the processor to output the previous address (the address that was on the bus in the cycle before $\overline{\text{MHOLD}}$ was asserted). The $\overline{\text{MHOLD}}$ signal must be maintained while the missed data is strobed into the processor with the $\overline{\text{MDS}}$ signal (it must be strobed externally because the internal processor clock is frozen by the $\overline{\text{MHOLD}}$).

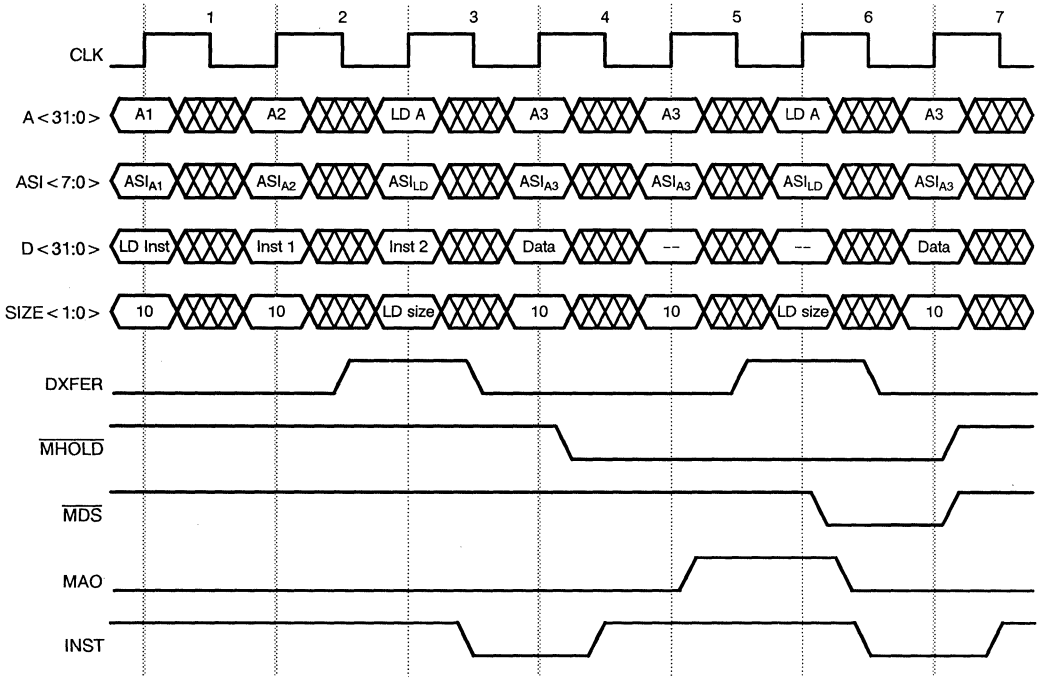


Figure 2–41. Load with Cache Miss Timing

2.6.11 Store with Cache Miss

The timing for a store with cache miss is similar to the load with cache miss situation, except that MAO and $\overline{\text{MDS}}$ are not required (see *Figure 2–42*). Because the processor outputs the store address twice, it already has the proper address on the bus when it's stopped by $\overline{\text{MHOLD}}$. $\overline{\text{MDS}}$ is not required because nothing needs to be strobed into the processor.

INULL is asserted for the second occurrence of the store address so that it doesn't trigger the miss circuitry during the time the cache is processing the miss on the first occurrence of that address.

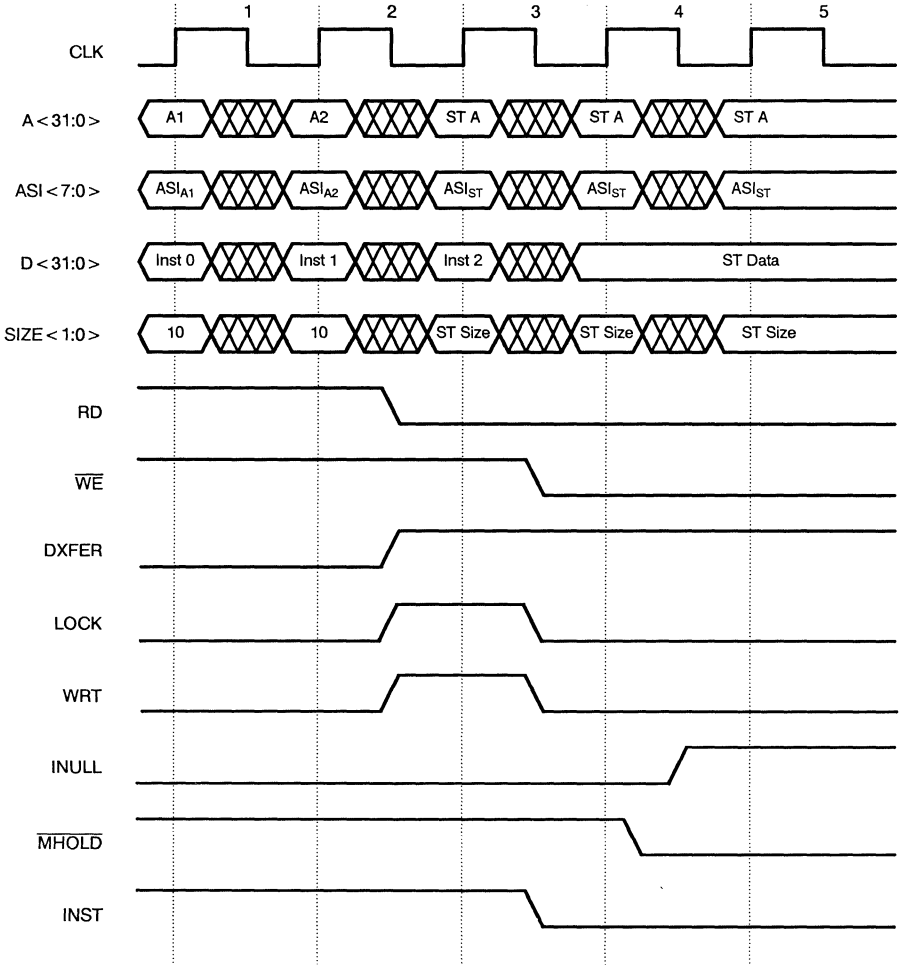


Figure 2–42. Store with Cache Miss Timing (1 of 2)

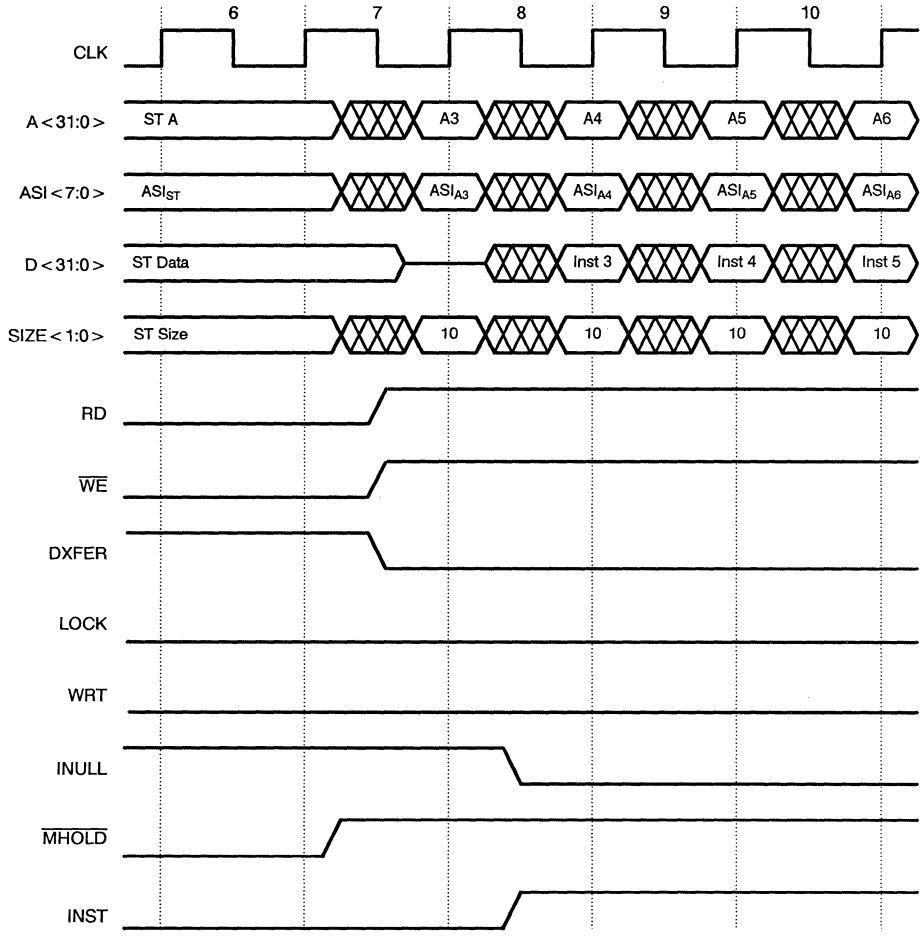


Figure 2–42. Store with Cache Miss Timing (2 of 2)

2.6.12 Memory Exceptions

Load with memory exception timing is shown in *Figure 2–43*. As with a cache miss, memory logic must stop the processor by asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ in the next cycle. The $\overline{\text{MHOLD}}$ signal must be maintained while the memory exception ($\overline{\text{MEXC}}$) signal is strobed into the processor with the $\overline{\text{MDS}}$ signal (it must be strobed in externally because the internal processor clock is frozen by the $\overline{\text{MHOLD}}$). $\overline{\text{MEXC}}$ must be deasserted in the same clock cycle in which $\overline{\text{MHOLD}}$ is deasserted. Note that INULL is asserted in the cycle 8 instruction fetch to annul that fetch. This is the same action shown in cycle 2 of *Figure 2–30* for an internal trap. Store with memory exception has the same timing (see *Figure 2–44*) except INULL is asserted from the second store address through to the annulled cycle 8 instruction fetch.

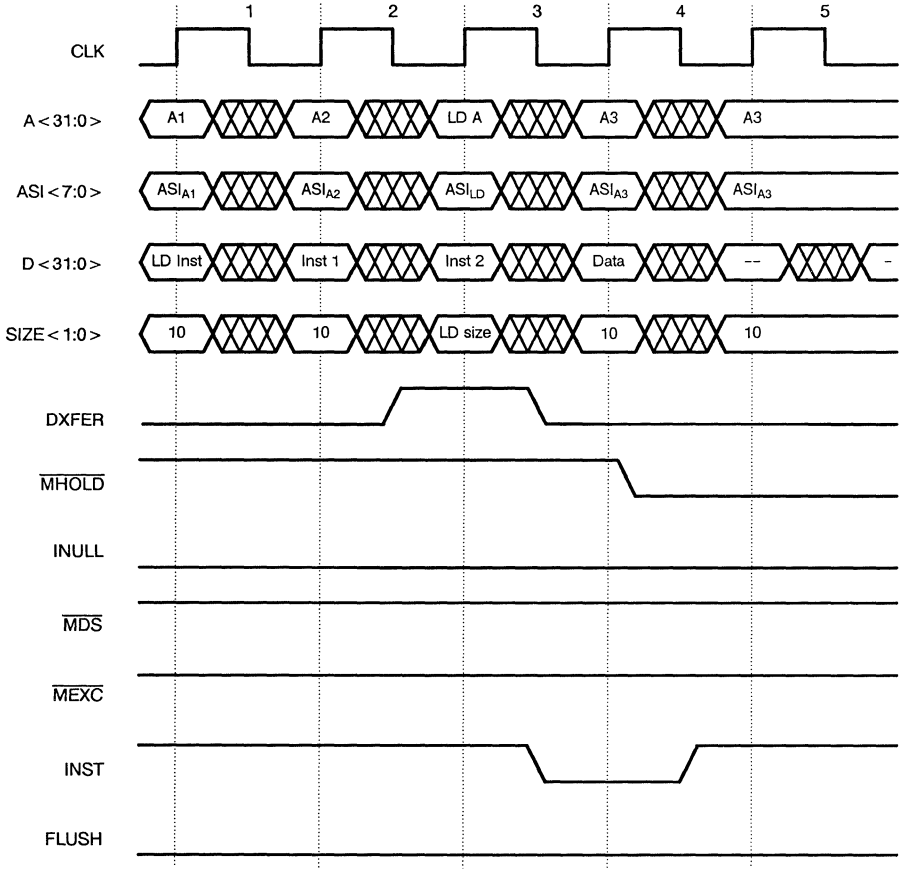


Figure 2–43. Load with Memory Exception Timing (1 of 2)

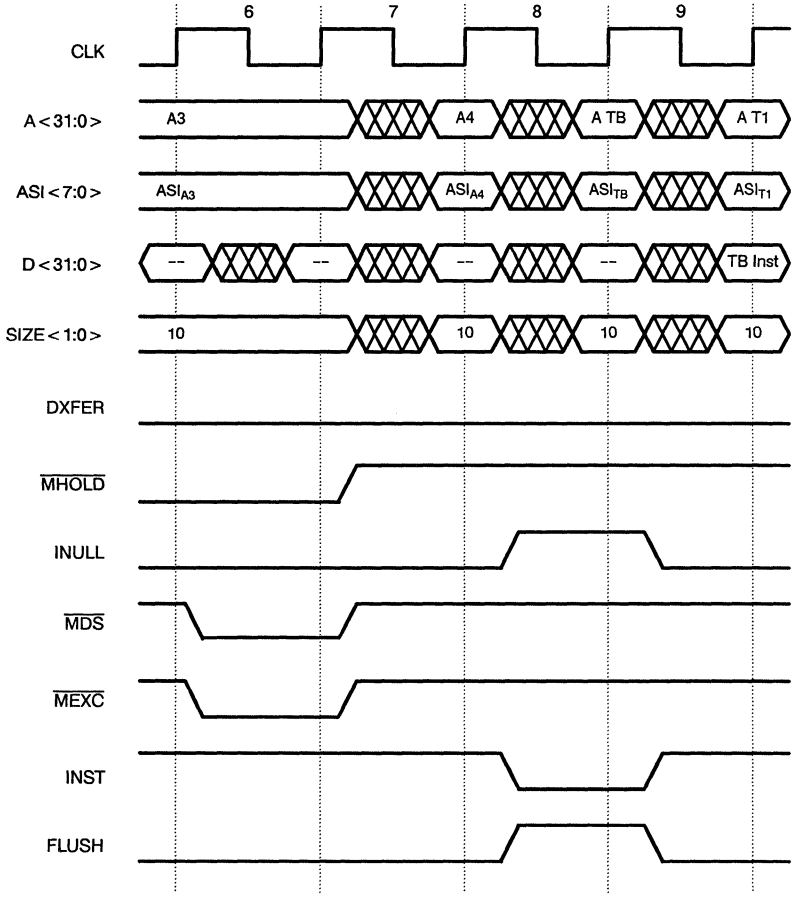


Figure 2–43. Load with Memory Exception Timing (2 of 2)

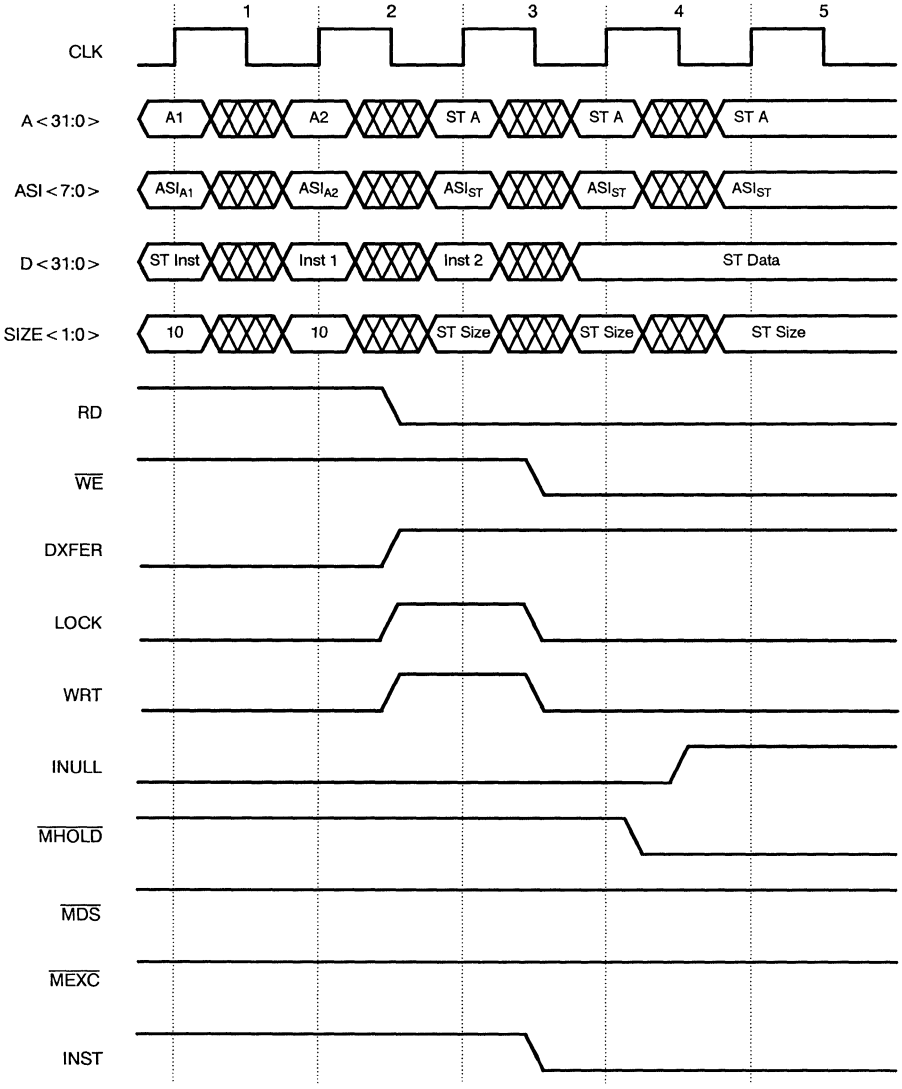


Figure 2–44. Store with Memory Exception Timing (page 1 of 2)

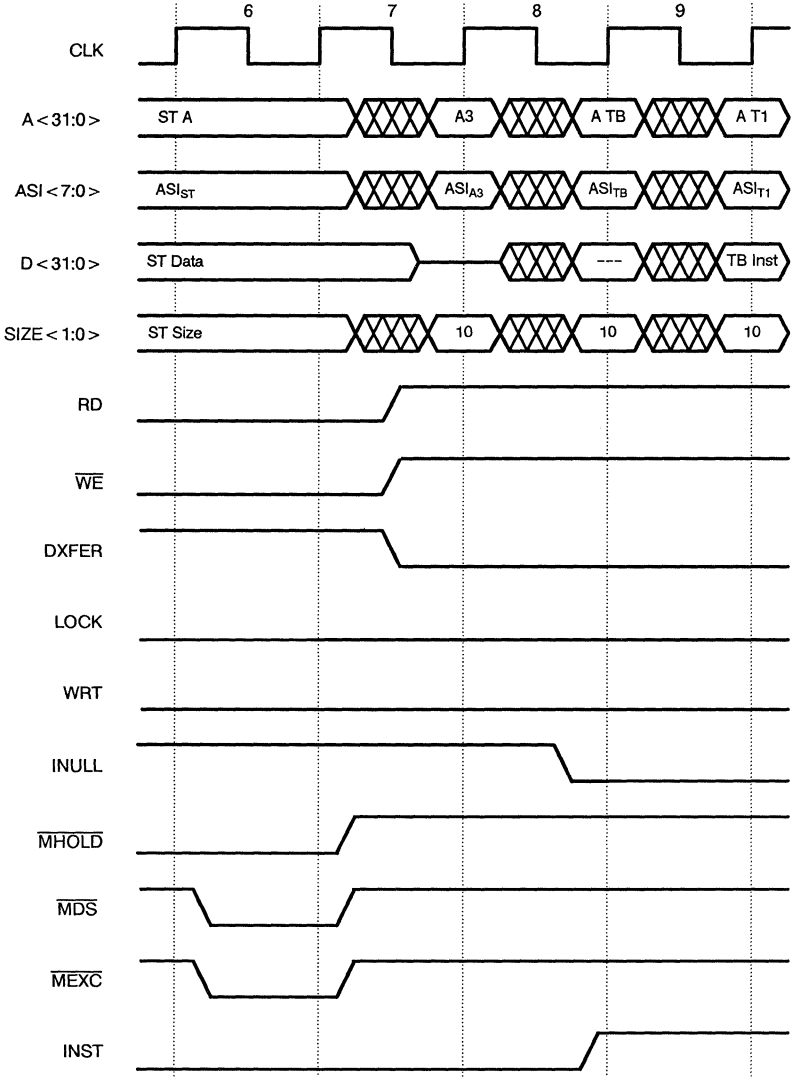


Figure 2–44. Store with Memory Exception Timing (page 2 of 2)

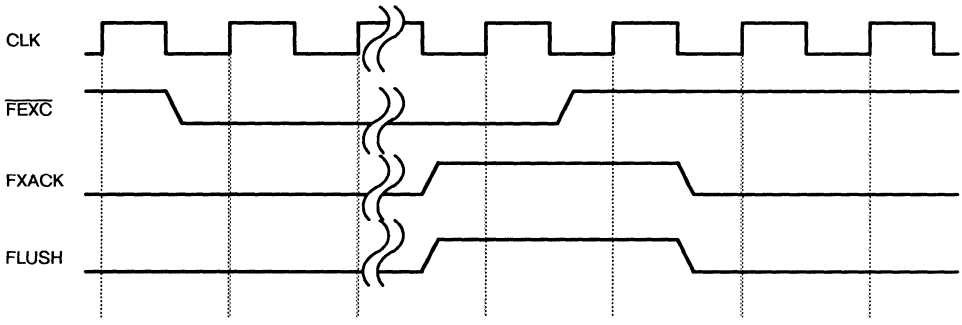


Figure 2–45. Floating-Point Exception Handshake Timing

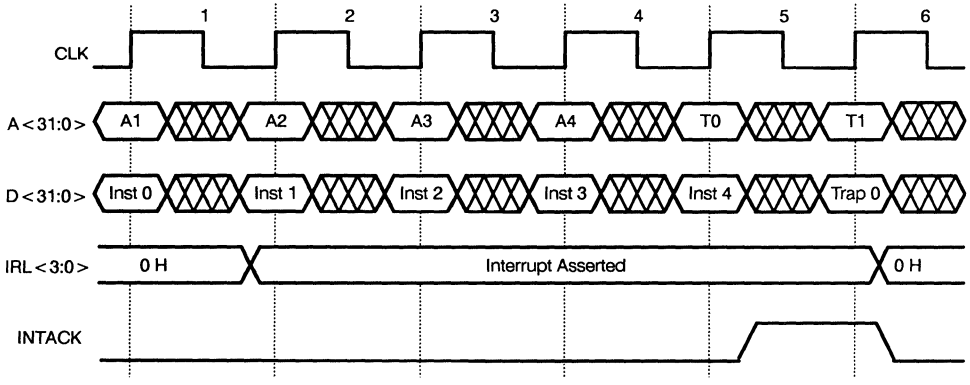


Figure 2–46. Asynchronous Interrupt Timing

2.6.13 Floating-Point Exceptions

The floating-point unit asserts $\overline{\text{FEXC}}$ to notify the CY7C601/611 that a floating-point exception has occurred and that it should take a trap on the next floating-point instruction that it encounters in the instruction stream (see *Figure 2–45*). The CY7C601/611 asserts FXACK to signal the FPU that the trap is being taken, and FLUSH to clean out the FPU's decode buffers. From this point on, the FPU will execute only floating-point store queue instructions until its queue is emptied by the trap handler.

$\overline{\text{FEXC}}$ is deasserted by the FPU after FXACK is asserted. FXACK is deasserted by the CY7C601/611 after $\overline{\text{FEXC}}$ is deasserted.

2.6.14 Interrupts

The asynchronous IRL<3:0> inputs are sampled on the rising edge of every clock. If the interrupt value represented by those inputs is greater than the masking value in the processor, and no higher priority trap supersedes it, the CY7C601/611 will take the interrupt. The IRL input level should be held stable until the processor asserts INTACK. *Figure 2–46* shows the timing for the best case response time where the IRL input value is asserted one clock and a set-up time before the execute stage of a single-cycle instruction. Refer to Section 2.7.3 for more information on interrupts.

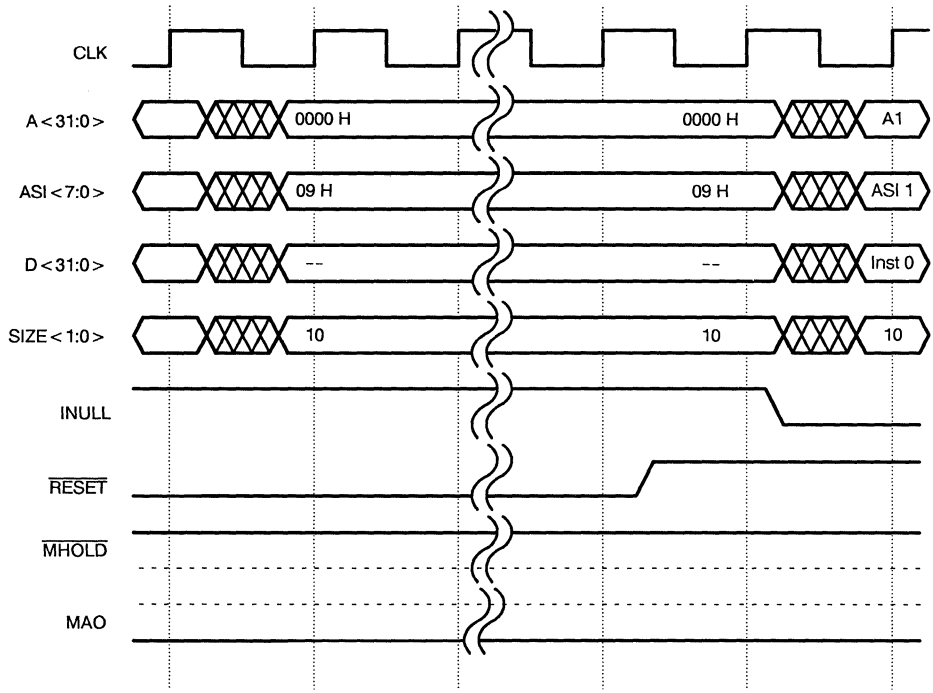


Figure 2–47. Power-On Reset Timing

2.6.15 Reset Condition

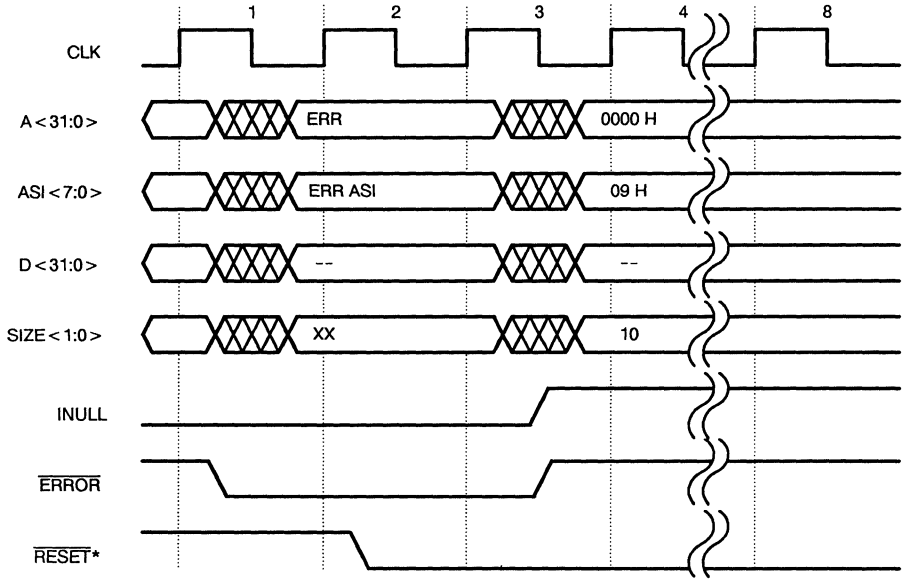
Figure 2–47 shows the timing for a power-on reset. $\overline{\text{RESET}}$ must be asserted for at least eight cycles so that the processor can synchronize the reset input and initialize its internal state. For $\overline{\text{RESET}}$ to be synchronized, the CLK signal must be active.

During the initialization, the processor disables traps (ET=0), sets the supervisor mode (S=1), and sets the program counter to location zero (PC=0, nPC=4).

2.6.16 Error Condition

Error mode is one of the three states in which the CY7C601/611 can exist. To get into the error mode, a synchronous trap must occur while traps are disabled (the processor state register's ET bit is set to zero). This essentially means that a trap which cannot be ignored occurs while another trap is being serviced. In order for that synchronous trap to be serviced, the processor goes through the normal operations of a trap (see Section 2.7), including setting the *tt* bits to identify the trap type. It then enters error mode, halts, and asserts the $\overline{\text{ERROR}}$ signal (see *Figure 2–48*).

The only way to leave error mode is to receive an external $\overline{\text{RESET}}$ signal, which forces the processor into reset mode. All information placed in the CY7C601/611's registers from the last execute mode (the trap operation) remains unchanged and the processor resumes operation at address zero. The reset trap handler can examine the trap type of the synchronous trap and deal with it accordingly.



*RESET must be asserted for a minimum of 8 clocks

(continued)

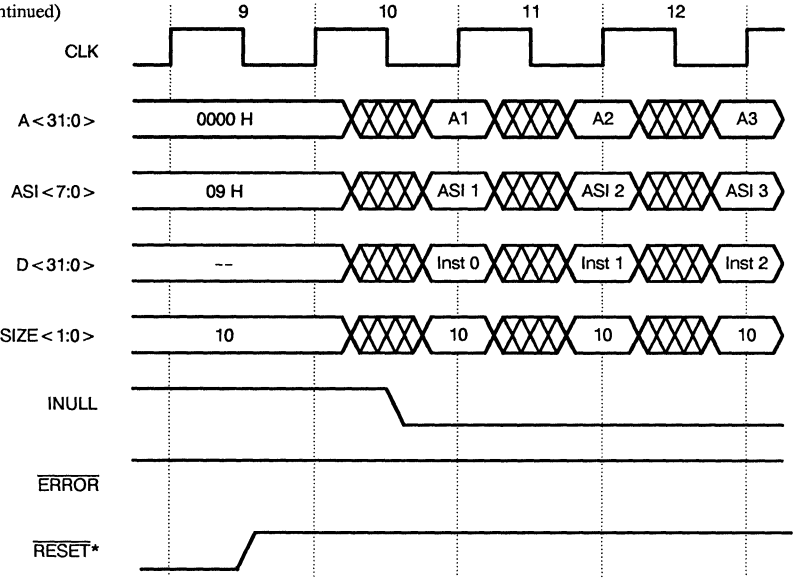


Figure 2–48. Error/Reset Timing

* MAO and MHOLD must be driven to a deasserted state when RESET is asserted.

Table 2–37. Externally Generated Synchronous Exception Traps

Trap	Initiating Signal	Condition
Data Access Exception	$\overline{\text{MEXC}}$	Memory error during data access
Instruction Access Exception	$\overline{\text{MEXC}}$	Memory error during instruction access
Floating-Point Exception	$\overline{\text{FEXC}}$	Floating-point unit error
Coprocessor Exception	$\overline{\text{CEXC}}$	Coprocessor unit error

2.7 Exception Model

The CY7C601/611 supports three types of traps: synchronous, floating-point/coprocessor, and asynchronous (also called interrupts). Synchronous traps are caused by hardware responding to a particular instruction or by the Trap on integer condition code (Ticc) instructions; they occur during the instruction that caused them.

Floating-point/coprocessor traps caused by a Floating-Point-operate (FPop) or CoProcessor-operate (CPop) instruction occur before that instruction is complete. However, because floating-point (and coprocessor) exceptions are pended until the next floating-point (coprocessor) instruction is executed, other non-floating-point (coprocessor) instructions may have executed before the trap is taken. See Section 3.3.3.1.

Asynchronous traps occur when an external event interrupts the processor. They are not related to any particular instruction and occur between the execution of instructions. See Section 2.7.3.

2.7.1 Reset

The reset trap is a special case of the external asynchronous trap type. It is asynchronous because it is triggered by asserting the $\overline{\text{RESET}}$ input signal. But from that point on, its behavior is entirely different from that of an asynchronous interrupt (see Section 2.7.3).

As soon as the CY7C601/611 recognizes the $\overline{\text{RESET}}$ signal, it enters reset mode and stays there until the $\overline{\text{RESET}}$ line is deasserted. The processor then enters execute mode and then the execute trap procedure. Here, it deviates from the normal action of a trap (Section 2.7.5) by modifying the enable traps bit (ET = 0), and the supervisor bit (S = 1). It then sets the PC to 0 (rather than changing the contents of the TBR), the nPC to 4, and transfers control to location 0. *All other PSR fields, and all other registers retain their values from the last execute mode.*

Note: Upon power-up reset the state of all registers other than the PSR are undefined.

If the processor got to reset mode from error mode, then the normal actions of a trap have already been performed, including setting the *tt* field to reflect the cause of the error mode. Because this field is not changed by the reset trap, a post-mortem can be conducted on what caused the error mode. The processor enters error mode whenever a synchronous trap occurs while traps are disabled.

2.7.2 Synchronous Traps

Synchronous traps are caused by the actions of an instruction, with the trap stimulus occurring either internally to the CY7C601/611 or from an external signal which was provoked by the instruction. These traps are taken immediately and the instruction that caused the trap is aborted *before* it changes any state in the processor.

The external signals that can cause a synchronous trap are listed in *Table 2–37.*

2.7.2.1 External Signals

Synchronous traps generated by the input signal $\overline{\text{MEXC}}$ (Memory Exception) occur during the execute phase of an instruction or occur immediately for data accesses. Traps generated by the $\overline{\text{FEXC}}$ and $\overline{\text{CEXC}}$ signals belong to the special floating-point/coprocessor category, and may not occur immediately. See Section 3.3.3.1.

2.7.2.1.1 instruction access exception

An instruction access exception trap is generated if a memory exception occurs (the $\overline{\text{MEXC}}$ input signal is asserted) during an instruction fetch.

2.7.2.1.2 data access exception

A data access exception trap is generated if a memory exception occurs (the $\overline{\text{MEXC}}$ input signal is asserted) during the data cycle of any instruction that moves data to or from memory.

2.7.2.2 Internal/Software

Synchronous traps generated by internal hardware are associated with an instruction. The trap condition is detected during the execute stage of the instruction and the trap is taken immediately, before the instruction can complete.

2.7.2.2.1 illegal instruction

An illegal instruction trap occurs:

- when the UNIMP instruction is encountered,
- when an unimplemented instruction is encountered (excluding FPops and CPops),
- in any of the situations below where the continued execution of an instruction would result in an illegal processor state:
  1. Writing a value to the PSR's CWP field that is greater than the number of implemented windows (with a WRPSR)
  2. Executing an Alternate Space instruction with its *i* bit set to 1
  3. Executing a RETT instruction with traps enabled (ET = 1)
  4. Executing an IFLUSH instruction with $\overline{\text{IFT}}$ = 0

Unimplemented floating-point and unimplemented coprocessor instructions do not generate an illegal instruction trap. They generate fp exception and cp exception traps, respectively.

2.7.2.2.2 privileged instruction

This trap occurs when a privileged instruction is encountered while the PSR's supervisor bit is reset (S = 0).

2.7.2.2.3 fp disabled

A fp disabled trap is generated when an FPop, FBfcc, or floating-point load/store instruction is encountered while the PSR's EF bit = 0, or if no FPU is present ($\overline{\text{FP}}$ input signal = 1).

2.7.2.2.4 cp disabled

A cp disabled trap is generated when a CPop, CBccc, or coprocessor load/store instruction is encountered while the PSR's EC bit = 0, or if no coprocessor is present ($\overline{\text{CP}}$ input signal = 1).

2.7.2.2.5 window overflow

This trap occurs when the continued execution of a SAVE instruction would cause the CWP to point to a window marked invalid in the WIM register.

2.7.2.2.6 window underflow

This trap occurs when the continued execution of a RESTORE instruction would cause the CWP to point to a window marked invalid in the WIM register. The window underflow trap type can also be set in the PSR during a RETT instruction, but the trap taken is a reset. See Section 2.7.1 on reset traps and Chapter 6 for the instruction definition for RETT.

2.7.2.2.7 memory address not aligned

Memory address not aligned trap occurs when a load or store instruction generates a memory address that is not properly aligned for the data type or if a JMPL instruction generates a PC value that is not word aligned (low-order two bits non-zero).

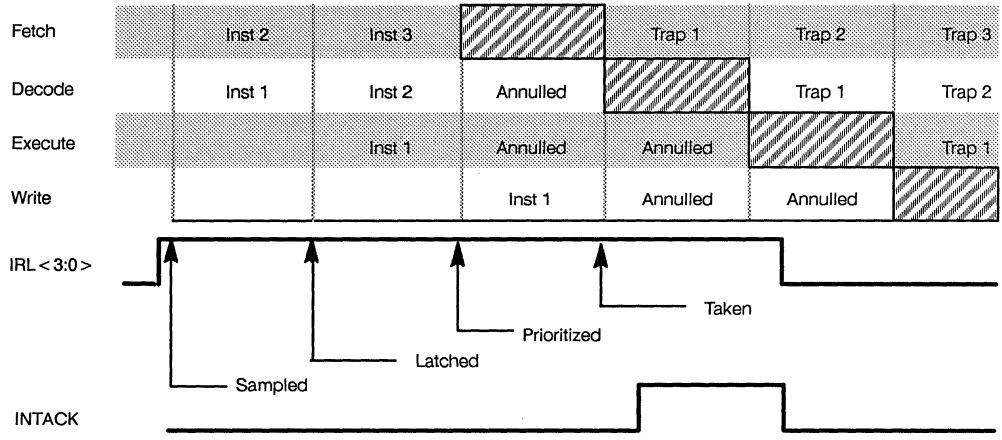


Figure 2–49. Best-Case Interrupt Response Timing

2.7.2.2.8 tag overflow

This trap occurs if execution of a TADDccTV or TSUBccTV instruction causes the overflow bit of the integer condition codes to be set. See the instruction definitions of TADDccTV and TSUBccTV and Section 2.3.3.2.3 for details.

2.7.2.2.9 trap instruction

This trap occurs when a Ticc instruction is executed and the trap conditions are met. There are 128 programmable trap types available within the trap instruction trap (see Chapter 6, Ticc instruction).

2.7.3 Interrupts (Asynchronous Traps)

Asynchronous traps occur in response to the Interrupt Request Level (IRL<3:0>) inputs. This type of trap is not associated with an instruction and is said to happen between instructions. This is because, unlike synchronous traps, an interrupt allows the instruction in whose execute stage it is prioritized to complete execution (see *Figure 2–49*). Any instruction that has entered the pipeline behind the instruction which was allowed to complete is annulled, but can be restarted again after returning from the trap.

2.7.3.1 Priority

The level, or priority, of the interrupt is determined by the value on the IRL<3:0> pins. For the interrupt to be taken, the IRL value must be greater than the value in the Processor Interrupt Level (PIL) field of the Processor State Register (PSR). A value of 0 indicates that no interrupt is requested. A value of 15 represents a non-maskable interrupt. All other IRL values between 0 and 15 represent interrupt requests which can be masked by the PIL field. The priority and trap type (*tt*) for each level is shown in *Table 2–38* in Section 2.7.5.3.

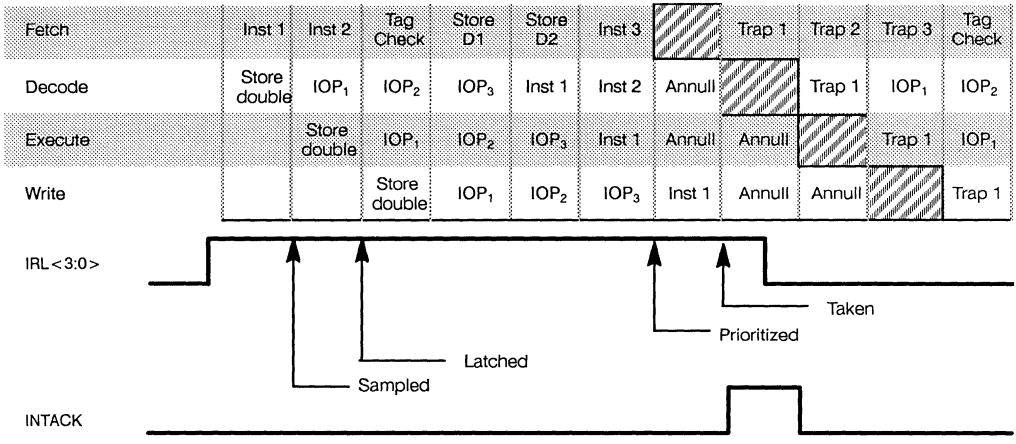


Figure 2–50. Worst-Case Interrupt Response Timing

2.7.3.2 Response Time

The CY7C601/611 samples the IRL inputs at the rising edge of every clock. In order to properly synchronize these asynchronous inputs, they are put through two synchronizing levels of D-type flip-flops. The outputs of the two levels must agree before the interrupt can be processed. If the outputs disagree, the interrupt request is ignored. This logic serves to filter transients on the IRL lines, but it means that the lines must be active for two consecutive clock edges to be accepted as valid.

Once the IRL input has been accepted, it is prioritized and the appropriate trap is taken during the next execute stage of the instruction pipeline. Best case interrupt response occurs when the interrupt is applied one clock plus one setup time before the execute phase of any instruction in the pipeline (see *Figure 2–49*). In this case, the first instruction of the interrupt service routine is fetched during the fourth clock following the application of an IRL value greater than the PIL field of the processor status register (PSR). This also holds for an IRL value of 0F H, which acts as a non-maskable interrupt.

The worst case interrupt response occurs when the detection of the IRL input just misses the cutoff point for the execute stage of a four-cycle instruction, such as a store double or atomic load-store (see *Figure 2–50*). In this case, the interrupt input must wait an additional three cycles for the next pipeline execute phase. In addition, if the IRL input just misses the sampling clock edge, an additional clock delay occurs. As a result, the first instruction of the service routine is fetched in the eighth clock following the application of IRL.

The best and worst case interrupt timing described above assumes that the processor is not stopped via the application of an external hold signal, and that the IRL input is not superceded by the occurrence of a synchronous (internal) trap.

2.7.3.3 Interrupt Acknowledge

As shown in *Figure 2–49*, and more clearly in *Figure 2–50*, the INTerrupt ACKnowledge (INTACK) output signal is asserted when the interrupt is *taken*, not when it is first detected and latched. Because of this delay, if the IRL < 3:0 > inputs are changed to reflect another interrupt condition before the corresponding INTACK for the latched condition is received, there could be some question as to which interrupt the INTACK is responding to. Therefore, external hardware should ensure that the IRL < 3:0 > inputs are held stable until an INTACK is received.

2.7.4 Floating-Point/Coprocessor Traps

Floating-point/coprocessor exception traps are considered a separate class of traps because they are both synchronous and asynchronous. They are asynchronous because they are triggered by an external signal ($\overline{\text{FEXC}}$ or $\overline{\text{CEXC}}$), and are

taken sometime after the floating-point or coprocessor instruction that caused the exception. This can happen because the CY7C601/611 and the FPU (coprocessor) operate concurrently. However, they are also synchronous, because they are tied to an instruction—the next floating-point or coprocessor instruction encountered in the instruction stream after the signal is received.

When the FPU (coprocessor) recognizes an exception condition, it enters an "exception pending mode" state. It remains in this state until the CY7C601/611 signals that it has taken an fp exception (cp exception) trap by sending back an FXACK (CXACK) signal. The FPU (coprocessor) then enters the "exception mode" state, remaining there until the floating-point (coprocessor) queue has been emptied by execution of one or more STDFQ (STDCQ) instructions.

Although the PC will always point to a floating-point or coprocessor instruction after an exception trap is taken, it doesn't point to the instruction that caused the exception. However, the instruction that did cause the exception is always the front entry in the queue at the time the trap is taken, and the entry includes both the instruction and its address. The remaining entries in the queue point to FPops (CPops) that have been started but have not yet completed. Once the queue has been emptied, these can be re-executed or emulated.

2.7.4.1 Floating-Point Exception

This trap occurs when the FPU is in exception pending mode and an FPop, FBfcc, or floating-point load/store instruction is encountered. The type of exception is encoded in the *tt* field of the Floating-point State Register (FSR). See Section 3.3.1.

2.7.4.2 Coprocessor Exception

This trap occurs when the Coprocessor is in exception pending mode and a CPop, CBccc, or coprocessor load/store instruction is encountered. The type of exception should be encoded in the *tt* field of the Coprocessor State Register (CSR). The nature of the exception is implementation dependent.

2.7.5 Trap Operation

Once a trap is taken, the following operations take place:

- Further traps are disabled (asynchronous traps are ignored; synchronous traps force an error mode).
- The S bit of the PSR is copied into the PS bit; the S bit is then set to 1.
- The CWP is decremented by one (modulo the number of windows) to activate a trap window.
- The PC and nPC are saved into r[17] and r[18], respectively, of the trap window.
- The *tt* field of the TBR is set to the appropriate value.
- If the trap is not a reset, the PC is written with the contents of the TBR and the nPC is written with TBR + 4. If the trap is a reset, the PC is set to address zero and the nPC to address four.

Unlike many other processors, the SPARC architecture does not automatically save the PSR into memory during a trap. Instead, it saves the volatile S bit into the PSR itself and the remaining fields are either altered in a reversible manner (ET and CWP), or should not be altered in the trap handler until the PSR has been saved to memory.

2.7.5.1 Recognition

In most cases, traps are "recognized" in the pipeline's execute stage. For a synchronous trap, the trap criteria are examined during the execute stage of an instruction, and the trap is taken immediately, before the write stage of that instruction takes place. This includes the fp disabled and cp disabled trap type. The special cases occur with those traps generated by external signals. A memory exception on an instruction fetch is detected at the beginning of the execute stage of instruction execution. Memory exceptions occurring on data accesses are detected on the rising clock edge of the data cycle.

Because asynchronous traps happen "between" instructions, their timing is slightly different. As long as the ET bit is set to one, the CY7C601/611 checks for interrupts. The interrupt is sampled on a rising clock edge and latched on the next rising clock edge. The processor compares the IRL<3:0> input value against the PIL field of the PSR, and if IRL is greater than PIL, or IRL is 15 (unmaskable), then it is prioritized at the end of the next execute stage of the pipeline. A trap keyed to the IRL level occurs after the write stage completes.

Floating-point/coprocessor exception traps are not recognized when the $\overline{FEXC}$ or $\overline{CEXC}$ signal is first sampled. The processor waits until it encounters a floating-point or coprocessor instruction in the instruction stream and then handles it as if it were an internal synchronous trap.

2.7.5.2 *Trap Addressing*

The Trap Base Register (TBR) is made up of two fields, the Trap Base Address (TBA) and the trap type (*tt*). The TBA contains the most-significant 20 address bits of the trap table, which is in external memory. The trap type field, which was written by the trap, not only uniquely identifies the trap, it also serves as an offset into the trap table when the TBR is written to the PC. The TBR address is the first address of the trap handler. However, because the trap addresses are only separated by four words (the least-significant four bits of TBR are zero), the program must jump from the trap table to the actual address of the particular trap handler.

Of the 256 trap types allowed by the 8-bit *tt* field, half are dedicated to hardware traps (0-127), and half are dedicated to programmer-initiated traps (Ticc). For a Ticc instruction, the processor must calculate the *tt* value from the fields given in the instruction, while the hardware traps can be set from a table such as the one below. See the Ticc instruction definition for details.

The *tt* field remains valid until another trap occurs.

2.7.5.3 *Trap Types and Priority*

Each type of trap is assigned a priority (see *Table 2–38*). When multiple traps occur, the highest priority trap is taken, and lower priority traps are ignored. In this situation, a lower priority trap must either persist or be repeated in order to be recognized and taken.

Table 2–38. Trap Type and Priority Assignments

Trap	Priority	Trap Type (tt)	Synchronous or Asynchronous
Reset	1	–	Async.
Instruction Access	2	1	Sync.
Illegal Instruction	3	2	Sync.
Privleged Instruction	4	3	Sync.
Floating-Point Disabled	5	4	Sync.
Coprocessor Disabled	6	36	Sync.
Window Overflow	7	5	Sync.
Window Underflow	8	6	Sync.
Memory Address not Aligned	9	7	Sync.
Floating-Point Exception	10	8	Sync.
Coprocessor Exception	11	40	Sync.
Data Access Exception	12	9	Sync.
Tag Overflow	13	10	Sync.
Trap Instructions (Ticc)	14	128 - 255	Sync.

Table 2–38. Trap Type and Priority Assignments (continued)

Trap	Priority	Trap Type (tt)	Synchronous or Asynchronous
Interrupt Level 15	15	31	Async.
Interrupt Level 14	16	30	Async.
Interrupt Level 13	17	29	Async.
Interrupt Level 12	18	28	Async.
Interrupt Level 11	19	27	Async.
Interrupt Level 10	20	26	Async.
Interrupt Level 9	21	25	Async.
Interrupt Level 8	22	24	Async.
Interrupt Level 7	23	23	Async.
Interrupt Level 6	24	22	Async.
Interrupt Level 5	25	21	Async.
Interrupt Level 4	26	20	Async.
Interrupt Level 3	27	19	Async.
Interrupt Level 2	28	18	Async.
Interrupt Level 1	29	17	Async.

2.7.5.4 Return From Trap

On returning from a trap with the RETT instruction, the following operations take place:

- The CWP is incremented by one (modulo the number of windows) to re-activate the previous window.
- The return address is calculated
- Trap conditions are checked. If traps have already been enabled (ET = 1), an illegal instruction trap is taken. If traps are still disabled but S = 0, or the new CWP points to an invalid window, or the return address is not properly aligned, then an error mode/reset trap is taken.
- If no traps are taken, then traps are re-enabled (ET = 1).
- The PC is written with the contents of the nPC, and the nPC is written with the return address.
- The PS bit is copied back into the S bit.

The last two instructions of a trap handler should be a JMPL followed by a RETT. This instruction couple causes a non-delayed control transfer back to the trapped instruction or to the instruction following the trapped instruction, whichever is desired. See the RETT instruction definition for details.

2.8 Coprocessor Interface

In the SPARC architecture, the integer unit is the basic processing engine, but provision is made for two coprocessor extensions. The extensions are in the form of instruction set extensions and a pair of identical signal interfaces. In the CY7C601, one of these instruction and signal interface extensions is dedicated to floating-point operations and the other is designated for a second coprocessor, either user defined or some future device offered by Cypress. Although signals and instructions have been named to reflect the assumption of how these two extensions will be used, either instruction set extension/signal interface may be used in any way desired.

The floating-point unit and its interface are described in Chapter 3. This section deals only with the second coprocessor interface.

In order for the CY7C601 to support a user-defined coprocessor, the coprocessor should contain certain elements defined by the SPARC architecture. These include an internal register set, a status register, a coprocessor queue, and a set of compatible interface pins. These elements are identical to the floating-point interface, and it is recommended that a user desiring to use the coprocessor interface thoroughly study the floating-point interface in Chapter 3 as an example of a coprocessor interface application.

2.8.1 Protocol

The coprocessor extensions to the architecture are designed to allow the coprocessor to operate concurrently with the integer unit and the floating-point unit. To keep operations synchronized, address and data buses are shared. The initial CY7C601 instruction decode determines which unit should execute the instruction. The CY7C601 executes its own instructions, but signals the coprocessor to continue the decode and execution if it recognizes a coprocessor instruction. For coprocessor loads and stores, the CY7C601 supplies the memory address and the coprocessor receives or supplies the data. The coprocessor must deal with resource or data dependencies, signaling the problem to the CY7C601 by freezing the instruction pipeline with the $\overline{\text{CHOLD}}$ signal.

The signal interface between the CY7C601 and the coprocessor consists of shared address, data, clock, reset, and control signals, plus a special set of signals that provide synchronization and minimal status information between the coprocessor and the CY7C601.

2.8.1.1 Coprocessor Interface Signals

The SPARC architecture defines two sets of signals intended for interfacing with two coprocessors. The CY7C601 assigns one set of coprocessor signals for specific use by the floating-point unit, and the other set of coprocessor signals for a user-defined coprocessor. All floating-point interface signal names begin with an *F*, and all coprocessor interface signal names begin with a *C*. Both sets of interface signals share the INST signal, which identifies a CY7C601 instruction fetch. The two groups of signals are symmetric, have identical timing requirements, and are listed in *Table 2–33*.

Instruction fetch is signaled by the CY7C601 using the INST signal. The coprocessor uses INST as an input to enable latching of an instruction on the data bus. The coprocessor latches all instructions fetched by the CY7C601, regardless of instruction type. The coprocessor is expected to use a two-stage instruction/address buffer as described in Section 3.2 on the floating-point/integer unit interface. The CY7C601 asserts CINS1 or CINS2 at the beginning of the decode stage of instruction execution of a coprocessor instruction. The CINS1 or CINS2 signals are used to start the execution of a coprocessor instruction and select which of the two most recently fetched instructions stored in the two-stage instruction buffer is to be executed by the coprocessor.

The CY7C601 requires the $\overline{\text{CP}}$ signal to be driven low in order for the integer unit to recognize the presence of a coprocessor. Attempting to execute coprocessor instructions with $\overline{\text{CP}}$ high will cause the CY7C601 to execute a *cp disabled* trap.

Hardware interlocking for coprocessor instruction execution is provided with the $\overline{\text{CHOLD}}$ signal. This signal is asserted by the coprocessor to freeze the CY7C601. This signal is asserted in cases where the CY7C601 must be halted to prevent it from causing a condition from which the coprocessor cannot recover. An example of this would be fetching multiple coprocessor instructions that would otherwise overrun the coprocessor queue. The coprocessor would be expected to assert $\overline{\text{CHOLD}}$ until it could handle additional instructions.

Coprocessor interrupts are asserted with the $\overline{\text{CEXC}}$ signal. This signal is asserted by the coprocessor upon the detection of an exception case. The CY7C601 will continue normal execution until the execution stage of the next coprocessor instruction. At that time, the CY7C601 will acknowledge the interrupt with CXACK, and begin coprocessor trap execution.

Coprocessor branch on condition code (CBcc) instructions are executed by the CY7C601 integer unit based on the value of the CCC < 1:0 > signals supplied by the coprocessor. These signals are typically set by the execution of a coprocessor compare instruction (defined by the designer). The CCCV signal supplied by the coprocessor indicates whether the state of the CCC < 1:0 > signals is valid. CCCV is normally asserted, but is deasserted when a coprocessor compare instruction is executed and remains deasserted until that instruction is completed. The deassertion of this signal causes the CY7C601 to halt execution. This interlock prevents the CY7C601 from branching on invalid condition codes. The SPARC architecture requires at least one non-coprocessor instruction between a coprocessor compare and a coprocessor branch on condition code (CBcc) instruction.

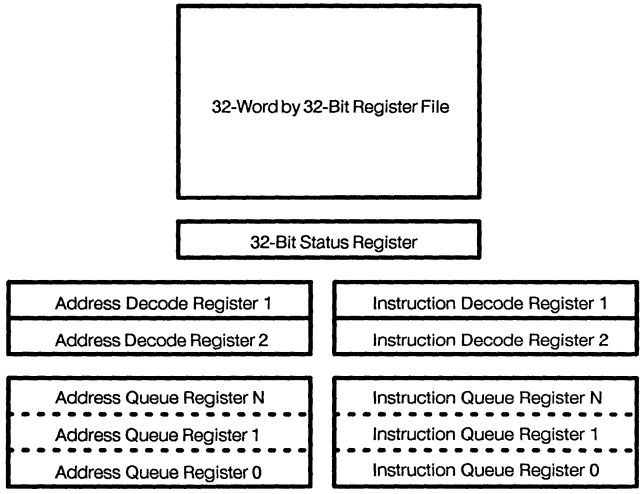


Figure 2–51. Coprocessor Register Model

2.8.2 Register Model

The coprocessor register model specified by the SPARC architecture is shown in *Figure 2–51*. The coprocessor has its own 32 x 32-bit working register file from which all operands for CPop instructions originate and to which all results return. The contents of these registers are transferred to and from memory under control of the CY7C601, using coprocessor load/store instructions.

The Coprocessor State Register (CSR) contains the current status of the coprocessor. The exact nature of the exception bits and trap types are implementation dependent. The CSR is read and written indirectly through memory using the LDCSR and STCSR instructions.

The coprocessor queue is necessary to properly handle traps with concurrently operating units. The first-in, first-out queue records all pending coprocessor instructions and their addresses at the time of a coprocessor exception. The front entry of the queue contains the unfinished instruction that caused the exception. The rest of the queue contains unfinished CPops which would be restarted or emulated after the trap handler returns control to the main program.

The address and instruction decode buffers hold instructions and their addresses until the CY7C601 determines if they belong to the coprocessor. If one of the held instructions belongs to the coprocessor, the CY7C601 sends the appropriate CINS signal to move the instruction into the coprocessor execute stage. The address and a copy of the instruction also move into the queue at this point and remain there until the instruction completes.

When a trap is taken, the CY7C601 asserts the FLUSH signal, causing the coprocessor to dump any instructions in the decode buffers. FLUSH does not affect instructions which are already in the queue.

2.8.3 Exceptions

Exactly what conditions will generate a cp exception trap are implementation dependent. However, most implementations would probably include Unfinished CPop as a condition that would cause an exception.

An Unfinished CPop trap is generated when the coprocessor cannot complete execution because the data has exceeded the capabilities of the coprocessor and/or has generated an inappropriate result.

2.9 CY7C611 Integer Unit for Embedded Control

The CY7C611 is a SPARC Integer Unit designed for embedded control applications. It is a functional equivalent of the CY7C601 with a reduced pin out for lower cost applications. The CY7C611 retains all internal features of the CY7C601, and maintains complete binary code compatibility with all other SPARC processors. The CY7C611 differs from the CY7C601 in that the address bus has been reduced to 24 bits, the ASI signals have been reduced to three bits, and several control signals not required for lower cost systems have been eliminated. The CY7C611 supports the floating-point interface, but does not include the coprocessor interface. The CY7C611 is packaged in a low-cost 160-pin plastic quad flat package (PQFP) and is available in speeds of 25 MHz.

CY7C601 signals not available on the CY7C611 are listed in *Table 2–39* below. The signal summary for the CY7C611 is listed in *Table 2–40*. All CY7C611 signals are identical to their CY7C601 counterparts, and the information regarding the CY7C601 in this chapter is also valid for the CY7C611.

Note that the EC (enable coprocessor) bit of the PSR register for the CY7C611 is permanently forced to zero.

A user-defined coprocessor can be connected to the CY7C611 instead of a floating-point unit, if desired. All floating-point interface signals are identical in function to their coprocessor counterparts. In order to use the floating-point interface to support a user-defined coprocessor, the floating-point instructions must be used to exercise the coprocessor. This will require software remapping of coprocessor instructions. The CY7C601 and CY7C611 do not decode the nine-bit opf field of a floating-point operate instruction. This can be used to map coprocessor instructions to valid and invalid FPop instructions (as specified by the op3 and opf fields of the op code) without causing an invalid FP instruction trap, since the invalid FP instruction must recognized by the floating-point unit.

Table 2–39. Signal Differences Between CY7C601 and CY7C611

CY7C601 Signals Not Available on CY7C611	
A<31:24>	Address bits 31 through 24
$\overline{\text{AOE}}$	Address Output Enable
ASI<7:3>	ASI bits 7 through 3
CCC<1:0>	Coprocessor Condition Codes <1:0>
CCCV	Coprocessor Condition Codes Valid
$\overline{\text{CEXC}}$	Coprocessor Exception
$\overline{\text{CHOLD}}$	Coprocessor Hold
CINS1	Coprocessor Instruction Stage 1
CINS2	Coprocessor Instruction Stage 2
$\overline{\text{COE}}$	Control Output Enable
$\overline{\text{CP}}$	Coprocessor Present
CXACK	Coprocessor Exception Acknowledge
$\overline{\text{DOE}}$	Data Output Enable
DXFER	Data Transfer
$\overline{\text{IFT}}$	Instruction Cache Flush Trap

Table 2–40. CY7C611 Signal Summary

CY7C611 Signal Summary			
Signal Name	**Signal Description**	**Input/Output**	**Active**
A<23:0>	Address Bus	Three-StateOutput	
ASI<2:0>	Address Space Identifier	Three-State Output	
$\overline{\text{BHOLD}}$	Bus Hold	Input	Low
CLK	Clock	Input	
D<31:0>	Data	Three-State Bidir.	
$\overline{\text{ERROR}}$	IU Error Mode	Three-State Output	Low
FCC<1:0>	Floating-Point Condition Codes	Input	
FCCV	Floating-Point Condition Codes Valid	Input	High
$\overline{\text{FEXC}}$	Floating-Point Exception	Input	Low
$\overline{\text{FHOLD}}$	Floating-Point Hold	Input	Low
FINS1	Floating-Point Instruction Stage 1	Three-State Output	High
FINS2	Floating-Point Instruction Stage 2	Three-State Output	High
FLUSH	Flush FP Instruction	Three-State Output	High
$\overline{\text{FP}}$	Floating-Point Present	Input	Low
FPSYN	FP Synonym Mode	Input	High
FXACK	FP Exception Acknowledge	Three-State Output	High
IRL<3:0>	Interrupt Level <3:0>	Input	
INST	Instruction Fetch Cycle	Three-State Output	High
INULL	Instruction Cycle Nullify	Three-State Output	High
INTACK	Interrupt Acknowledge	Three-State Output	High
LDSTO	Atomic Load-Store Operation	Three-State Output	High
LOCK	Multicycle Bus Lock	Three-State Output	High
MAO	Memory Address Output Select	Input	High
$\overline{\text{MDS}}$	Memory Data Strobe	Input	Low
$\overline{\text{MEXC}}$	Memory Exception	Input	Low
$\overline{\text{MHOLDA}}$	Memory Hold A	Input	Low
$\overline{\text{MHOLDB}}$	Memory Hold B	Input	Low
RD	Read	Three-State Output	High
$\overline{\text{RESET}}$	Reset	Input	Low
SIZE<1:0>	Bus Transaction Size	Three-State Output	
$\overline{\text{TOE}}$	Test Output Enable	Input	Low
WRT	Advanced Write	Three-State Output	High
$\overline{\text{WE}}$	Write	Three-State Output	Low

Chapter 3

CY7C602 Floating-Point Unit

The CY7C602 Floating-Point Unit (FPU) is a high-performance, single-chip implementation of the SPARC reference floating-point unit. The CY7C602 FPU is designed to provide execution of single and double-precision floating-point instructions concurrently with execution of integer instructions by the CY7C601 Integer Unit (IU). The CY7C602 is compliant to the ANSI/IEEE–754 floating-point standard.

The CY7C602 provides a 64-bit internal datapath, a 64-bit ALU, and a 64-bit multiply/divide/square-root unit for efficient execution of double-precision floating-point instructions. For efficient data management, the CY7C602 provides thirty-two 32-bit floating-point registers. These 32-bit registers can be concatenated for use as 64-bit registers for double-precision operations. The internal 64-bit architecture of the CY7C602 allows high speed execution of both single- and double-precision operations. The CY7C602 is capable of a peak performance of 6.15 MFLOPS (double-precision) at a clock speed of 40 MHz.

The SPARC floating-point/integer unit interface supports concurrent execution of integer and floating-point instructions. The tightly coupled floating-point/integer unit interface requires the integer unit to provide all addressing and control signals for memory access. All instructions are fetched by the integer unit, and these instructions are simultaneously latched and decoded by both the CY7C601 and CY7C602. Execution of a floating-point instruction is enabled by CY7C601, which signals the CY7C602 to begin execution of the floating-point instruction when that instruction reaches the execute stage of the CY7C601 instruction pipeline. In the case of a floating-point load or store instruction, the CY7C601 executes the FP load or store in conjunction with the CY7C602 by asserting address and control signals for memory access while the CY7C602 loads or stores the data. All other floating-point instructions execute independently of the integer unit and in parallel with integer instruction execution.

The floating-point/integer unit interface provides hardware interlocking to ensure synchronization between the CY7C601 and CY7C602. Hardware interlocking ensures software compatibility among SPARC systems with different levels of floating-point performance.

3.1 CY7C602 Functional Description

Figure 3–1 illustrates the functional block diagram for the CY7C602. The fetch unit captures instructions and their addresses from the D(31:0) and A(31:0) buses. The decode unit contains logic to decode the floating-point instruction opcodes. The execution unit handles all instruction execution. The execution unit includes a floating-point queue (FP queue), which contains stored floating-point operate (FPop) instructions (see Section 3.3.2) under execution and their addresses. The execution unit controls the load unit, the store unit, and the datapath unit.

The load unit holds data that is fetched from memory via the data bus before it is written into the register file. The register file contains the 32 f registers. The exceptions/floating-point status register (FSR) unit keeps the status of completing FPops, as well as the operating mode of the CY7C602. The store unit holds data that is supplied to the data bus during a store operation. The dependency checking unit checks for conditions where the FPU must freeze the CY7C601 integer unit pipeline so that an incoming instruction does not overflow the floating-point queue (described below). The datapath unit contains arithmetic logic used by FPops to operate on the data in the register file and is comprised of a 64-bit ALU and a 64-bit multiply/divide/square-root/compare unit. *Figure 3–2* gives a more detailed block diagram of the CY7C602.

The CY7C602 provides three types of registers: *f* registers, FSR, and the FP queue. The *f* registers are the thirty-two floating-point operand registers, each 32-bits in size. Adjacent even-odd *f* register pairs (for instance, *freg0* and *freg1*) can be concatenated to support double-precision operands. The FSR is a 32-bit status and control register. It keeps track of rounding modes, floating-point trap types, queue status, condition codes, and various IEEE exception information. The floating-point queue contains the floating-point instructions currently under execution, along with their corresponding addresses. The floating-point queue provides an efficient method of handling floating-point exceptions. When an FPop instruction causes a floating-point exception, the queue contains the offending instruction/address pair along with any other instructions that have started execution. The CY7C601 integer unit acknowledges the floating-point exception, enters a floating-point trap routine, empties the queue, and corrects the exception case. After the exception case is cor-

rected, unfinished floating-point instructions found in the floating-point queue are either executed or emulated in the trap handler before returning to normal execution.

The CY7C602 depends upon the CY7C601 to assert all addresses and control signals for memory access. Floating-point loads and stores are executed in conjunction with the CY7C601, which provides addresses and control signals while the CY7C602 supplies or stores the data. Instruction fetch for integer and floating-point instructions is provided by the CY7C601. When the CY7C601 integer unit asserts an address for an instruction fetch, it asserts the INST signal one clock later. The CY7C602 floating-point unit uses INST to determine when a valid instruction is present on the D(31:0) bus. The instruction, which appears on the data bus on the next clock cycle, is latched and paired with its corresponding address (refer to *Figure 3–3*). In any given cycle, the two previous instruction/address pairs are stored by the CY7C602, regardless of whether the instruction is an integer or floating-point instruction. Either of these two instruction/address pairs may be selected for execution by the CY7C601 upon asserting the FINS1 or FINS2 signal. The CY7C601/CY7C602 interface uses this two stage address/ instruction buffer to accommodate delays in the instruction pipeline of the CY7C601 integer unit. The FINS1 or FINS2 signals select between the output of the two stages of the address/instruction buffer, enabling a floating-point instruction to begin execution by the CY7C602.

Upon decoding a floating-point instruction, the CY7C601 will assert the FINS1 or the FINS2 signal to enable the CY7C602 to begin execution. The FINS1 or FINS2 signal is asserted during the decode stage of the floating-point instruction, and is recognized by the CY7C602 at the beginning of the execute stage of the floating-point instruction. This ensures synchronization of the decode and execute stages of a floating-point instruction between instruction pipelines of the CY7C601 and the CY7C602.

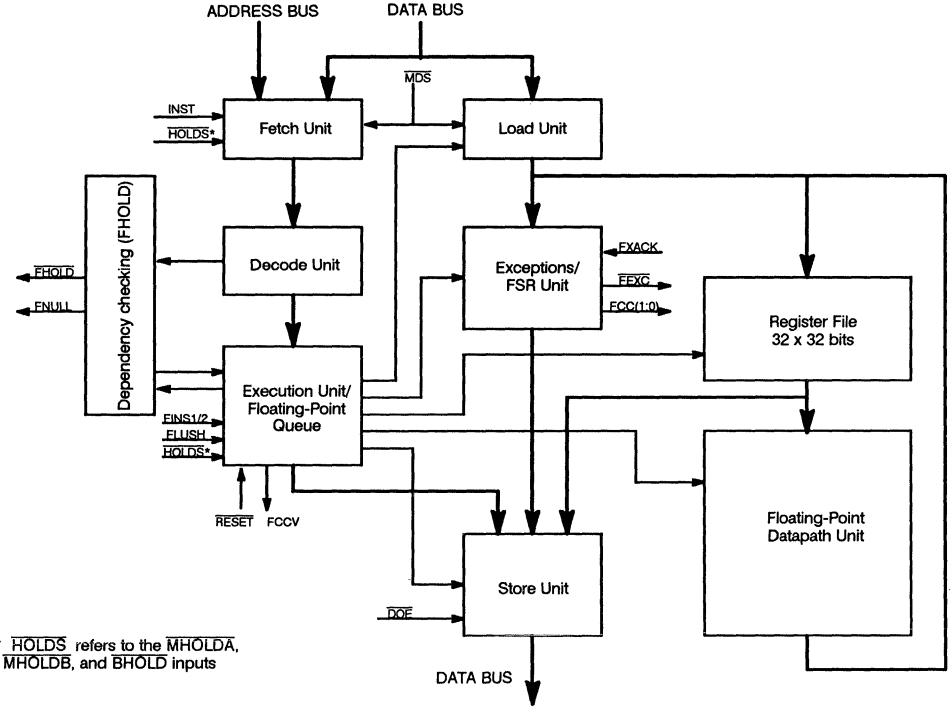


Figure 3–1. CY7C602 Functional Block Diagram

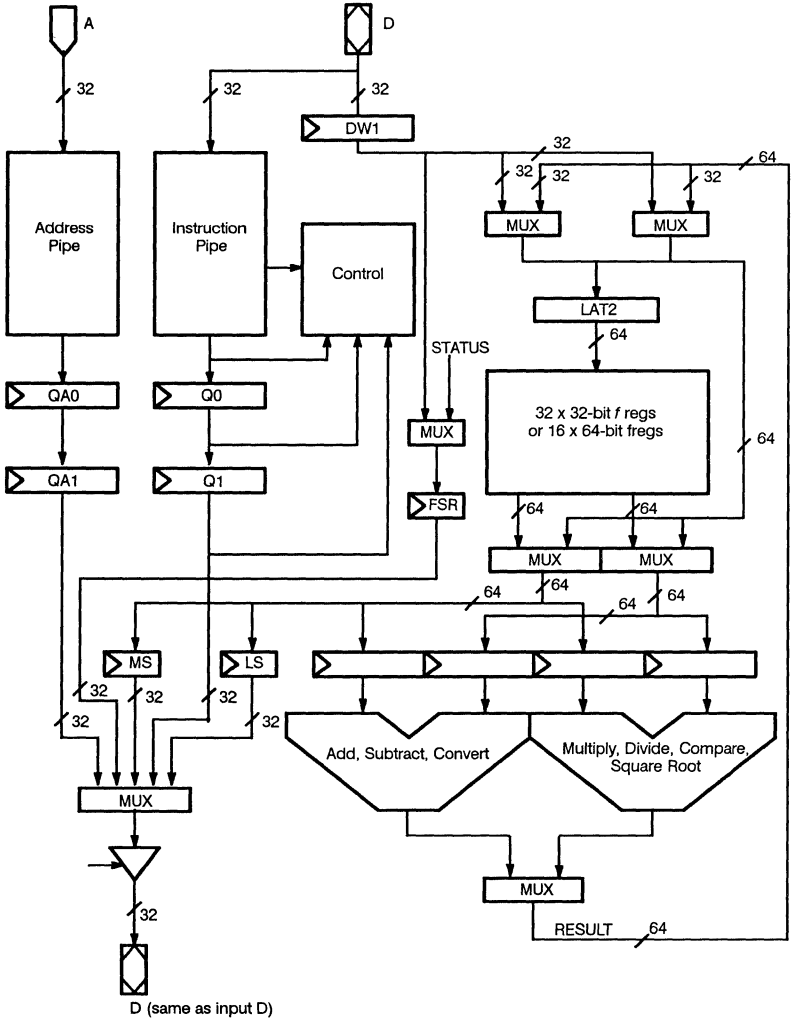


Figure 3–2. CY7C602 Block Diagram

3.2 Floating-Point/Integer Unit Interface

The CY7C602 is designed to directly interface with the CY7C601 without external glue logic. *Figure 3–4* illustrates the signals required to interconnect the CY7C601 and CY7C602. The control signals illustrated in *Figure 3–4* are used to interface with the remainder of the CPU system components. The FNULL, $\overline{\text{RESET}}$, $\overline{\text{BHOLD}}$, $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$, $\overline{\text{MDS}}$, and $\overline{\text{DOE}}$ signals are used by the CY7C604 or CY7C605 for cache interface and virtual bus arbitration. The signal descriptions for the CY7C602 signals are described in Section 3.4.

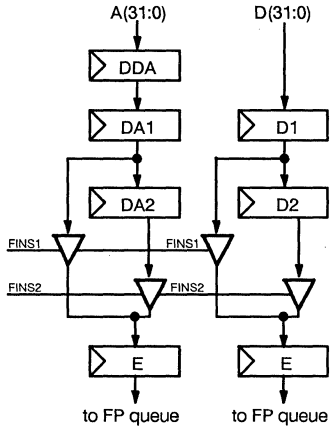


Figure 3–3. CY7C602 Address/Instruction Pipe

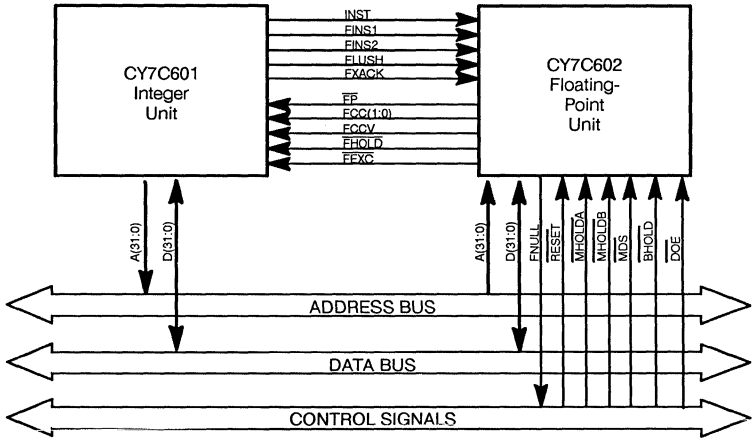


Figure 3–4. CY7C601 – CY7C602 Hardware Interface

3.2.1 CY7C602 Instruction Fetch and Execution

The CY7C602 uses a four-stage instruction pipeline consisting of fetch, decode, execute, and write stages (F, D, E, and W). The instruction pipelines for the CY7C601/611 and the CY7C602 are concurrent and synchronized; a floating-point instruction will be in the same stage in both processors. Multiple cycle instructions such as floating-point operate instructions (FPops) leave the pipeline after the W stage and enter the FP queue until completion.

Addresses for both integer unit and floating-point unit instructions are supplied by the CY7C601. The CY7C602 FPU latches all instructions and the corresponding addresses from the D(31:0) and A(31:0) buses. The CY7C602 uses the INST signal, supplied by the CY7C601, to identify an instruction fetch by the integer unit.

Decode of the latched instruction occurs on the next clock cycle, with both the IU and the FPU decoding the instruction simultaneously. During the decode stage of the floating-point instruction, the FPU checks for operand and resource dependencies. When the CY7C601 integer unit decodes a FPop, it asserts the FINS1 or FINS2 signal. This occurs before the end of the decode stage, and is used by the CY7C602 to initiate the execution of a floating-point instruction. If the CY7C602 has detected an operand or resource dependency during the decode stage, the FPU will assert $\overline{\text{FHOLD}}$ as the instruction begins the execution stage. This freezes the integer unit's pipeline until the FPU can resolve the dependency.

If no resource or operand dependencies exist, the decoded floating-point instruction begins execution. Instructions entering execution are stored in the FP queue, where they are held until execution is completed. Note that if the FP queue is full during an instruction's decode stage, the CY7C602 asserts $\overline{\text{FHOLD}}$ as the instruction enters the execution stage in order to halt the CY7C601. $\overline{\text{FHOLD}}$ is released when space becomes available in the FP queue.

The following tables describe the execution phases of CY7C602 instructions. Additional cycles beyond the F, D, E, and W stages are denoted as Wh (Write hold). Wh stages are equivalent to the additional cycles held by IOPs in the CY7C601/611.

Table 3–1. Load instruction execution

Cycle	Action
D stage	Decode instruction, check operand dependencies
E stage	FHOLD if necessary
W stage	Capture data from D(31:0) bus (LDF, LDFSR), capture MSW from D(31:0) bus (LDDF).
Wh1 stage	Write data into register FSR (LDF, LDFSR), capture LSW from D(31:0) bus (LDDF)
Wh2 stage	Write data into register (LDDF)

Table 3–2. Store instruction execution

Cycle	Action
D stage	Decode instruction, check operand dependencies
E stage	FHOLD if necessary, read data from FSR register or FP queue
W stage (mid-cycle)	Drive data onto D(31:0) bus (STF, STFSR), drive MSW or FP queue address onto D(31:0) bus (STDF, STDFQ)
Wh1 stage (mid-cycle)	Stop driving D(31:0) bus (STF, STFSR), drive LSW or FP queue opcode onto D(31:0) bus (STDF, STDFQ)
Wh2 stage (mid-cycle)	Stop driving D(31:0) bus

Table 3–3. FPop execution

Cycle	Action
D stage	Decode FPop, check resource and operand dependencies
E stage	FHOLD if necessary, read operand(s) from register file
W stage	Read any additional operands from register file; start computing results
FP Queue • • •	Compute, FPop in queue • • •
FP Queue	Check exception status
FP Queue	Update FSR, write results or signal FP exception trap if necessary

3.2.1.1 Instruction Fetch

As the CY7C601 fetches an instruction, the CY7C602 captures it at the same time from the D(31:0) bus. The address corresponding to this instruction is captured from the A(31:0) in the previous cycle. The INST signal is used to determine when a valid instruction is present on the D(31:0) bus, and when a valid address has been fetched from the A(31:0) bus in the previous cycle. *Figure 3–5* illustrates an example of an instruction fetch with a cache hit. The transactions on the address and data buses show two instruction fetches followed by a data fetch.

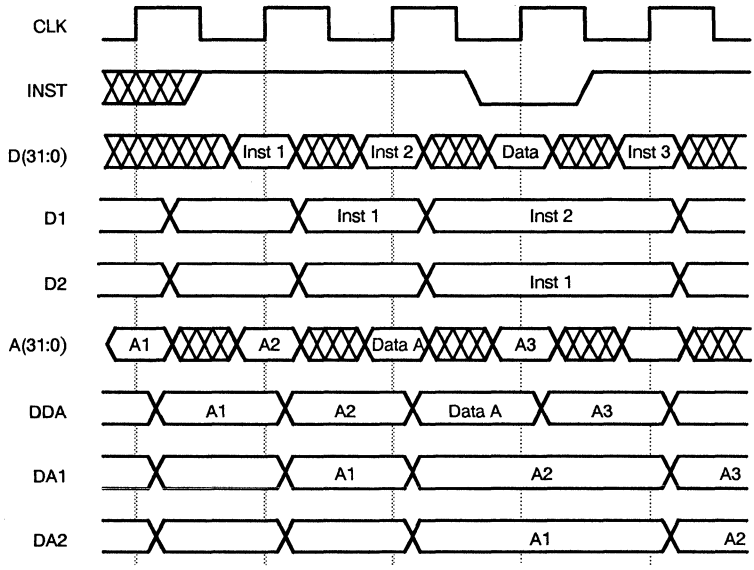


Figure 3–5. Instruction Fetch (Cache Hit)

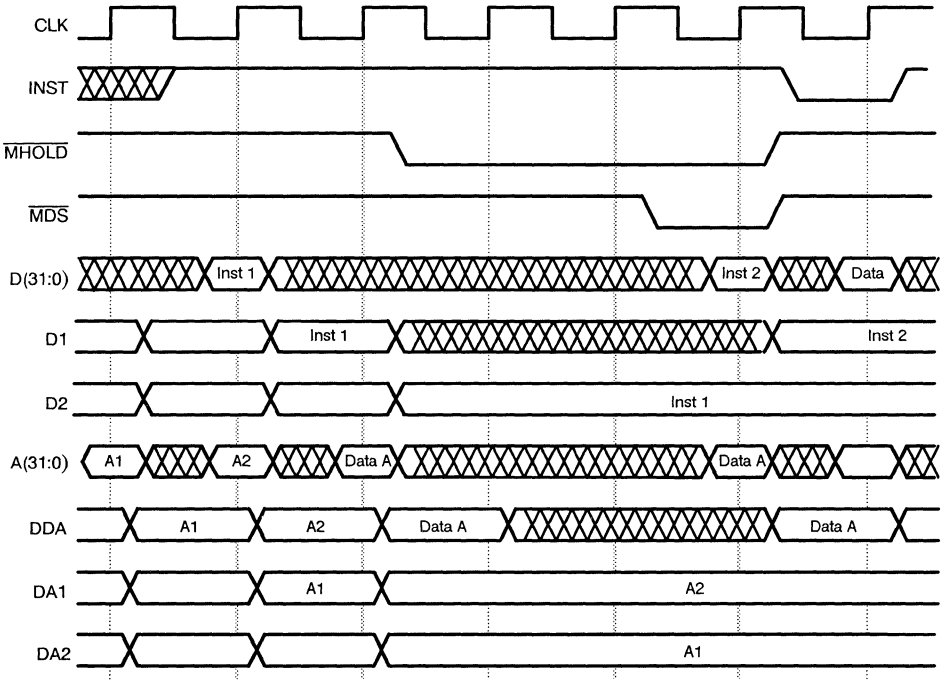


Figure 3–6. Instruction Fetch (Cache Miss on A2)

In the case of an instruction cache miss, a memory hold signal ($\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, or $\overline{\text{BHOLD}}$) is driven low by the cache system starting in the cycle following the instruction fetch. The instruction which was captured from the D(31:0) bus is invalid and is replaced when the system returns a valid instruction on the D(31:0) bus. The hold signal lasts for several cycles during which time the $\overline{\text{MDS}}$ signal is asserted by the cache system, notifying the CY7C602 that the valid instruction is available on the D(31:0) bus. $\overline{\text{MDS}}$ is also used when there is a cache miss on data (via load instructions) so the instruction is reloaded only if INST was asserted in the previous non-hold cycle. The same sequence of transactions in *Figure 3–5* are used in *Figure 3–6*, except that the second instruction fetch (Inst 2) experiences a cache miss.

3.2.1.2 Instruction Execution

The FINS1 and FINS2 signals notify the CY7C602 when to launch a floating-point instruction. When FINS1/FINS2 is received, the floating-point instruction is in the D stage of the CY7C601 integer unit pipeline. The example in *Figure 3–7* shows a situation where both FINS1 and FINS2 are used. A load instruction is immediately followed by two FPops. The FPops are fetched while the load instruction is executing. Because the load takes more than one cycle to execute, the starting of the FPops are deferred, and thus two instructions are held in the instruction buffers of the CY7C602. When the CY7C601 reaches the D stage of the first FPop (Inst 2), it issues FINS2 to start the FPop. When the D stage of the second FPop (Inst 3) is reached, FINS1 is issued to start the second FPop.

FINS1 and FINS2 are never asserted in the same cycle. Both FINS1 and FINS2 are ignored in the following conditions:

1. FLUSH is asserted.
2. $\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, $\overline{\text{BHOLD}}$,$\overline{\text{CHOLD}}$, or $\overline{\text{FHOLD}}$ is asserted.
3. FCCV or CCCV is deasserted.

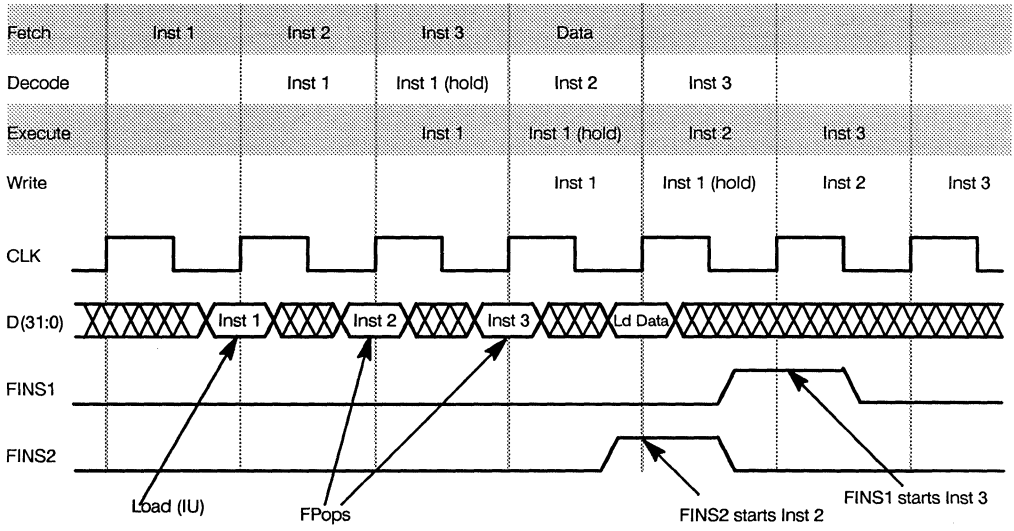


Figure 3–7. Floating-Point Instruction Dispatching

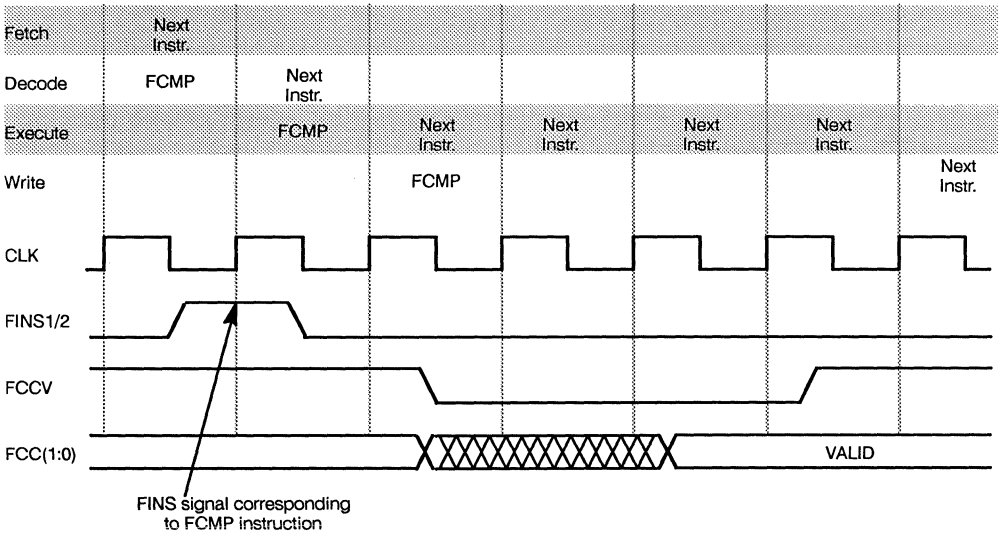


Figure 3–8. Floating-Point Compare (FCMP) Execution

3.2.1.2.1 Floating-Point Compare Execution

Floating-point compare instructions cause the instruction pipeline to be frozen by the use of FCCV, starting from the E stage of the instruction following the compare instruction until the FCC condition codes become valid. FCCV is deasserted, causing the CY7C601/611 to halt execution until FCCV is asserted. *Figure 3–8* illustrates the timing of FCCV relative to the FCMP instruction and the FCC condition codes.

FCCV is deasserted in the W stage of the FCMP instruction. The instruction that immediately follows the FCMP is held in its E stage until FCCV is reasserted. FCC(1:0) is valid one cycle before FCCV is reasserted. For unimplemented compare instructions, the CY7C602 freezes the instruction pipeline and causes an unimplemented FPop trap, which the CY7C601 takes immediately.

3.2.1.2.2 FPop Queuing

When a FPop has passed the first cycle of the W stage and FLUSH has not been asserted, the FPop enters the FP queue. Note that the W stage of an FPop may be extended to more than one cycle if a hold condition exists. As an FPop completes execution successfully and results are written to the register file, it is removed from the FP queue. The front entry of the FP queue contains the instruction/address pair of the oldest FPop which is still being executed by the CY7C602.

3.2.2 Instruction Pipeline Flush

When a trap or interrupt occurs in the integer unit, normal program execution is halted and control is transferred to the trap handler. The instruction in the E stage of the pipeline and any instructions fetched after it are aborted and must be restarted after the trap handler is done (or emulated in the trap handler). Instructions that have not yet been transferred to the FP queue are aborted by the CY7C602 when the trap occurs. The CY7C601 asserts the FLUSH signal in the W stage of the instruction to be aborted (refer to *Figure 3–9*). FPops which were issued before this instruction continue execution (and are in the queue) while instructions issued after it are aborted.

The following figures illustrate how each type of floating-point instruction is affected by the FLUSH signal. *Figure 3–10* illustrates the effect of the FLUSH signal during a load floating-point instruction (LDF). A FLUSH signal asserted anytime on or before the last Wh stage of a load instruction causes the load to abort, leaving the contents of the floating-point register file unchanged.

Figure 3–11 illustrates the effect of FLUSH on a store floating-point instruction (STF). A FLUSH signal asserted on or before the last Wh stage of a store instruction causes the store to abort and the CY7C602 to stop driving the D(31:0) bus by the middle of the next clock cycle.

Figure 3–12 illustrates the effect of FLUSH on a FPop instruction. A FLUSH signal asserted anytime on or before the W stage of a FPop instruction causes the FPop to abort, leaving the contents of the register file and the FSR unchanged by that instruction. FPops that have passed the W stage but are still executing (stored in the FP queue) are not affected.

Figure 3–13 illustrates the effect of FLUSH on a floating-point compare. FLUSH asserted in the W stage of a FCMP instruction causes the FCMP to abort, leaving the FSR unchanged by that instruction. FCCV is reasserted in the next clock cycle.

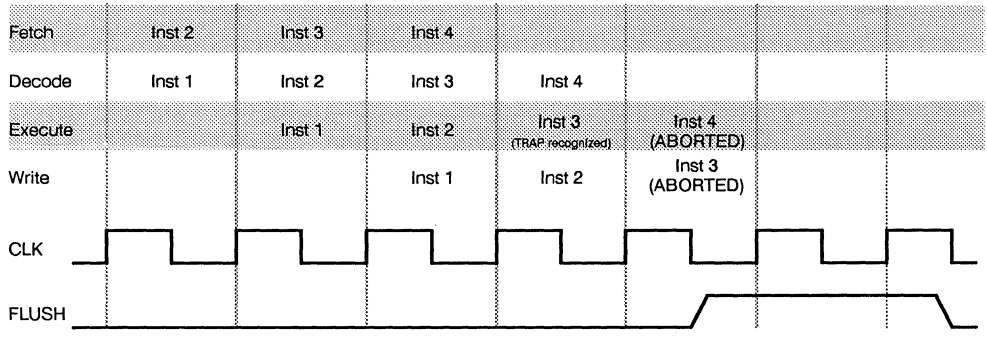


Figure 3–9. Floating-Point Instruction Pipeline During A Trap

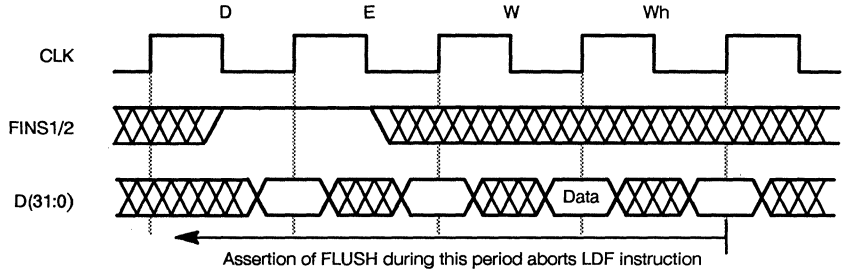


Figure 3–10. Effect of FLUSH on LDF Instruction

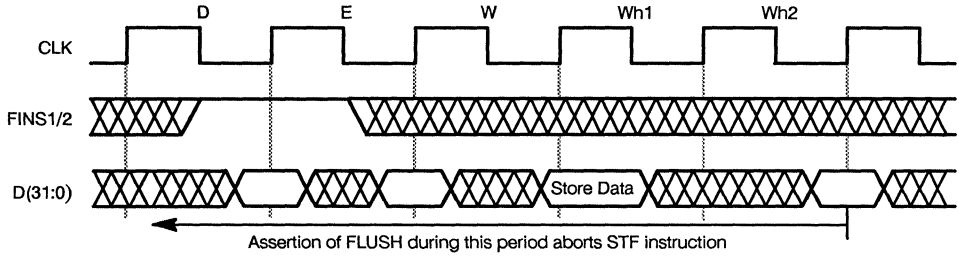


Figure 3–11. Effect of FLUSH on STF Instruction

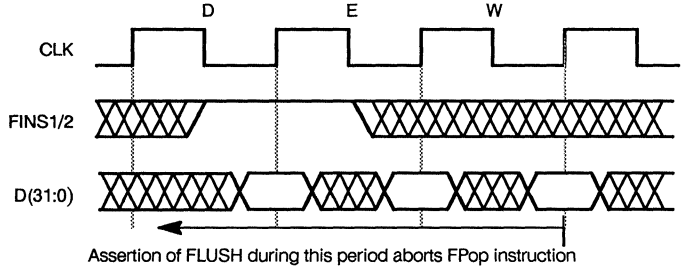


Figure 3–12. Effect of FLUSH on FPop Instruction

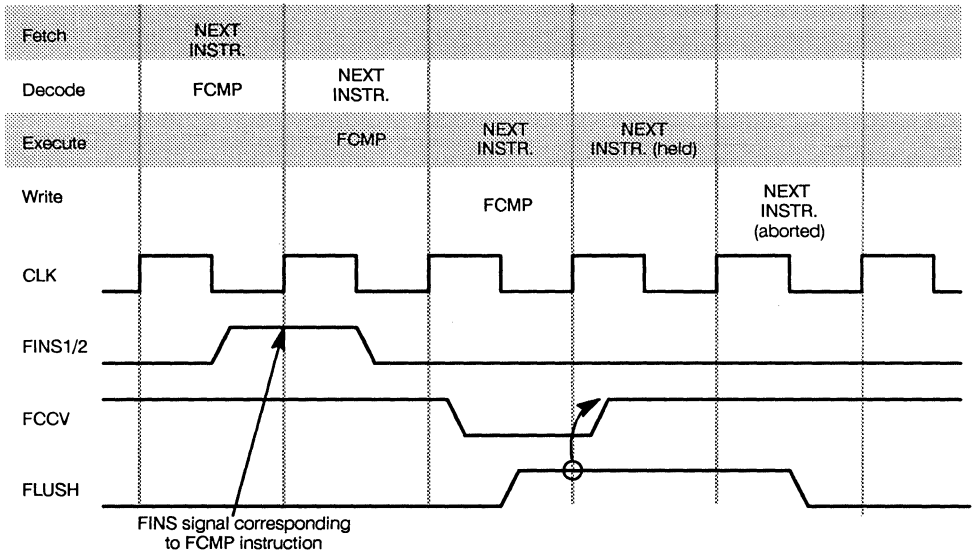


Figure 3–13. Effect of FLUSH on FCMP Instruction

3.2.2.1 Hold Signals

If $\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, $\overline{\text{BHOLD}}$, $\overline{\text{CHOLD}}$, or $\overline{\text{FHOLD}}$ is active, or FCCV or CCCV is inactive, the instruction pipelines of the CY7C601 and CY7C602 are frozen. $\overline{\text{FHOLD}}$ and FCCV are generated by the CY7C602, $\overline{\text{CHOLD}}$ and CCCV are generated by the coprocessor, and the others are generated by the system.

In the CY7C602, "freezing" or "holding" the instruction pipeline means that instructions that are still being tracked by the CY7C601 are not allowed to continue executing. The instructions are allowed to continue execution when all of the hold signals are inactive and all of the condition code valid signals are active. Holds affect all load/store instructions, and only FPops which are in the F, D, and E stages of the instruction pipeline. Hold signals do not affect the execution of FPops in the FP queue.

3.2.2.2 Interlocking with $\overline{\text{FHOLD}}$

In some situations it is necessary to stop the CY7C601 pipeline, either because a FP load/store instruction must be suspended due to an operand dependency, or because the CY7C602 cannot accept any more instructions due to a resource dependency. $\overline{\text{FHOLD}}$ is used to freeze the instruction pipeline in these cases. *Table 3–4* describes mandatory conditions under which $\overline{\text{FHOLD}}$ is asserted.

Operand dependencies listed in *Table 3–4* apply to all FPops that are defined in the architecture. For example, suppose an unimplemented FPop is in the FP queue, waiting to cause an exception. If a store instruction is issued to the CY7C602 to store the contents of the unimplemented FPop's destination register, the store instruction must cause a $\overline{\text{FHOLD}}$ so that the wrong data is not stored. The unimplemented FPop eventually causes a trap that is taken by the CY7C601 in the E stage of the store instruction.

The following simplification could be applied when handling all unimplemented FPops: when an unimplemented FPop has been issued to the CY7C602 but has not yet caused a trap, assert $\overline{\text{FHOLD}}$ on the next floating-point instruction issued until $\overline{\text{FEXC}}$ is asserted. There is no loss in performance because any FPops entering the FP queue after the unimplemented FPop would be re-executed after the unimplemented FPop has been taken care of in the trap handler.

Table 3–4. $\overline{\text{FHOLD}}$ Resource/Operand Dependency Cases

Resource Dependencies:		
If the CY7C602 will not have FP queue entries available to accommodate additional FPops, the CY7C602 asserts $\overline{\text{FHOLD}}$ to stop the CY7C601 from issuing any more instructions to the CY7C602.		
Operand Dependencies:		
LDF, LDDF	*Load data from memory to f register*	Load instructions must not overwrite the source or destination registers of any FPop that has not completed execution. In other words, the rd field of the load instruction must not refer to the same *f* register as any valid rs1, rs2 or rd field of an outstanding FPop. The source registers of FPops (rs1, rs2) may not be altered because an FP exception trap would require that the source registers be unaltered for the trap handler.
STF, STDF	*Store data from f register to memory*	If a store instruction accesses an *f* register that is the destination register of an FPop that has not yet finished execution, the store instruction waits until all outstanding FPops with that register as a destination are complete.
LDFSR, STFSR	*Load/store data between memory and floating-point status register*	If any instructions are currently executing in the CY7C602 when a LDFSR/STFSR instruction is issued by the CY7C601, the CY7C602 holds until all instructions have completed execution and are no longer in the FP queue.

If the CY7C602 goes into exception mode, $\overline{\text{FHOLD}}$ is deasserted. If there is a floating-point sequence error (see Section 3.3.3), $\overline{\text{FHOLD}}$ is asserted for one cycle. This is the only case where $\overline{\text{FHOLD}}$ is asserted in the exception mode.

If a floating-point trap condition occurs while $\overline{\text{FHOLD}}$ is asserted, $\overline{\text{FHOLD}}$ is deasserted at least one cycle after $\overline{\text{FEXC}}$ is asserted. Similarly, if FCCV is deasserted, it is reasserted at least one cycle after $\overline{\text{FEXC}}$ is asserted. For the $\overline{\text{FHOLD}}$ case, the CY7C601 takes the FP trap on the FP instruction that triggered the $\overline{\text{FHOLD}}$.

3.2.2.3 FNULL Signal

FNULL is used to signal a pipeline delay of the CY7C601 by the CY7C602. FNULL replaces FCCV and $\overline{\text{FHOLD}}$ for informing the system that the pipeline is being held. FNULL is asserted when either $\overline{\text{FHOLD}}$ is asserted or FCCV is deasserted. This signal is used as an input by the CY7C604/605 to monitor pipeline freezes initiated by the CY7C602.

3.3 CY7C602 Programming Model

3.3.1 CY7C602 Registers

The CY7C602 has three types of user accessible registers: the *f* registers, the FP queue, and the Floating-point Status Register (FSR). The *f* registers are the CY7C602 data registers. The FSR is the CY7C602 status and operating mode register. The FP queue contains the CY7C602 instructions that have started execution and are awaiting completion. The following section describes these registers in detail.

3.3.1.1 f Registers

The CY7C602 provides 32 registers for floating-point operations, referred to as *f* registers. These registers are 32 bits in length, which can be concatenated to support 64-bit double words. Extended precision instructions are not supported in the CY7C602, but the extended precision data format and its position in the SPARC FPU is defined for the SPARC architecture. *Figure 3–14* illustrates the data organization for the *f* registers.

Integer and single precision data requires a single 32-bit *f* register. Double precision data requires 64 bits of storage and occupies an even-odd pair of adjacent *f* registers. Extended precision data requires 128 bits of storage and occupies a group of four consecutive *f* registers, always starting with register f0, f4, f8, f12, f16, f20, f24, or f28.

The CY7C602 forces register addressing to match the data type specified by the floating-point instruction. This ensures data alignment in the *f* register file for double and extended precision data. *Figure 3–15* illustrates how the CY7C602 uses the five register address bits in a floating-point instruction for the different types of data. Single data word transfers (integer, single-precision floating-point) can be stored in any register. Consequently, all five bits of the register address specified in the floating-point instruction are valid. Double precision data must reside in an even-odd pair of adjacent registers. By ignoring the LSB of the register address for a FPop requiring a register pair, the CY7C602 ensures data alignment. In a similar manner, the two LSBs of the register address are ignored in a SPARC FPU that supports extended precision data.

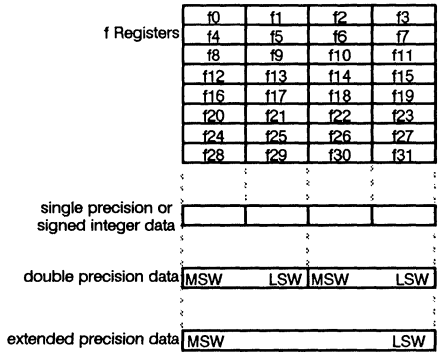


Figure 3–14. *f* Register Organization

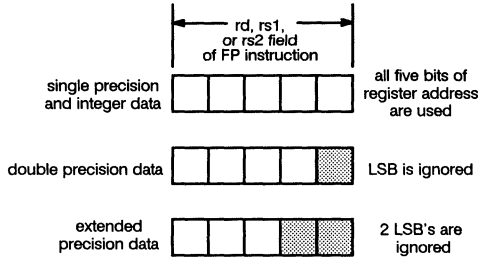


Figure 3–15. *f* Register Addressing

3.3.1.2 FP Queue

The CY7C602 maintains a floating-point queue of instructions that have started execution, but have yet to complete execution. The FP queue is used to accommodate the multiple clock nature of floating-point instructions and to support the handling of FP exceptions.

When the CY7C602 encounters an exception case, it asserts $\overline{\text{FEXC}}$ and enters pending exception mode. The CY7C602 remains in pending exception mode until the CY7C601/611 encounters another floating-point instruction, at which time the CY7C601/611 asserts the FXACK signal to force the CY7C602 into exception mode. When the CY7C602 enters the exception mode, floating-point execution halts until the FP queue is emptied. This allows the CY7C601 to store the floating-point instructions under execution when the exception case occurred. Emptying the FP queue frees the CY7C602 for use by the trap handler without losing the pre-exception state of the CY7C602.

The FP queue contains the 32-bit address and 32-bit FPop instruction of up to two instructions under execution. Floating-point load and store instructions and FP branch instructions are not queued. The front entry of the FP queue is accessible by executing the store double floating-point queue (STDFQ) instruction. The FP queue acts as a FIFO stack, pushing later entries to the top of the stack as the top entry is removed (or executed). A load FP queue instruction does not exist, as the FP queue must be loaded by launching instructions.

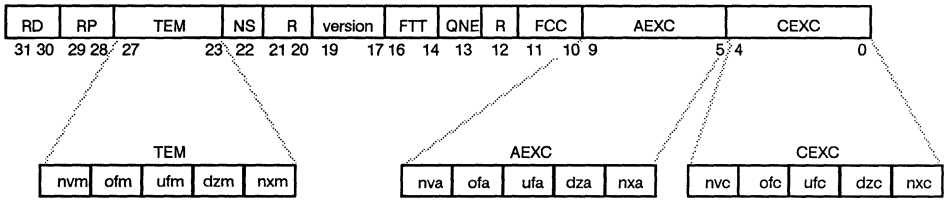


Figure 3–16. Floating-Point Status Register

3.3.1.3 Floating-Point Status Register (FSR)

The following paragraphs describe the bit fields of the floating-point status register (FSR). Refer to *Table 3–5* (following page) for bit assignments for the FSR fields.

RD *FSR(31:30).* Rounding Direction: These two bits define the rounding direction used by the CY7C602 during an FP arithmetic operation.

RP *FSR(29:28).* Rounding Precision: These two bits define the rounding precision to which *extended-precision* results are rounded. This bit is included in accordance with the ANSI/IEEE STD-754-1985. The CY7C602 does not currently support rounding of extended-precision results and this bit does not affect CY7C602 operation.

TEM *FSR(27:23).* Trap Enable Mask: These five bits enable traps caused by FPops. These bits are ANDed (1 = enable, 0 = disable) with the bits of the CEXC (current exception field) to determine whether to force a floating-point exception to the CY7C601. All trap enable fields correspond to the similarly named bit in the CEXC field (see below). The TEM field only affects which bits in the CEXC field will cause the $\overline{\text{FEXC}}$ signal to be asserted.

NS *FSR(22).* Non-Standard floating point: This bit enables non-standard floating-point operations in the CY7C602. When enabled, the CY7C602 inserts zeros for denormalized floating-point numbers before using them in a floating-point operation. The CY7C602 also writes back zero if a denormalized number results from an operation. This is not consistent with the IEEE-754-1985 specification, and is therefore, non-standard.

version *FSR(19:17).* The version number is used to identify the SPARC floating-point processor type. This field is set to 011 (3H) for the CY7C602, and is read-only.

FTT *FSR(16:14).* Floating-point Trap Type: This field identifies the floating point trap type of the current FP exception. This field can be read and written, and must be cleared by software.

QNE *FSR(13).* Queue Not Empty: This bit signals whether the FP queue is empty. (0 = empty, 1 = not empty)

FCC *FSR(11:10).* Floating-point Condition Codes: These two bits report the FP condition codes (see *Table 3–5*).

AEXC *FSR(9:5).* Accumulated EXCeptions: This field reports the accumulated FP exceptions that are masked by the TEM field. All masked exception cases are ORed with the contents of the AEXC and accumulated as status. All accumulated fields have the same definition as the corresponding field for CEXC (see below). This field can be read and written, and must be cleared by software (see *Table 3–5*).

CEXC *FSR(4:0).* Current EXCeptions: This field reports the current FP exceptions. This field is automatically cleared upon the execution of the next floating-point instruction. CEXC status is not lost upon assertion of a floating-point exception, because instructions following a valid exception are not executed by the CY7C602. The five CEXC bits are:

- *nvc = 1* indicates invalid operation exception. This is defined as an operation using an improper operand value. An example of this is 0/0.
- *ofc = 1* indicates overflow exception. The rounded result would be larger in magnitude than the largest normalized number in the specified format.
- *ufc = 1* indicates underflow exception. The rounded result is inexact, and would be smaller in magnitude than the smallest normalized number in the indicated format.
- *dzc = 1* indicates division-by-zero: X/0, where X is subnormal or normalized. Note that 0/0 does not set the dzc bit.
- *nxc = 1* indicates inexact exception. The rounded result differs from the infinitely precise correct result.

R *FSR21, 20, and 12.* Reserved - always set to 0.

Table 3–5. Floating-Point Status Register Summary

Field	Values	FSR bits	Description	Loadable by LDFSR
RD	0 - Round to nearest (tie-even) 1 - Round to 0 2 - Round to $+\infty$ 3 - Round to $-\infty$	31:30	Rounding Direction	yes
RP	0 - Extended precision 1 - Single precision 2 - Double precision 3 - Reserved	29:28	Extended Rounding Precision	yes
TEM	0 - Disable trap 1 - Enable trap NVM OFM UFM DZM NXM	27:23 27 26 25 24 23	Trap Enable Mask invalid operation trap mask overflow trap mask underflow trap mask divide by zero trap mask inexact trap mask	yes
NS	0 - Disable 1 - Enable	22	Non-standard Floating-point: 0 = IEEE mode; multiplier and ALU generate denormalized operand exceptions and produce unrounded normalized values on underflow exceptions. 1 = FAST mode; multiplier and ALU flush denormalized operands to zero and round underflow results to zero.	yes
version	0 - 7	19:17	FPU version number	no
FIT	0 - None 1 - IEEE Exception 2 - Unfinished FPop 3 - Unimplemented FPop 4 - Sequence Error 5 - 7 Reserved	16:14	Floating-point trap type	no
QNE	0 - queue empty	13	Queue Not Empty	no
FCC	0 - = 1 - < 2 - > 3 - Unordered	11:10	Floating-point Condition Codes	yes
AEXC	 NVA OFA UFA DXA NXA	9:5 9 8 7 6 5	Accrued Exception Bits accrued invalid exception accrued overflow exception accrued underflow exception accrued divide by zero exception accrued inexact exception	yes
CEXC	 NVC OFC UFC DZC NXC	4:0 4 3 2 1 0	Current Exception Bits current invalid exception current overflow exception current underflow exception current divide by zero exception current inexact exception	yes
r	Always set to 0	21, 20, 12	reserved bits	no

3.3.2 CY7C602 Floating-Point Instructions

SPARC floating-point instructions are separated into three groups: floating-point load/store, floating-point branch (FBfcc), and floating-point operate instructions (FPops). Floating-point load/store instructions are used to transfer data to and from the data registers (*f* registers). FP load/store instructions also allow the CY7C601/611 integer unit to read and write the floating-point status register (FSR) and to read the front entry of the floating-point queue. Floating-point load and store instructions are executed by both the CY7C601/611 and the CY7C602; the CY7C601/611 supplying all address and control signals for memory access and the CY7C602 loading or storing the data.

Floating-point branch (FBfcc) instructions (and coprocessor branch instructions (CBccc)) are executed by the CY7C601/611, since the CY7C601/611 is responsible for generating address and control signals for memory access. Conditional FBfcc branches are based upon the FCC(1:0) signals supplied by the CY7C602. FCC(1:0) is set by executing a FCMP instruction, which belongs to the FPop group of instructions. Floating-point branch instructions will cause the CY7C601/611 to recognize a pending floating-point exception in the same manner as other floating-point instructions (see Section 3.3.3).

FPops include all other floating-point instructions executed by the CY7C602. Floating-point operate instructions (FPops) include basic numeric operations (add, subtract, multiply, and divide), conversions between data types, register to register moves, and floating-point number comparison. FPops operate only on data in the floating-point registers.

The SPARC architecture supports four data types: 32-bit signed integer, single-precision FP, double-precision FP, and extended-precision FP. Extended precision instructions are defined in the SPARC architecture, but are not supported in the CY7C602. The CY7C602 supports execution of extended precision floating-point instructions by asserting an unimplemented instruction trap. This allows the CY7C601 to trap to a software emulation of extended precision floating-point.

Seven load/store instructions are executed by the CY7C602. The following describes the CY7C602 load/store instructions:

LDF and LDDF transfer data from memory to *f* registers 32 and 64 bits at a time, respectively.

STF and STDF transfer data from the *f* registers to memory in data widths of 32 and 64 bits.

LSFSR and STFSR allow the FSR to be read and written to.

STDFQ is a privileged instruction which allows the FP queue to be read.

All FPops operate only on data located in the *f* registers. The FPops are divided into four groups: basic arithmetic operations, compares, format conversions, and register-to-register moves. Move operations do not cause exceptions. The converts, moves and the square root instruction use only a single source operand. FP compare instructions modify only the FCC(1:0) signals. FPops are dispatched in one cycle in the CY7C601, and require multiple cycles to execute in the CY7C602.

Floating-point performance can be improved in the CY7C602 by scheduling FPop instructions such that the floating-point ALU and the floating-point multiply/divide/compare/square-root units are concurrently operating. With the exception of data dependencies, the ALU and multiply/divide/compare/square-root units are independent and can execute separate instructions without requiring the other unit to complete execution. Therefore, an FPop using the ALU followed by a FPop using the multiply/divide/compare/square-root unit does not require the previous instruction to finish before starting (assuming there are no data dependencies).

Table 3–6 and *Table 3–7* illustrate the CY7C602 instructions and their execution cycle count. For further information on the SPARC floating-point instructions, please refer to Chapter 6, SPARC Instruction Set.

Table 3–6. Floating-Point Load and Store Instruction Cycle Count

Mnemonic	Operation	Cycles
LDF	load floating-point	2
LDDF	load double floating-point	3
LDFSR	load FSR	2
STF	store floating-point	3
STDF	store floating-point double	4
STFSR	store FSR	3
STDFQ	store double FP queue	4

Table 3–7. Floating-Point Operate (FPops) Instruction Cycle Count

Mnemonic	Operation	Cycles
FABSs	absolute value	4
FADDs	add single	5
FADDd	add double	5
FCMPs	compare single	4
FCMPd	compare double	4
FCMPEs	compare single and exception if unordered	4
FCMPEd	compare double and exception if unordered	4
FDIVs	divide single	23
FDIVd	divide double	37
FMOVs	move	4
FMULs	multiply single	5
FMULd	multiply double	7
FNEGs	negate	4
FSQRTs	square root single	34
FSQRTd	square root double	63
FSUBs	subtract single	5
FSUBd	subtract double	5
FdTOi	convert double to integer	5
FdTOs	convert double to single	5
FiTOs	convert integer to single	9
FiTOd	convert integer to double	5
FsTOi	convert single to integer	5
FsTOd	convert single to double	5

3.3.3 CY7C602 Internal Operation

The CY7C602 operates in one of three modes: execution mode, pending exception mode, and exception mode (see *Figure 3–17*). After reset, the CY7C602 enters execution mode, which is the normal mode of operation. When the CY7C602 encounters a floating-point exception condition, the CY7C602 asserts $\overline{\text{FEXC}}$ and enters the pending exception mode. All FPop instructions under execution at this point are suspended. The CY7C601 asserts FXACK and enters the floating-point trap when the next floating point instruction is encountered. Upon receiving FXACK, the CY7C602 FPU enters exception mode. The CY7C602 returns to execution mode as soon as the trap handler empties the FP queue using STore Double Floating-point Queue instructions (STDFQ).

3.3.3.1 Exception Handling

Upon encountering an exception condition, the CY7C602 asserts $\overline{\text{FEXC}}$ to notify the CY7C601/611 that a floating-point exception has occurred and enters the pending exception mode. The CY7C601/611 enters the trap handler on the next floating-point instruction it encounters in the instruction stream, asserting FXACK to signal to the CY7C602 that the trap is being taken. At this point, the CY7C602 enters exception mode (see *Figure 3–17*).

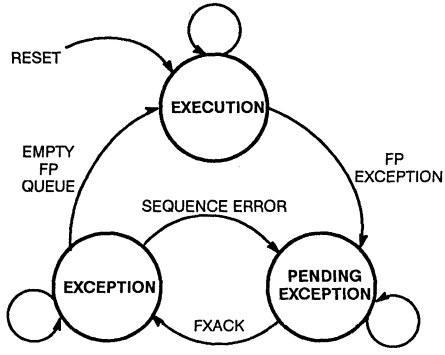


Figure 3–17. FPU Operation Modes

Upon receiving FXACK from the CY7C601, the mode of the CY7C602 changes from pending exception to exception mode. All FPops in the CY7C602 stop executing during pending exception and exception modes. While in exception mode, the CY7C602 will execute only store floating-point instructions until the FP queue is emptied. All floating-point store instructions are allowed while in this operating mode. Any load or FPop issued to the CY7C602 while in this mode causes a sequence error and returns the CY7C602 to exception pending mode. Once the queue is emptied by successive STDFQ instructions, the CY7C602 returns to execution mode.

Due to the latency of floating-point instruction execution, an exception caused by a FPop occasionally may not occur until one or more FP instructions have been fetched and executed (or entered into the FP queue for execution). This is a case where $\overline{\text{FEXC}}$ is not asserted before the next floating-point instruction is fetched and executed. In this case, $\overline{\text{FEXC}}$ is asserted as soon as the exception case is recognized, and the CY7C601/611 acknowledges the FP exception during the execute stage of the next floating-point instruction fetched after $\overline{\text{FEXC}}$ is asserted.

Figure 3–18 illustrates the handshake of signals between the CY7C601 and the CY7C602 during a floating-point exception. The qne (queue not empty) bit of the FSR is shown in *Figure 3–18* to illustrate the dependency of clearing the FP queue to return to execution mode.

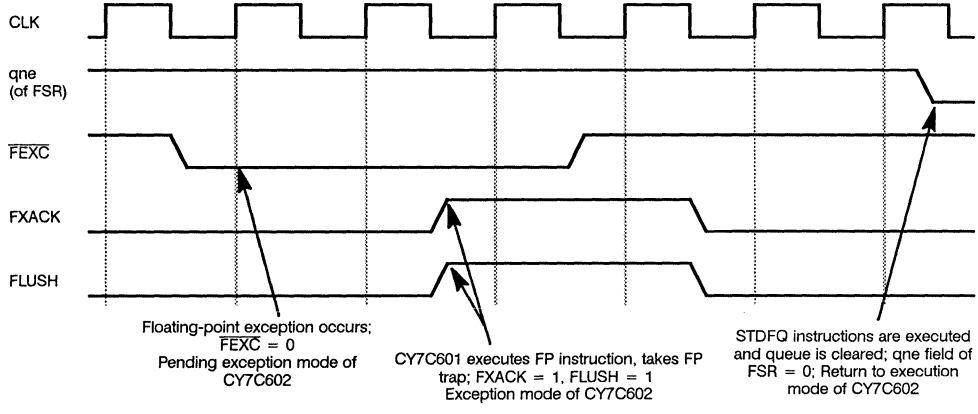


Figure 3–18. Floating-Point Exception Handshake

3.3.4 CY7C602 IEEE-754 Compliance

The CY7C602 meets the requirements of the IEEE Std. 754-1985 for floating-point arithmetic. Accuracy of the results of its operations are within ±½ LSB, as specified by the IEEE standard. The following sections describe the IEEE format as implemented on the CY7C602.

3.3.4.1 IEEE Definitions

The following terms are used extensively in describing the IEEE-754 floating-point data formats. This section is directly quoted from the *IEEE Standard for Binary Floating-Point Arithmetic.*

biased exponent — The sum of the exponent and a constant (bias) chosen to make the biased exponent's range nonnegative. (Note in the remainder of this section, the term "exponent" refers to a biased exponent.)

binary floating-point number — A bit string characterized by three components: a sign, a signed exponent and a significand. Its numerical value, if any, is the signed product of its significand and two raised to the power of its exponent.

Denormalized — Denormalized numbers are those numbers whose magnitude is smaller than the smallest magnitude representable in the format. They have a zero exponent and a denormalized non-zero fraction. Denormalized fraction means that the hidden bit is zero.

The CY7C602 cannot directly operate on denormalized operands. The CY7C602 asserts an unfinished FPop exception when an operation results in a denormalized number.

denormalized number — (DNRM) A non-zero floating-point number whose exponent has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero. (Denormalized numbers are also referred to as subnormal in this text.)

fraction — The field of the significand that lies to the right of its implied binary point.

NaN — Not a number, a symbolic entry encoded in floating-point format. They are used to signal invalid operations and as a way of passing status information through a series of calculations. NaNs arise in one of two ways: they can be generated by the CY7C602 upon an invalid operation or they may be supplied by the user as an input operand. NaN is further subdivided into two categories: quiet and signaling. Signaling NaNs signal the invalid operation exception whenever they appear as operands. Quiet NaNs propagate through almost every arithmetic operation without signaling exceptions.

Normalized — Most calculations are performed on normalized numbers. For single-precision, they have a biased exponent range of 1 to 255, which results in a true exponent range of -126 to $+127$. The normalized number type implies a normalized significand (hidden bit is 1).

significand — The component of a binary floating-point number that consists of an explicit or implicit lead ing bit to the left of its implied binary point and a fraction field to the right.

true exponent — The component of a binary floating-point number that normally signifies the integer power to which 2 is raised in determining the value of the represented number.

Zero — The IEEE zero has all fields except the sign field equal to zero. The sign bit determines the sign of zero (i.e., the IEEE format defines a $+0$ and a -0).

3.3.4.2 IEEE Floating-point Data Formats

The CY7C602 directly supports single- and double-precision floating-point data formats. Extended-precision formats are defined as part of the SPARC architecture, but are not directly executed by the CY7C602. Extended-precision instructions encountered by the CY7C602 cause an unimplemented instruction trap to be asserted by the CY7C602. This allows software to emulate extended-precision instructions through the use of a trap handler. Single-, double-, and extended-precision formats are described in this section.

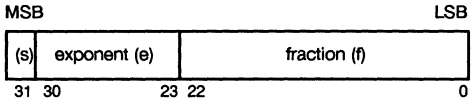


Figure 3–19. Single-Precision Floating-Point Format

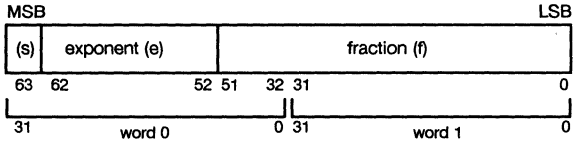


Figure 3–20. Double-Precision Floating-Point Format

Single-Precision Floating-Point

Single-precision floating-point data are 32-bits wide and consist of three fields: a single sign bit (s), an eight-bit biased exponent (e), and a 23-bit fraction (f). *Figure 3–19* illustrates the single-precision floating-point format.

The IEEE standard defines single-precision floating-point numbers according to the following conventions:

(+0, -0)	If e = 0 and f = 0, then the value V = $(-1)^s * (0)$ Note that two representations of zero exist, one positive and one negative
DNRM (denormalized)	If e = 0 and f $\neq$ 0, then the value V = DNRM
Normalized	If 0 < e < 225, then value V = $(-1)^s * (2^{e-127}) * (1.f)$ Note that 1.f is the significand. The one to the left of the binary point is the so-called "hidden bit." This bit is not stored as part of the floating-point word; it is implied. For a number to be normalized, it must have this one to the left of the binary point.
(+∞, -∞)	If e = 255 and f = 0, then value V = $(-1)^s$ (∞)
NaN (not a number)	If e = 255 and f $\neq$ 0, then value V = NaN.
	The value is a quiet NaN if the first bit of the fraction is 1, and a signaling NaN if the first bit of the fraction is 0 (at least one bit must be non-zero).

Double-Precision Floating-Point

Double-precision floating-point data are 64-bits wide and consist of three fields: a single sign bit (s), an eleven-bit biased exponent (e), and a 52-bit fraction (f). *Figure 3–20* illustrates the double-precision floating-point format.

The IEEE standard defines double-precision floating-point numbers according to the following conventions:

(+0, -0)	If e = 0 and f = 0, then value V = $(-1)^s * (0)$
DNRM	If e = 0 and f $\neq$ 0, then value V = DNRM
Normalized	If 0 < e < 2047, then value V = $(-1)^s * (2^{e-1023}) * (1.f)$
(+∞, -∞)	If e = 2047 and f = 0, then value V = $(-1)^s * (\infty)$
NaN	If e = 2047 and f $\neq$ 0, then value V = NaN.
	The value is a quiet NaN if the first bit of the fraction is 1, and a signaling NaN if the first bit of the fraction is 0 (at least one bit must be non-zero).

Extended-Precision Floating-Point

Extended-precision floating-point data are 128 bits wide and consist of six fields: a single sign bit (s), a 15-bit biased exponent (e), 16 reserved bits, a single hidden bit(j), a 63-bit fraction, and 32 additional reserved bits. The extended-precision floating-point differs from the other precision types in that the "hidden bit" is no longer hidden. The value of the hidden bit is explicitly defined as j, which defines the number as normalized or denormalized.

The IEEE standard defines extended-precision floating-point numbers according to the following conventions:

(+0, -0)	If e = 0 and f = 0, then value V = $(-1)^s * (0)$
DNRM	If e = 0 and f ≠ 0, then value V = DNRM
Normalized	If 0 < e < 32767, then value V = $(-1)^s * (2^{e-16383}) * (1.f)$
(+∞, -∞)	If e = 32767 and f = 0, then value V = $(-1)^s * (\infty)$
NaN	If e = 32767 and f ≠ 0, then value V = NaN
	The value is a quiet NaN if the first bit of the fraction is 1, and a signaling NaN if the first bit of the fraction is 0 (at least one bit must be non-zero).

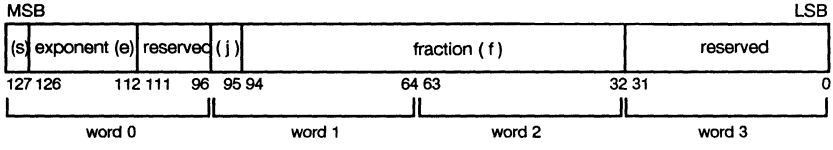


Figure 3–21. Extended-Precision Floating-Point Format

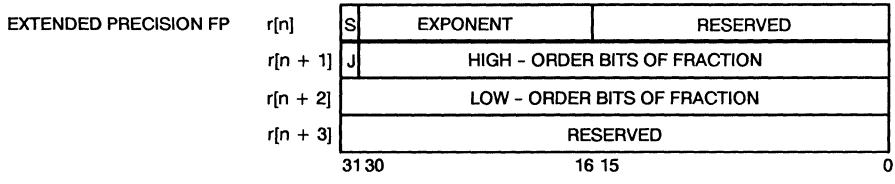


Figure 3–22. Extended-Precision Data Organization in Registers

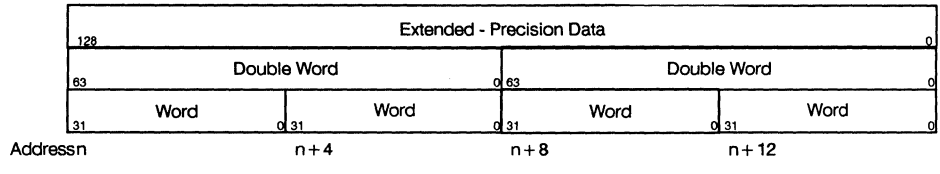


Figure 3–23. Extended-Precision Data Organization in Memory

3.3.5 CY7C602 Exception Cases

The following section describes the CY7C602 exception cases, including exceptions specified by the IEEE-754 standard.

Unfinished FPop. This exception case can occur when operations on normalized floating-point numbers either encounter a denormalized operand or produce a denormalized result. This exception case is asserted upon executing any FPop encountering a NaN as one of the operands. The CY7C602 also asserts this trap when a floating-point to integer conversion overflow occurs.

Unimplemented FPop. This exception is asserted by the CY7C602 upon encountering a defined SPARC FPop instruction that is not supported by the CY7C602. This includes all operations using extended-precision format operands. The trap handler is expected to emulate the unimplemented instruction.

Sequence Error. This exception is asserted by the CY7C602 when a floating-point instruction (other than FP store) is attempted after the CY7C602 has entered either pending exception or exception mode. The CY7C602 suspends all instruction execution with the exception of FP stores until the FP exception has been acknowledged and the FP queue has been cleared.

IEEE Exceptions. This class of exceptions is defined as part of the IEEE-754 Standard. The five exceptions defined as IEEE Exceptions are reported in the CEXC and AEXC fields of the FSR. These exceptions are: invalid, overflow, underflow, division-by-zero, and inexact. The only exceptions that can coincide are inexact with overflow and inexact with underflow. The following paragraphs discuss these exception cases.

Invalid Operation. The invalid operation exception is signaled if an operand is invalid for the operation to be performed. The result, when the exception occurs without a trap, shall be a quiet NaN provided the destination has a floating-point format. The invalid operations are:

1. Any operation on a signaling NaN
2. Addition or subtraction: Magnitude subtraction of infinities such as $(+\infty) + (-\infty)$
3. Multiplication: $0 \times \infty$
4. Division: $0/0$ or ∞/∞
5. Square root if the operand is less than zero
6. Conversion of a binary floating-point number to an integer or decimal format when overflow, infinity, or NaN precludes a faithful representation in that format and this cannot otherwise be signaled
7. Floating-point compare operations: when one or more of the operands are NaN

Division-by-zero. If the divisor is zero and the dividend is a finite nonzero number, then the division by zero exception shall be signaled. The result, when no trap occurs, shall be a correctly signed ∞.

Overflow. The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. The result, when no trap occurs, shall be determined by the rounding mode and the sign of the intermediate result as follows:

1. Round to nearest carries all overflows to ∞ with the sign of the intermediate result.
2. Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result.
3. Round toward $-\infty$ carries positive overflows to the format's largest positive finite number, and carries negative overflows to $-\infty$.
4. Round toward $+\infty$ carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$.

Underflow. The CY7C602 does not assert an underflow exception. Underflow cases are covered in the unfinished FPop trap, which is asserted in any case where a denormalized number is used as an operand. The unfinished FPop trap handler must resolve the underflow condition and update this bit to reflect correct accumulated exception status (AEXC field of FSR).

Inexact. The inexact exception is generated whenever there is a loss of accuracy (or significance) in the result. The CY7C602 computes results to higher precision than the number of fraction bits in the format. If any of the fraction bits to the right of the LSB was one prior to rounding, the inexact exception is signaled.

3.4 CY7C602 Signal Descriptions

The following sections describe the external signals of the CY7C602. Active low signals are marked with an overbar, active high signals are not.

3.4.1 Integer Unit Interface Signals

$\overline{FP}$ *active-low output* Floating-point Present: This signal indicates to the CY7C601 that a FPU is present in the system. In the absence of a FPU, this signal is pulled up to VCC by a resistor. This is a static signal; it always asserts a low output. The CY7C601 generates a floating-point disable trap if $\overline{FP}$ is not asserted during the execution of a floating-point instruction.

FCC(1:0) *output* Floating-point Condition Codes: The FCC(1:0) bits indicate the current condition code of the FPU, and are valid only if FCCV is asserted. FBfcc instructions use the value of these bits during the execute cycle if they are valid. If the FCC(1:0) bits are not valid, then FCCV is released, which halts the CY7C601 until the FCC bits become valid.

Table 3–8. FCC(1:0) Condition Codes

FCC1	FCC0	Condition
0	0	equal
0	1	Op1 < Op2
1	0	Op1 > Op2
1	1	Unordered

3

FCCV *output* Floating-point Condition Codes Valid: The CY7C602 asserts the FCCV signal when the FCC(1:0) represent a valid condition. The FCCV signal is deasserted when a pending floating-point compare instruction exists in the floating-point queue. FCCV is reasserted when the compare instruction is completed and FCC bits are valid.

$\overline{FHOLD}$ *output* Floating-point HOLD: The $\overline{FHOLD}$ signal is asserted by the CY7C602 if it cannot continue execution due to a resource or operand dependency. The CY7C602 checks for all dependencies in the decode stage, and if necessary, asserts $\overline{FHOLD}$ in the next cycle. The $\overline{FHOLD}$ signal is used by the CY7C601 to freeze its pipeline in the same cycle. The CY7C602 must eventually de-assert $\overline{FHOLD}$ to release the CY7C601 pipeline.

$\overline{FEXC}$ *output* Floating-point EXCeption: The $\overline{FEXC}$ is asserted if a floating-point exception has occurred. It remains asserted until the CY7C601 acknowledges that it has taken a trap by asserting FXACK. Floating-point exceptions are taken only during the execution of a floating-point instruction. The CY7C602 releases $\overline{FEXC}$ when it receives FXACK.

FXACK *input* Floating-point eXception ACKnowledge: The FXACK signal is asserted by the CY7C601 to acknowledge to the CY7C602 that the current FP trap is taken.

INST *input* INSTruction fetch: The INST signal is asserted by the CY7C601 whenever a new instruction is being fetched. It is used by the CY7C602 to latch the instruction on the D(31:0) bus into the FPU instruction buffer. The CY7C602 has two instruction buffers (D1 and D2) to save the last two fetched instructions (see *Figure 3–3*). When INST is asserted, the new instruction enters the D1 buffer and the old instruction is pushed into the D2 buffer.

FINS1 *input* Floating-point INStruction in buffer 1: The FINS1 signal is asserted by the CY7C601 during the decode stage of a FPU instruction if the instruction is stored in the D1 buffer of the CY7C602. The CY7C602 uses this signal to launch the instruction in the D1 buffer into its execute stage instruction register.

FINS2 *input* Floating-point INStruction in buffer 2: The FINS2 signal is asserted by the CY7C601 during the decode stage of a FPU instruction if the instruction is stored in the D2 buffer of the CY7C602. The CY7C602 uses this signal to launch the instruction in the D2 buffer into its execute stage instruction register.

FLUSH *input* Floating-point instruction fLUSH: The FLUSH signal is asserted by the CY7C601 to signal to the CY7C602 to flush the instructions in its instruction registers. This may happen when a trap is taken by the CY7C601. The CY7C601 will restart the flushed instructions after returning from the trap. FLUSH has no effect on instructions in the floating-point queue. In addition to freezing the FPU pipeline, the CY7C602 uses FLUSH to shut off the D bus drivers during store operations. To ensure correct operation of the CY7C602, FLUSH must not change state more than once during a clock cycle.

3.4.2 Coprocessor Interface Signals

$\overline{\text{CHOLD}}$ *input* Coprocessor HOLD: The $\overline{\text{CHOLD}}$ signal is asserted by the coprocessor if it cannot continue execution. The coprocessor must check all dependencies in the decode stage of the instruction and assert the $\overline{\text{CHOLD}}$ signal, if necessary, in the next cycle. The coprocessor must eventually deassert this signal to unfreeze the CY7C601 and CY7C602 pipelines. The $\overline{\text{CHOLD}}$ signal is latched with a transparent latch in the CY7C602 before it is used.

CCCV *input* Coprocessor Condition Codes Valid: The coprocessor asserts the CCCV signal when the CCC(1:0) represent a valid condition. The CCCV signal is deasserted when a pending coprocessor compare instruction exists in the coprocessor queue. CCCV is reasserted when the compare instruction is completed and the CCC(1:0) bits are valid. The CY7C602 will enter a wait state if CCCV is deasserted. The CCCV signal is latched with a transparent latch in the CY7C602 before it is used.

3.4.3 System/Memory Interface Signals

A(31:0) *input* Address bus (31:0): The address bus for the CY7C602 is an input-only bus. The CY7C601 supplies all addresses for instruction and data fetches for the CY7C602. The CY7C602 captures addresses of floating-point instructions from the A(31:0) bus into the DDA register. When INST is asserted by the CY7C601, the contents of the DDA is transferred to the DA1 register.

D(31:0) *input/output* Data bus (31:0): The D(31:0) bus is driven by the FPU only during the execution of floating-point store instructions. The store data is sent out unlatched and must be latched externally before it is used. Once latched, store data is valid during the second data cycle of a store single access and on the second and third data cycle of a store double access. The data alignment for load and store instructions is done inside the FPU. A double word is aligned on an eight-byte boundary. A single word is aligned on a four-byte boundary.

$\overline{\text{DOE}}$ *input* Data Output Enable: The $\overline{\text{DOE}}$ signal is connected directly to the data output drivers and must be asserted during normal operation. Deassertion of this signal three-states all output drivers on the data bus. This signal should be deasserted only when the bus is granted to another bus master, i.e, when either $\overline{\text{BHOLD}}$, $\overline{\text{CHOLD}}$, $\overline{\text{MHOLDA}}$, or $\overline{\text{MHOLDB}}$ is asserted.

$\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$ *input* Memory HOLD: Asserting $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ freezes the CY7C602 pipeline. Either $\overline{\text{MHOLDA}}$ or $\overline{\text{MHOLDB}}$ is used to freeze the FPU (and the IU) pipelines during a cache miss (for systems with cache) or when slow memory is accessed.

$\overline{\text{BHOLD}}$ *input* Bus HOLD: This signal is asserted by the system's I/O controller when an external bus master requests the data bus. Assertion of this signal will freeze the FPU pipeline. External logic should guarantee that after deassertion of $\overline{\text{BHOLD}}$, the state of all inputs to the chip is the same as before $\overline{\text{BHOLD}}$ was asserted.

$\overline{\text{MDS}}$ *input* Memory Data Strobe: The $\overline{\text{MDS}}$ signal is used to load data into the FPU when the internal FPU pipeline is frozen by assertion of $\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, or $\overline{\text{BHOLD}}$.

FNULL *output* Fpu NULLify cycle: This signal signals to the memory system when the CY7C602 is holding the instruction pipeline of the system. This hold would occur when $\overline{\text{FHOLD}}$ is asserted or FCCV is deasserted. This signal is used by the memory system in the same fashion as the integer unit's INULL signal. The system needs this signal because the IU's INULL does not take into account holds requested by the FPU.

$\overline{\text{RESET}}$ *input* RESET: Asserting the $\overline{\text{RESET}}$ signal resets the pipeline and sets the writable fields of the floating-point status register (FSR) to zero. The $\overline{\text{RESET}}$ signal must remain asserted for a minimum of eight cycles.

CLK *input* CLocK: The CLK signal is used for clocking the FPU's pipeline registers. It is high during the first half of the processor cycle and low during the second half. The rising edge of CLK defines the beginning of each pipeline stage in the FPU.

Chapter 4

CY7C604 / CY7C605 Cache Controller and Memory Management Units

The CY7C604 (CMU) and CY7C605 (CMU-MP) are combined memory management unit (MMU) and cache controllers with on-chip cache tag memory. The CY7C604 and CY7C605 are designed as an integral part of the CY7C600 family to provide a high-performance solution for cache and virtual memory support. The CY7C604 is designed for uniprocessor systems, providing control for a 64-kbyte virtual cache. The CY7C604/605 cache is extendible to 256 kbytes through the addition of cache RAMs and CY7C604/605s. Expansion of the CY7C604/605 cache increases the number of TLB (Translation Lookaside Buffer) entries available to the system for MMU address translation, as well as increasing the number of cache tag entries available to the cache. Another feature of the CY7C604 is cache locking, which provides deterministic response time for real-time systems controlling time-critical processes. The CY7C604, as well as the CY7C605, provides the SPARC reference MMU and supports the SPARC Mbus standard for interfacing to physical memory.

The CY7C605, a derivative of the CY7C604, is designed to support the requirements of multiprocessing systems. The CY7C605 provides two separate cache tag memories, as compared to the single cache tag memory used on the CY7C604. The second cache tag memory is physically addressed and allows concurrent bus snooping without stalling the CY7C601. This allows the CY7C605 to maintain cache coherency with other cache systems without degrading CPU performance. The CY7C605 supports the Mbus level 2 cache coherency protocol, which is modeled after the acclaimed IEEE Futurebus. The CY7C605 is pin compatible with the CY7C604, which allows a CY7C604-based CPU to be used in a multiprocessor system by substituting the CY7C604 with the CY7C605 and enhancing the system software.

The MMU portion of the CY7C604 and CY7C605 provides translation from a 32-bit virtual address range (4 gigabytes) to a 36-bit physical address (64 gigabytes), as provided in the SPARC reference MMU specification. Virtual address translation is further extended with the use of a context register, which is used to identify up to 4096 contexts or tasks. The cache tag entries and TLB entries contain context numbers to identify tasks or processes. This minimizes unnecessary cache tag and TLB entry replacement during task switching.

The MMU features a 64-entry translation lookaside buffer. The TLB acts as a cache for address mapping entries used by the MMU to map a virtual address to a physical address. These mapping entries, referred to as page table entries or PTEs, allow one of four levels of address mapping. A PTE can be defined as the address mapping for a single 4-kbyte page, a 256-kbyte region, a 16-Mbyte region, or a 4-Gbyte region. The TLB entries are lockable, allowing important TLB entries to be excluded from replacement.

The MMU performs its address translation task by comparing a virtual address supplied by the CY7C601 (Integer Unit) to the address tags in the TLB entries. If the virtual address and the value of the context register match a TLB entry, a TLB "hit" occurs. When this occurs, the physical address stored in the TLB is used to translate the virtual address to a physical address. The access type (read/write of data or instruction) and privilege level (user/supervisor) are checked during translation. If a TLB hit occurs but access-level protection is violated, the MMU signals an exception and the operation ends.

If the virtual address or context does not match any valid TLB entry, a TLB "miss" occurs. This causes a table walk to be performed by the MMU. The table walk is a search performed by the MMU through the address translation tables stored in main memory. The MMU searches through several levels of tables for the PTE corresponding to the virtual address. Upon finding the PTE, the MMU translates the address and selects a TLB entry for replacement, where it then stores the PTE.

The 64-kbyte virtual cache is organized into 2048 lines of 32 bytes each. The term "virtual cache" refers to the direct addressing of the cache by the integer unit (CY7C601) with the virtual address bus. Virtual address bits VA(15:5) select the cache line, and virtual address bits VA(4:2) select the 32-bit word of the cache line, as illustrated in *Figure 4–1.* The CY7C604/605 provides access control for the cache by checking the context and virtual address against the cache tags. If the virtual address, access-level, and context match the cache tag for the cache line addressed, a cache hit occurs and the access is enabled. If the virtual address or context do not match the cache tag for the cache line, a cache miss occurs and the cache controller accesses main memory for the required data.

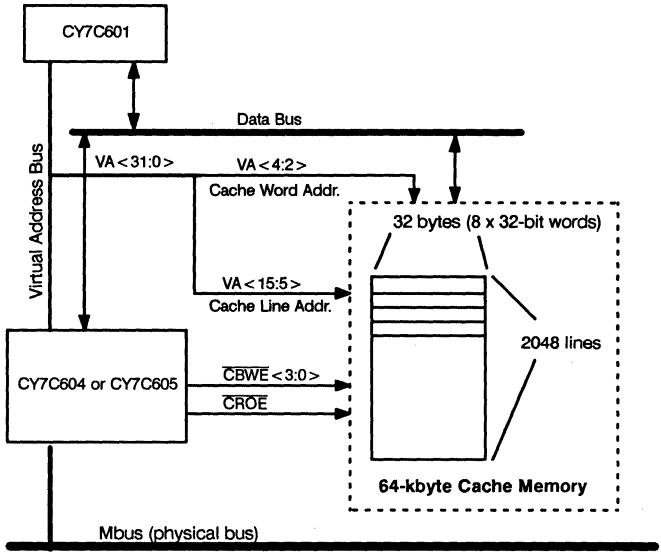


Figure 4–1. Virtual 64-kbyte Cache

The CY7C604/605 cache controller supports two modes of caching : write-through with no write allocate and copy-back with write allocate. Write-through mode is a simpler style of cache management that causes write accesses to the cache to be written through to main memory upon each write access. The advantage of this method is that the cache always remains coherent with main memory. Its disadvantage is that each write to the cache is echoed to main memory, which increases traffic on the system bus. Another disadvantage to write-through is that the processor is delayed by the time required to arbitrate the system bus and write the data to main memory. However, in the case of the CY7C604/605, this disadvantage is significantly offset by the inclusion of write buffers. The write buffers can store up to four doubleword accesses, allowing the CY7C601 to continue execution while data is written to main memory.

Copy-back cache mode causes write accesses to be written to the cache only. This causes the cache line to become modified. Modified cache lines are automatically written back to main memory only when the cache line is no longer needed. Copy-back mode is a more complex mode of cache management, but provides substantial system performance improvements over write-through due to decreased traffic on the system bus.

A 32-byte write buffer and a 32-byte read buffer are provided in the CY7C604/605 to fully buffer the transfer of a cache line. This feature allows the CY7C604/605 to simultaneously read a cache line from main memory as it is flushing a modified cache line from the cache. This feature is also used in write-through cache mode for write accesses to main memory. The write buffer avoids stalling the CY7C601 on writes to main memory by storing the write data until the physical bus becomes available. The write buffer writes the data to memory as a background task.

The CY7C604 and CY7C605 support the SPARC Mbus reference standard interface. The Mbus is a peer-level, high-speed, 64-bit, multiplexed address and data bus that supports a full peer-level protocol (i.e., multiple bus masters). The CY7C604/605 Mbus supports data transfers in transaction sizes of 1, 2, 4, 8, or 32 bytes. These data transfers are performed in either burst or non-burst mode, depending upon size. Data transactions larger than eight bytes (one doubleword) are transferred in burst mode, which consists of an address phase followed by four data phases. Non-burst transactions consist of an address phase followed by one data phase, and are used for data transactions of eight or less bytes. Bus mastership is granted and controlled by an external bus arbiter. The bus arbiter sets bus priorities, and grants access to a bus master. Additional information on the Mbus can be found in the Physical Bus section.

Mbus is divided into two levels of implementation: level 1 and level 2. Level 1, implemented on the CY7C604, is the uniprocessor version of Mbus. Level 1 is a subset of level 2, which is the multiprocessor version of Mbus. The CY7C605 supports level 2 Mbus. Level 2 Mbus includes the IEEE Futurebus cache coherency protocol, which has been recognized in the industry as a superior method of supporting multiprocessing systems. Level 2 Mbus defines five cache states for describing cache line status. Transactions on the Mbus are monitored or "snooped" by the CY7C605 and other bus agents

on the level 2 Mbus to maintain ownership status for each cache line. Transactions on the level 2 Mbus are made with respect to the cache line ownership status to ensure consistency for shared data images.

The level 2 Mbus supports direct data intervention, which allows a cache system with the up-to-date version of a cache line to directly supply the data to another cache system without having to first update main memory. Direct data intervention provides a significant performance improvement over systems which do not support this feature. In addition, the CY7C605 provides support for memory systems with reflective memory controllers. A memory system with reflective memory control can recognize a cache-to-cache data transaction and automatically update itself without delaying the system. Another system concept supported by the CY7C605 is secondary caching. Secondary caching provides a performance advantage over systems directly using main memory, and provides an economic advantage over systems using large caches for each processing node.

4.1 Memory Management Unit

This section describes the SPARC reference MMU implemented on the CY7C604 and CY7C605. This function is identical for both the CY7C604 and CY7C605, and all details of Sections 4.1 and 4.2 apply to both.

The MMU provides virtual to physical address translation with the use of an on-chip translation lookaside buffer. The TLB is in reality a full Address Translation Cache for address translation entries stored from tables in main memory. These entries, referred to as page table entries or PTEs, contain the mapping information used by the MMU to translate the virtual addresses. Addresses presented to the MMU for translation are compared against the set of PTEs stored in the TLB. All entries in the TLB are simultaneously accessed through the use of advanced Content Addressable Memory (CAM) technology. If a match for the virtual address and context is found in a valid TLB entry and the access protection is not violated, a TLB hit occurs and the address is translated. A virtual address and context that matches a valid TLB entry but violates the memory access protections will cause the CY7C604/605 to generate a memory exception to the CY7C601. If the TLB entries do not match the address and context, or the TLB entry is invalid, then a TLB miss occurs. The MMU responds to the TLB miss by initiating a table walk to find the correct PTE stored in main memory for the virtual address.

The MMU uses a tree-structured table walk algorithm to find page table entries not found in the TLB. The table walk is a search through a series of four tables in main memory for the PTE corresponding to a virtual address. These tables are: the context table, the level 1 table, the level 2 table, and the level 3 table. The table walk uses the context table pointer register as a base register and the context number as a offset to point to an entry in the context table. At any address, the MMU finds either a PTE, which terminates its search, or a Page Table Pointer (PTP). A PTP is a pointer used in conjunction with a field in the virtual address to select an entry in the next level of tables. The table walk continues searching through levels of tables as long as PTPs are found pointing to the next table. The table walk terminates when a PTE is found, or an exception is generated if a PTE is not found after accessing the level 3 table. An exception is also generated if the table walk finds an invalid or reserved entry in the page tables. Upon finding the PTE, the CY7C604/605 stores it in an available TLB entry and translates the corresponding virtual address. The table walk processing is implemented in the CY7C604/605 hardware. It is self-initiated, and is transparent to the user.

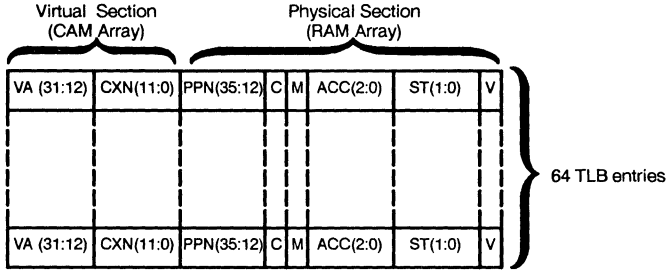


Figure 4–2. Translation Lookaside Buffer (TLB)

4.1.1 Translation Lookaside Buffer (TLB)

The CY7C604/605 uses a 64-entry fully associative TLB for address translation. The TLB consists of two sections: a virtual section and a physical section, as shown in *Figure 4–2*. The virtual section is compared against the virtual address and the contents of the context register. A content addressable memory (CAM) is used as the virtual section of the TLB. The CAM provides simultaneous comparison of all 64 TLB entries with the current virtual address and context. The physical section of the TLB is a RAM array, and its entries are addressed by a valid compare output from a CAM entry. If a CAM entry matches the virtual address and context, the corresponding RAM entry in the TLB provides the physical address for use by the CY7C604/605.

The virtual section of a TLB entry consists of 20 bits of virtual address (VA(31:12)) and a 12-bit context number (CXN(11:0)). The physical section of a TLB entry consists of a 24-bit physical page number (PPN(35:12)), a cacheable bit (C), a modified bit (M), a three-bit field for page access-level protection (ACC(2:0)), a two-bit short translation field (ST(1:0)), and one valid bit (V).

As described by the SPARC reference MMU specification, bits 31 through 12 of the virtual address are translated to an expanded physical address using bits 35 through 12. The translation of these bits depends upon the ST field of the TLB entry (or PTE) and the MMU operation mode (refer to page 4–13). Bits 11 through 0 of the virtual address are not translated, and are defined as the page offset for the 4-kbyte memory page.

A TLB entry (PTE) can be defined to map a virtual address into one of four sizes of addressing regions using the ST field. The four sizes of addressing regions are: 4-kbyte, 256-kbyte, 16-Mbyte, or 4-Gbyte. *Table 4–1* illustrates the values assigned to the ST(1:0) field.

The value of the short translation bits affects both the addresses generated using the TLB entry and the virtual addresses allowed to match with the TLB entry. The virtual address supplied by the integer unit is divided into four fields : index 1, index 2, index 3, and page offset, as illustrated in *Figure 4–3*. For ST = (1,1) (4-Gbyte addressing range), only the context register is used to match a TLB entry. Setting ST = (1,1) essentially causes the CAM array to ignore the index 1, 2, and 3 fields of the virtual address. Consequently, the address generated using the TLB entry only supplies the upper four bits of the 36-bit physical address. Index 1, 2, and 3 fields, along with the page offset, are passed along to the physical address unchanged.

The three remaining values of the ST field "turn on" comparison of the three index fields. The index fields that are required to match a TLB entry also become the fields that are replaced by the TLB entry during virtual to physical translation. Setting ST = (1,0), (16-Mbyte addressing region), requires the TLB to match the context and index 1 fields of the virtual address to the TLB entry. The TLB entry with ST = (1,0) will supply the upper four address bits and replace the index 1 field of the virtual address with a physical address field. The index 2, 3, and page offset fields are passed along to the physical address from the virtual address. Setting ST = (0,1) and (0,0) adds index 2 and index 3 fields to the comparison, respectively. Setting ST = (0,0) causes the TLB to require matching of the context, index 1, 2, and 3, and will replace all but the page offset when translating the virtual address.

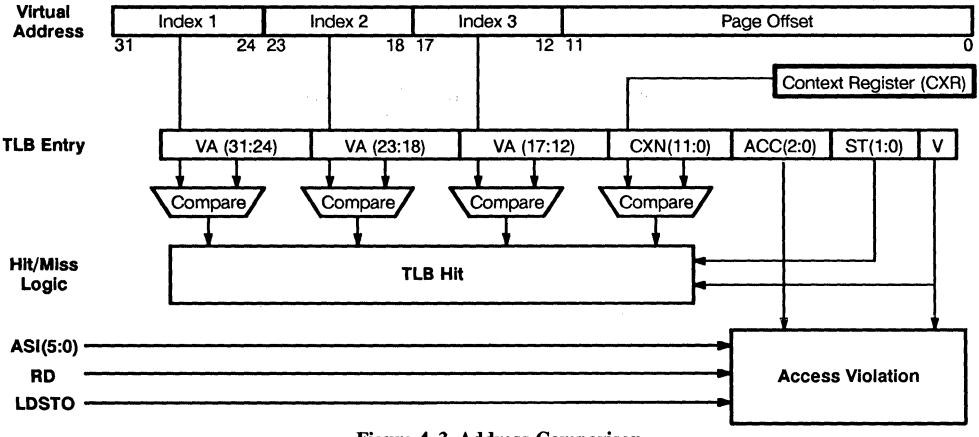


Figure 4–3. Address Comparison

Table 4–1. Short Translation Bits—ST(1:0)

ST1	ST0	Address Mapping
0	0	4-kbyte (page size)
0	1	256-kbyte
1	0	16-Mbyte
1	1	4-Gbyte

Physical addresses are generated using the contents of the PPN field of the TLB entry. The portion of the PPN field used to map the virtual address to a physical address is dependent upon the ST(1:0) bit field, as described above. If a 4-kbyte linear addressing range is specified by the ST(1:0) bits, then the entire 24 bit field is used as the upper 24 bits of the physical address. When a 256-kbyte linear addressing range is specified, the upper 18 bits of the PPN(35:18) field are used in the physical address. The remaining bits of the physical address are supplied from the virtual address. The upper 12 bits of the PPN(35:24) field are used for a 16-Mbyte addressing region. If a 4-Gbyte region is selected, only the upper four bits of the PPN(35:32) field are used in the address translation. The page offset field of the virtual address is always used as the lower twelve bits of the physical address.

The cacheable bit (C) indicates whether the memory addressed by the TLB entry is cacheable or not. If the MMU is enabled, the value of the C bit is output on the MC pin (MAD(43)) of the Mbus during the address phase of a transaction. The Mbus is described in the Physical Bus section.

The modified bit (M) in the TLB is set when the CY7C601 modifies the memory page. This bit may be checked by an operating system to determine the modified status of a memory area.

The access-level protection (ACC) bits are described in *Table 4–2.* The ACC bits define the access-level protection for the addressing region controlled by the TLB entry. Access-level protection is checked during a TLB access. If a TLB hit occurs but access-level protection is violated, the MMU generates a synchronous fault and the operation terminates (see Section 4.9, Synchronous Faults).

The valid bit (V) reports the valid status of the TLB entry. These bits are cleared upon power on reset ($\overline{POR}$) to invalidate the TLB entries. These bits are also cleared for a TLB entry flush.

Programmer's Note: When loading the TLB entries under software control (i.e., TLB entries loaded by the integer unit with ASI = 6), care must be taken to ensure that multiple TLB entries cannot map to the same virtual address. This may inadvertently occur when combining TLB entries that map different sizes of addressing regions. For example, a 4-kbyte region described by a TLB entry could be included in a TLB entry for a 16-Mbyte region. Violation of this restriction will result in an invalid output from the TLB. Note that this case cannot happen when the TLB entries are automatically loaded by the CY7C604/605 during a table walk, as the TLB is checked for a "hit" first.

Table 4–2. Access-Level Protection Bits—ACC(2:0)

ACC	User Access	Supervisor Access
0	Read Only	Read Only
1	Read / Write	Read / Write
2	Read / Execute	Read / Execute
3	Read / Write / Execute	Read / Write / Execute
4	Execute Only	Execute Only
5	Read Only	Read / Write
6	No Access	Read / Execute
7	No Access	Read / Write / Execute

4.1.1.1 TLB Look-up

A virtual address to be translated by the CY7C604/605 is compared against each entry in the TLB as shown in *Figure 4–3*. If a TLB hit (match) occurs and access-level requirements are satisfied, then the TLB outputs the physical address and the cacheable bit. This physical address is output by the CY7C604/605 onto the Mbus (see Section 4.12, Physical Bus) if the cache has been disabled or if the page is non-cacheable. If the cache controller is enabled and a cache miss occurs, the physical address of the cache miss is used to access the new cache line in main memory for cache line replacement.

The short translation bits specify a linear address mapping range of 4-kbytes, 256-kbytes, 16-Mbytes, or 4-Gbytes for each TLB entry. The short translation bits also determine the index fields of the virtual address that are matched with the TLB entry to determine a TLB hit. For a TLB entry with a linear address range of 4 kbytes, index fields 1, 2, and 3 of the virtual address and the context register are compared against the TLB entry. A TLB entry with a 256-kbyte linear addressing range requires a match of the context and of the index 1 and index 2 fields. A 16-Mbyte linear addressing range requires a match of the index 1 field and the context. The 4-Gbyte linear address mapping requires only a context match to produce a TLB hit.

If the modified bit is not set in a TLB entry, write or load-store accesses that match the TLB entry and meet all access-level requirements will cause a table walk. (see Table Walk, Section 4.1.2.) If the modified (M) bit is not set for a write access, then the table walk sets the modified bit in the page table pointer entry for the memory region. This information is used by an operating system to ensure that modified regions of memory are stored in alternate memory media (typically a disk drive) before they are overwritten during memory page swap operations.

If there is a matched entry, but the access-level requirements are not satisfied, then a synchronous address fault exception is asserted. *Context number matching is not required if the access-level field (ACC) is either 6 or 7 and the memory access is a supervisor mode access (ASI = 9,B H).* This produces a means of mapping the kernel of an operating system into the same virtual address locations of every context.

The TLB ignores access-level checking during MMU probe operations, copy-back flush cycles, and alias detection cycles.

4.1.1.2 TLB Entry Replacement and Locking

The CY7C604/605 supports a random replacement algorithm to replace a TLB entry during TLB miss processing. The random replacement is implemented by using a counter to point to one of the 64 TLB entries. A 6-bit replacement counter (RC) is incremented by one during each clock cycle to point to one of the TLB entries as shown in *Figure 4–4*. Upon encountering a TLB miss, the CY7C604/605 uses the counter value to address a TLB entry to be replaced. The hardware automatically replaces an entry pointed to by the replacement counter (RC) during TLB miss processing.

Locking of TLB entries is supported with a 6-bit initial replacement counter (IRC). The number of locked entries is specified by setting the value of the IRC. The value of the IRC is used as a counter preset for the replacement counter. Once the replacement counter (RC) reaches the maximum value, it wraps to the initial replacement counter (IRC) value. Upon power-on reset ($\overline{\text{POR}}$), both the IRC and RC are initialized to zero.

Locked TLB entries can be changed (read/write) only through the alternate space load/store instructions with ASI = 6 (see Diagnostics Support, page 4–43.) These locked entries will not participate in the random replacement algorithm during TLB miss processing. The IRC should be initialized to the number of lockable entries by writing to the TLB replacement control register (TRCR).

Programming Note: When changing the IRC, the RC should also be written with the same value. This ensures that the RC is always pointing to the replacement area of the TLB.

4.1.1.3 TLB Entries (TLBEs)

Both the virtual and physical sections of each TLB entry can be accessed (read/write) through single load or store instructions. Software has the option to write and to lock high-usage or high-priority TLB entries to optimize system response time (Refer to MMU TLB Entries, page 4–43, for more details.)

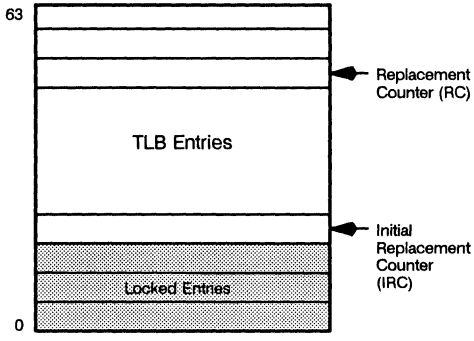


Figure 4–4. TLB Replacement and Locking

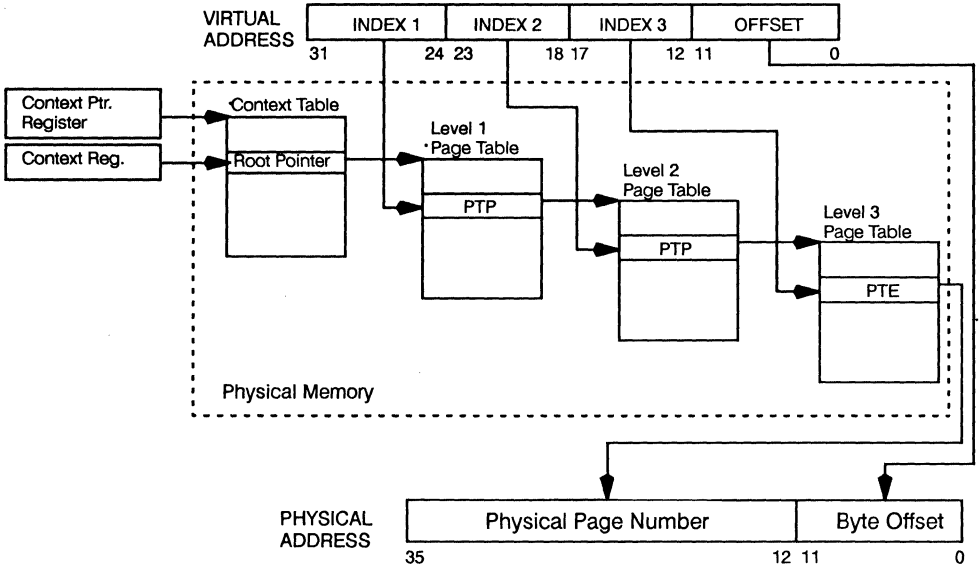


Figure 4–5. Four-Level Table Walk (4-kbyte Addressing)

4.1.2 Table Walk

The CY7C604/605 supports tree-structured, 4-level table walk processing (including the context table level) as shown in *Figure 4–5*. All of the virtual to physical address mapping tables are located in physical memory. These tables are accessed in the case of a TLB miss or of a write or load-store operation with a cleared M (modified) bit in the TLB entry.

Upon starting a table walk, the CY7C604/605 walks through a series of tables to find a page table entry (PTE). The page table entry contains the physical page number, the access-level permission, cacheable, modified, and referenced bits for the address generating the table walk. (Refer to page 4–10 for information on PTEs.) A table walk caused by a TLB miss causes the CY7C604/605 to update an available TLB entry with the new PTE. A table walk forced by a write or load-store operation on an unmodified memory region causes the CY7C604/605 to set the modified bit in the page table entry and in the TLB entry.

The table walk begins with an access to the context table. The CY7C604/605 uses the context table pointer register (CTPR) as a base register to point to the beginning of the context table. The context register (CXR) is used as an index register to point to the table entry. The upper twenty-two bits of the CTPR are concatenated with the twelve bits of the CXR to provide a 36-bit address. The lowest two bits of all addresses pointing to a page table entry or pointer are always forced to zero.

If a page table entry (PTE) is found at the context table level, the table walk terminates. The PTE is stored in the TLB and, if necessary, the modified bits and/or the reference bits are updated. If a page table entry is not found, then a Page Table Pointer (PTP) must be located at the address pointed to in the context table. (See page 4–9 for more information on PTPs and PTEs.) The page table pointer is used as the base address for the next table.

If a PTE is not found, the table walk continues by accessing the level 1 table using the PTP as a base address and the index 1 field from the virtual address as an index pointer. It is possible to find a PTE instead of a page table pointer at any level during the table walk. The index 1 field (virtual address (31:24)) is used to select an entry in the level 1 table. If a page table entry is not found at this location, a page table pointer stored at this entry is used as the base address for the level 2 table. The index 2 field (virtual address (23:18)) is used to select an entry in the level 2 table. The entry in the level 2 table, if not a page table entry, is used as the base address for the level 3 table. The index 3 field (virtual address (17:12)) is used to select an entry in the level 3 table, which must be a page table entry.

If a page table entry is not found after the level 3 table access, a synchronous fault exception is asserted. A synchronous fault exception is also generated if an invalid entry is found at any level of the table walk. The table walk terminates immediately when an exception is generated.

The level at which the table walk terminates is related to the size of addressing region associated with the entry. A table walk that finds its page table entry in the context table corresponds to an addressing region of 4-Gbyte. Each level deeper into the table walk corresponds to a smaller size of address mapping. A PTE for a 16-Mbyte addressing region will be found in a level 1 table. A 256-kbyte PTE will be found in a level 2 table. Only an addressing region of 4 kbytes will require a table walk of four levels to find the correct page table entry).

An example of a table walk for a 256-kbyte linear address space is shown in *Figure 4–6*. The value of the short translation bits are related to the level at which the table walk terminates. The short translation bits decrease from (1,1) for a table walk with a context table PTE to (0,0) for a table walk with a level 3 table PTE. (Refer to *Table 4–1.)*

Each table walk access is performed as a non-burst transaction on the Mbus (physical bus). The Mbus busy ($\overline{\text{MBB}}$) signal is asserted from the beginning of the table walk to the end of the table walk process. This locks the Mbus and prevents another bus master from gaining the bus until the table walk is complete. The MLOCK bit in the address phase of the Mbus transaction will be set (refer to Section 4.12.5), indicating a locked transaction. During these transactions, the C bit in the SCR register is output on the MC signal of the Mbus. There will be write transactions during the table walk only if the reference bit (R) and/or the modified bit (M) has to be set in the page tables.

If there is an invalid page table entry (ET = 0) at any level, an invalid address error exception occurs and the table walk terminates immediately. If an external bus error occurs, a reserved entry (ET = 3) is detected, or a PTP entry is detected in level 3, a translation error exception occurs, and the table walk terminates immediately. If an access-level protection occurs, the table walk is terminated and a protection/privilege violation exception is asserted.

The reference bit (R) and the modified bit (M) are set according to the access type. In order to record the exceptions in the synchronous fault status registers properly, the table walk hardware must indicate the fault type and the level at which the fault occurred (Refer to Section 4.9 for more details). For access-level checking during the table walk, load-store cycles are treated as write cycles. The table walk state diagram is shown in *Figure 4–10*.

During MMU probe operations, copy-back flush cycles, and alias detection cycles, the table walk controller ignores access-level checking.

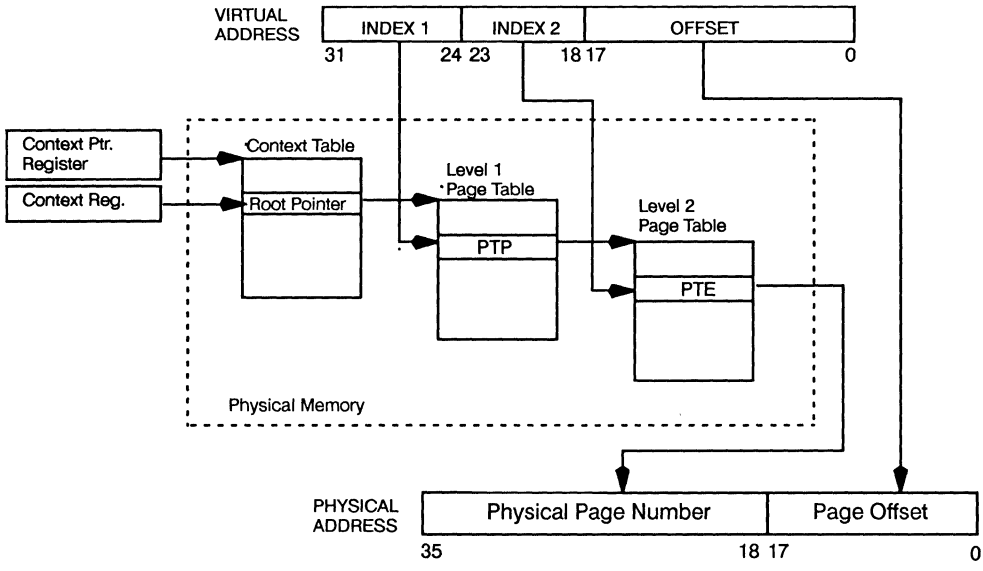


Figure 4–6. Three-Level Table Walk (256-kbyte Addressing)

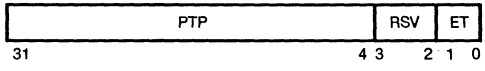
4

4.1.3 Page Table Pointer (PTP)

A Page Table Pointer (PTP), as shown in *Figure 4–7*, may be found in the context, level 1, or level 2 tables. The PTP is used in conjunction with an index field of the virtual address to point to the next level of table in a table walk. The PTP found at the context level is called the root pointer. Bits 31 through 6 of the root pointer are output on bits 35 through 10 of the Mbus (MAD(35:10)) and are concatenated with the eight bits of the index 1 field of the virtual address to access the entry in the first level page table. (Refer to *Figure 4–6*.) The lowest two bits of the address are equal to zero, as addressing is aligned on word boundaries.

Similarly, bits 31 through 4 of the PTP in level 1 or level 2 tables are output on bits 35 through 8 of the Mbus (MAD(35:8)). The index 2 or index 3 fields are concatenated with the PTP to yield the address of the next table entry. The ET field (*see Table 4–3*) describes the entry type: invalid, page table pointer, or page table entry.

In order to reduce the penalty for a TLB miss, the root pointer from the context level table and two PTPs from the level 2 table are cached in the PTP cache. The PTPs from the most recent data and instruction misses using a four-level table walk are cached for later use. The TLB checks the PTP cache upon a TLB miss, and uses the cached PTP to access the level 3 table if an entry matches the access. The PTP cache is discussed in more detail in Section 4.1.5.



PTP = Page Table Pointer ET = Entry type
RSV = Reserved

Figure 4–7. Page Table Pointer

Table 4–3. Page Table Entry Type

ET	Entry Type
0	Invalid
1	Page Table Pointer
2	Page Table Entry
3	Reserved

4.1.4 Page Table Entry (PTE)

The Page Table Entry (PTE) is shown in *Figure 4–8* and may be found in the context, level 1, level 2 or level 3 tables. The page table entry contains the address mapping information used by the MMU to translate a range of virtual addresses to physical addresses.

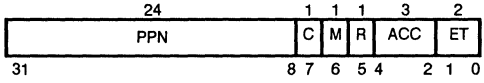
The level of the table in which the PTE is found is related to the addressing range associated with the PTE. A PTE found in the context table will map a 4-Gbyte addressing region. A level 1 PTE will map a 16-Mbyte addressing region. A level 2 PTE corresponds to a mapping region of 256 kbytes. A level 3 PTE maps a 4-kbyte addressing region.

The addressing region mapped to the PTE determines how many bits in the PPN field of the PTE are used to form the physical address. PTE(31:28) from a context level table PTE are output on bits 35 through 32 of the physical address bus (MAD(35:32)) to offer 4-Gbytes of linear address mapping. Similarly, PTE(31:20) from a level 1 table PTE are asserted on bits 35 through 24, and provides 16 Mbytes of linear addressing. PTE(31:14) from a level 2 table PTE are asserted on bits 35 through 18, and PTE(31:8) from a level 3 table PTE are asserted on bits 35 through 12 to offer 256K and 4 kbytes of linear address mapping, respectively. The remainder of the PPN field not used for address translation is reserved. The remaining physical address bits not specified by the PPN field are supplied from the virtual address.

The ACC bits describe the access-level and privilege protection assigned to the PTE. These bits are described in *Table 4–2*. The referenced (R) bit is set in the PTE when the CY7C604/605 has read the value of the PTE in a table walk. The CY7C604/605 automatically sets this bit upon access of the PTE. The modified (M) bit is set upon a write or load-store access of a previously unmodified memory region. This information is commonly used by an operating system to flag regions of memory that must be written to mass storage before being replaced by another memory page.

The cacheable (C) bit indicates whether or not the memory region addressed by the PTE is allowed to be cached. This bit may be used to prevent shared memory pages from being cached, thereby avoiding potential aliasing problems. It also may be used to prevent caching of memory mapped input/output devices.

The ET field, illustrated in *Table 4–3*, is used by the CY7C604/605 to determine the type of table entry during a table walk. The ET field is set to 2 to indicate a PTE, and is set to 1 to indicate a PTP. If the CY7C604/605 encounters a table entry with ET = 0 during a table walk, the CY7C604/605 generates an invalid address error. The CY7C604/605 generates a translation error if ET = 3 (reserved) is encountered in a table entry during a table walk.



PPN = Physical Page Number
C = Cacheable bit
M = Modified bit
R = Referenced bit
ACC = Access protection bits
ET = Entry type

Figure 4–8. Page Table Entry Format

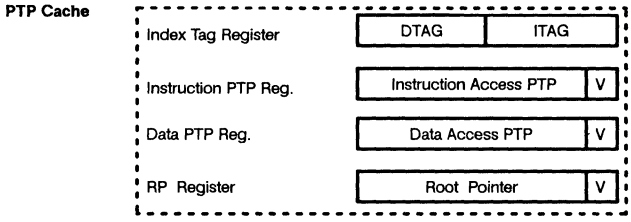


Figure 4–9. Page Table Pointer Cache

4.1.5 Page Table Pointer Cache (PTPC)

In order to reduce the penalty for a TLB miss, the CY7C604/605 supports a three-PTP entry page table pointer cache. The Page Table Pointer Cache (PTPC) caches the most recently used PTPs, as shown in *Figure 4–9*. The three entries are: the Root Pointer Register (RPR), the Instruction access level 2 PTP (IPTP), and the Data access level 2 PTP (DPTP). The IPTP and DPTP registers are referenced by a fourth register, the Index Tag Register (ITR). These entries are cached during table walk processing for a TLB miss.

The root pointer for a context is cached in the RPR. The RPR remains valid until the ConteXt Register (CXR) or the Context Table Pointer Register (CTPR) value is changed. The instruction access PTP register contains the latest level 2 PTP for an instruction access. This PTP is cached from the last TLB miss requiring a four-level table walk for an instruction access. The Data Access PTP Register contains the latest level 2 PTP for a data access. This PTP is also cached from the last four-level table walk for a data access. The IPTP and DPTP registers are invalidated when another table walk that accesses level 3 of the page tables is forced for an instruction or data access or a TLB flush. They also are invalidated when either the context register or context pointer register is changed. Refer to page 4-38 for more information on these registers.

Figure 4–9 illustrates the PTPC. The index tag register (ITR) is used to reference the IPTP and DPTP registers. The ITAG and DTAG fields of the index tag register are used by the CY7C604/605 to compare against an address generating a TLB miss. Once a level 2 page table pointer is cached for an instruction or a data access, the same PTP is used if the index 1 and index 2 fields of the virtual address match the index 1 and index 2 tag fields of the ITAG or DTAG. The IPTP and DPTP registers are updated only if a TLB miss occurs that does not match the ITAG or DTAG and also generates a table walk that accesses level 3 of the page tables.

Once a root pointer is cached for a particular context, the same root pointer can be used as long as the context is not changed. If the table walk finds a context level or level 1 or level 2 entry PTE (i.e., is not a four-level table walk), then no caching of level 2 pointers is performed.

Whenever the context is changed, the entire PTPC (all three entries) is invalidated. Upon power-on reset, all the PTPC entries are invalidated. When the ContexT Pointer Register (CTPR) is written, the page table pointer cache is invalidated by clearing the V bits in the IPTP, DPTP, and RPR registers. Any TLB flush invalidates the IPTP and DPTP registers of the PTP Cache.

The IPTP and DPTP registers are not updated during table walks caused by address alias detection and copy-back flush cycles.

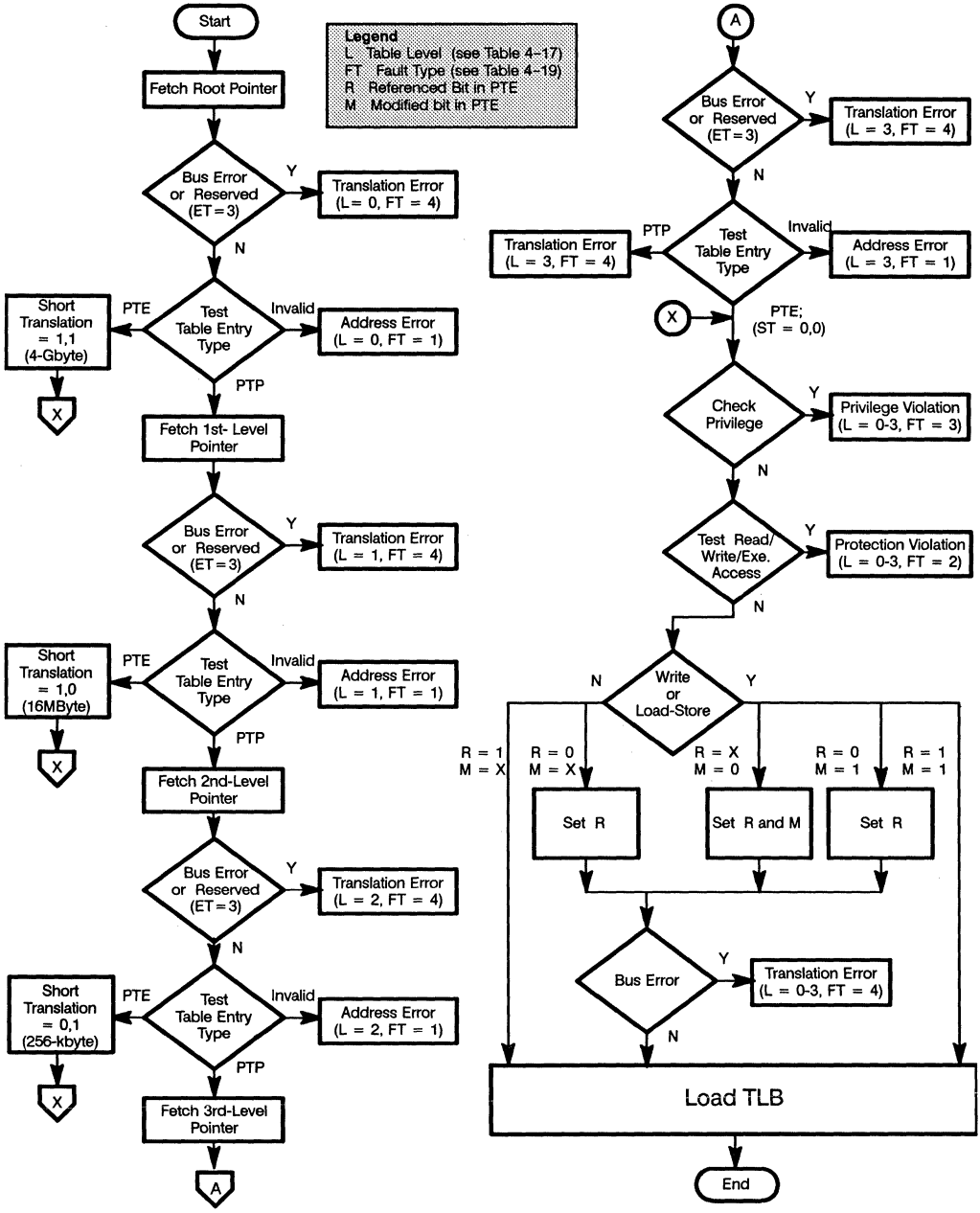


Figure 4–10. Table Walk Algorithm

4.2 MMU Operation Modes

This section describes the different modes of operation of the CY7C604/605, the conditions under which they occur, and what information is reflected on the pins. The operation mode for the MMU (and cache controller) is controlled by the system control register (SCR). Please refer to Sections 4.4.1 and 4.4.2 for further information on the SCR.

The following symbols are used throughout the chart:

MC(MAD(43))	Mbus Cacheable indicator signal (Refer to Pin Definitions, Section 4.10)	**UN**	Unassigned ASI
MBL(MAD(45))	Mbus Boot/Local indicator signal (Refer to Pin Definitions, Section 4.10)	**RES**	Reserved ASI and ASI defined but not implemented (see *Table 4-15*)
ASI	Address Space Identifier code for current access from CY7C601	**PA**	Physical Address
SCR[C]	Cacheable bit of SCR	**VA**	Virtual Address
X	Not Defined or Don't Care	**BM, ME, CE**	Bits in System Control Register (SCR)
		PTE[C]	Cacheable bit of page table pointer

Table 4–4. MMU Operation Modes

MMU Operation Modes									
Mode	**Conditions**				**Results**				
	ASI	**BM**	**ME**	**CE**	**Physical Addressing**		**Caching**	**MC**	**MBL**
Local	1	X	X	X	PA<35:32> = 0	PA<31:0> = VA<31:0>	Not Cached	0	1
UN, RES	UN, RES	X	X	X	Ignore	Ignore	Ignore	N/A	N/A
By-pass	20-2F	X	X	X	PA<35:32> = ASI<3:0>	PA<31:0> = VA<31:0>	Not Cached	0	0
Pass-Through	8,9,A,B	0	0	X	PA<35:32> = 0	PA<31:0> = VA<31:0>	Not Cached	SCR [C]	0
Boot (Instr. access)	8,9	1	X	X	PA<35:28> = FF	PA<27:0> = VA<27:0>	Not Cached	SCR [C]	1
Boot (Data access)	A,B	1	0	X	PA<35:32> = 0	PA<31:0> = VA<31:0>	Not Cached	SCR [C]	1
Translation 1 (Data Access and Cache Disabled)	A,B	X	1	0	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Not Cached	PTE [C]	0
Translation 2 (Data Access and Cache Enabled)	A,B	X	1	1	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Cached if PTE[C] =1	PTE [C]	0
Translation 3 (Instruction Access and Cache Disabled)	8,9	0	1	0	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Not Cached	PTE [C]	0
Translation 4 (Instruction Access and Cache Enabled)	8,9	0	1	1	PA<35:12> = PTE<31:8>*	PA<11:0> = VA<11:0>*	Cached if PTE[C] =1	PTE [C]	0

* Concatenation field sizes vary depending upon the short translation (ST) bits to provide 4G, 16M, 256K, 4 kbytes of linear address mapping. Refer to Section 4.1.1 for further details.

The MMU provides three types of operating modes: boot modes, direct-access modes, and translation modes. Two boot modes are defined for the MMU, one for data accesses, and one for instruction accesses. The boot modes force the upper eight bits of the physical address to FF H for instruction accesses. The upper four bits are forced to zero for data accesses. These two modes also assert the Mbus Boot mode/Local indicator (MBL) signal. This signal can be used in the system to enable a memory region used only for system boot and configuration. This allows the system a secure method of accessing bootstrap ROM and shadow RAM separate from the main memory space.

The direct access modes allow the integer unit to access the main memory without address translation by the MMU. These modes include: local, by-pass, and pass-through. Local mode enables the MBL signal and forces the upper four bits of the physical address to zero. The lower 32 bits of the physical address are supplied directly from the virtual address bus. This mode allows the integer unit to access the boot mode memory (if supported in the system) without changing the state of the System Control Register (SCR). Local mode is enabled by using a load or store alternate instruction with ASI* = 1 H.

Bypass mode allows complete access to the main memory space. MBL is not enabled, and the lower four bits of the ASI are used as the upper bits of the physical address. The remaining 32 bits are supplied directly from the virtual address bus. The state of the SCR does not have to be modified. This mode is mapped into the ASI space as ASI = 20 - 2F H.

Pass-through mode describes the CY7C604/605 operation with the MMU disabled. The upper four address bits of the physical address are forced to zero. The MBL signal is not asserted. This mode does not require non standard ASI assignments (i.e., ASI = 8,9,A,B H), but the boot mode (BM) and MMU enable (ME) bits of the SCR must be cleared.

The translation modes are considered to be the normal operating modes of the MMU. This group includes four modes of translation operations: Translation 1 - 4. Translation 1 and 2 are the non-cached and cached data access modes. Translation 3 and 4 are the non-cached and cached instruction access modes. The cached and non-cached modes are identical in results for both data and instruction accesses, with the exception that the data access modes ignore the Boot Mode (BM) bit of the SCR. This feature allows the system to enable the MMU for data accesses, yet still access instructions from the boot memory space without changing the BM bit.

* The SPARC architecture reference supports the concept of Address Space Identifiers (ASI), which provide an extension of the standard addressing space. These bits are used to enable special addressing modes, or to provide access to registers and other features of the CY7C604. Refer to section on ASI and Register Mapping for more information.

4.2.1 MMU Flush and Probe Operations

4.2.1.1 Flush Operations

The flush operation allows software invalidation of selected entries in the TLB. TLB entries are flushed by executing a Store Alternate ASI instruction using ASI = 3 H and supplying a virtual address in the format shown in *Figure 4–11*. The context number is given by the context register (CXR). All TLB entries that match the virtual address, context, and TLB flush type will be flushed (invalidated) simultaneously. The flush type is specified in bits 11-8 of the virtual address for the flush operation.

The CY7C604/605 supports five different types of TLB flushing operations. These types are: page, segment, region, context, and entire flush. The five types of flushing are listed in *Table 4–5*, and define the address comparison required to match a TLB entry for flushing. The Short Translation (ST) bits in the TLB entries are ignored for TLB matching. All TLB entries matching the compare criterion of the flush type are invalidated, including those locked by the IRC.

Virtual Address Format:

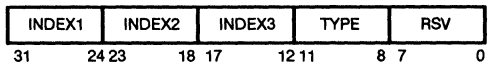


Figure 4–11. MMU Flush Address Format

Table 4–5. TLB Entry Flushing

Type	Flush	Compare Criterion
0	Page	Context (or ACC = 6, 7), Index 1, Index 2, and Index 3
1	Segment	Context (or ACC = 6, 7), Index 1, and Index 2
2	Region	Context (or ACC = 6, 7), and Index 1
3	Context	Context (user pages with ACC = 0 to 5)
4	Entire	None
5 to F	Reserved	

4.2.1.2 Probe Operation

The probe operation allows testing the TLB and page tables for a PTE entry corresponding to a virtual address. The operation is initiated by executing a load alternate ASI instruction with ASI = 3 H, the appropriate virtual address, and the context number. The context is specified by the context register. Upon starting a probe operation, the TLB is probed first. If there is a TLB hit, it returns the 32-bit physical section of the matched entry. The returned entry fields are formatted such that it is identical to a PTE (see Section 4.1.4 on page 4–10, for PTE format information). If a matching entry could not be found in the TLB, a table walk is started and an appropriate 32-bit value (PTE) is returned and loaded into the TLB.

A probe operation causes the Reference bit (R) to be set in the PTE by means of a table walk. When a probe operation hits the TLB, the R bit is always returned as set.

The context register and access-level protection checking are ignored for TLB matching and during the probe operation table walk. The table walk hardware checks for invalid address error and translation error exceptions and records appropriate fields in the SFSR register as in the normal table walk process. If a bus error occurs or an invalid or reserved entry is detected during the table walk, a 32-bit zero value is returned as status. If a zero value is returned, the UC, TO, BE, L, and FT fields of the SFSR are updated accordingly, but the operation does not cause an exception to the CY7C601.

4.3 CY7C604 / CY7C605 Cache Controllers

The differences between the CY7C604 and CY7C605 become evident in the features of their respective cache controllers. The CY7C604 cache controller is designed for a uniprocessor system, and provides cache locking for real-time system support. The CY7C605 cache controller is enhanced to accommodate the requirements of a multiprocessing system. The CY7C605 provides bus snooping and a Futurebus style of cache coherency protocol. The CY7C605 is designed to provide high visibility into its cache operations from the perspective of the shared physical bus in order to simplify support by a secondary cache system. The following sections discuss the CY7C604 and CY7C605 cache controllers. Sections specific to the CY7C604 or CY7C605 are marked with that part number only. Sections applying to both the CY7C604 and the CY7C605 are marked "CY7C604/605."

4.3.1 CY7C604/605 Cache Modes

The CY7C604/605 virtual cache can be programmed for either write-through with no write allocate or copy-back with write allocate. The two cache modes differ in how they treat cache write accesses. Write-through cache mode causes write hits to the cache to be written to both cache and main memory. Write-through write cache misses only update main memory and invalidate the cache tag, but do not modify the cache.

A write access in copy-back mode only modifies the cache. The writing of the modified cache line to main memory is deferred until the cache line is no longer required. Copy-back cache mode has the advantage of reducing traffic on the system bus. Bus traffic is reduced since all updates to memory are deferred and are performed subsequently only as absolutely required. In addition, all such data transfers are made utilizing the more efficient burst mode. The following describes the two cache modes in detail.

4.3.1.1 CY7C604/605 Write-Through Mode with No Write Allocate

For write-through cache mode, write access cache hits cause both the cache and main memory to be updated simultaneously. A write access cache miss causes only main memory to be updated (no write allocate). The selected cache line is invalidated for a write access cache miss. Write-through caching mode normally requires a processor to delay during a write miss while the data is written to main memory. The CY7C604/605 provides write buffers to prevent this delay in most cases. The write buffers store the write access and write the data to main memory as a background task. (Refer to page 4-31 for further information on the write buffers.)

During read access cache hits, the cached data is read out and supplied to the CY7C601. In the case of a read access cache miss, a cache line is fetched from main memory to load into the cache and the required data is supplied to the CY7C601.

4.3.1.2 CY7C604/605 Copy-Back Mode with Write Allocate

When the cache is configured for copy-back mode, only the cache is updated on write access cache hits (i.e., main memory is not updated). The modified bit of the cache tag for the cache line is set on a copy-back write access (write hit or after a write miss is corrected). During write access cache misses, if the selected cache line is clean (not modified), a cache line is fetched from main memory to load into the cache and only the cache is updated. If the selected cache line is modified, the selected cache line is flushed out to update main memory. The CY7C604/605 simultaneously fetches the new cache line from main memory and stores it into the read buffer as it flushes the modified cache line from the cache and stores it into its write buffer. After the modified cache line has been flushed, the CY7C604/605 writes the modified cache line out of its write buffer into main memory while the new cache line is stored into the cache memory from the read buffer.

During read access cache hits, the cached data is read out and supplied to the CY7C601. During read access cache misses, if the selected cache line is clean (not modified), a cache line is fetched from main memory to load into the cache. If the selected cache line is modified, the selected cache line is flushed out to the CY7C604/605 write buffer, and a new cache line is fetched from main memory and stored into the read buffer. The new cache line is then stored in the cache from the read buffer, while the modified cache line stored in the write buffer is written out to main memory.

4.3.2 CY7C604 Cache Controller

The cache controller provides cache memory access control for a 64-kbyte direct mapped virtual cache. The cache controller is designed to use two CY7C157 Cache RAMs for the cache memory. These cache RAMs are 16-kbyte x 16 SRAMs with on-chip address and data latches and timing control. The CY7C601 cache can be expanded to a maximum of 256 kbytes by adding additional groups of one CY7C604 and two CY7C157s. Using multiple CY7C604s to expand the cache is referred to as a multichip configuration for the CY7C604, and is described in the Section 4.5, Multichip Configuration.

The cache is organized as 2048 cache lines of 32 bytes each. The CY7C604 has 2048 cache tag entries on-chip, one tag entry for each cache line. Addressing for the virtual cache is provided directly from the virtual address bus. The virtual address field VA(15:5) selects one of the 2048 lines of the cache. This address field also selects one of the corresponding cache tag entries in the CY7C604. A cache hit occurs when the upper sixteen bits of the virtual address and the context register match with the virtual address and context stored in the selected cache tag entry. The lowest five bits of the virtual address bus (VA(4:0)) select one or more of the 32 bytes in the cache line. Cache data replacement is always performed by replacing cache lines.

The cache is designed to provide data with every read access asserted on the virtual bus, regardless of the cache controller. The CY7C604 controls cache read access by holding the CY7C601 with $\overline{\text{MHOLD}}$ if a cache hit is not detected by the cache controller. The cache controller then reads the new cache line from main memory, and supplies the correct data to the CY7C601. After the correct data is latched into the CY7C601 by strobing the $\overline{\text{MDS}}$ signal, the CY7C601 is released and execution proceeds normally.

Writes to the cache are controlled by the CY7C604, which decodes the lowest two bits of the virtual address, the SIZE(1:0) signal, and checks for a cache hit to enable the correct cache byte write enable signals. If a cache write hit occurs, the CY7C604 decodes the correct $\overline{\text{CBWE}}$ signals for the write access, and outputs these to the CY7C157 Cache RAM write enables. If the cache mode is set to write-through (see Cache Modes, Section 4.3.1), the write data is also written to main memory. If a write cache miss occurs for write-through cache mode, the data is written to main memory and the cache is not updated. If the write cache miss occurs during copy-back cache mode (see *Figure 4-14*) and the selected cache line is not modified, the missed cache line is fetched from main memory. If a write cache miss occurs during copy-back mode and the selected cache line is modified, the CY7C604 simultaneously flushes the modified cache line into the write buffers while it fetches the new cache line from main memory. After the cache line has been replaced, the write access is enabled by the CY7C604. The modified cache line is written to main memory from the write buffers as a background task.

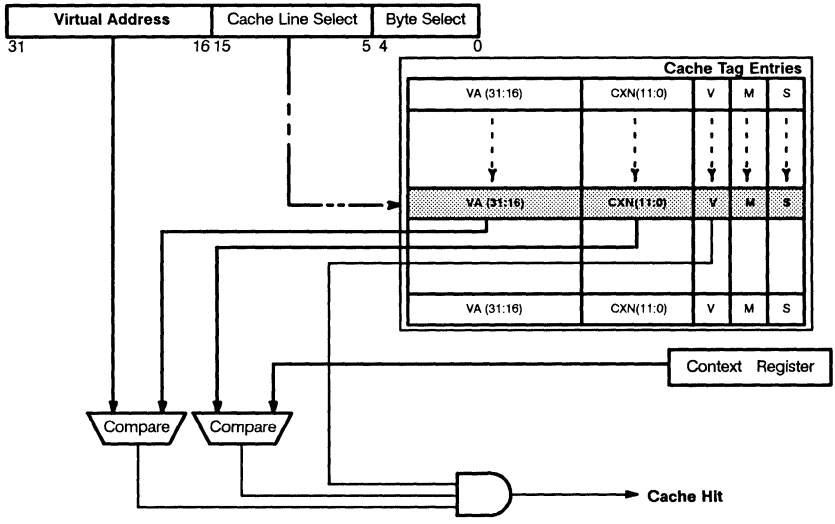


Figure 4–12. CYC7604 Cache Tag Comparison

4.3.2.1 CY7C604 Cache Tag

The CY7C604 features 2048 direct-mapped cache tag entries, as shown in *Figure 4–12*. The on-chip cache tag and the TLB are accessed simultaneously. Each entry in the cache consists of 16 bits of virtual address (VA(31:16)), a 12-bit context number (CXN(11:0)), one valid bit (V) and one modified bit (M). The valid bit (V) is set or cleared to indicate the validity of the cache tag entry. The modified bit (M) of a cache tag entry is set during copy-back mode after a write access to the cache line. This indicates that the cache line has been modified. The modified bit has no meaning for write-through cache mode. The cache line select field (VA(15:5)) is used to select a cache line entry and its corresponding cache tag entry. The address field VA(31:16) and context register are compared against the virtual address and the context fields of the selected cache tag entry. If a match occurs, then a cache hit is generated. If a match is not found, then a cache miss is generated. To complete an access successfully, both the cache tag and the TLB must be hit with appropriate access-level permission. Upon Power-On Reset ($\overline{\text{POR}}$), all cache tag entries are invalidated (all V bits are cleared).

A Supervisor bit (S) is included in the cache tag entry. For cache tag entries which are accessible by the supervisor only (access-level field 6 or 7), the S bit is set. During a cache tag look up, if the access is supervisor mode and the the S bit is set, the context number comparison is ignored and the context match is forced. This operation is similar to a TLB look up with access-level field set to either 6 or 7.

4.3.2.2 CY7C604 Address Aliasing

Two or more virtual addresses mapped to the same physical address is known as *aliasing*. This must be detected to maintain data consistency in a virtual cache system. The SPARC reference system software convention permits the use of aliases in address spaces that are modulo with respect to the system's underlying cache size. In order to allow the efficient caching of physical memory pages where such aliases may occur, the CY7C604 supports automatic address aliasing protection.

The CY7C604 tests for address aliasing during copy-back read or copy-back write cache misses or during write-through read misses. The MMU must be enabled to allow the CY7C604 to test and correct address aliases.

To detect address aliasing, the virtual address of the selected cache tag entry is translated through the MMU. The translated physical address is compared with the physical address of the missed cache access. If the physical address of the selected cache tag entry and the physical address of the cache miss match, then address aliasing is detected.

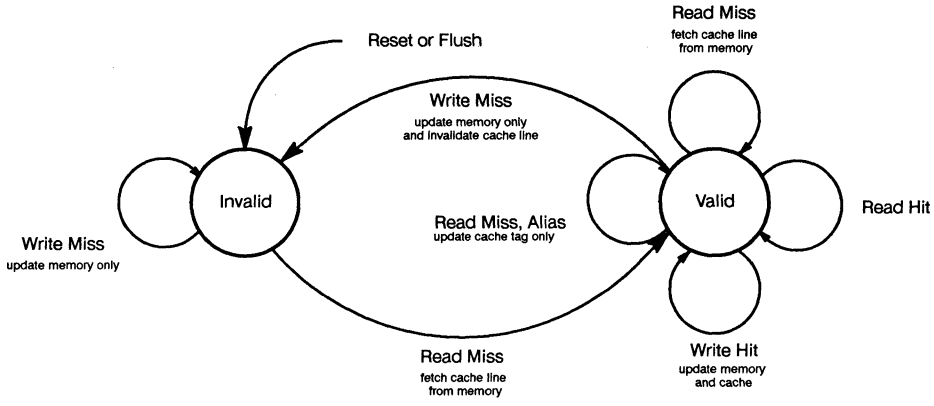


Figure 4–13. CY7C604 Write-Through with No Write Allocate

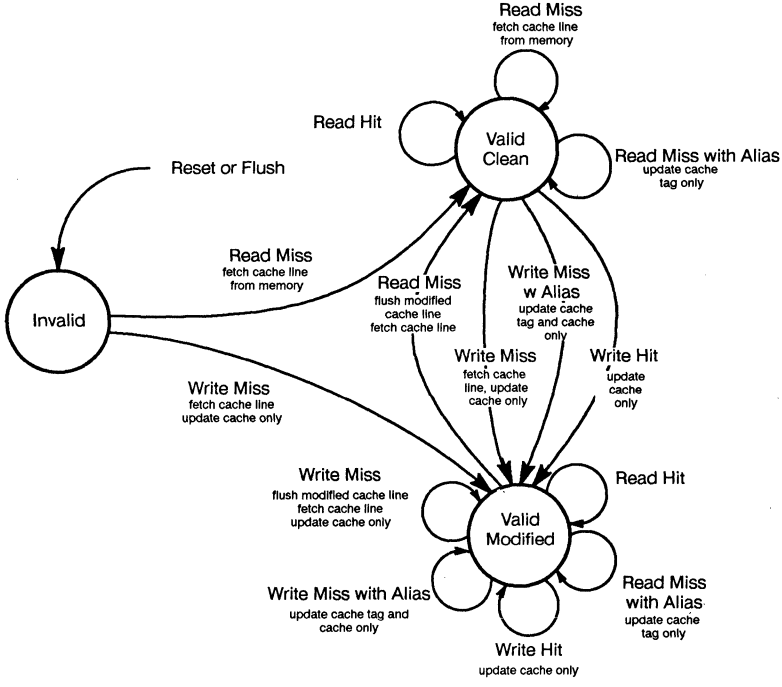


Figure 4–14. CY7C604 Copy-Back with Write Allocate

The SPARC system software convention ensures that the aliasing maps to the same cache line address for a particular CY7C604. Coupled with this convention, the cache controller hardware automatically prevents any existence of address aliases in the virtual caches.

Aliasing is checked during a cache miss. If detected, an alias is corrected by updating the selected cache tag entry with the new virtual address. The CY7C604 then halts the cache miss processing and provides an access to the cache, as with a cache hit. If no alias is detected, the cache miss processing proceeds normally. The state diagrams for write-through and copy-back cache modes with alias detection and correction are illustrated in *Figure 4–13* and *Figure 4–14.*

In copy-back mode, address aliasing is checked during a read- or a write-access cache miss. For an alias detected during a read-access cache miss, the selected cache tag entry is updated with the virtual address that caused the cache miss. The cache miss processing is halted, and the CY7C601 is supplied with data from the cache.

If an address alias is detected during a write access cache miss, the selected cache tag entry is updated with the new virtual address that caused the cache miss. The modified bit is set if it was not set previously. The cache miss processing is halted, and the cache write access is enabled.

In write-through mode, address aliasing is checked only on read-access cache misses. If an address alias is detected on a read-access cache miss, the old cache tag entry is replaced with the new virtual address. The cache miss is halted, and the cache supplies the data requested.

In write-through cache mode, address aliasing is not checked during write-access cache misses. In order to avoid potential address aliasing, the selected cache line is invalidated. Address aliasing is not checked in this case in order to avoid unnecessary performance degradation.

To detect address aliasing, the selected cache line address is translated through the TLB. Protection checking is ignored during this translation. The translation may occasionally cause a TLB miss. If this happens in a write-through read miss case, the alias checking and the TLB miss are ignored. In a copy-back read miss or a write miss when the selected cache line is clean, alias checking and TLB miss processing are ignored. To provide data consistency, the table walk is performed in order to detect address aliasing in a copy-back read miss or a write miss when the selected cache line is modified.

4.3.2.3 CY7C604 Cache Lock

The CY7C604 supports a cache lock mechanism that allows the system to lock all entries in the cache. This feature is provided to allow deterministic response times for real-time systems. The cache lock function affects only cache miss operations, since it locks out cache line replacement of valid entries. Since alias detection is not enabled, shared memory pages must be declared as non-cacheable when the cache is locked . The following description summarizes each case in detail:

a. *Write-through read miss and selected entry is invalid:* A new cache line is fetched from main memory to load into the cache and the requested data is supplied to CY7C601 as in normal operation mode.

b. *Write-through read miss and selected entry is valid:* The requested data is obtained from main memory as a non-burst transaction on the Mbus and supplied to the CY7C601, but is not loaded into the cache.

c. *Write-through write miss:* The selected cache line is invalidated in order to prevent data inconsistency due to potential address aliasing.

d. *Copy-back read miss and selected entry is invalid:* A new cache line is fetched from main memory to load into the cache and the requested data is supplied to CY7C601 as in a normal operation.

e. *Copy-back read miss, selected entry is valid:* The requested data is obtained from main memory as a non-burst transaction on the Mbus and supplied to the CY7C601, but is not loaded into the cache.

f. *Copy-back write miss and selected entry is invalid*: A new cache line is fetched from main memory to load into the cache and the CY7C601 data is stored in the cache as in a normal operation.

g. *Copy-back write miss and selected entry is valid:* The CY7C601 data is stored in the main memory as a non-burst transaction on the Mbus, but the cache is not updated.

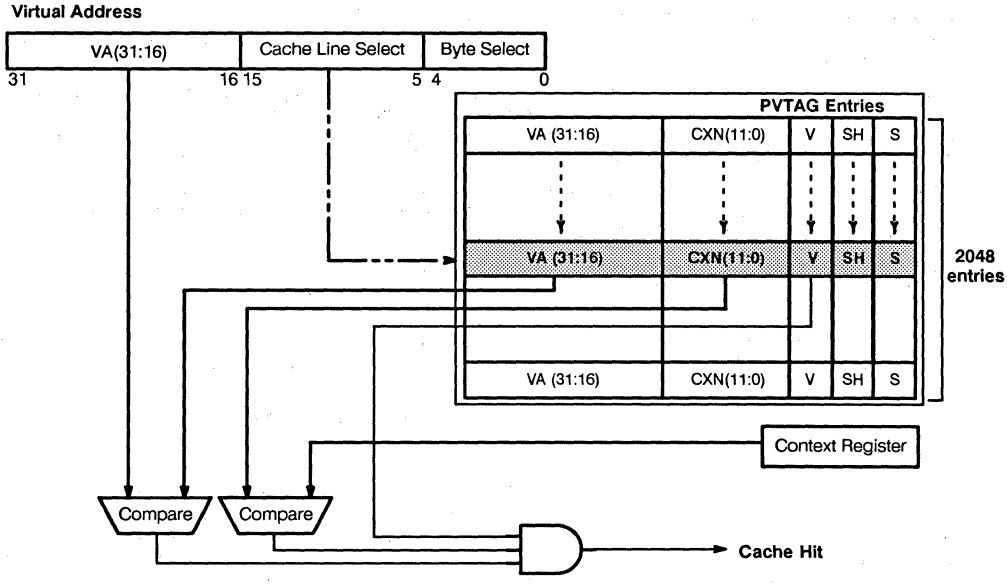


Figure 4–15. CY7C605 Processor Virtual Cache Tag (PVTAG) Comparison

4.3.3 CY7C605 Cache Controller

The cache controller provides cache memory access control for a 64-kbyte direct-mapped virtual cache. The cache controller performs this task by comparing memory accesses against the address and status entries in a cache tag memory. The CY7C605 provides two separate cache tag memories for access comparison. Cache memory accesses from the processor are compared against the Processor Virtual cache TAG (PVTAG) memory. Bus snooping operations are compared against the Mbus Physical cache TAG (MPTAG) memory. The use of two cache tag memories allows the cache controller to service processor cache accesses concurrently with bus snooping cache tag accesses. This feature of the CY7C605 provides significant performance improvements over cache systems sharing a single cache tag memory between the processor cache access and the bus snooping operations. Single cache tag systems typically must stall the processor when a bus snooping operation is required, causing serious performance degradation.

The cache controller is designed to use two CY7C157 cache RAMs for the cache memory. These cache RAMs are 16-kbyte x 16 SRAMs with on-chip address and data latches and timing control. Two CY7C157s and one CY7C605 comprise an entire 64-kbyte cache system with physical bus interface and read and write buffers.

The cache is organized as 2048 cache lines of 32 bytes each. The CY7C605 has 2048 cache tag entries in both the PVTAG and MPTAG, one entry in each cache tag memory per cache line. Addressing for the virtual cache is provided directly from the virtual address bus. The virtual address field (VA(15:5)) selects one of the 2048 lines of the cache (refer to *Figure 4–15*). This address field also selects the cache tag entry in the PVTAG dedicated to the selected cache line. A cache hit occurs when the upper sixteen bits of the virtual address and the context register match with the virtual address and context stored in the selected cache tag entry in PVTAG. The lowest five bits of the virtual address bus (VA(4:0)) select one or more of the 32 bytes in the cache line. Cache data replacement is always performed by replacing cache lines.

The cache is designed to provide data with every read access asserted on the virtual bus, regardless of the cache controller. The CY7C605 controls cache read access by holding the CY7C601 with $\overline{\text{MHOLD}}$ if a cache hit is not detected by the cache controller. The cache controller then reads the new cache line from main memory, and supplies the correct data to the CY7C601. After the correct data is latched into the CY7C601 by strobing the $\overline{\text{MDS}}$ signal, the CY7C601 is released and execution proceeds normally.

Writes to the cache are controlled by the CY7C605, which decodes the lowest two bits of the virtual address, the SIZE(1:0) signal, and checks for a cache hit to enable the correct cache byte write enable signals. If a cache write hit occurs, the CY7C605 decodes the correct $\overline{\text{CBWE}}$ signals for the write access, and outputs these to the CY7C157 cache RAM write enables. If the cache mode is set to write-through (see Section 4.3.1, Cache Modes), the write data is also written to main memory. If a write cache miss occurs for write-through cache mode, the data is written to main memory and the cache is not updated. If the write cache miss occurs during copy-back cache mode, the cache line is fetched from main memory. If the cache line stored in the cache when the write cache miss occurred has been modified, the old cache line is written to main memory before the cache line is replaced by the new data. After the cache line has been replaced, the write access is enabled by the CY7C605.

4.3.3.1 CY7C605 Cache Tag

The CY7C605 features two separate cache tag arrays: the processor virtual cache tag memory (PVTAG) and the Mbus physical cache tag memory (MPTAG). Cache controllers using only one cache tag array must delay the processor when bus snooping requires access to the cache tags. The inclusion of two independent cache tag memories allows the CY7C605 to support processor accesses to cache while simultaneously performing bus snooping on the Mbus.

4.3.3.1.1 CY7C605 Processor Virtual Cache Tag (PVTAG)

The PVTAG consists of 2048 direct-mapped cache tag entries, as shown in *Figure 4–16*. The PVTAG and the TLB are accessed simultaneously. Each entry in the cache consists of 16 bits of virtual address (VA(31:16)), a 12-bit context number (CXN(11:0)), one valid bit (V), and one shared bit (SH). The valid bit (V) is set or cleared to indicate the validity of the cache tag entry. The shared bit (SH) of a cache tag entry is set when bus snooping indicates that the cache line is shared. The cache line select field (VA(15:5)) is used to select a cache line entry and its corresponding cache tag entry. The address field VA(31:16) and context register are compared against the virtual address and the context fields of the selected cache tag entry. If a match occurs, then a cache hit is generated. If a match is not found, then a cache miss is generated. To complete an access successfully, both the cache tag and the TLB must be hit with appropriate access-level permission. On Power-On Reset ($\overline{\text{POR}}$), all cache tag entries are invalidated (all V bits are cleared).

A supervisor bit (S) is included in the cache tag entry. For cache tag entries which are accessible by the supervisor only (access-level field 6 or 7), the S bit is set. During a cache tag look up, if the access is supervisor mode and the the S bit is set, the context number comparison is ignored and the context match is forced. This operation is similar to a TLB look up with access-level field set to either 6 or 7.

4.3.3.1.2 CY7C605 Mbus Cache Tag (MPTAG)

The MPTAG consists of 2048 direct-mapped, physical address cache tag entries (refer to *Figure 4–16*). Each entry in the cache consists of 24 bits of physical address (PA<35:12>), a valid bit (V), a shared bit (SH), and a modified bit (M).

The 2048 MPTAG entries are virtual address indexed. The index field for MPTAG, as supplied by the Mbus, is formed by concatenating the superset virtual address bits (15:12) (MAD(49:46)) with physical address bits (11:5) (MAD(11:5)) (refer to *Figure 4–17*). The format of the Mbus address bus cycle is described in Section 4.12.5 in Section 4.12.5.

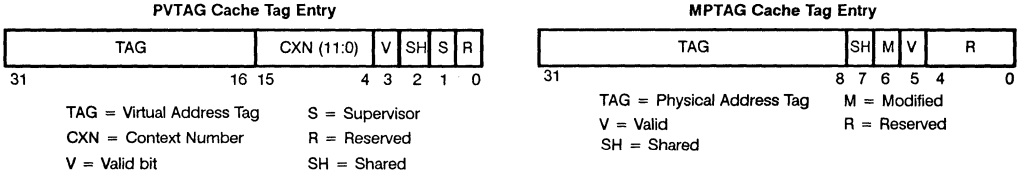


Figure 4–16. CY7C605 Cache Tag Entries

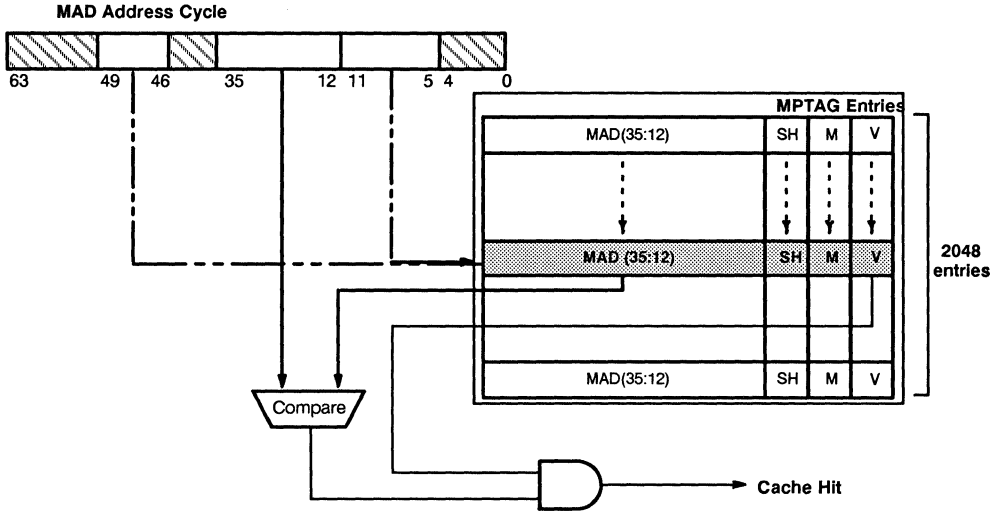


Figure 4–17. CY7C605 Mbus Physical Cache Tag (MPTAG) Comparison

During a MPTAG compare operation, the physical address field (35:12) of the access is compared against the physical address field of the MPTAG entry selected by the virtual address index. If a match occurs and the valid bit is set, a cache hit is generated. If a match is not found, or the valid bit is not set, a cache miss is generated. On Power-On Reset (POR), all the MPTAG cache entries are invalidated (V bits are cleared).

4.3.3.2 CY7C605 Multiprocessing Support

The CY7C605 is specifically designed to support multiprocessing systems. The CY7C605 accomplishes this by providing features necessary to maintain cache coherency with a second-level memory system (typically main memory or a secondary cache) and other caching systems on the shared bus.

The CY7C605 supports two modes of caching: write-through and copy-back. Operation in write-through caching mode causes main memory to be modified with each write access to the cache. This avoids the issue of lack of coherency between the individual cache systems and main memory, but greatly increases memory bus traffic. The effect of this increased bus traffic is a degrading of the performance of a multiprocessor system as the processing nodes compete for memory bus bandwidth. This problem is greatly reduced when copy-back caching mode is used.

Operation in copy-back mode causes all changes to a cache line to be held until the line is flushed from the cache. This minimizes bus traffic to only those transactions necessary to maintain the cache. However, by allowing the cache line to be modified without updating main memory, a problem arises when other processing nodes require an up-to-date copy of that memory location. The problem of modified cache lines is solved by the enforcement of a cache coherency protocol.

The CY7C605 implements a cache coherency protocol specified by the SPARC reference standard Mbus level-2 interface. This protocol is modeled after that used by the IEEE Futurebus. In this protocol, each cache line is described by one of five states: invalid (I), exclusive clean (EC), exclusive modified (EM), shared clean (SC), and shared modified (SM). The following describes these five cache states:

Invalid (I): Cache line is not valid.

Exclusive Clean (EC): Only this cache module has a valid copy of this cache line, other than the next level of memory (main memory or secondary cache). No other cache module on the same level of memory has a valid copy of this cache line.

Exclusive Modified (EM): Only this cache module has a valid copy of this cache line. This cache module is the OWNER of the cache line, and has the responsibility to update the next level of memory (main memory or secondary cache) and also to supply data if any other cache references this memory location.

Shared Clean (SC): The same cache line may exist in more than one cache module. The next level of memory may or may not contain a valid copy of this cache line, depending upon whether this cache line has been modified in any other cache.

Shared Modified (SM): The same cache line may exist in more than one cache module, but this cache module is the OWNER of the cache line. The next level of memory does not have a valid copy of this cache line, and this cache module has the responsibility to update the next level of memory and to supply any other cache that may reference this same memory location.

These five states are described by three state bits (valid (V), shared (SH), and modified(M)) in each MPTAG cache tag entry (refer to *Figure 4–16*). The PVTAG cache tag entries are described by two state bits: valid (V), and shared (SH). The PVTAG cache tag entries corresponding to the same cache lines can be in one of three states: invalid, exclusive valid, and shared valid.

Under write-through cache mode, only the valid and invalid states apply to either the MPTAG or PVTAG cache tag entries. The shared and modified bits in the MPTAG are ignored by the CY7C605 when in write-through mode.

4.3.3.3 CY7C605 Cache State Transitions

The following sections describe the five cache line states (invalid, exclusive clean, exclusive modified, shared clean, and shared modified) and the transitions these states undergo due to transactions on the Mbus. Each numbered transition in a section corresponds to a numbered transition on the state diagram for that section. Note that state transitions are dependent upon both the cache transaction and the state of the Mbus signals: memory shared ($\overline{\text{MSH}}$), and memory inhibit ($\overline{\text{MIH}}$).

All processor transactions described in this section affect the processor serviced by the CY7C605. All coherent transactions affect all bus agents on the Mbus with a copy of the shared cache line. For further information on Mbus transactions, please refer to Section 4.12.

4.3.3.3.1 Copy-Back Invalid

Processor Read Miss: CY7C605 issues a coherent read transaction on the Mbus. The CY7C605 will read the cache line from the second-level memory and then load it into the cache RAM. Then the data is supplied to the processor in the cycle following the last cache line entry.

1. If $\overline{\text{MSH}}$ = HIGH, then invalid changes to exclusive valid in PVTAG and invalid changes to exclusive clean in MPTAG.
2. If $\overline{\text{MSH}}$ = LOW, then invalid changes to shared valid in PVTAG and invalid changes to shared clean in MPTAG.

Processor Write Miss: CY7C605 issues a coherent read and invalidate transaction on the Mbus. The CY7C605 reads the cache line from the second-level memory and loads it into the cache RAM. Then the processor data is written into the cache RAM in the cycle following the last cache line entry.

3. Invalid changes to exclusive valid in PVTAG and invalid changes to exclusive modified in MPTAG.

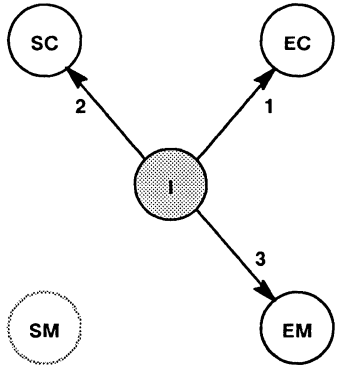


Figure 4–18. Copy-Back Invalid

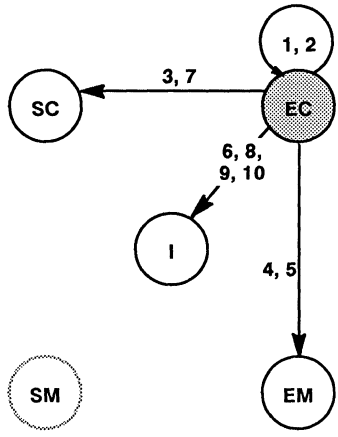


Figure 4–19. Copy-Back Exclusive Clean

4.3.3.3.2 Copy-Back Exclusive Clean

Processor Read Hit: The CY7C605 will supply data to the CY7C601 immediately.

1. PVTAG entry is exclusive valid; exclusive clean in MPTAG: NO STATE CHANGE.

Processor Read Miss: The CY7C605 will issue a coherent read transaction on the Mbus. The CY7C605 will read the cache line from the second-level memory and then load it into the cache RAM. Then the data is supplied to the CY7C601 in the cycle following the last cache line entry.

2. If $\overline{\text{MSH}}$ = HIGH, then exclusive valid in PVTAG; exclusive clean in MPTAG.

3. If $\overline{\text{MSH}}$ = LOW, then shared valid in PVTAG; exclusive clean changes to shared clean in MPTAG.

Processor Write Hit: The CY7C605 will update the cache immediately with the CY7C601 data.

4. PVTAG entry is exclusive valid; exclusive clean changes to exclusive modified in MPTAG.

Processor Write Miss: The CY7C605 will issue a coherent read and invalidate transaction on the Mbus. The CY7C605 will read the cache line from the second-level memory and then load it into the cache RAM. Then the processor data is written into the cache RAM in the cycle following the last cache line entry.

5. PVTAG entry is exclusive valid; exclusive clean changes to exclusive modified in MPTAG.

Software Flush (Store alternate instruction with ASI = 10H to 14H; see Section 4.3.7): The CY7C605 will invalidate both the PVTAG and MPTAG cache tag entries.

6. Exclusive valid is changed to invalid in PVTAG; exclusive clean is changed to invalid in MPTAG.

Coherent Read: During the A+2 cycle of the Mbus coherent read transaction, the CY7C605 will assert $\overline{\text{MSH}}$ and change the state of the cache line from exclusive clean to shared clean.

7. Assert $\overline{\text{MSH}}$; exclusive clean is changed to shared clean in MPTAG and shared valid in PVTAG.

Coherent Read and Invalidate: Both the PVTAG and the MPTAG cache tag entries in the CY7C605 are invalidated.

8. Exclusive valid is changed to invalid in PVTAG; exclusive clean is changed to invalid in MPTAG.

Coherent Invalidate: Both the PVTAG and the MPTAG entries in the CY7C605 are invalidated.

9. Exclusive valid is changed to invalid in PVTAG; exclusive clean is changed to invalid in MPTAG.

Coherent Write and Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

10. Exclusive valid is changed to invalid in PVTAG and exclusive clean is changed to invalid in MPTAG.

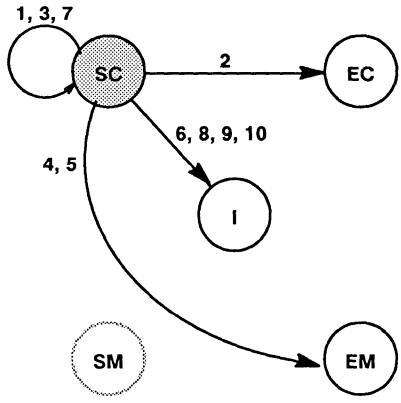


Figure 4–20. Copy-Back Shared Clean

4.3.3.3.3 Copy-Back Shared Clean

Processor Read Hit: The CY7C605 will supply data immediately to the CY7C601.

1. PVTAG entry is shared valid; shared clean in MPTAG: NO STATE CHANGE.

Processor Read Miss: The CY7C605 will issue a coherent read transaction on the Mbus. The CY7C605 will read the cache line from the second-level memory and load it into the cache RAM. Then the data is supplied to the CY7C601 in the cycle following the last cache line entry.

2. If $\overline{\text{MSH}}$ = HIGH, then exclusive valid in PVTAG and shared clean is changed to exclusive clean in MPTAG.

3. If $\overline{\text{MSH}}$ = LOW, then shared valid in PVTAG and shared clean in MPTAG.

Processor Write Hit: The CY7C605 issues a coherent invalidate transaction on the Mbus. The CY7C605 will update the cache immediately with the processor data.

4. PVTAG entry is exclusive valid; shared clean is changed to exclusive modified in MPTAG.

Processor Write Miss: The CY7C605 will issue a coherent read and invalidate transaction on the Mbus. The CY7C605 will read the cache line from the second-level memory and then load the data into the cache RAM. The processor data is written into the cache RAM in the cycle following the last cache line entry.

5. PVTAG entry is changed to exclusive valid; shared clean is changed to exclusive modified in the MPTAG.

Software Flush: The CY7C605 will invalidate both the PVTAG and MPTAG cache tag entries.

6. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

Coherent Read: During the A+2 cycle of the Mbus coherent read transaction, the CY7C605 will assert the $\overline{\text{MSH}}$.

7. Assert $\overline{\text{MSH}}$; shared clean in MPTAG and shared valid in PVTAG.

Coherent Read and Invalidate: Both the PVTAG and the MPTAG cache tag entries will be invalidated.

8. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

Coherent Invalidate: Both the PVTAG and MPTAG cache tag entries are invalidated.

9. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

Coherent Write and Invalidate: Both the PVTAG and MPTAG cache tag entries are invalidated.

10. Shared valid is changed to invalid in PVTAG; shared clean is changed to invalid in MPTAG.

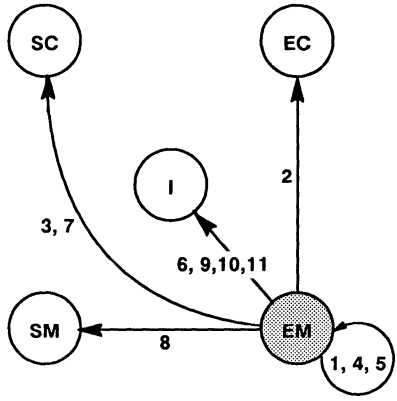


Figure 4–21. Copy-Back Exclusive Modified

4.3.3.3.4 Copy-Back Exclusive Modified

Processor Read Hit: The CY7C605 will supply data to the processor immediately.

1. PVTAG entry is exclusive valid; exclusive modified in MPTAG: NO STATE CHANGE.

Processor Read Miss: The CY7C605 will initiate a coherent read transaction followed by a write block transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. Then the data will be supplied to the processor in the cycle following the last cache line entry into the cache RAM. The modified cache line has to be written to update the second-level memory. The Mbus Busy ($\overline{MBB}$) signal is asserted from the beginning of the coherent read transaction to the end of the write transaction on the Mbus.

2. If $\overline{MSH}$ = HIGH, then the PVTAG entry is exclusive valid, and the MPTAG entry is changed from exclusive modified to exclusive clean.

3. If $\overline{MSH}$ = LOW, then the PVTAG entry is changed to shared valid, and the MPTAG entry is changed from exclusive modified to shared clean.

Processor Write Hit: The CY7C605 will update the cache immediately with the processor data.

4. PVTAG entry is exclusive valid; exclusive modified remains as exclusive modified in MPTAG.

Processor Write Miss: The CY7C605 will initiate a coherent read and invalidate transaction followed by a write block transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load it into the cache RAM. The processor data is written into the cache RAM in the cycle following the last cache line entry into the cache RAM. The modified cache line must be written into the second-level memory in order to update the memory. The $\overline{MBB}$ signal is asserted from the beginning of the coherent read and invalidate transaction to the end of the write transaction on the Mbus.

5. PVTAG entry remains exclusive valid; the MPTAG entry remains exclusive modified.

Software Flush: The CY7C605 initiates a coherent write and invalidate transaction on the Mbus. The CY7C605 will write the modified cache line to update the second-level memory and then it invalidates both the PVTAG and MPTAG cache tag entries.

6. Exclusive valid is changed to invalid in PVTAG; exclusive modified is changed to invalid in MPTAG.

Coherent Read: During the A + 2 cycle of the coherent read transaction on the Mbus, the CY7C605 asserts both the $\overline{MSH}$ and $\overline{MIH}$ signals. This CY7C605 is the OWNER of the cache line, and is responsible to supply the data for the coherent read transaction on the Mbus.

7. If the memory reflection (MR) bit of the system control register (SCR) is set, the CY7C605 changes the state of the MPTAG cache tag entry from exclusive modified to shared clean, and the PVTAG entry from exclusive valid to shared valid.

8. If the memory reflection (MR) bit of the SCR is cleared, the CY7C605 changes the state of the MPTAG entry from exclusive modified to shared modified. The PVTAG entry is changed to shared valid.

Coherent Read and Invalidate: During the A+2 cycle of a coherent read and invalidate transaction on the Mbus, the CY7C605 asserts both the $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ signals. This CY7C605 is the OWNER of the cache line, and is responsible to supply the data for the coherent read transaction on the Mbus. Both the PVTAG and MPTAG cache tag entries are invalidated.

9. Exclusive valid is changed to invalid in the PVTAG entry; exclusive modified is changed to invalid in the MPTAG entry.

Coherent Invalidate: Both the PVTAG and MPTAG cache tag entries in the CY7C605 are invalidated.

10. Exclusive valid is changed to invalid in the PVTAG entry; exclusive modified is changed to invalid in the MPTAG entry.

Coherent Write and Invalidate: Both the PVTAG and the MPTAG cache tag entries are invalidated.

11. Exclusive valid is changed to invalid in the PVTAG entry; exclusive modified is changed to invalid in the MPTAG entry.

4.3.3.3.5 Copy-Back Shared Modified

Processor Read Hit: The CY7C605 will supply data immediately to the CY7C601.

1. PVTAG entry is shared valid; shared modified in MPTAG: NO STATE CHANGE.

4

Processor Read Miss: The CY7C605 will initiate a coherent read transaction followed by a write block transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. Then the data will be supplied to the processor in the cycle following the last cache line entry into the cache RAM. The modified cache line has to be written to update the second-level memory. The $\overline{\text{MBB}}$ signal is asserted from the beginning of the coherent read transaction to the end of the write transaction on the Mbus.

2. If $\overline{\text{MSH}}$ = HIGH, the PVTAG entry changes to exclusive valid. The MPTAG entry is changed from shared modified to exclusive clean.

3. If $\overline{\text{MSH}}$ = LOW, then the PVTAG entry changes to shared valid, and the MPTAG entry is changed from shared modified to shared clean.

Processor Write Hit: The CY7C605 initiates a coherent invalidate transaction on the Mbus. The CY7C605 will update the cache immediately with the processor data.

4. The PVTAG entry changes to exclusive valid; the entry in the MPTAG is changed from shared modified to exclusive modified.

Processor Write Miss: The CY7C605 will initiate a coherent read and invalidate transaction followed by a write block transaction of the previously modified cache line. The CY7C605 will read the cache line from the second-level memory and load it into the cache RAM. The processor data is written into the cache RAM in the cycle following the last cache line entry into the cache RAM. The modified cache line must be written into the second-level memory in order to update the memory. The $\overline{\text{MBB}}$ signal is asserted from the beginning of the coherent read and invalidate transaction to the end of the write transaction on the Mbus.

5. PVTAG entry is exclusive valid; the MPTAG entry is changed from shared modified to exclusive modified.

Software Flush: The CY7C605 initiates a coherent write and invalidate transaction on the Mbus. The CY7C605 will write the modified cache line to update the second-level memory and then it invalidates both the PVTAG and MPTAG cache tag entries.

6. Shared valid is changed to invalid in PVTAG; shared modified is changed to invalid in MPTAG.

Coherent Read: During the A+2 cycle of the coherent read transaction on the Mbus, the CY7C605 asserts both the $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ signals. This CY7C605 is the OWNER of the cache line, and is responsible for supplying the data for the coherent read transaction on the Mbus.

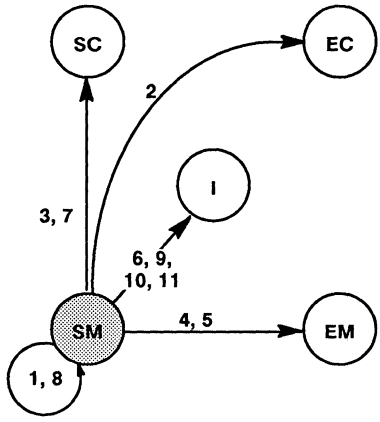


Figure 4–22. Copy-Back Shared Modified

7. If the memory reflection (MR) bit of the system control register (SCR) is set, the CY7C605 changes the state of the MPTAG from shared modified to shared clean, and the PVTAG entry is shared valid.
8. If the MR bit of the SCR is not set, then the PVTAG remains shared valid and the MPTAG remains shared modified.

Coherent Read and Invalidate: During the A+2 cycle of a coherent read and invalidate transaction on the Mbus, the CY7C605 asserts both the $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ signals. This CY7C605 is the OWNER of the cache line, and is responsible for supplying the data for the coherent read transaction on the Mbus. Both the PVTAG and MPTAG cache tag entries are invalidated.

9. Shared valid is changed to invalid in the PVTAG entry; shared modified is changed to invalid in the MPTAG entry.

Coherent Invalidate: Both the PVTAG and MPTAG cache tag entries in the CY7C605 are invalidated.

10. Shared valid is changed to invalid in the PVTAG entry; shared modified is changed to invalid in the MPTAG entry.

Coherent Write and Invalidate: Both the PVTAG and the MPTAG cache tag entries are invalidated.

11. Shared valid is changed to invalid in the PVTAG entry; shared modified is changed to invalid in the MPTAG entry.

4.3.3.3.6 Write-Through Invalid

Processor Read Miss: The CY7C605 issues a block read transaction on the Mbus. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. The data will be supplied to the processor in the cycle following the last cache line entry written to the cache RAM.

1. The PVTAG and MPTAG entries are changed from invalid to valid.

Processor Write Miss: The CY7C605 will issue a write-buffered coherent write and invalidate transaction on the Mbus.

2. The PVTAG and MPTAG entries remain invalid.

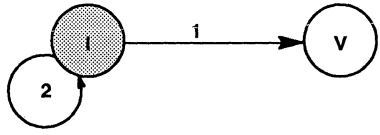


Figure 4–23. Write-Through Invalid

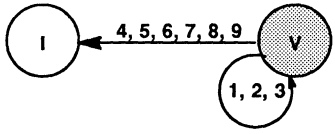


Figure 4–24. Write-Through Valid

4.3.3.3.7 Write-Through Valid

Processor Read Hit: The CY7C605 will supply data to the CY7C601 immediately.

1. The PVTAG and MPTAG entries remain valid: NO STATE CHANGE.

Processor Read Miss: The CY7C605 issues a coherent read transaction on the Mbus. The CY7C605 will read the cache line from the second-level memory and load the data into the cache RAM. The data will be supplied to the processor in the cycle following the last cache line entry written to the cache RAM.

2. The PVTAG and MPTAG entries remain valid.

Processor Write Hit: The CY7C605 issues a write-buffered coherent write and invalidation transaction on the Mbus. The CY7C605 will write data into the cache.

3. The PVTAG and MPTAG entries remain valid.

Processor Write Miss: The CY7C605 issues a write-buffered coherent write and invalidate transaction on the Mbus. The CY7C605 will not write to the cache and invalidates the cache line in order to avoid potential data inconsistency due to aliasing.

4. The PVTAG and MPTAG entries change from valid to invalid.

Software Flush: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

5. The PVTAG and MPTAG entries change from valid to invalid.

Coherent Read: During the A+2 cycle of the Mbus coherent read transaction, the CY7C605 asserts $\overline{\text{MSH}}$.

6. Assert $\overline{\text{MSH}}$; the PVTAG and MPTAG entries remain valid.

Coherent Read and Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

7. The PVTAG and MPTAG entries change from valid to invalid.

Coherent Write and Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

8. The PVTAG and MPTAG entries change from valid to invalid.

Coherent Invalidate: The CY7C605 invalidates both the PVTAG and MPTAG cache tag entries.

9. The PVTAG and MPTAG entries change from valid to invalid.

4.3.3.3.8 Bus Snooping

The CY7C605 bus snooper watches Mbus transactions and snoops into the MPTAG array for certain transactions, as listed in *Table 4–6*.

4.3.3.4 CY7C605 Address Aliasing

Two or more virtual addresses mapped to the same physical address is known as *aliasing*. This must be detected to maintain data consistency in a virtual cache system. The SPARC reference system software convention permits the use of aliases in address spaces that are modulo with respect to the system's underlying cache size. In order to allow the efficient caching of physical memory pages where such aliases may occur, the CY7C605 supports automatic address aliasing protection.

Table 4–6. Mbus Snooping Transactions

Cache Mode	Transaction Type	Snoop
Copy-Back	Coherent Read & Invalidate	yes
	Coherent Write & Invalidate	yes
	Coherent Read	yes
	Coherent Invalidate	yes
	Read	no
	Write	no
Write-Through	Coherent Read & Invalidate	yes*
	Coherent Write & Invalidate	yes
	Coherent Read	yes*
	Coherent Invalidate	yes
	Read	no
	Write	no

*These transactions are not generated by the CY7C605, but the CY7C605 will snoop these transactions if generated by another bus master

The SPARC system software convention ensures that the aliased entry maps to the same cache line address for each CY7C605 in the multiprocessor system. Coupled with this convention, the cache controller hardware automatically prevents any existence of address aliases in the virtual caches.

The CY7C605 tests for address aliasing during all cache misses except write-through mode write misses. Address aliasing cannot occur unless the MMU is enabled (ME bit of SCR). To detect address aliasing in the CY7C605, the physical address of the missed cache access is compared with the selected MPTAG entry.

If the physical address of the selected MPTAG entry and the physical address of the cache miss match, then address aliasing is detected. If detected, an alias is corrected by updating the selected cache tag entry with the new virtual address. The CY7C605 then halts the cache miss processing and provides an access to the cache, as with a cache hit. If no alias is detected, the cache miss processing proceeds normally.

For an alias detected during a read-access cache miss, the selected cache tag entry is updated with the virtual address that caused the cache miss. The cache miss processing is halted, and the CY7C601 is supplied with data from the cache.

If an address alias is detected during a copy-back mode write-access cache miss, the selected cache tag entry is updated with the new virtual address causing the cache miss. The modified bit is set if it was not set previously. The cache miss processing is halted, and the cache write access is enabled.

In write-through write-access cache misses, address aliasing is not checked. However, in order to avoid potential address aliasing, the selected cache line is invalidated. Address aliasing is not checked in write-through cache mode in order to avoid unnecessary performance degradation.

4.3.4 CY7C604/CY7C605 Cache Control Signals

The CY7C604/605 controls the virtual cache through control signals supplied to the CY7C601 and to the cache RAMs. The signals used by the cache controller to control the CY7C601 consist of $\overline{\text{MHOLD}}$, $\overline{\text{MDS}}$, and $\overline{\text{IOE}}$. $\overline{\text{MHOLD}}$ is used to stall the CY7C601 until the CY7C604/605 can service the CY7C601 memory access request, such as during cache miss processing or during table walks. $\overline{\text{MDS}}$ is used by the CY7C604/605 to strobe data into the CY7C601 when $\overline{\text{MHOLD}}$ is asserted. This causes the CY7C601 to latch data on the data bus despite being stalled by the assertion of $\overline{\text{MHOLD}}$. $\overline{\text{IOE}}$ is used as the enable signal for the $\overline{\text{AOE}}$ and $\overline{\text{DOE}}$ inputs of the CY7C601. When $\overline{\text{IOE}}$ is deasserted, the address and data bus output drivers of the CY7C601 are disabled. This feature is used to force the CY7C601 off of the virtual address and data buses.

The signals used to control the cache RAM consist of the cache byte write enable ($\overline{\text{CBWE}}$) and cache read output enable ($\overline{\text{CROE}}$) signals. $\overline{\text{CROE}}$ is asserted low to enable the output of the cache RAMs during a cache read. $\overline{\text{CBWE}}$(3:0) is asserted low to enable writing to the cache RAMs. The multiple $\overline{\text{CBWE}}$ signals allow the cache controller to enable byte, halfword, or word writes to the cache RAM. Single byte or halfword reads are handled by the CY7C601, which reads an entire 32-bit word and internally discards unwanted bytes.

During a cache read miss, the CY7C604/605 halts the CY7C601 by asserting $\overline{\text{MHOLD}}$. The CY7C604/605 also deasserts $\overline{\text{IOE}}$, which is used to disable the CY7C601 data bus and address bus output drivers. The cache controller fetches the

new cache line from main memory, asserting $\overline{\text{CBWE}}$(3:0) and the cache line addresses to write the data into the cache. Then the CY7C604/605 places the missed read data word on the data bus and toggles the $\overline{\text{MDS}}$ (Memory Data Strobe) signal. Toggling $\overline{\text{MDS}}$ forces the integer unit to latch the data on the data bus. The cache read miss terminates by reasserting the $\overline{\text{IOE}}$ signal and then releasing the $\overline{\text{MHOLD}}$ signal. $\overline{\text{IOE}}$ is typically reasserted one or more clocks before the $\overline{\text{MHOLD}}$ signal is deasserted, thus allowing the CY7C601 to output the next address onto the virtual address bus. This provides the address set-up time for the next memory access after $\overline{\text{MHOLD}}$ is released. Read misses are handled in the same manner for both copy-back and write-through modes of caching.

Cache write misses for write-through mode generally do not affect the operation of the CY7C601 due to the presence of write buffers in the CY7C604/605 (refer to the following section on the write buffer). In the case of a write miss, the write data is written to the write buffer instead of the cache memory and the cache tag for the cache line is invalidated. The write buffer writes the data to memory as a background task. The CY7C601 is stalled for a write-through write miss only if the write buffer is full. This occurs when the CY7C601 overruns the four doubleword buffers in the write buffer. In this case, $\overline{\text{MHOLD}}$ is asserted until space is made by the write buffer as it writes its contents into main memory.

On a write miss, if the cache mode is copy-back and the cache line is clean, the cache line is replaced in a similar manner as in the cache read miss described above. $\overline{\text{MHOLD}}$ is asserted to stall the CY7C601 and $\overline{\text{IOE}}$ is deasserted to force the CY7C601 off the data and address buses. A new cache line is read from main memory, and the cache is updated by writing the data into the cache. This is accomplished by supplying the cache addresses, cache line data from main memory, and asserting the $\overline{\text{CBWE}}$ signals to write the data. The write cache miss terminates by reasserting $\overline{\text{IOE}}$, which causes the missed write data and address to reappear on their respective buses. The CY7C604/605 then strobes $\overline{\text{CBWE}}$(3:0) according to the address and SIZE(1:0) signals to write the data into the cache. The copy-back write miss procedure terminates by deasserting $\overline{\text{MHOLD}}$, which allows the processor to return to execution.

If the cache line is modified, the modified cache line is read out of the cache and stored into the write buffer during the same time the new cache line is fetched from main memory and stored in the read buffer (refer to the following sections on write and read buffers). $\overline{\text{MHOLD}}$ is asserted and $\overline{\text{IOE}}$ deasserted to force the CY7C601 into a halted and inactive state. The cache controller asserts $\overline{\text{CROE}}$ and the cache addresses to flush the modified cache line into the write buffer. The cache controller then writes the new cache line into the cache from the read buffer while simultaneously writing the modified cache line into main memory from the write buffer. This is accomplished by supplying the cache addresses for the cache line data, and asserting the $\overline{\text{CBWE}}$(3:0) signals to write the data into the cache. The copy-back write miss for a modified cache line terminates by releasing $\overline{\text{IOE}}$ to allow the missed write data and address to reassert on the data and address buses. The CY7C604/605 asserts the $\overline{\text{CBWE}}$(3:0) signals to write the data into the cache. The $\overline{\text{MHOLD}}$ signal is then deasserted to allow the CY7C601 to return to processing. See Section 4.11 for virtual bus timing diagrams.

4.3.5 CY7C604/605 Write Buffer

The CY7C604/605 supports four write buffers on chip, as shown in *Figure 4–25*. In write-through mode, each buffer can store two 32-bit words, which efficiently supports store double operations. A physical address tag is associated with each of the four buffers in write-through mode. Upon a write access, the write buffers are loaded with the data to be written to main memory. This allows the CY7C601 to continue operation without stalling due to memory access delays on the physical bus.

In copy-back mode, the same buffers are configured to store a 32-byte cache line with a single physical address as shown in *Figure 4–26*. This allows for faster cache line flushes during modified cache line replacement. The modified cache line is flushed into the write buffer as the new cache line is simultaneously fetched from main memory. In either case, the contents of the buffers are transferred to main memory as a background task. On Power-On Reset ($\overline{\text{POR}}$), all of the write buffers are invalidated.

Non-cacheable writes use the four write buffers in the same manner as write-through cache transaction, even if copy-back mode is enabled. However, a copy-back cache line and non-cacheable data cannot simultaneously occupy the write buffer.

The CY7C604/605 requests Mbus ownership as soon as one of the write buffers is valid. For each write buffer transfer, the CY7C604/605 re-arbitrates the Mbus again. A modified cache-line flush is considered as one transaction. When the bus is still granted to the CY7C604/605 (i.e., bus parking), the CY7C604/605 can transfer the data immediately without any bus re-arbitration (so there are no dead clocks between transactions). Once all of the write buffers are full, further writes from the CY7C601 are held until a buffer is empty. If there is a read access cache miss, the CY7C601 is held until all of the write buffers are written back into main memory in order to maintain data consistency. After the write buffers are cleared, the CY7C604/605 resumes the task of fetching the cache line for the cache read miss.

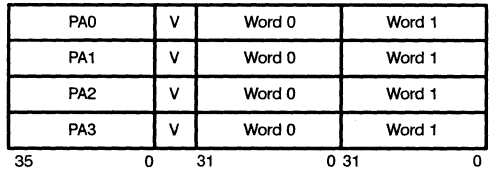


Figure 4–25. Write Buffers (Write-Through Mode or Non-Cacheable Write)

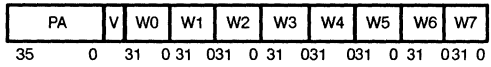


Figure 4–26. Write Buffer (Copy-Back Mode)

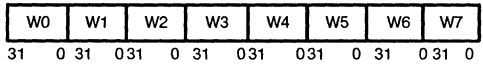


Figure 4–27. Read Buffer (Copy-Back Mode)

4.3.6 CY7C604/605 Read Buffer

The CY7C604/605 provides a read buffer of 32 bytes (one cache line) in order to support simultaneous writing of a modified cache line to main memory and reading of a new cache line from main memory into the cache under copy-back mode. The read buffer is shown in *Figure 4–27*. The read buffers are invalidated on power-on reset.

4.3.7 CY7C604/605 Cache Flushing Operations

The CY7C604/605 supports five different levels of cache flushing operations, as illustrated in *Table 4–7*. The cache flushing operations are dependent upon the cache mode and state. Flushing under copy-back cache mode for a modified cache line means flushing the cache line into main memory and invalidating the cache tag entry. If the cache line is clean (copy-back mode), or is in write-through cache mode, flushing only invalidates the cache tag entry.

Unlike a TLB flush operation, all cache flushing operations flush only one cache line at a time. Each cache line can be flushed on the basis of a page, segment, region, context, or user mode, as illustrated in *Table 4–7*. The levels of address matching for a cache line flush vary from a full 4-kbyte page level match of address and context, to a match of user mode only.

The cache line selected for operation is indexed as in normal cache access operations (VA(15:5)). If the cache flush operation does not cause a match of the cache tag entry, no action occurs. The five types of cache flush operations are: page flush, segment flush, region flush, context flush, and user flush. These different levels of cache flush are mapped with the ASI bits. The store alternate space instructions for the CY7C601 must be used to assert the ASI value that corresponds with the level of cache flush operation desired. The combination of the ASI and a store operation using the virtual address specify the cache flush operation and the cache line to be matched for flushing. During flush operations, the context register provides the context number to be compared.

Table 4–7. Cache Flush Operations

Cache Flush	ASI	Compares:
PAGE	10 H	Context (or Supervisor S = 1,), Index 1, Index 2, and Index 3 (bits 17 and 16)
SEGMENT	11 H	Context (or Supervisor S = 1), Index 1, and Index 2
REGION	12 H	Context, (or Supervisor S = 1), and Index 1
CONTEXT	13 H	Context and User (S = 0)
USER	14 H	User (S = 0)

Table 4–8. Cacheable/Non-Cacheable accesses

Access	Condition
Not cached	ASI = 20-2F H (By-pass) or ASI = 1 (Local)
	ASI = UN, RES (unassigned/reserved)
	BM = 1 and ME = x and CE = x and ASI = 8,9 H
	BM = x and not (ME = 1 and CE = 1 and PTE[C] = 1)
	LDSTO cycles in write-through mode
	Table walk cycles
	Cache lock miss accesses which have valid entries, but no alias
Cached	BM = 0 and ME = 1 and CE = 1 and ASI = 8,9,A,B H and PTE[C] = 1
	BM = 1 and ME = 1 and CE = 1 and ASI = A,B H and PTE[C] = 1

4.3.8 CY7C604/605 Cacheable/Non-Cacheable Memory Accesses

Pages that are declared as non-cacheable (C = 0 in the page table entry (PTE)) are not cached in the cache RAM and, as such, there are no associated cache tag entries in the CY7C604/605. For data consistency and implementation reasons, the CY7C604/605 assumes the following cycles are also non-cacheable:

a. LDSTO cycles in write-through mode (CY7C604 only)

b. table walk accesses

c. cache-missed accesses during cache-lock mode (CY7C604 only)

d. boot mode accesses (except user/supervisor data accesses when the MMU is enabled and the cache is enabled)

e. pass-through mode accesses

f. by-pass mode accesses

g. accesses while the cache is disabled

h. local-mode accesses

i. when MMU is disabled (ME bit of SCR = 0)

Table 4–8 shows the CY7C604/605 operation conditions for cacheable and non-cacheable accesses. Refer to the section on MMU operation modes for additional information.

4.3.9 CY7C604/605 Mbus Cacheable (MC) Bit

One of the CY7C604/605 output signals is a Mbus cacheable bit, which is embedded in the Mbus address phase as MAD(43) (Refer to Section 4.12, Physical Bus for more information on Mbus.) The Mbus cacheable bit indicates the cacheable status of a memory access by the CY7C604/605. This information is consistent with the cache visibility philosophy of the CY7C604/605 and is made available for use by a secondary cache tag array.

When the MMU is enabled, the MC bit is set by the state of the C bit in the corresponding PTE entry. When the MMU function of the CY7C604/605 is disabled, the C bit of the SCR register sets the value of the MC bit. The C bit of the SCR register is loaded by the CY7C601, and it defines the cacheable status of memory accesses when the MMU is disabled. *Table 4–9* illustrates the state of the MC bit for various CY7C604/605 operation conditions.

Table 4–9. State Table for MC (Memory Cacheable) Bit

MC	Condition
0	ASI = 20-2F H or ASI = 1 H
not applicable	ASI = UN, RES
SCR[C]	Not one of the above and ME = 0 or Not one of the above and (BM = 1 and ASI = 8,9 H) or Not one of the above and table walk
PTE[C]	Not one of the above

4.3.10 CY7C604/605 LDSTO (Atomic Load-Store Instruction) cycles

In order to maintain data consistency under write-through cache mode, LDSTO (atomic load-store) cycles are treated as non-cacheable transactions (CY7C604 only). All LDSTO accesses are forced into main memory in this case. The C bit in the TLB entry is output on the Mbus as the MC (MAD(43)) bit. If a cache hit occurs on a LDSTO cycle with the cache in write-through mode, the cache line is invalidated. If the MMU is disabled, the C bit in the SCR is output on the MC signal of the Mbus.

In copy-back mode, LDSTO cycles are treated as normal memory accesses and are cached according to the C bit of the PTE associated with the access.

LDSTO operations on the physical bus (Mbus) are repeated if interrupted by a relinquish and retry before the load operation of the LDSTO has been completed. However, if the relinquish and retry occurs after the load operation has completed, only the store operation of the LDSTO is repeated.

4.3.11 CY7C604/605 Cache Byte Write Enables

The CY7C604/605 supports four separate byte write enables ($\overline{\text{CBWE}}$(3:0)) to control write accesses to the cache RAM (CY7C157). These signals are generated using the lower two bits of the virtual address (VA(1:0)) and size (SIZE(1:0)) information during write accesses.

The decoding of the SIZE(1:0) and VA(1:0) bits is shown in *Table 4–10*. The $\overline{\text{CBWE0}}$ signal controls the most significant byte (MSB), which is located at a word-aligned address N. $\overline{\text{CBWE3}}$ controls the least-significant byte, located at address N + 3. All of the byte write enables are asserted for a cache line load into the cache RAM during a cache miss.

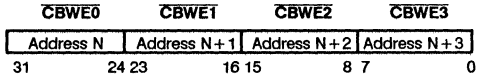


Figure 4–28. $\overline{\text{CBWE}}$ Byte Assignments

Table 4–10. Byte Write Enables

Size(1:0)	A(1:0)	$\overline{\text{CBWE3}}$	$\overline{\text{CBWE2}}$	$\overline{\text{CBWE1}}$	$\overline{\text{CBWE0}}$
00	00	1	1	1	0
00	01	1	1	0	1
00	10	1	0	1	1
00	11	0	1	1	1
01	00	1	1	0	0
01*	01*	1	1	1	1
01	10	0	0	1	1
01*	11*	1	1	1	1
10	00	0	0	0	0
10*	01*	1	1	1	1
10*	10*	1	1	1	1
10*	11*	1	1	1	1
11	00	0	0	0	0
11*	01*	1	1	1	1
11*	10*	1	1	1	1
11*	11*	1	1	1	1

*Denotes an illegal combination of Size(1:0) and A(1:0).

4.4 CY7C604 / CY7C605 Registers

This section describes the control and data registers for the CY7C604/605. All registers for the CY7C604 and CY7C605 are identical with the exception of the system control register (SCR). Sections or diagrams specific to the CY7C604 or CY7C605 are named with that part name only, whereas sections or diagrams common to both will be named using CY7C604/605.

All values in all control registers are read/write (with the exception of the Implementation and Version fields of the SCR). Control registers are accessible by use of the alternate space load or store instructions with ASI = 4. Please refer to Section 4.8, ASI and Register Mapping, for more information on register addressing.

Programmer's Note: To ensure software compatibility with future versions of the CY7C604/605, reserved fields in a register should be written as zeros and masked out when read.

4.4.1 CY7C604 System Control Register (SCR)

The system control register, as shown in *Figure 4–29,* defines the operation modes for the cache controller and MMU. Refer to Section 4.2, MMU Operational Modes, for additional information on the operation modes of the MMU. The following describes the functions of the bit fields in the SCR.

IMPL, VER The Implementation number (SCR(31:28)) and the Version number (SCR(27:24)) fields are hardwired; they are read only fields and writes to those fields are ignored. The assignments for the CY7C604 these fields are:

Implementation number field: 0001
Version number field: 0000

MCA(1:0) *Multichip address field* (SCR(23:22)) provides the address field in multichip configuration. Refer to the Section on Multichip Configuration for more information.

MCM(1:0) *Multichip mask field* (SCR(21:20)) provides a masking facility to mask certain multichip address (MCA) bits in order to provide a facility to build systems with a different number of CY7C604s (from 1 to 4).

MV *Multichip configuration valid bit* (SCR(19)) indicates that the MCA and MCM fields are valid (see Multichip Configuration, Section 4.5).

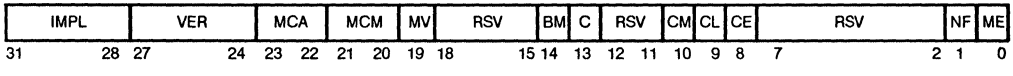
BM *Boot-mode bit* (SCR(14)) indicates the system is in boot mode. This bit is set to 1 to indicate boot mode. This bit is automatically set upon power-on reset.

C *Cacheable bit* (SCR(13)) indicates whether the access is cacheable or not when the MMU is disabled (this bit is independent of the CE bit, see Cacheable/Non-Cacheable Memory Accesses, Section 4.3.8 for more details.) This bit is set to 1 if accesses on the physical bus (with the MMU disabled) are to be considered cacheable.

CM *Cache-mode bit* (SCR(10)) indicates whether the cache is operating under write-through no write allocate policy or copy-back write allocate policy. This bit is set to 1 to enable copy-back cache mode. Setting this bit to 0 will enable write-through cache mode.

CL *Cache-lock bit* (SCR(9)) indicates whether the entire cache is locked or not (see Section 4.3.2.3 on Cache Lock, page 4–19). This bit is set to 1 to lock the cache.

CE *Cache-enable bit* (SCR(8)) indicates whether the virtual cache is enabled or not. This bit is set to 1 to enable the cache controller.



IMPL = Specific Implementation of the MMU
VER = Version of Specific Implementation (typically mask revision)
MCA (0:1) = Multichip Address
MCM (0:1) = Multichip Mask
MV = Multichip Valid
BM = Boot Mode
C = Cacheable (when MMU disabled)
CM = Cache Mode
CL = Cache Lock
CE = Cache Enable
NF = No Fault
ME = MMU Enable
RSV = Reserved

Figure 4–29. CY7C604 System Control Register (SCR)

NF *No-fault bit* (SCR(1)) prevents supervisor data accesses from signaling data faults to the CY7C601. When the NF bit is set, exception-generating logic (in both the TLB and the table walk) does not indicate supervisor data faults to the CY7C601 (via $\overline{\text{MEXC}}$), but status and address information is recorded in the SFSR and SFAR registers as in normal data access operations. When the NF bit is not set, the CY7C604 reports the supervisor data exceptions.

ME *MMU-enable bit* (SCR(0)) indicates whether the MMU is enabled or not. This bit is set to 1 to enable the MMU.

Upon power-on reset, all writable control bits except the BM bit are cleared. This sets the CY7C604 into the following state: cache disabled (CE = 0), cache unlocked (CL = 0), write-through mode (CM = 0), non-cacheable (C = 0), boot-mode enabled (BM = 1), multichip disabled (MV = 0), no fault disabled (NF = 0), and MMU disabled (ME = 0).

4.4.2 CY7C605 System Control Register (SCR)

The System Control Register, as shown in *Figure 4–30*, defines the operation modes for the cache controller and MMU. Refer to page 4–13 for additional information on the operation modes of the MMU. The following describes the functions of the bit fields in the SCR.

IMPL, VER The Implementation number (SCR(31:28)) and the Version number (SCR(27:24)) fields are hardwired; they are read only fields and writes to those fields are ignored. The assignments for the CY7C605 are:

Implementation number field: 0001
Version number field: 1111

MCA(1:0) *Multichip address field* (SCR(23:22)) provides the address field in multichip configuration. Refer to Section 4.5 on Multichip Configuration for more information.

MCM(1:0) *Multichip mask field* (SCR(21:20)) provides a masking facility to mask certain multichip address (MCA) bits in order to provide a facility to build systems with a different number of CY7C605s (from 1 to 4).

MV *Multichip configuration valid bit* (SCR(19)) indicates that the MCA and MCM fields are valid (see Multichip Configuration, Section 4.5).

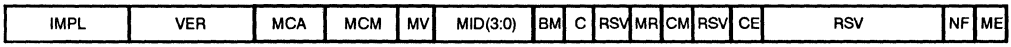
MID(3:0) *Module identification number* (SCR(18:15)) identifies the processor module during transactions on the Mbus (refer to Section 4.12). This four bit module identification number is embedded in the Mbus address phase of all Mbus transactions initiated by the CY7C605.

BM *Boot-mode bit* (SCR(14)) indicates the system is in boot mode. This bit is set to 1 to indicate boot mode. This bit is automatically set upon power-on reset.

C *Cacheable bit* (SCR(13)) indicates whether the access is cacheable or not when the MMU is disabled (this bit is independent of the CE bit, see Cacheable/Non-cacheable Memory Accesses, Section 4.3.8, for more details.) This bit is set to 1 if accesses on the physical bus (with the MMU disabled) are to be considered cacheable.

MR *Memory Reflection* (SCR(11)) MR = 1 indicates that the main memory system on the Mbus supports memory reflection. MR affects the status of the MPTAG cache tag bits as described in the cache state transitions section starting on page 4-23.

CM *Cache-mode bit* (SCR(10)) indicates whether the cache is operating under write-through no write allocate policy or copy-back write allocate policy. This bit is set to 1 to enable copy-back cache mode. Setting this bit to 0 will enable write-through cache mode.



IMPL = Specific Implementation of the MMU
VER = Version of Specific Implementation (typically mask revision)
MCA (1:0) = Multichip Address
MCM (1:0) = Multichip Mask
MV = Multichip Valid
MID(3:0) = Module Identifier (3:0)
BM = Boot Mode
C = Cacheable (when MMU disabled)
MR = Memory Reflection
CM = Cache Mode
CE = Cache Enable
NF = No Fault
ME = MMU Enable
RSV = Reserved

Figure 4–30. CY7C605 System Control Register (SCR)

CE *Cache-enable bit* (SCR(8)) indicates whether the virtual cache is enabled or not. This bit is set to 1 to enable the cache controller.

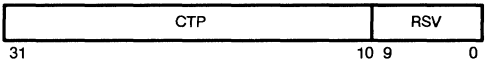
NF *No-fault bit* (SCR(1)) prevents supervisor data accesses from signaling data faults to the CY7C601. When the NF bit is set, exception-generating logic (in both the TLB and the table walk) does not indicate supervisor data faults to the CY7C601 (via $\overline{\text{MEXC}}$), but status and address information is recorded in the SFSR and SFAR registers as in normal data access operations. When the NF bit is not set, the CY7C605 reports the supervisor data exceptions.

ME *MMU-enable bit* (SCR(0)) indicates whether the MMU is enabled or not. This bit is set to 1 to enable the MMU.

Upon power-on reset, all writable control bits except the BM bit are cleared. This sets the CY7C605 into the following state: cache disabled (CE = 0), write-through mode (CM = 0), non-cacheable (C = 0), boot-mode enabled (BM = 1), memory reflection disabled (MR = 0), no fault disabled (NF = 0), and MMU disabled (ME = 0).

4.4.3 CY7C604/605 Context Table Pointer Register (CTPR)

The context table pointer points to the context table in physical memory. The table is indexed by the contents of the context register. The context table pointer appears on bits 35 through 14 of the Mbus (MAD(35:14)) during the first fetch of TLB miss processing. Once the root pointer is cached in the PTPC (Page Table Pointer Cache), no fetching of the root pointer is required until the context is changed (see *Figure 4–31).*

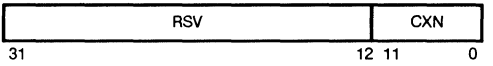


CTP = Context Table Pointer
RSV = Reserved

Figure 4–31. CY7C604/605 Context Table Pointer Register

4.4.4 CY7C604/605 Context Register (CXR)

The context register defines a virtual address space associated with the current process. The CXR is a twelve-bit register, which supports 4096 contexts. This register is used to define the current context for the CY7C604/605. Nearly all CY7C604/605 operations are dependent upon matching the value of this register to a cache tag entry or TLB entry.

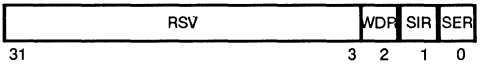


CXN = Context Number
RSV = Reserved

Figure 4–32. CY7C604/605 Context Register

4.4.5 CY7C604/605 Reset Register (RR)

The RR register contains information regarding whether Watch Dog Reset (WDR), Software Internal Reset (SIR) or Software External Reset (SER) occurred. This is a read/write register, and setting the software internal reset bit (SIR) or the software external reset (SER) causes the corresponding reset. Refer to CY7C604/605 Reset, Section 4.7, for more details on reset processing. Upon power-on reset, the WDR, SIR, and SER bits in the RR will be cleared. Reading the RR will also clear these bits.



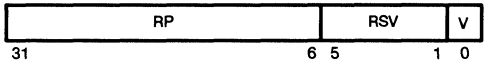
RSV = Reserved
WDR = Watch Dog Reset
SIR = Software Internal Reset
SER = Software External Reset

Figure 4–33. CY7C604/605 Reset Register

4.4.6 CY7C604/605 Root Pointer Register (RPR)

The RPR is the context level table page table pointer (PTP) and is cached in the Page Table Pointer Cache. Refer to Section 4.1.5 on page 4–11 for information on the page table pointer cache.

On power-on reset, the V bit is cleared. When the current context is changed by writing to the Context Pointer Register (CXR), the V bit of the RPR is cleared. The V bit is also cleared when the CTPR register is written.

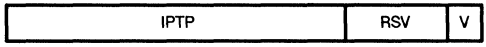


RP = Root Pointer
RSV = Reserved
V = Valid

Figure 4–34. CY7C604/605 Root Pointer Register

4.4.7 CY7C604/605 Instruction access PTP (IPTP)

The IPTP is the instruction access level 2 table page table pointer (PTP) and is part of the Page Table Pointer Cache. On power-on reset, the V bit is cleared.

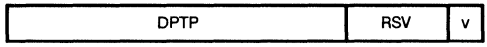


IPTP = Instruction Access PTP
RSV = Reserved
V = Valid

Figure 4–35. CY7C604/605 Instruction Access PTP Register

4.4.8 CY7C604/605 Data access PTP (DPTP)

The DPTP is the data access level 2 table page table pointer (PTP) and is a register in the Page Table Pointer Cache. On power-on reset, the V bit is cleared.

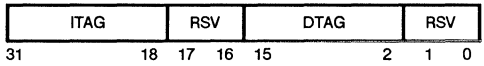


DPTP = Data Access PTP
RSV = Reserved
V = Valid

Figure 4–36. CY7C604/605 Data Access PTP Register

4.4.9 CY7C604/605 Index Tag Register (ITR)

The ITR contains the tag (index1 and index2) fields of the IPTP and DPTP entries. Refer to Section 4.1.5 on page 4–11 for information on the PTP cache.

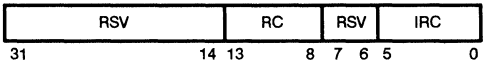


RSV = Reserved
ITAG = Instruction Access PTP Tag
DTAG = Data Access PTP Tag

Figure 4–37. CY7C604/605 Index Tag Register

4.4.10 CY7C604/605 TLB Replacement Control Register (TRCR)

The TRCR contains the Replacement Counter (RC) and Initial Replacement Counter (IRC) fields as shown in *Figure 4–38*. These fields are used in order to support random replacement and to support locking capabilities of the TLB. Refer to Section 4.1.1.2 on page 4–6for information on TLB entry locking. Upon power-on reset, both the RC and IRC fields are initialized to zero.



RSV = Reserved
RC = Replacement Counter
IRC = Initial Replacement Counter

Figure 4–38. CY7C604/605 TLB Replacement Control Register

4.4.11 CY7C604/605 Synchronous Fault Status Register (SFSR)

The synchronous fault status register, illustrated in *Figure 4–39*, contains fault-associated information for synchronous faults. Synchronous faults are faults that occur during an integer unit access of memory. Synchronous faults include almost all possible faults for the CY7C604/605. This type of fault is synchronous to the operations of the CY7C601. For the CY7C604/605, this fault type covers all cases except those caused by delayed writes of data stored in the write buffers. These faults are asynchronous to the operation of the CY7C601, and are named asynchronous faults.

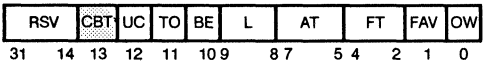
An example of a synchronous fault is a privilege violation fault caused by attempting an unauthorized memory access. These faults are discussed in detail in Section 4.9. Upon encountering a synchronous fault, the CY7C604/605 asserts the $\overline{\text{MEXC}}$ signal, along with $\overline{\text{MHOLD}}$ and $\overline{\text{MDS}}$. Synchronous faults are the only exception type that assert the $\overline{\text{MEXC}}$ signal.

In the CY7C604, the copy-back translation error (CBT) bit indicates that a translation error occurred during a table walk for the flush of a modified cache line of a copy-back mode cache miss. The SFAR contains the address of the missed cache access, not the modified cache line address that caused the translation error. When this type of error occurs, the cache tag remains valid, and the cache line remains modified. Note that this bit is not used in the CY7C605, and is reserved. The physical address for a cache line is always available in the CY7C605, therefore making the CBT bit unnecessary in a CY7C605 based system.

The uncorrectable error (UE), timeout error (TO), and bus error bits (BE) report error status as encoded in the $\overline{\text{MERR}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MRDY}}$ signals. (Refer to the Section 4.12 on Mbus for further information.) The level bits (L) describe the level in a table walk process at which the fault occurred (if applicable). These bits are described in *Table 4–17* on page 4–49.

The access type bits (AT(2:0)) describes the access type that caused the fault. This field specifies user/supervisor access and whether the access is load or store of data or instruction. The AT bits are described in *Table 4–18* in the section on synchronous faults. The fault type bits (FT) describe the fault type, and are illustrated in *Table 4–19* on page 4–49. The fault address valid bit is set when the address in the synchronous fault address register (SFAR) is a valid fault address. The over-write bit (OW) is set in the case of a double fault where the fault status stored in the SFSR does not correspond with the fault first trapped on by the CY7C601. This is discussed in detail in the section on synchronous faults, page 4–47.

Upon power-on reset, the UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. Reading the synchronous fault status register clears all fault status bits.



RSV = Reserved
UC = Uncorrectable Error
TO = Time Out Error
BE = Bus Error
*CBT = Copy-back Translation Error
(*CY7C604 only; reserved in CY7C605)
L = Level
AT = Access Type
FT = Fault Type
FV = Fault Address Valid
OW = Over Write

Figure 4–39. CY7C604/605 Synchronous Fault Status Register

4.4.12 CY7C604/605 Synchronous Fault Address Register (SFAR)

The synchronous fault address register contains the faulted virtual address.

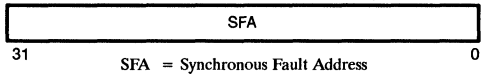


Figure 4–40. CY7C604/605 Synchronous Fault Address Register

4.4.13 CY7C604/605 Asynchronous Fault Status Register (AFSR)

Asynchronous faults are those faults caused by a delayed memory access initiated by the CY7C604/605. This type of error can only be caused by a delayed write to main memory initiated by the write buffer. Asynchronous faults cause the $\overline{\text{CMER}}$ signal to be asserted, which can be used as an interrupt to the CY7C601.

The UC, TO, and BE bits are identical to those in the SFSR. They are set by the information encoded into the $\overline{\text{MERR}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MRDY}}$ signals of the Mbus (see Section 4.12.4). The asynchronous fault address bits provide the upper four bits of the physical address not captured in the Asynchronous Fault Address Register (AFAR), which is a thirty-two bit register.

The Asynchronous Fault Occurred (AFO) bit is set when an asynchronous fault is encountered. Once the asynchronous fault occurred bit is set, no further asynchronous faults are recorded until the AFO bit is cleared, which is accomplished by reading the asynchronous fault address register (see *Figure 4–41).* The UC, TO, BE, and AFO bits in the AFSR will be cleared upon power-on reset. Reading the AFSR will also clear these bits.

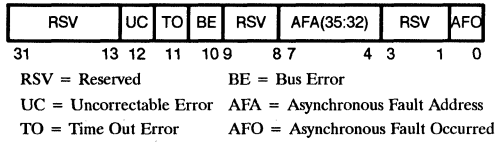


Figure 4–41. CY7C604/605 Asynchronous Fault Status Register

4.4.14 CY7C604/605 Asynchronous Fault Address Register (AFAR)

The AFAR contains bits 31 through 0 of the physical address for asynchronous faults (bus errors). Asynchronous faults can occur during delayed write accesses or during background cache line flush operations in copy-back mode (see *Figure 4–42).* The address in the AFAR is concatenated with the four AFA bits in the AFSR to define the entire 36-bit physical address.

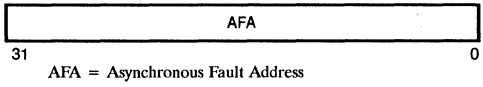


Figure 4–42. CY7C604/605 Asynchronous Fault Address Register

4.5 CY7C604 / CY7C605 Multichip Configuration

The CY7C604/605 is designed to allow expansion of the 64-kbyte cache by adding additional CY7C604/605s, each controlling two CY7C157 cache RAMs. A system using an expanded cache is required to configure the CY7C604/605s for multichip operation. Multichip operation is defined by the MultiChip Address field (MCA(1:0)), MultiChip Mask field (MCM(1:0)), and the Multichip Valid bit (MV) of the System Control Register (SCR). The two-bit MCA and MCM fields control the addresses to which the CY7C604/605 is allowed to respond. The multichip valid bit enables the multichip mode for the CY7C604/605, and is to be set when the MCA and MCM fields are configured for the system.

System initialization under multichip operation mode is handled by designating one of the CY7C604/605s to respond to all addresses from the CY7C601 until the CY7C604/605s have been initialized. This CY7C604/605 is referred to as the boot mode CY7C604/605. The other CY7C604/605s remain inactive until multichip operation has been set.

The boot mode CY7C604/605 is responsible for accesses to memory during system initialization. The boot mode CY7C604/605 responds to all memory accesses until multichip operation is enabled by setting the multichip fields of the SCR. The other CY7C604/605s remain inactive for all memory accesses until their SCR has been enabled for multichip mode. The non-boot mode CY7C604/605s three-states $\overline{MDS}$ and $\overline{MEXC}$.

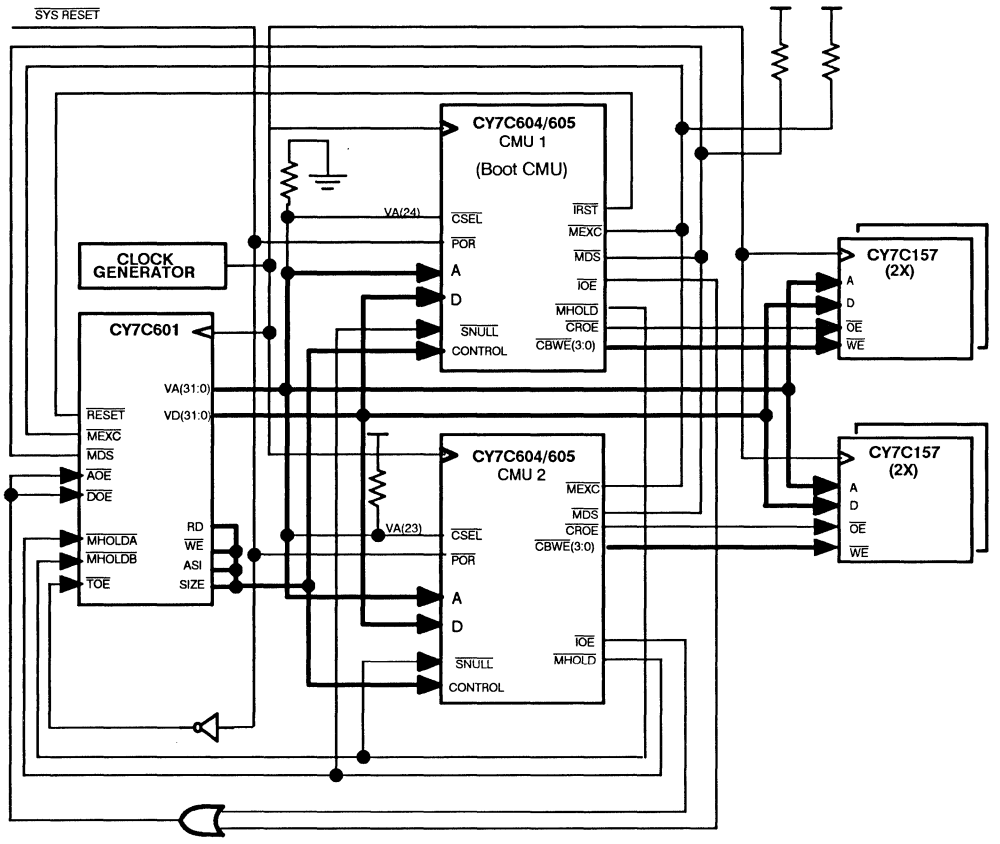


Figure 4–43. Two-CMU Multichip Configuration

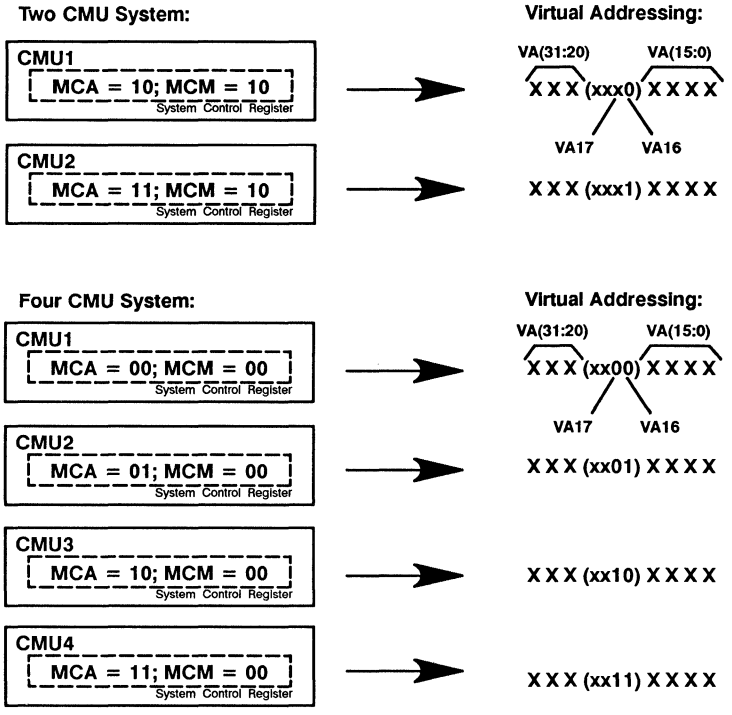


Figure 4–44. Examples of Multichip Addressing

The boot mode CY7C604/605 is selected by forcing LOW the $\overline{\text{CSEL}}$ signal as the power-on reset ($\overline{\text{POR}}$) signal is deasserted. The remaining CY7C604/605s are connected such that the $\overline{\text{CSEL}}$ signals are forced HIGH when the $\overline{\text{POR}}$ signal is deasserted. Each CY7C604/605 latches the state of its $\overline{\text{CSEL}}$ signal upon rising clock edge after $\overline{\text{POR}}$ is deasserted, and remains in either boot mode or becomes inactive until the multichip fields of its SCR have been set. (See $\overline{\text{CSEL}}$ power-on reset timing diagrams in Sections 7.4.7 and 7.5.7.) A single CY7C604/605 system should tie the $\overline{\text{CSEL}}$ signal to ground to ensure correct operation upon reset.

While multichip operation is not enabled, CY7C604/605 registers are addressed by using a combination of $\overline{\text{CSEL}}$, the register address, and ASI = 4. The $\overline{\text{CSEL}}$ signal of each CY7C604/605 is tied to one of the upper virtual address signals, thereby mapping the CY7C604/605 registers to different virtual addresses. These virtual addresses mapped using the $\overline{\text{CSEL}}$ signals are ignored by the CY7C604/605 after the multichip fields of the SCR are initialized. The non-boot mode CY7C604/605s will ignore all register accesses except to SCR until the multichip mode is enabled for the CY7C604/605.

All boot-mode CY7C604/605 registers can be accessed without enabling the multichip operation mode. Register access is accomplished by using a load or store alternate instruction with ASI = 4. Section 4.8 on ASI and Register Mapping describes the address mapping for the CY7C604/605. Note that after the multichip fields of the SCR have been set, $\overline{\text{CSEL}}$ is ignored for register addressing. All register accesses are mapped according to the MCM and MCA fields after the MV bit has been set.

The multichip fields of the SCR for the non-boot mode CY7C604/605s should be configured and enabled before the SCR for the boot mode CY7C604/605 is enabled. This prevents problems with the boot mode CY7C604/605 interfering during the configuration of the non-boot mode CY7C604/605s.

Figure 4–43 illustrates a 128-kbyte cache using two CY7C604/605s in a multichip configuration. Note that VA24 of the virtual address is connected to the $\overline{\text{CSEL}}$ input of CMU1 and is pulled to ground with a resistor. This signal is used to

access the CMU1 registers before multichip operation has been enabled. Using a pull-down resistor also accomplishes the task of forcing the $\overline{\text{CSEL}}$ signal for CMU1 to low, which is latched on the rising clock edge after $\overline{\text{POR}}$ is deasserted to enable the CY7C604/605 as the boot mode CMU. VA23 is connected to the $\overline{\text{CSEL}}$ input for CMU2. This signal is pulled up with a resistor to ensure that it is forced HIGH when the system reset signal is released. The virtual address bus (VA(31:0)) is three-stated by using the system reset signal to drive $\overline{\text{TOE}}$ HIGH, thereby forcing the CY7C601 off the address bus.

The $\overline{\text{SNULL}}$ input signal causes the CY7C604/605 to ignore an address on the virtual address bus. This input is used in multichip operation to keep a CY7C604/605 from responding to addresses output on the virtual address bus by other CY7C604/605s. The $\overline{\text{MHOLD}}$ output signal from a CY7C604/605 is used as the $\overline{\text{SNULL}}$ input for the remaining CY7C604/605s. *Figure 4–43* illustrates the $\overline{\text{MHOLD}}$ to $\overline{\text{SNULL}}$ connections for a two-CY7C604/605 system.

The multichip address bits (MCA(1:0)) of the System Control Register (SCR) select the state of the VA(17:16) bits that must be matched for multichip addressing. The multichip mask bits (MCM(1:0)) select which of the VA(17:16) bits can be ignored. The combination of the two fields define the address mapping for the CY7C604/605. The multichip valid bit (MV) must be set when writing to the MCA and MCM fields in order to enable multichip mode. *Figure 4–44* illustrates two examples of how these fields are used to define the address mapping for multiple CY7C604/605 systems.

4.6 CY7C604/605 Diagnostic Support

4.6.1 CY7C604/605 MMU TLB Entries

TLB entries can be accessed with a load or store alternate instruction with the TLB entry address and ASI = 6H. This feature is supported for diagnostic purposes and to provide CY7C601 access to locked TLB entries. The virtual and physical sections of each entry in the TLB can be accessed by the CY7C601 as a single-word read or write. The address mapping for the TLB entries is shown in *Table 4–11*. The format of CAM word and RAM word entries in the TLB is shown in *Figure 4–45*.

Table 4–11. TLB Entry Address Mapping

Address	TLB Entry Register
0 H	Entry 0 RAM Word
4 H	Entry 0 CAM Word
8 H	Entry 1 RAM Word
C H	Entry 1 CAM Word
10 H	Entry 2 RAM Word
14 H	Entry 2 CAM Word
• • •	• • •
1F0 H	Entry 62 RAM Word
1F4 H	Entry 62 CAM Word
1F8 H	Entry 63 RAM Word
1FC H	Entry 63 CAM Word
200-FFFFFFF8 H	Reserved

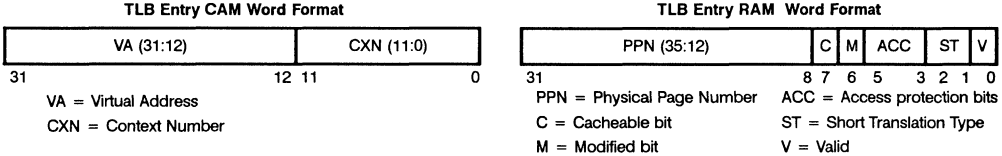


Figure 4–45. TLB Entry Format

Table 4–12. Cache Tag Entry Address Mapping

Address	Cache Tag Entry
000x H	0
002x H	1
004x H	2
006x H	3
•	•
•	•
•	•
FFEx H	2047

(x = don't care)

4.6.2 CY7C604/605 Cache Tag Entries

CY7C604 tag entries are accessed using a load or store alternate instruction with the cache tag entry address and ASI = 0E H. The CY7C605 PVTAG is accessed using a load or store alternate instruction specifing the entry address and ASI = 0E H. CY7C605 MPTAG entries are accessed in a similar manner using ASI = 30 H. Each tag entry can be read as a load single or can be written as a store single from the CY7C601. The address mapping for the cache tag entries is shown in *Table 4–12*. The format of a CY7C604 tag entry is shown in *Figure 4–46*. The CY7C605 PVTAG and MPTAG entry formats are illustrated in *Figure 4–47*.

4.6.3 CY7C604/605 Cache Data Entries

Cache data entries can be accessed from the cache RAM by using a load or store alternate instruction asserting the virtual address and ASI = 0F H. The CY7C604/605 cache controller causes a forced hit from the cache tag during these accesses. All data widths are supported for a read or write to the cache ram.

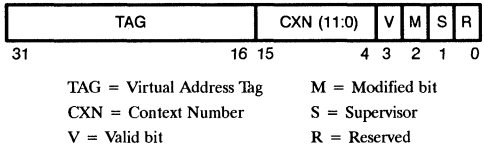


Figure 4–46. CY7C604 Cache Tag Entry Format

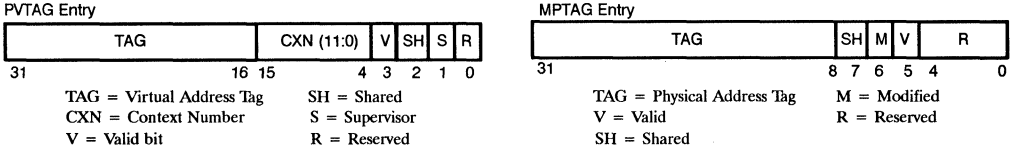


Figure 4–47. CY7C605 Cache Tag Entry Format

4.7 CY7C604/605 Reset

4.7.1 Power-On Reset ($\overline{POR}$)

Upon power-on reset, the entire system is forced into a defined state. The TLB and the cache tag(s) in the CY7C604/605 are invalidated, all valid bits in control registers are cleared, and certain bits in the ASFR and SFSR are cleared as described in the previous sections. The CY7C604/605 asserts $\overline{IRST}$ to the integer unit for as long as $\overline{POR}$ is asserted. $\overline{MRST}$ is not asserted. $\overline{POR}$ must be asserted for a minimum of 8 clocks. The bits in the reset register (RR) are cleared. Upon power-on reset, the UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. The SCR fields in the CY7C604/605 will have the following state after a power-on reset:

Table 4–13. CY7C604/605 Power-On Reset States

IMPL	Unchanged
VER	Unchanged
MCA(1:0)	Unchanged
MCM(1:0)	Unchanged
MV	0
BM	1
C	0
CM	0
CL	0
CE	0
NF	0
ME	0
MR	0

4.7.2 Watch-Dog Reset (WDR)

When the CY7C601 encounters a trap while traps are disabled, the CY7C601 enters into an error state, asserts the $\overline{ERROR}$ signal, and then halts. The only way to restart the CY7C601 in the error state is to assert its $\overline{RESET}$ signal. The CY7C604/605 does this by performing a watch-dog reset, which asserts the $\overline{IRST}$ signal for 1024 clock cycles. $\overline{MRST}$ is not asserted. The TLB and the cache tag(s) in the CY7C604/605 are not invalidated. The WDR (RR[2]) bit in the RR register is set. All SCR fields except boot mode (BM) are unchanged. BM is set to 1 after a watch-dog reset.

4.7.3 Software Internal Reset (SIR)

The operating system can reset the CY7C601 by setting the SIR bit in the reset register. The CY7C604/605 asserts $\overline{IRST}$ for 1024 clock cycles to reset the CY7C601. The TLB and the cache tag are not invalidated. All SCR fields except BM are unchanged, and BM is set to 1 after a software internal reset. The contents of the reset register are unchanged and the SIR bit will remain set. Refer to page 4–83 for timing diagrams for the SIR and SER resets.

4.7.4 Software External Reset (SER)

The operating system can reset the system separately from the CY7C601 by writing 1 into the SER bit of the RR register. Only the writing of a 1 into the SER bit will cause $\overline{MRST}$ to be asserted. The CY7C604/605 asserts $\overline{MRST}$ for 1024 Mbus clock cycles to reset the system. The TLB and the cache tag are not invalidated. The SCR register remains unchanged. The CY7C604/605 will wait for its write buffers to empty before asserting $\overline{MRST}$ on a software external reset. The contents of the reset register are unchanged and the SER bit will remain set.

$\overline{MRST}$ will not be asserted on a software external reset until the write buffers have been flushed. Writing both the SIR and SER bits in the reset register will cause the assertion of both $\overline{IRST}$ and $\overline{MRST}$. A reset routine can poll the reset register to determine the source of any reset.

4.7.5 CY7C604/605 Reset in Multichip Configuration

In a multichip configuration, the CY7C604/605 that is responsible for handling boot mode can also assume the responsibility to handle the Reset operations described above. The $\overline{IRST}$ to the CY7C601 and the $\overline{MRST}$ to the external system are connected only to this responsible CY7C604/605. The reset signals from the other CY7C604/605s are not connected. The $\overline{ERROR}$ pin of the CY7C601 should be connected to all CY7C604/605s thereby putting all CY7C604/605s in the same state during watch dog reset. Only the $\overline{IRST}$ of the boot-handling CY7C604/605 is connected to the $\overline{RESET}$ input of the CY7C601.

When performing a software internal reset in a multichip configuration, the reset register SIR bit should be set in all the non-boot-handling CY7C604/605s before SIR is set in the boot-handling CY7C604/605. This places all CY7C604/605s contained in the system in the same mode before the CY7C601 is reset. A software external reset in a multichip configuration can be performed by writing the SER bit in the boot-handling CY7C604/605 only. It is not necessary to alter the non-boot-handling CY7C604/605s.

4.8 CY7C604/605 ASI and Register Mapping

The CY7C604/605 uses the address space identifier bus (ASI < 5:0 >) to provide access by the CY7C601 to internal registers and resources, such as the cache tag and the TLB. The CY7C604/605 also uses the ASI bus to map restricted memory access functions, such as local and pass-through memory addressing modes. Register access to the CY7C604/605 requires using a load or store alternate instruction with ASI = 04 H in addition to the register address, given in *Table 4–14. Table 4–15* illustrates the ASI mapping for the CY7C604/605.

Table 4–14. CY7C604/605 Register Address Mapping

VA (15:0)	CY7C604/605 Registers
0 H	System Control Register (SCR)
100 H	Context Table Pointer Register (CTPR)
200 H	Context Register (CXR)
300 H	Synchronous Fault Status Register (SFSR)
400 H	Synchronous Fault Address Register (SFAR)
500 H	Asynchronous Fault Status Register (AFSR)
600 H	Asynchronous Fault Address Register (AFAR)
700 H	Reset Register (RR)
800 - F00 H	Reserved
1000 H	Root Pointer Register (RPR)
1100 H	Instruction Access PTP (IPTP)
1200 H	Data Access PTP (DPTP)
1300 H	Index Tag Register (ITR)
1400 H	TLB Replacement Control Register (TRCR)
1500 - FF00 H	Reserved

Table 4–15. Standard ASI Assignments

ASI	Function	ASI	Function
0 H	Reserved	12 H	Flush combined cache line (region)*
1 H	Mbus extended address space*	13 H	Flush combined cache line (context)*
2 H	Unassigned	14 H	Flush combined cache line (user)*
3 H	MMU flush/probe*	15 H	Reserved
4 H	MMU registers*	16 H	Reserved
5 H	MMU diagnostics instruction only TLB	17 H	Block copy
6 H	MMU diagnostics instruction/data TLB*	18 H	Flush data cache line (page)
7 H	MMU diagnostics I/O TLB	19 H	Flush data cache line (segment)
8 H	User instruction*	1A H	Flush data cache line (region)
9 H	Supervisor instruction*	1B H	Flush data cache line (context)
A H	User data*	1C H	Flush data cache line (user)
B H	Supervisor data*	1D H	Reserved
C H	Cache tag for instruction cache	1E H	Reserved
D H	Cache data for instruction cache	1F H	Block zero
E H	Cache tag combined(inst/data) cache* (PVTAG)**	20-2F H	MMU passthrough physical address*
F H	Cache data for combined cache*	30 H	MPTAG cache tag entries **
10 H	Flush combined cache line (page)*	31-7F H	Unassigned
11 H	Flush combined cache line (segment)*	80-FF H	Reserved

*Indicates functions supported by the CY7C604 and CY7C605

**Indicates function is specific to the CY7C605

4.9 Synchronous Faults

Synchronous faults are grouped into three classes: instruction access faults, data access faults, and translation table access faults. The translation table access faults are further divided into translation instruction access faults and translation data access faults. The SPARC architecture causes the timing and priority of these fault classes to be handled differently. Due to delays caused by the instruction pipeline, the CY7C601 can possibly encounter a second fault before the CY7C601 enters a trap to correct the first. Depending upon the class of fault encountered, the status and address of a fault may be allowed to overwrite information for a previous fault that has not yet generated a trap. This potential condition requires a trap handler that can correct the various combinations of fault conditions. This section describes these potential fault conditions.

The case of a pair of faults occurring presents a problem in reporting the correct fault status. This problem is solved by use of an overwrite (OW) bit in the SFSR and by prioritizing which types of faults may overwrite a previous fault. The OW bit signals the trap handler that the status and address stored in the fault registers are not valid for the trap that the CY7C601 has entered. The SFSR logic sets the OW bit according to a state sequence based on the fault handling of the CY7C601 and the type of faults encountered.

Since the CY7C601 delays entering a trap handler for an instruction fault, a trap caused by another fault will overwrite the trap information for the initial instruction fault. If the second fault causes a trap in the CY7C601 before the initial instruction fault trap is entered, the OW bit is not set. This is because the information in the fault registers will be correct for the first trap reading the registers. However, if the initial instruction trap is entered before the second fault trap is entered, the OW bit will be set. This is because the first trap reading the fault status registers will have the fault data for the second trap. The OW bit is set only if the trap that will be executed first by the CY7C601 does not match the status information stored in the SFSR. The setting of the OW bit is entirely based upon the types of faults and their order of occurrence. *Table 4–16* illustrates the possible fault cases and their effect on OW.

Table 4–16. OW Bit States

First Fault	Second Fault	Update SFSR	OW
single fault		yes	0
instruction	instruction	yes	1
instruction	data	yes	0
instruction	translate instr.	yes	1
instruction	translate data	yes	0
data	instruction	no	0
data	data	yes	1*
data	translate	yes	1
translate	instruction, data	no	0
translate	translate	no	0

*NOT POSSIBLE with CY7C601 (and related processors)

The CY7C601 delays a trap caused by an instruction access fault until that instruction reaches the execute stage. However, since data accesses are not pipelined, the CY7C601 jumps to a trap immediately upon encountering a data access fault.

Faults are allowed to overwrite another fault status dependent upon priority. An instruction fault is allowed to overwrite only another instruction fault. It is not allowed to overwrite either a data fault or a translation fault. Data faults may overwrite an instruction fault, but not a translation fault. Data faults cannot overwrite another data fault, since the CY7C601 traps immediately upon encountering a data fault. Translation faults may overwrite any type of fault, but cannot be overwritten. Translation faults may not overwrite another translation fault.

All double fault cases are recoverable by re-executing the instruction or access that caused the fault whose status has been overwritten. If an instruction access fault occurs and the OW bit is set, the system software must determine the cause by probing the MMU and/or memory.

Upon encountering a synchronous fault, the SFSR records the bus error status (bus error, timeout, and uncorrectable error) when a bus error occurs during memory accesses. The level field (L), as shown in *Table 4–17*, is set to the page table level of the entry that caused the fault, if the fault is associated with a table walk. The access type (AT) field, illustrated in *Table 4–18*, defines the type of access that caused the fault. The fault type field FT (see *Table 4–19*) defines the type of the current fault.

A translation table access fault (FT = 4) occurs if an MMU page table access causes an external system error. This also occurs if a reserved entry type (ET = 3 in the PTE) is found in any level of the table walk. A translation table access fault (FT = 4) also can occur if a PTP (page table pointer) is found in level 3, instead of a PTE. If the page table entry is invalid (ET = 0 in the PTE), the fault type is an invalid address error (FT = 1). *Table 4–20* illustrates the fault type (FT) assigned for valid TLB entries or PTE entries (ET = 2) that cause a fault condition. These fault conditions are always either a protection error (read/write of data or instruction) or a privilege violation (user/supervisor access) error.

The copy-back translation fault bit (CBT) is set if there is an error occurring during a table walk for a modified cache line replacement or during a modified cache line flush operation. The fault address valid bit (FAV) is set to one if the content of the synchronous fault address register is valid. The SFAR may not be valid for instruction faults. The SFAR is always valid for data faults and translation errors.

If multiple fault types apply to the same fault occurrence, the highest priority fault is recorded. The highest fault priority is a translation fault (priority 2), as shown in *Table 4–21*. Priority 1 is reserved for an internal fault.

Upon power-on reset, the UC, TO, BE, FT, FAV, and OW bits in the SFSR will be cleared. Reading the synchronous fault status register clears all fault status bits.

Table 4–17. Fault Register Level Field

L	Level
0	Entry in Context Field
1	Entry in Level 1 Table
2	Entry in Level 2 Table
3	Entry in Level 3 Table

Table 4–18. Fault Register Access Type Field

AT	Access Type
0	Load from User Data Space
1	Load from Supervisor Data Space
2	Load/Execute from User Instruction Space
3	Load/Execute from Supervisor Instruction Space
4	Store to User Data Space
5	Store to Supervisor Data Space
6	Store to User Instruction Space
7	Store to Supervisor Instruction Space

Table 4–19. Fault Register Fault Type Field

FT	Fault Type
0	None
1	Invalid Address Error
2	Protection Error
3	Privilege Violation Error
4	Translation Error
5	Bus Access Error
6	Not Generated
7	Reserved

Table 4–20. Fault Type (FT) for PTE[ET] = 2

AT	ACC							
	0	1	2	3	4	5	6	7
0	0	0	0	0	2	0	3	3
1	0	0	0	0	2	0	0	0
2	2	2	0	0	0	2	3	3
3	2	2	0	0	0	2	0	0
4	2	0	2	0	2	2	3	3
5	2	0	2	0	2	0	2	0
6	2	2	2	0	2	2	3	3
7	2	2	2	0	2	2	2	0

Table 4–21. Fault Register Error Priorities

Priority	Error
1	Internal Error
2	Translation Error
3	Invalid Address Error
4	Privilege Violation Error
5	Protection Error
6	Bus Access Error

4.9.1 Synchronous Fault Cases

The following seventeen cases describe the combinations of fault cases that can occur:

Case 1: *Instruction fault with no further faults.* The CY7C601 trap is delayed until the CY7C601 tries to execute the instruction.

The trap is taken immediately if the instruction access is actually a data access that is interpreted by the CY7C604/605 as an instruction access due to asserting ASI = 8 or 9 with a load alternate instruction. In this case, the trap handlers cannot probe main memory using the PC of the instruction. If the instruction is a load alternate instruction, the trap handler has to calculate the effective address to probe. The SFAR has the valid address if the OW bit is not set.

Case 1: Single-Instruction Fault		
OW	0	
FAV	1	SFAR has valid address
FT	1	Invalid error occurred (ET = 0 during table walk)
	2	Protection error occurred (either TLB or table walk)
	3	Privilege violation error occurred (either TLB or table walk)
	5	Bus access error occurred (external bus error: UC or TO or BE is set).
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which fault occurred during table walk (only valid with FT = 1)

Case 2: *Double instruction fault.* Instruction fault (1) followed by another instruction fault (2); CY7C601 traps on instruction fault (1).

If the instruction fault (2) is due to a load access with ASI 8,9 (load alternate), it overwrites the fault associated information of fault (1). In this case the SFAR has a valid address for the data access of the load alternate instruction.

The fault address of fault (1) can be obtained from the PC in the CY7C601 for the trap handler with the exception of the following case.

A possible case is that of a data access interpreted by the CY7C604/605 as an instruction access because of the use of a load or store alternate instruction with ASI = 8, 9. Before the CY7C601 takes the trap on the data access fault (which is recorded as an instruction fault in the CY7C604/605), another instruction fault may occur. The second instruction will overwrite the data access fault information, because it is recorded as an instruction fault in the CY7C604/605. In this case, the trap handler can not just probe on the PC of the instruction. If the instruction is a load alternate instruction, the trap handler has to calculate the effective address to probe and the SFAR will not contain the fault address of the data access fault.

Case 2: Double-Instruction Fault		
OW	1	
FAV	1	SFAR has valid address for fault (2)
FT	1,2,3,5	Fault type of fault (2)
AT	2,3	Access type of fault (2)
L	0,1,2,3	Level at which fault (2) occurred during table walk (only valid with FT = 1)

Case 3: *Single data fault.* CY7C601 trap (taken immediately)

Case 3: Single Data Fault		
OW	0	
FAV	1	SFAR has valid address
FT	1	Invalid error occurred (ET = 0 during table walk)
	2	Protection error occurred (either TLB or table walk)
	3	Privilege violation error occurred (either TLB or table walk)
	5	Bus error occurred (external bus error, UC or TO or BE is set)
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which fault occurred during table walk (only valid with FT = 1)

Case 4: *Instruction fault followed by data fault.* CY7C601 traps on the data fault

The history of the instruction fault is lost, but the same fault can be obtained again, once the return from the trap handler of the data fault is completed.

Case 4: Instruction Fault then Data Fault		
OW	0	
FAV	1	SFAR has valid address for data fault
FT	1,2,3,5	Fault type of data fault
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which data fault occurred during table walk (only valid with FT = 1)

Case 5: *Data fault followed by instruction fault.* The instruction fault cannot overwrite the data fault. The instruction fault will occur again, once the return from the data fault trap handler is completed. CY7C601 will trap on data fault.

Case 5: Data Fault then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for data fault
FT	1,2,3,5	Fault type of data fault
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which data fault occurred during table walk (only valid with FT = 1)

Case 6: *Data fault followed by data fault.* (NOT POSSIBLE with CY7C601.)

Case 7: *Translation fault (instruction access); no further faults.* The CY7C601 trap is delayed until the CY7C601 tries to execute the instruction or is taken immediately if the access is data due to a load alternate instruction.

Case 7: Translation Fault on Instruction Access		
OW	0	
FAV	1	SFAR has valid address for translation fault.
FT	4	Translation error occurred (bus error or ET = 3 or PTP in level 3 during table walk)
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 8: *Translation fault (data access).* The CY7C601 trap is taken immediately.

Case 8: Translation Fault on Data Access		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	Translation error occurred (bus error or ET = 3 or PTP in level 3 during table walk)
AT	0,1,4,5,6,7	
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 9: *Instruction fault followed by translation fault (instruction.)* The CY7C601 traps on the instruction fault.

The fault address of the instruction fault can be obtained from the PC in the CY7C601 for the trap handler with the exception of the following case.

A data access fault can be recorded as an instruction fault if a load alternate instruction with ASI = 8, 9 causes a fault. Before the CY7C601 takes the trap on the data access fault (which is recorded as an instruction fault in the CY7C604/605), a translation fault may occur due to an instruction access. This will overwrite the data access fault information.

Case 9: Instruction Fault then Translation Fault (I)		
OW	1	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 10: *Translation fault (instruction access) followed by instruction fault.* The CY7C601 traps on the translation fault. The instruction fault cannot overwrite the translation fault.

Case 10: Translation Fault (I) then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 11: *Translation fault1 (instruction access) followed by translation fault2 (instruction).* The CY7C601 traps on translation fault1.

Case 11: Translation Fault (I) then Translation Fault (I)		
OW	0	
FAV	1	SFAR has valid address for first translation fault
FT	4	
AT	2,3	Load/Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which first translation fault occurred during table walk

The second translation fault cannot overwrite the first translation fault.

Case 12: *Translation fault1 (instruction access) followed by translation fault2 (data access).* The CY7C601 traps on translation fault2. The translation fault2 cannot overwrite translation fault1.

Case 12: Translation Fault (I) then Translation Fault (D)		
OW	0	
FAV	1	SFAR has valid address for translation fault1
FT	4	
AT	2,3	Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault1 occurred during table walk

Case 13: *Translation fault (instruction access) followed by data fault.* The CY7C601 traps on the data fault. The data fault cannot overwrite the translation fault.

Case 13: Translation Fault (I) then Data Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 14: *Data fault followed by translation fault (instruction access).* The CY7C601 traps on the data fault.

Before the CY7C601 takes the trap on the data access fault, a translation fault may occur due to an instruction access. This will overwrite the data access fault information.

Case 14: Data Fault then Translation Fault (I)		
OW	1	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	2,3	Execute from User/Supervisor instruction space
L	0,1,2,3	Level at which translation fault occurred during table walk

Case 15: *Instruction fault followed by translation fault (data).* The CY7C601 will trap on the data fault.

Case 15: Instruction Fault then Translation Fault (D)		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0,1,4,5,6,7	
L	0,1 2,3	Level at which translation fault occurred during table walk

Case 16: *Translation fault (data) followed by instruction fault.* The CY7C601 will trap on the data fault.

Case 16: Translation Fault (D) then Instruction Fault		
OW	0	
FAV	1	SFAR has valid address for translation fault
FT	4	
AT	0,1,4,5,6,7	
L	0,1 2,3	Level at which translation fault occurred during table walk

Case 17: *Translation fault (data) followed by translation fault (instruction).* The CY7C601 will trap on the data fault.

Case 17: Translation Fault (D) then Translation Fault (I)		
OW	0	
FAV	1	SFAR has valid address for data translation fault
FT	4	
AT	0,1,4,5,6,7	
L	0,1 2,3	Level at which translation fault occurred during table walk

4.10 CY7C604/605 Pin Definitions

The functional pinouts for the CY7C604 and CY7C605 are shown in *Figure 4–48*. Note that all three-state output signals are driven to their inactive state before they are released to three-state. All signals described are common to both the CY7C604 and CY7C605 unless otherwise stated.

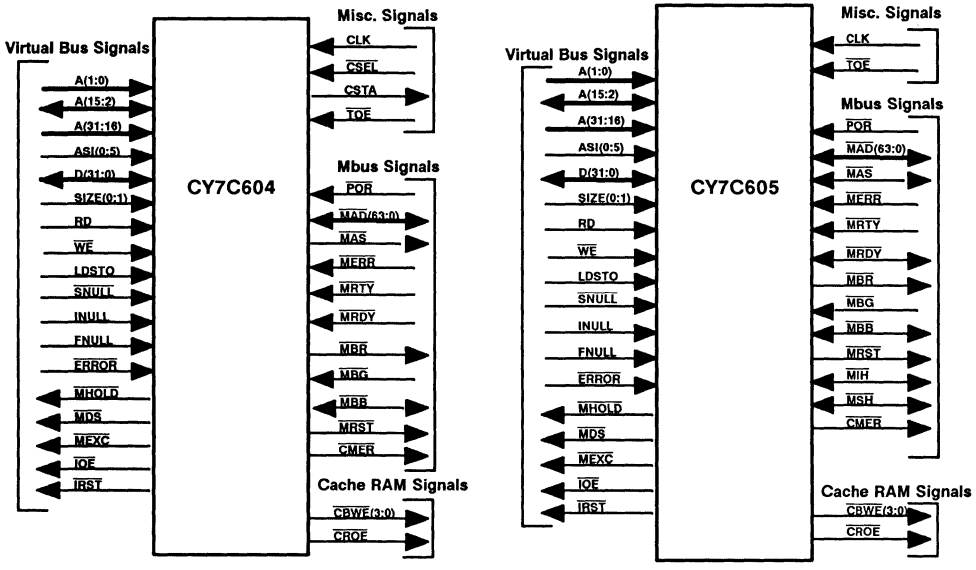


Figure 4–48. CY7C604 and CY7C605 I/O Signals

CY7C604/605 Virtual Bus Signals

Signal Name	I/O	Description
A(31:16)	I	Virtual Address bus. A(31:16) are input signals during normal read/write accesses and are latched into the CY7C604/605 on the rising edge of clock.
A(15:2)	I/O	Virtual Address bus. Three-state input/output signals. A(15:2) are input signals during normal read/write accesses and are latched into the CY7C604/605 on the rising edge of the clock. They are output signals during cache line loads into the CACHE RAM and modified cache-line reads from the CACHE RAM.
A(1:0)	I	Virtual Address bus. A(1:0) are input signals during normal read/write accesses and are latched on the rising edge of clock.
ASI(5:0)	I	Address Space Identifiers. The ASI bits are used to: 1. Identify various types of accesses (user/supervisor, instruction/data) 2. Access CY7C604/605 registers 3. Initiate MMU Flush/Probe operation 4. Identify CACHE Flush operations 5. Recognize diagnostic operations 6. Recognize pass physical address space

Signal Name	I/O	Description
D(31:0)	I/O	Virtual Data bus. Three-state input/output signals. D(31:0) are input signals during CY7C601 normal write accesses, modified cache-line reads from the CACHE RAM, CY7C604/605 register writes or CY7C604/605 diagnostic accesses. They are output signals during cache line loads into CACHE RAM, CY7C604/605 register reads, non-cacheable loads, or CY7C604/605 diagnostic accesses.
$\overline{\text{ERROR}}$	I	Error (active LOW) signal from the CY7C601. When this signal is asserted, it indicates the CY7C601 has halted due to entering the error state. The CY7C604/605 reads this signal and initiates a watchdog reset. (Refer to Section 4.7.2 for more details.)
FNULL	I	Floating-point unit NULLification cycle (active HIGH). When FNULL is active, the current access is ignored.
INULL	I	Integer unit NULLification cycle (active HIGH). When INULL is active, the current access is ignored.
$\overline{\text{IOE}}$	O	Integer unit Output Enable (active LOW). This signal is continually driven HIGH or LOW. This signal is connected to the $\overline{\text{AOE}}$ and $\overline{\text{DOE}}$ inputs of the CY7C601. When deasserted (HIGH), the $\overline{\text{IOE}}$ will place the address (A(31:0)), address space identifiers (ASI(7:0)), and data (D(31:0)) drivers of the CY7C601 in a three-state condition.
$\overline{\text{IRST}}$	O	Integer unit Reset (active LOW) is asserted to reset the integer unit. (Refer to Section 4.7.2 for more details.) This signal is continually driven HIGH or LOW.
LDSTO	I	Load-Store Atomic operation indicator (active HIGH). Asserted by the CY7C601 during atomic load store cycles and is sampled by the CY7C604/605 on the rising edge of the clock.
$\overline{\text{MDS}}$	O	Memory Data Strobe (active LOW) is asserted for one clock to strobe data into the CY7C601 during a cache miss. $\overline{\text{MHOLD}}$ must be low when $\overline{\text{MDS}}$ is asserted. It is driven off of the falling edge of the clock. This is a three-state output.
$\overline{\text{MEXC}}$	O	Memory Exception (active LOW) is asserted for one clock whenever a privilege or protection violation is detected. $\overline{\text{MHOLD}}$ and $\overline{\text{MDS}}$ must be low when $\overline{\text{MEXC}}$ is asserted. This is a three-state output.
$\overline{\text{MHOLD}}$	O	Memory Hold (active LOW) is asserted by the CY7C604/605 whenever it requires additional time to complete the current access such as during cache miss etc. It is driven off of the falling edge of the clock.
RD	I	Read cycle indicator (active HIGH). Asserted by the CY7C601 during read cycles and is sampled by the CY7C604/605 on the rising edge of the clock. This signal is also used to generate cache read output enable ($\overline{\text{CROE}}$)
SIZE(1:0)	I	SIZE of access indicator. Specifies the data width of the CY7C601 access and is sampled by the CY7C604/605 at the rising edge of the clock.
$\overline{\text{SNULL}}$	I	System NULLification cycle (active LOW). When $\overline{\text{SNULL}}$ is active, the current access is ignored.
$\overline{\text{WE}}$	I	Write Enable to indicate write cycle (active LOW). Asserted by the CY7C601 during write cycles and is sampled by the CY7C604/605 on the rising edge of the clock. This signal is also used to generate cache byte-write enables ($\overline{\text{CBWE}}$(3:0)).

Mbus Signals

Signal Name	I/O	Description
$\overline{\text{CMER}}$	O	CMU Error (active LOW). This signal is asserted if any bus error has occurred during writes to main memory. A system can use this signal to cause an interrupt. This signal has the same timing specifications as the Mbus control signals and is asserted for one clock. This signal is constantly driven.
MAD(63:0)	I/O	Mbus Address and Data (three-state bus). During the address phase of a transaction, MAD(35:0) contains the physical address PA(35:0). The remaining signals MAD(63:36) contain the transaction-associated information, as shown below: **MAD(39:36) Transaction Type** 0 H — Mbus write 1 H — Mbus read 2 H* — Coherent invalidate 3 H* — Coherent read 4 H* — Coherent write and invalidate 5 H* — Coherent read and invalidate 6–F H — Reserved *CY7C605 ONLY **MAD(42:40) Transaction Size** 0 — Byte (8 bits) 1 — Halfword (16 bits) 2 — Word (32 bits) 3 — Doubleword (64 bits) 4 — 16 Bytes** 5 — 32 Bytes 6 — 64 Bytes** 7 — 128 Bytes** **Not supported by CY7C604/605. MAD(43) (MC) Mbus Cacheable (active HIGH). Indicates the current Mbus transaction is cacheable. MAD(44) (MLOCK) Mbus LOCK (active HIGH). Indicates the currrent Mbus transaction is a locked transaction. MAD(45) (MBL) Mbus Boot mode/Local indicator. MBL is high during the address phase of boot mode transactions. The instruction fetch and data accesses to the Mbus while the MMU is disabled in boot mode are considered BOOT MODE transactions. The data transactions on the Mbus required for Load/Store Alternate instructions with ASI = 01 are considered LOCAL transactions. MAD(63:46) Reserved during the address phase (driven HIGH). During the data phase of the transaction the MAD(63:0) lines contain the 64 bits of data being transferred.
$\overline{\text{MAS}}$	O (604) I/O(605)	Mbus Address Strobe (active LOW). Asserted by the bus master during the first cycle of every bus transaction to indicate the address phase of that transaction. This is a three-state output.
$\overline{\text{MBB}}$	I/O	Mbus Bus Busy (active LOW). Asserted by the current Mbus master during an entire transaction and, if required, during both the read and write transactions of indivisible accesses. The potential bus master devices sample $\overline{\text{MBB}}$ in order to obtain bus mastership as soon as the current master releases the bus. This is a three-state output.

Signal Name	I/O	Description
$\overline{MBG}$	I	Mbus Bus Grant (active LOW). Asserted by external arbiter when the Mbus is granted to a master. This signal is continually driven.
$\overline{MBR}$	O	Mbus Bus Request (active LOW). Asserted by potential Mbus master devices to acquire bus mastership. This signal is continually driven.
$\overline{MERR}$	I	Mbus Error (active LOW). Asserted or deasserted by an Mbus slave during every data phase of a transaction. This signal is three-stated when released.
$\overline{MIH}$ (605 ONLY)	I/O	Memory InHibit (active LOW). Asserted by the CY7C605 for Mbus transactions where the cache owns the data that has been requested on the Mbus. This signal is monitored during bus snooping by the CY7C605. Refer to section 4.12 for further details.
$\overline{MRDY}$	I (604) I/O (605)	Mbus Ready (active LOW). Asserted or deasserted by an Mbus slave during every data phase of a transaction. This signal is to be three-stated when released.
$\overline{MRST}$	O	Mbus Reset (active LOW). Asserted for 1024 clock cycles by only one source on the Mbus to initialize all devices on the Mbus. This signal is continually driven.
$\overline{MRTY}$	I	Mbus Retry (active LOW). Asserted or deasserted by an Mbus slave during every data phase of a transaction. This signal is three-stated when released.

$\overline{MERR}$	$\overline{MRDY}$	$\overline{MRTY}$	Action
H	H	H	Nothing
H	H	L	Relinquish and Retry*
H	L	H	Data Strobe
H	L	L	Reserved
L	H	H	Bus Error
L	H	L	Time Out
L	L	H	Uncorrectable Error
L	L	L	Retry*

* See Section 4.12 on Mbus.

$\overline{MSH}$ (605 ONLY)	I/O	Memory SHared (active LOW). Asserted by the CY7C605 after detecting a data request on the Mbus for which the CY7C605 has a copy. This signal is monitored by the CY7C605 during bus snooping. Refer to Section 4.12 for further information.
$\overline{POR}$	I	Power-On Reset (active LOW). The $\overline{POR}$ initializes all on-chip logic to a known state, invalidates all the TLB entries, and all cache tag entries. It must be asserted for a minimum of 8 clocks. It also causes the CY7C604/605 to assert $\overline{IRST}$ to reset the CY7C601.

Cache RAM Signals

Signal Name	I/O	Description
$\overline{\text{CBWE}}$(3:0)	O	Cache Byte Write Enables (active LOW). During normal write operations, certain byte enable signals are asserted depending upon the size and A(1:0) inputs. During a cache line load all four byte enable signals are asserted. These signals can also be driven by using a store alternate instruction with ASI = 0F H. This feature is supported for diagnostic purposes. This output is continually driven (not three-stated). $\overline{\text{CBWE0}}$ controls the most significant byte (MSB) and $\overline{\text{CBWE3}}$ controls the least significant byte (LSB). Refer to page 4–34 for further information on this signal.
$\overline{\text{CROE}}$	O	Cache RAM Output Enable (active LOW). Asserted during normal read operations with ASI = 8, 9, A, B, and during modified cache line read operations. This signal is also asserted during cache data read operations with ASI = 0F H for diagnostic purposes. This signal is continually driven.

Miscellaneous Signals

Signal Name	I/O	Description
CLK	I	System Clock. This is the same clock used by the 7C601 integer unit.
$\overline{\text{CSEL}}$ (604 only)	I	Chip Select (active low). In multi-CMU systems, $\overline{\text{CSEL}}$ on each CY7C604 is connected to different address lines (any one from A(31:16)) to initialize the Multichip Configuration. In single-CMU systems, $\overline{\text{CSEL}}$ should be connected to ground in order to permanently enable the CY7C604. In multi-CMU systems, $\overline{\text{CSEL}}$ should be connected to ground or VCC through a resistor during power-on reset. This is required in order to enable only one boot mode CMU. (Refer to Multichip Configuration, Section 4.5, for more details.)
CSTA (604 only)	O	Cache Status. This pin provides the status of cache. In write-through, the CSTA indicates whether the write transaction on the Mbus is associated with a cache hit or not. For read transaction on the Mbus in either write-through or copy-back mode, the CSTA indicates whether the CY7C604 is replacing a valid cache line entry or not. This signal has the same timing specifications as the Mbus signals such as MC and has meaning only in the address phase of Mbus transactions. This signal is continually driven HIGH or LOW.

Cache Mode	CSTA	Condition
Write-through	1	read and valid cache line replacement
	0	read and invalid cache line replacement
	1	write and cache hit
	0	write and cache miss
Copy-back	1	read and valid cache line replacement
	0	read and invalid cache line replacement
	undef.	write

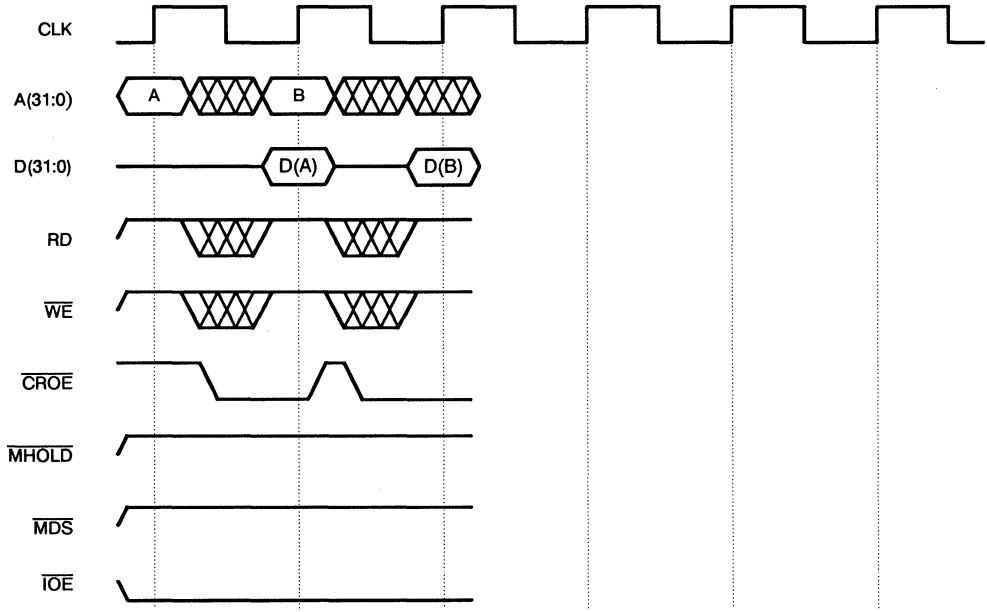
$\overline{\text{TOE}}$	I	Test/Output Enable (active LOW). When HIGH, this signal is used to three-state all output drivers of the CY7C604/605. $\overline{\text{TOE}}$ SHOULD BE TIED LOW DURING NORMAL OPERATION. It is used to isolate the CY7C604/605 from the rest of the system for debugging purposes.

4.11 Virtual Bus Operation

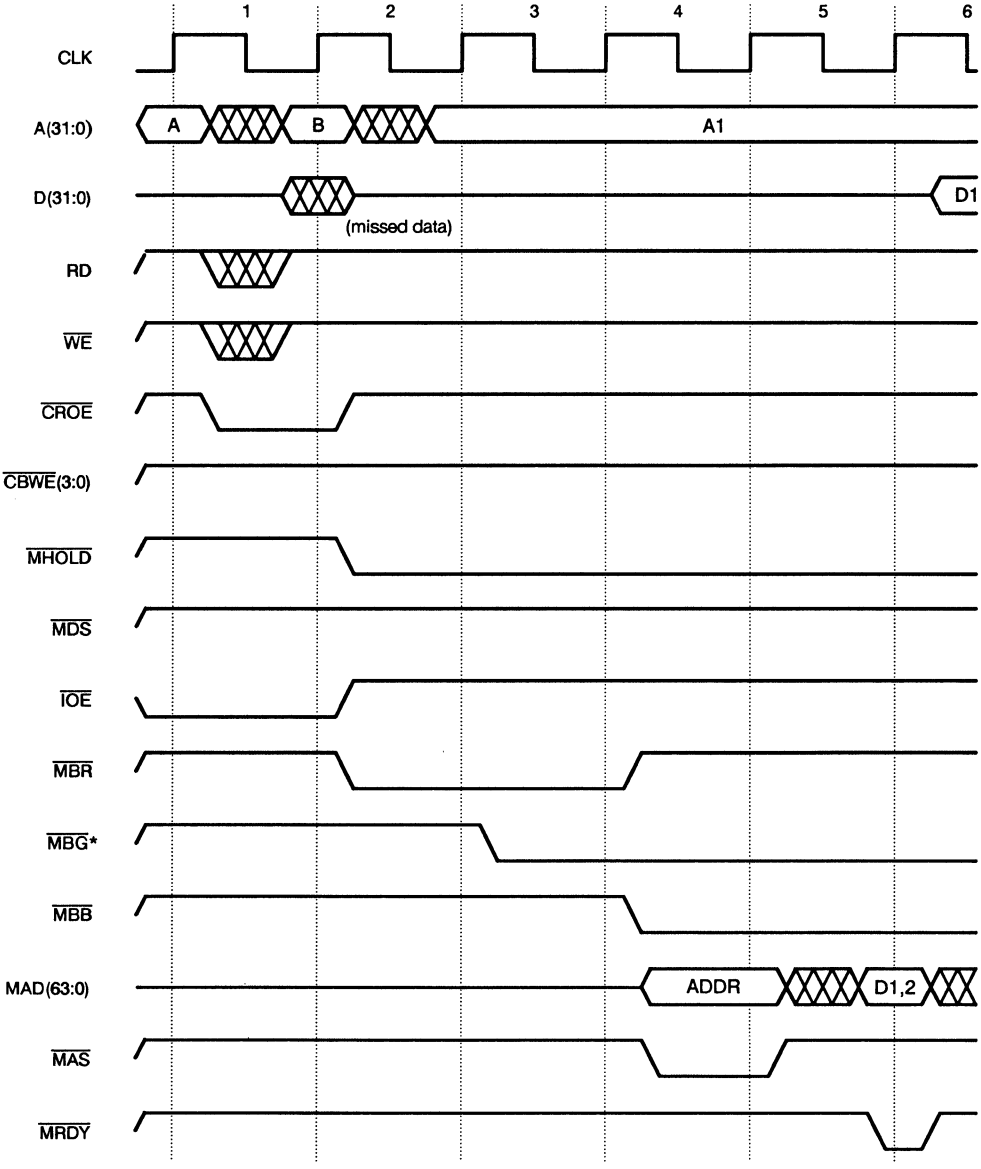
The following timing diagrams illustrate CY7C604/605 virtual bus operations:

	Page
Write-Through Read Cache Hit	4-60
Write-Through Read Cache Miss	4-61
Write-Through Read Cache Miss (Alias Detected)	4-64
Write-Through Write Cache Hit	4-65
Write-Through Write Cache Miss	4-66
Copy-Back, Read Cache Miss (Modified Cache Line)	4-67
Copy-Back Read Cache Miss, Modified or Non-Modified (Alias Detected)	4-72
Copy-Back Write Cache Miss, Modified or Non-Modified (Alias Detected)	4-73
Copy-Back Write Cache Hit	4-74
Write-Through Load Double Cache Hit	4-74
Write-Through Store Double Cache Hit	4-75
Table Walk (with Modified Bit Update)	4-76
Read Access with Protection/Privilege Violation	4-80
CY7C604/605 Diagnostic Cache Tag Write Access	4-80
CY7C604/605 Register Read	4-81
CY7C604/605 Register Write	4-81
Power-On Reset	4-82
Software External Reset	4-83
Software Internal Reset	4-83

Write-Through (Copy-Back) Read Cache Hit Timing Diagram

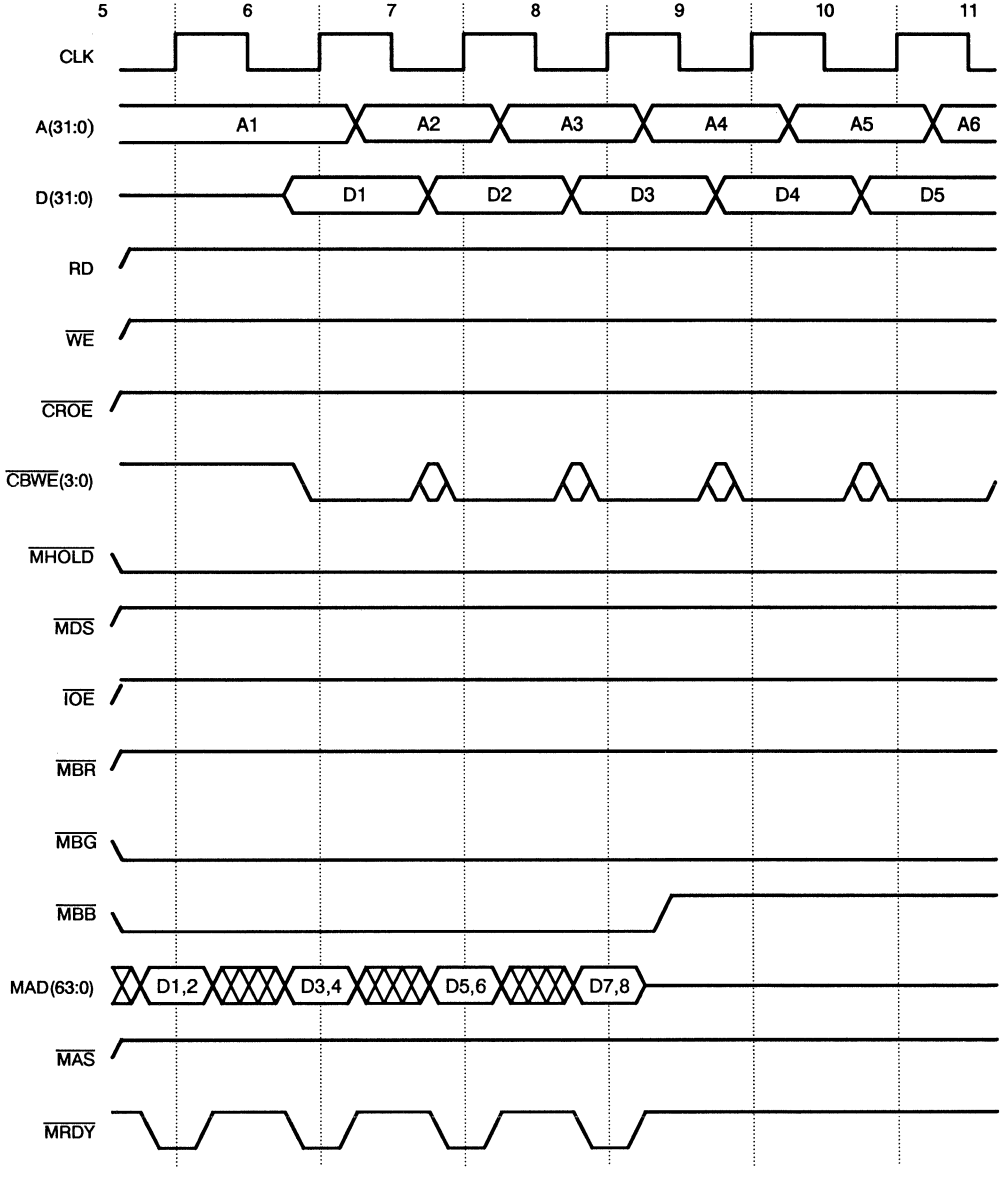


Write-Through (Copy-Back, Clean Cache Line) Read Cache Miss Timing Diagram (page 1 of 3)*

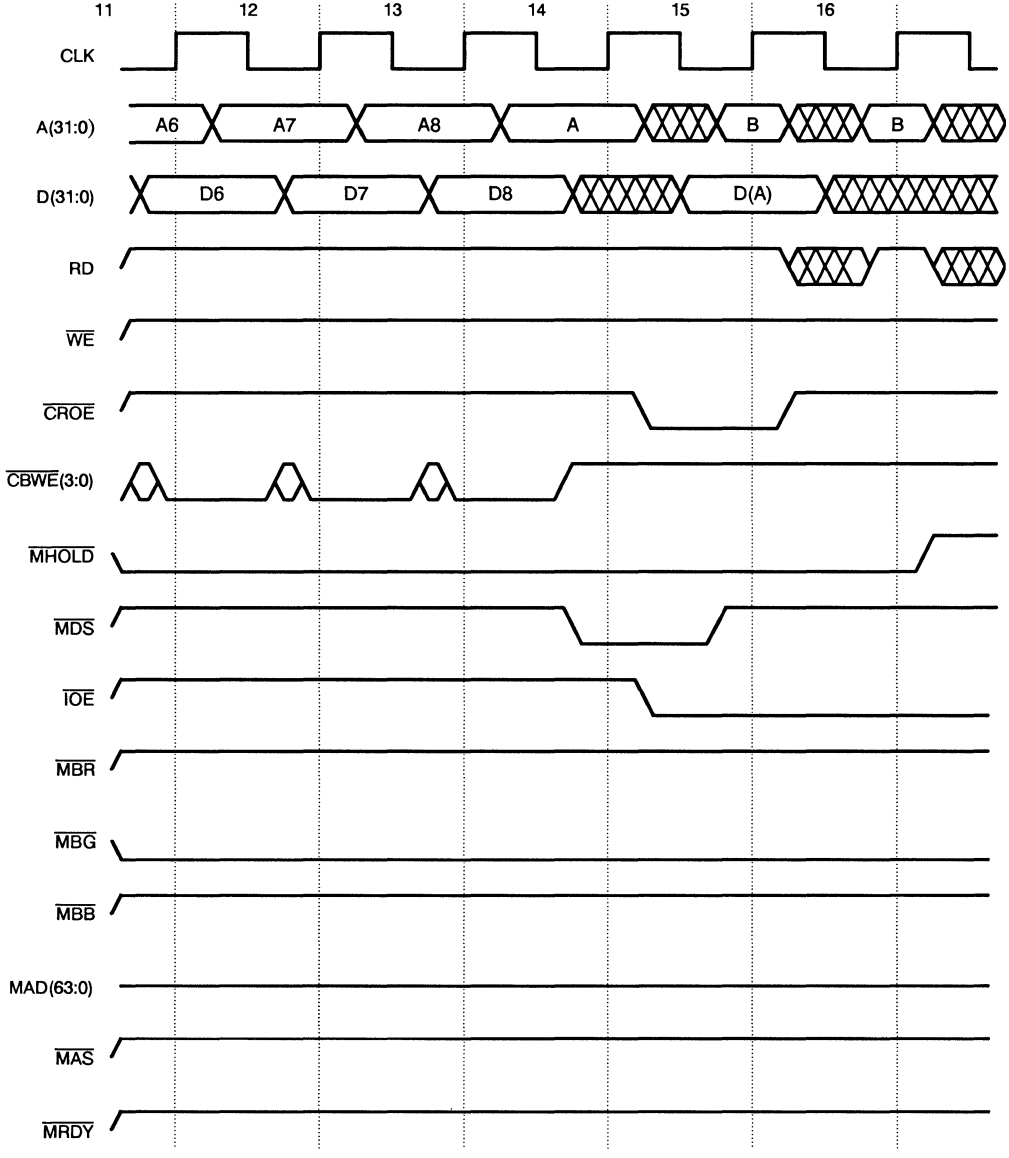


*Two clocks can be deleted from the cache miss timing if $\overline{MBG}$ is already granted.

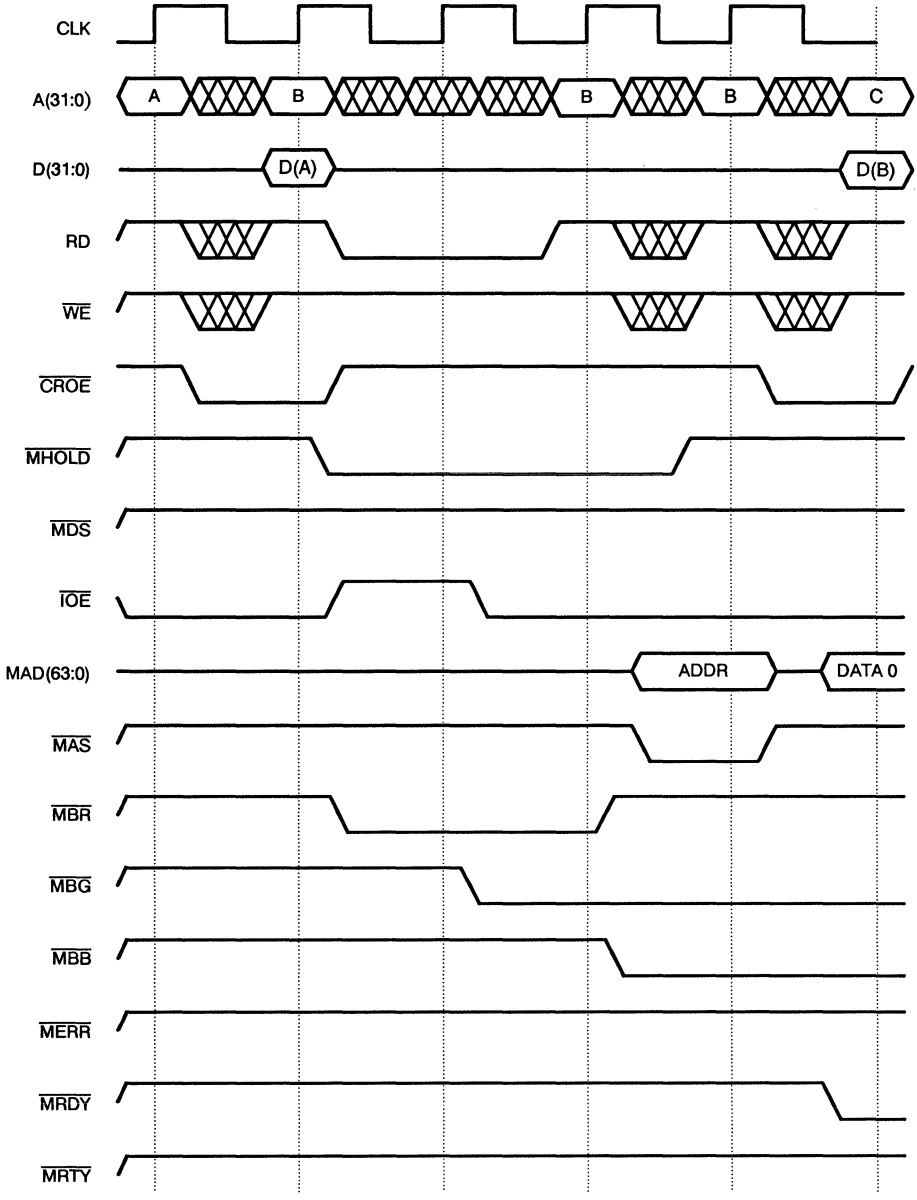
Write-Through (Copy-Back) Read Cache Miss Timing Diagram (page 2 of 3)



Write-Through (Copy-Back) Read Cache Miss Timing Diagram (page 3 of 3)

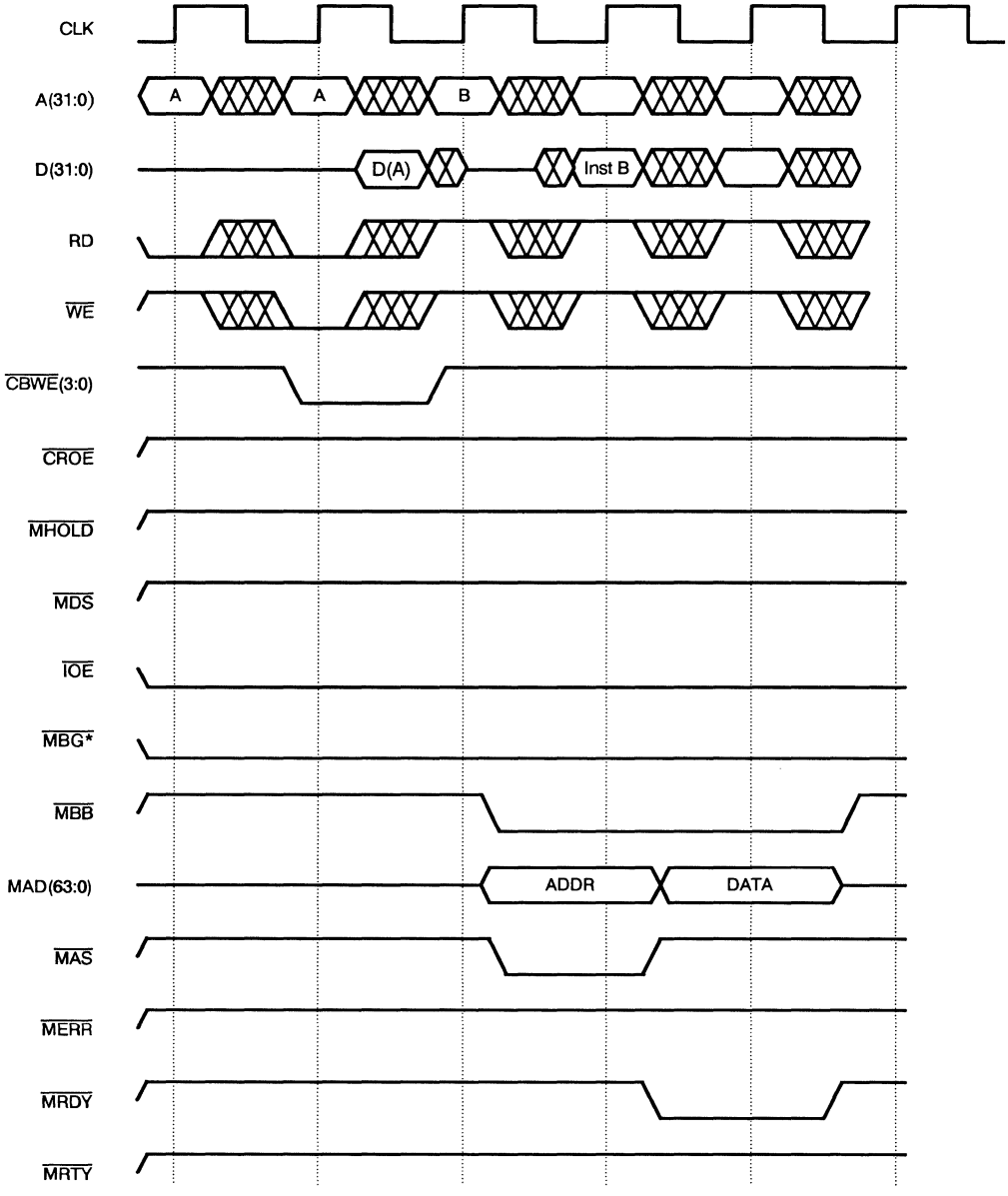


Write-Through Read Cache Miss (Alias Detected) Timing Diagram



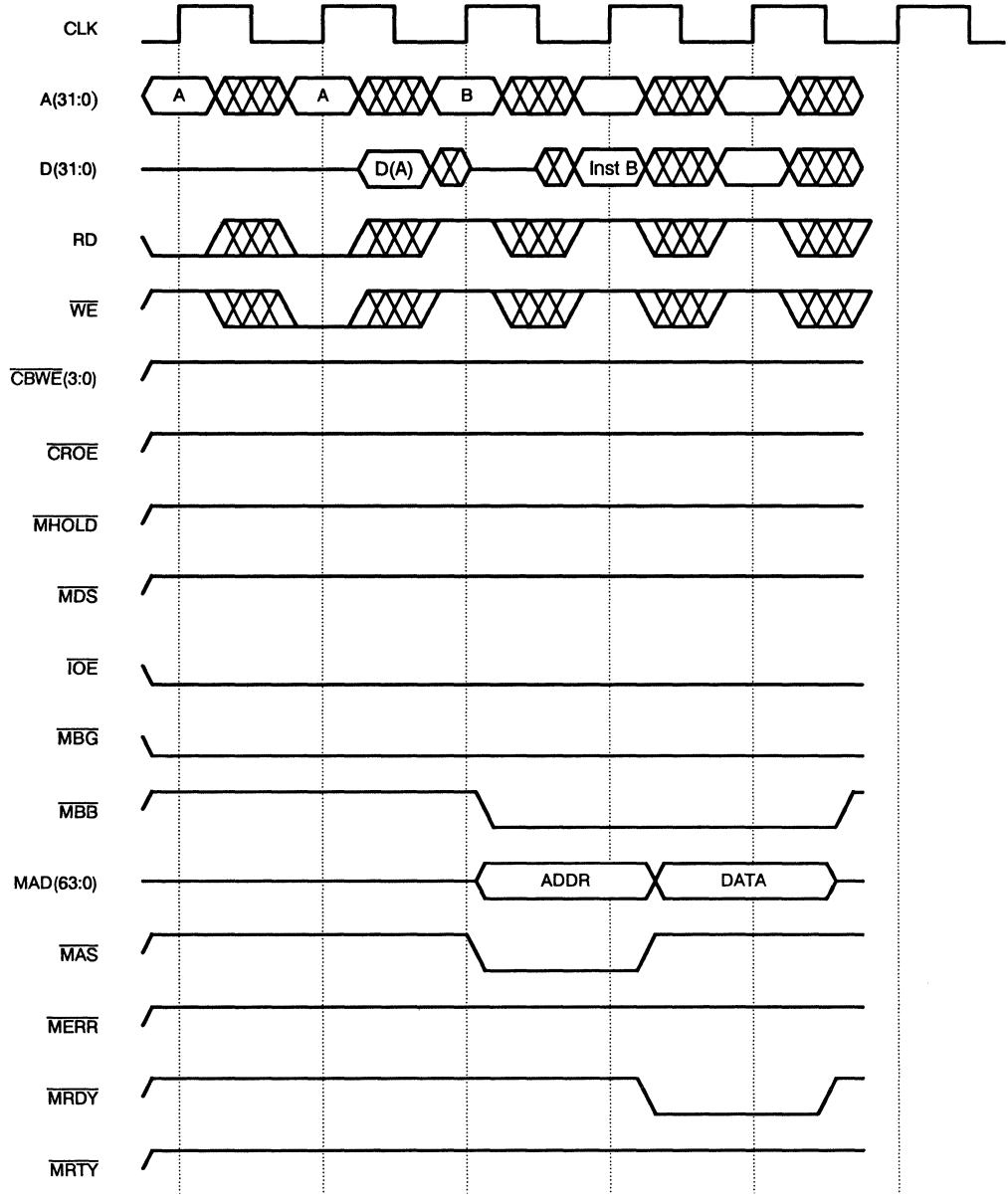
Note: Although aliasing is detected, the Mbus access is not aborted (the CY7C604/605 ignores the access). The Mbus transaction terminates normally.

Write-Through Write Cache Hit Timing Diagram

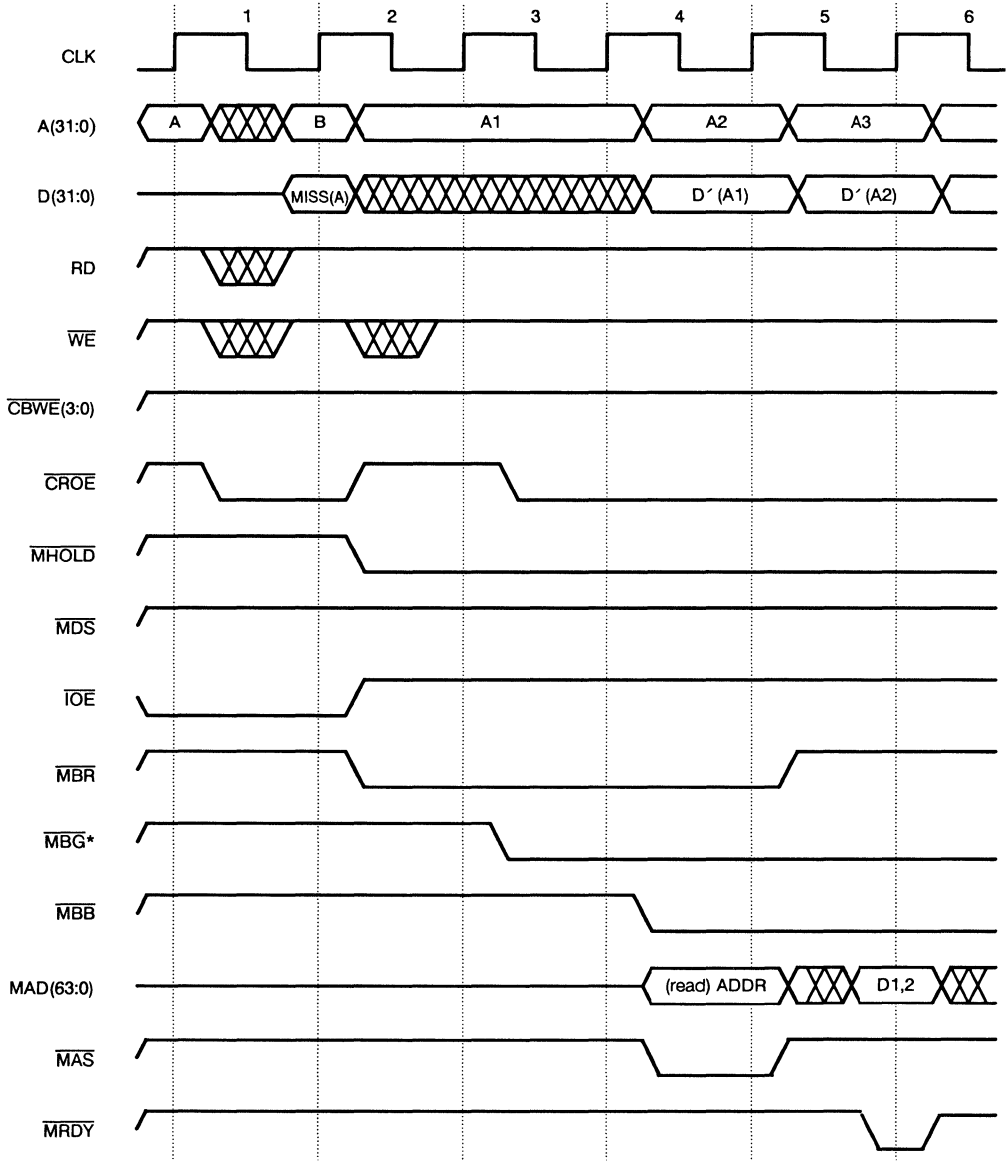


* This timing diagram is an example of bus parking (i.e., MBG granted by default to the CY7C604/605).

Write-Through Write Cache Miss Timing Diagram

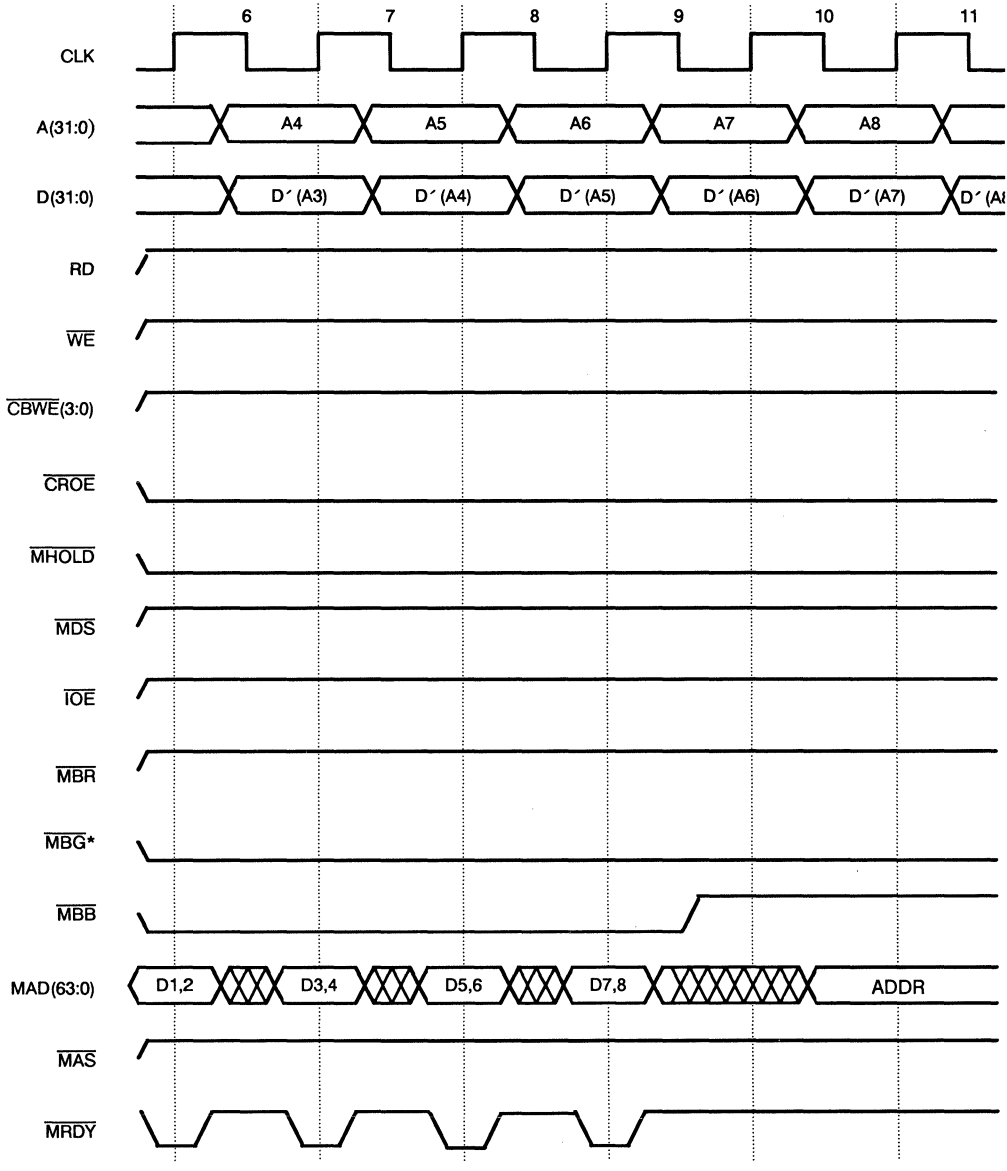


Copy-Back Cache Read Cache Miss, Modified Cache Line (page 1 of 5)*

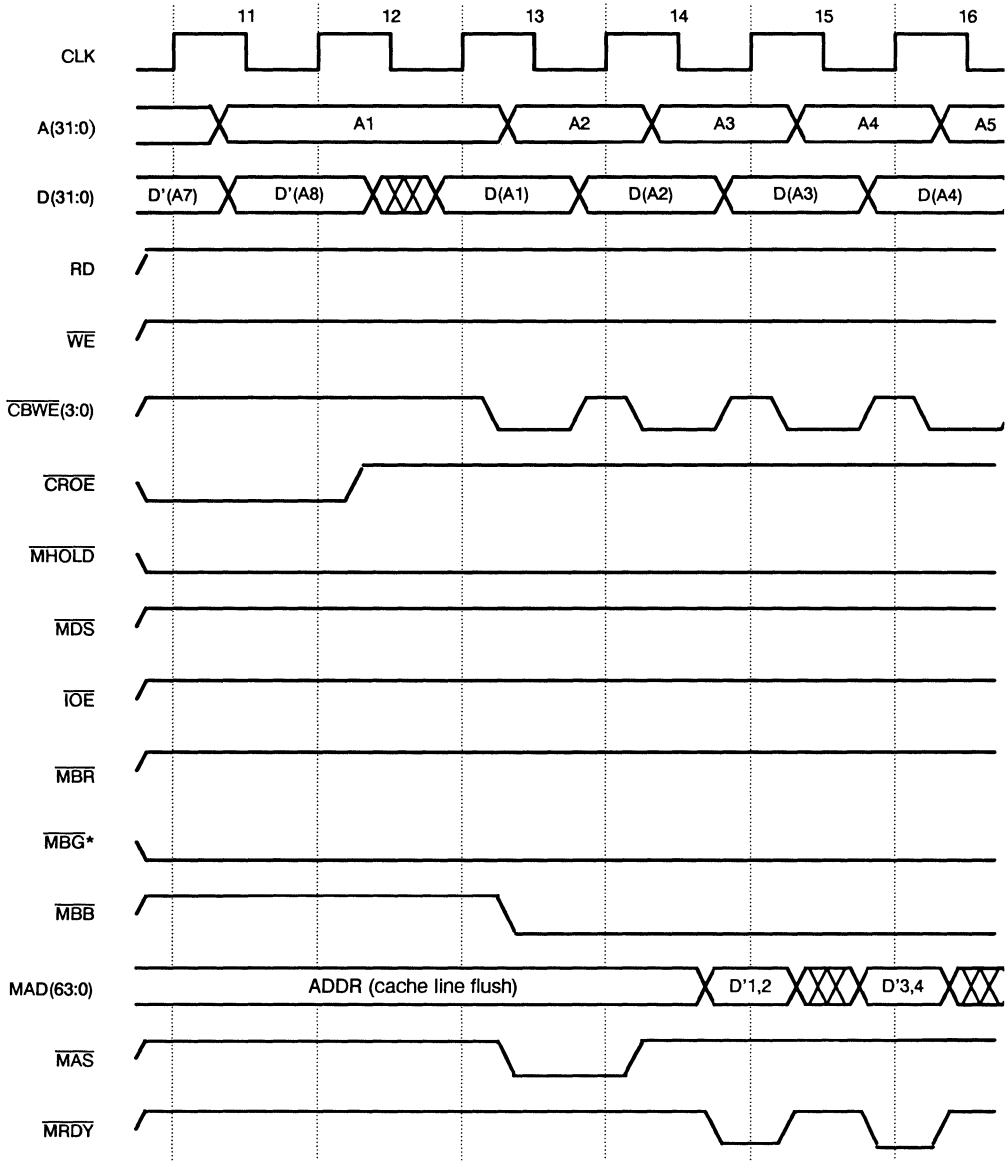


* Two clock cycles can be deleted from this timing diagram if the $\overline{MBG}$ signal is already asserted.

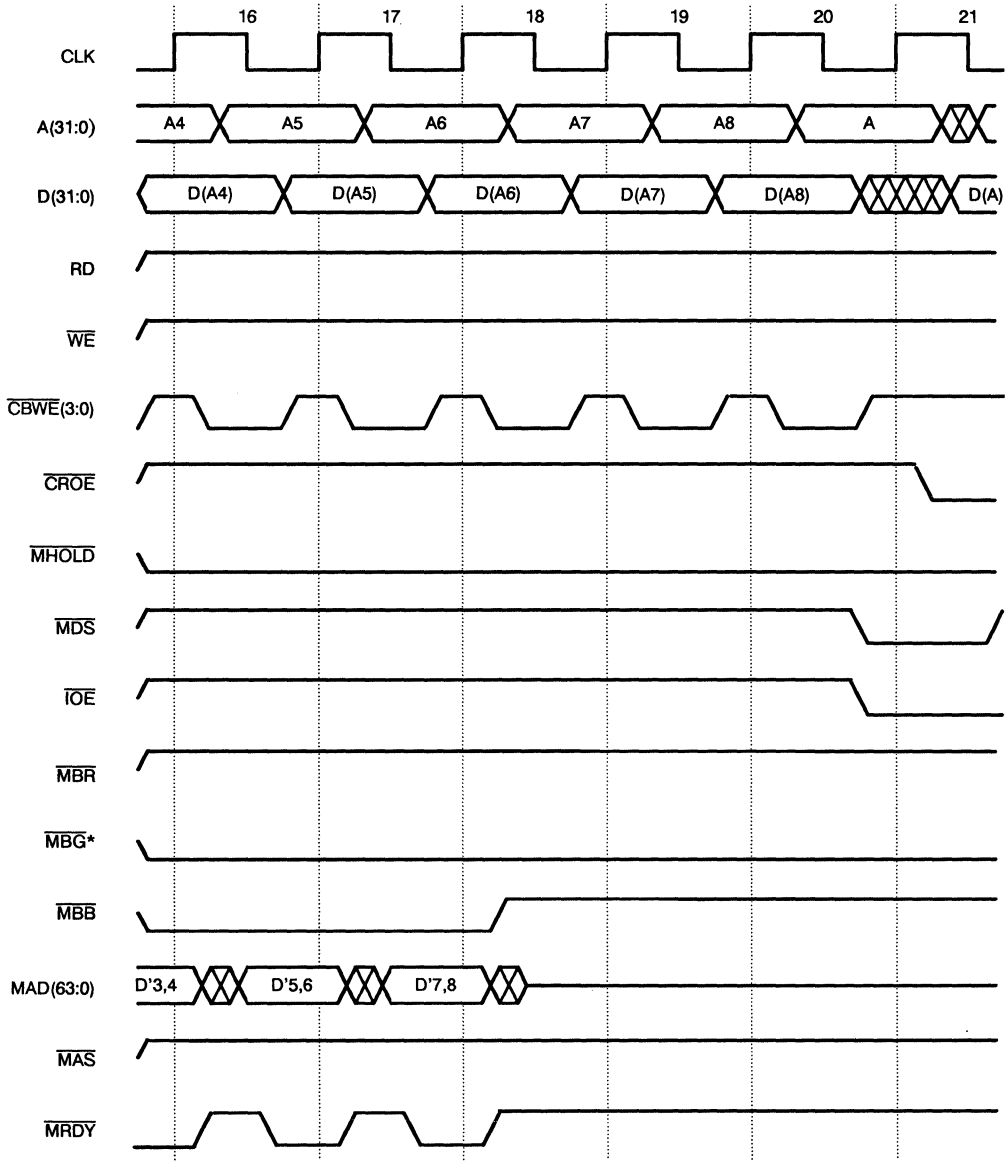
Copy-Back Cache Read Cache Miss, Modified Cache Line (page 2 of 5)



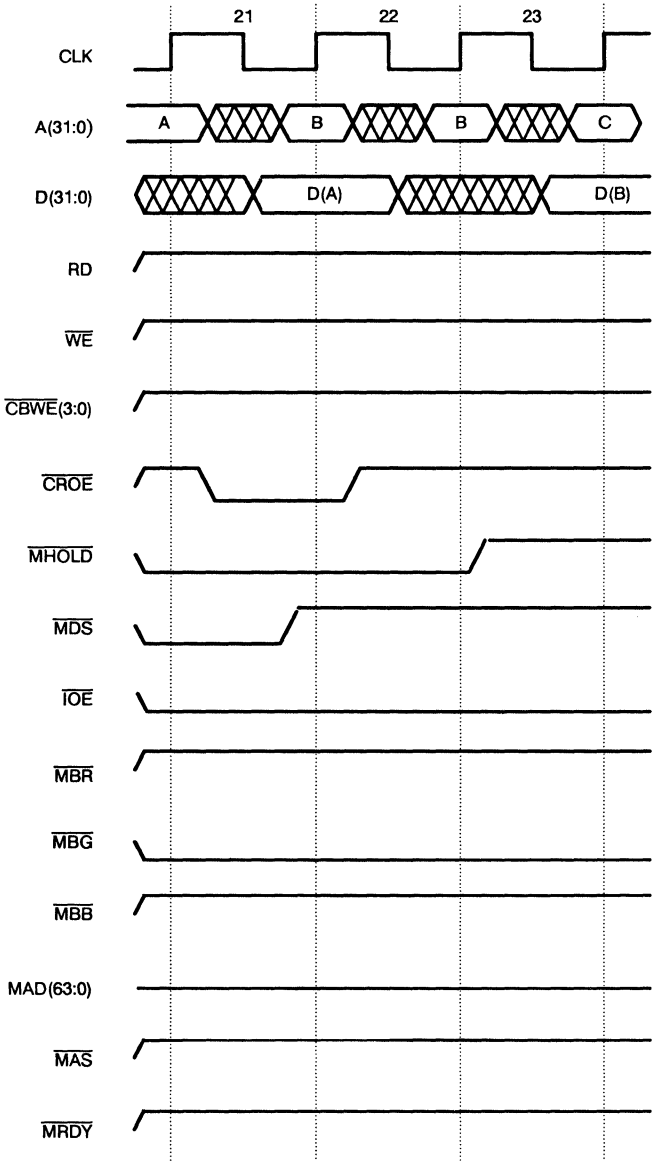
Copy-Back Cache Read Cache Miss, Modified Cache Line (page 3 of 5)



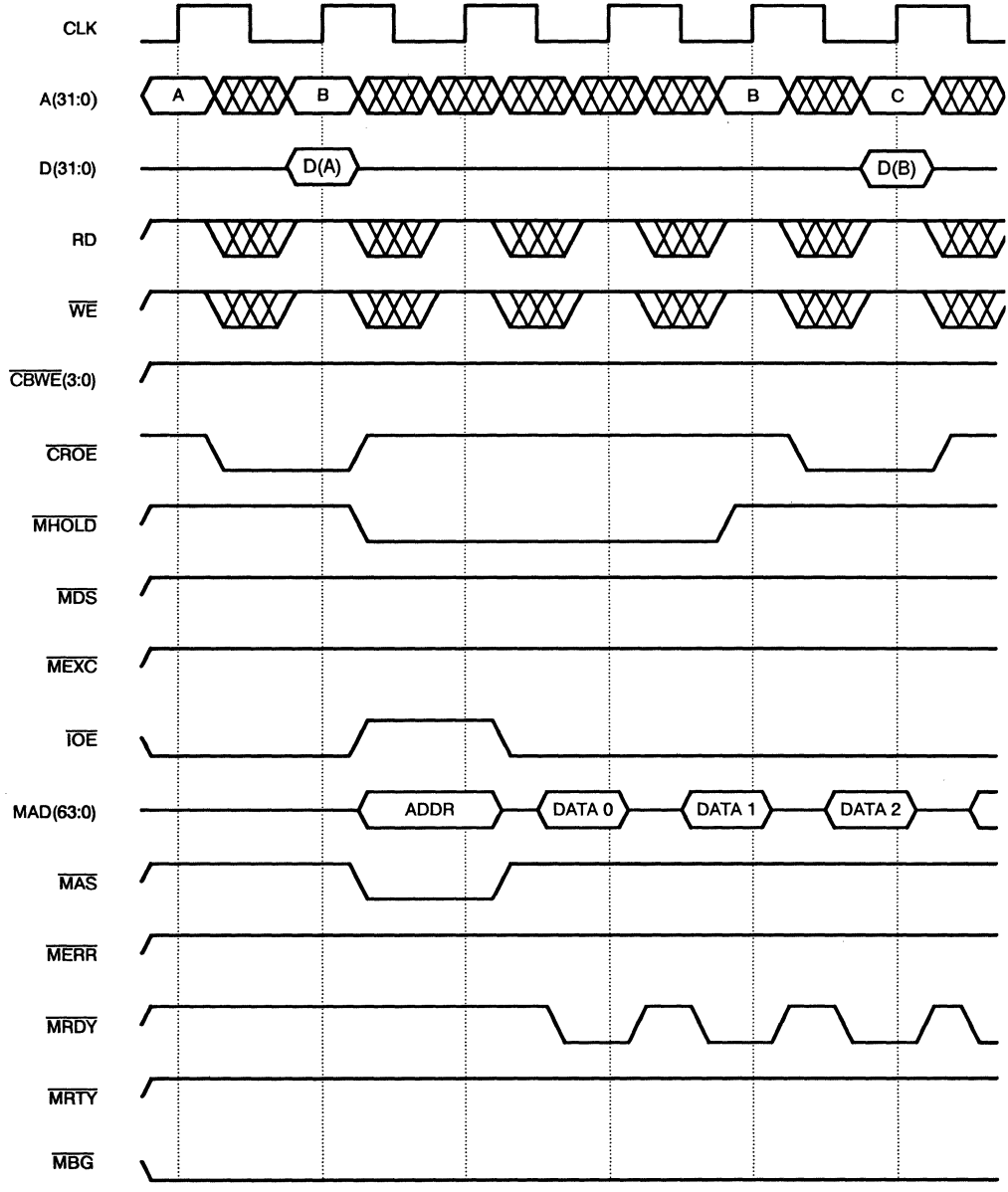
Copy-Back Cache Read Cache Miss, Modified Cache Line (page 4 of 5)



Copy-Back Cache Read Cache Miss, Modified Cache Line (page 5 of 5)

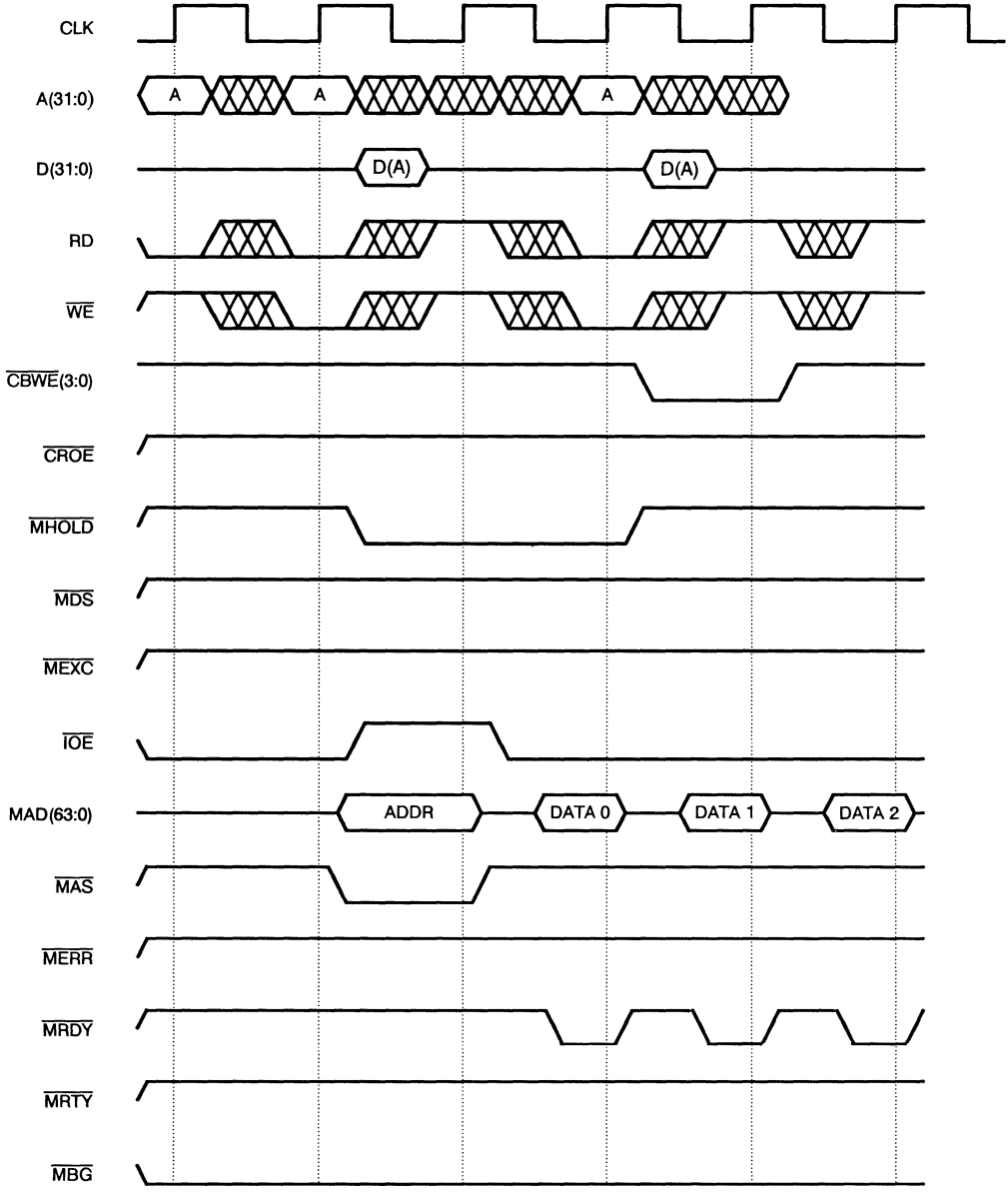


Copy-Back Read Cache Miss, Modified or Non-Modified (Alias Detected) Timing Diagram



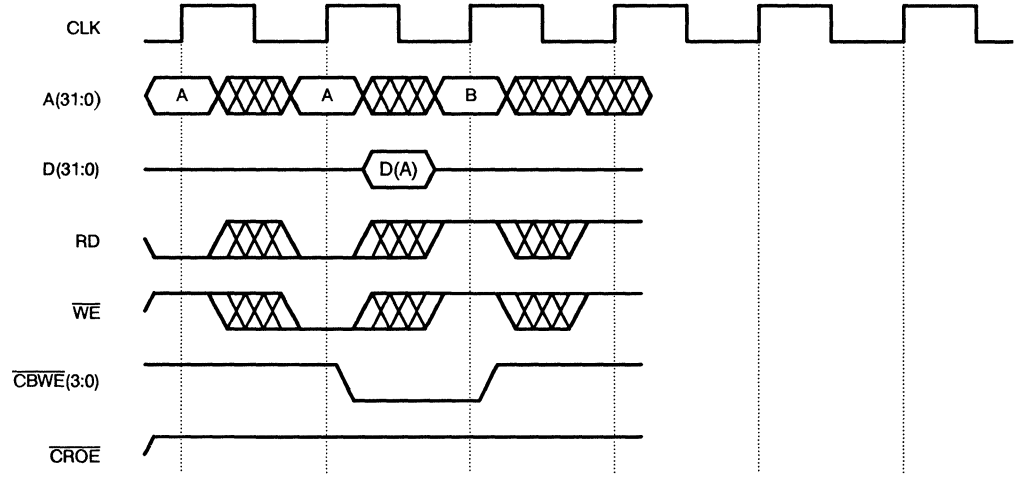
Note: Even though aliasing is detected, the Mbus is not aborted (the CY7C604/605 ignores the access). The Mbus transaction terminates normally. Timing assumes Mbus is parked (already granted).

Copy-Back Write Cache Miss, Modified or Non-Modified (Alias Detected) Timing Diagram

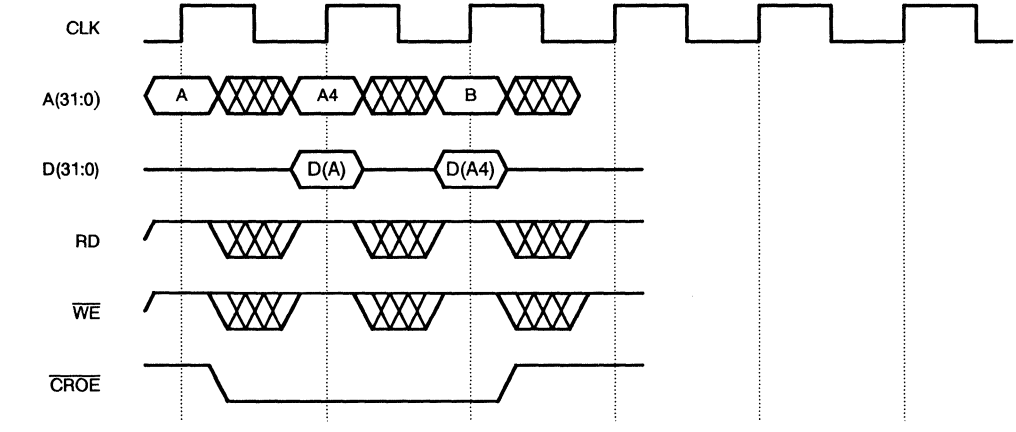


Note: Even though aliasing is detected, the Mbus is not aborted (the Mbus controller ignores the access). Timing assumes Mbus is parked (already granted).

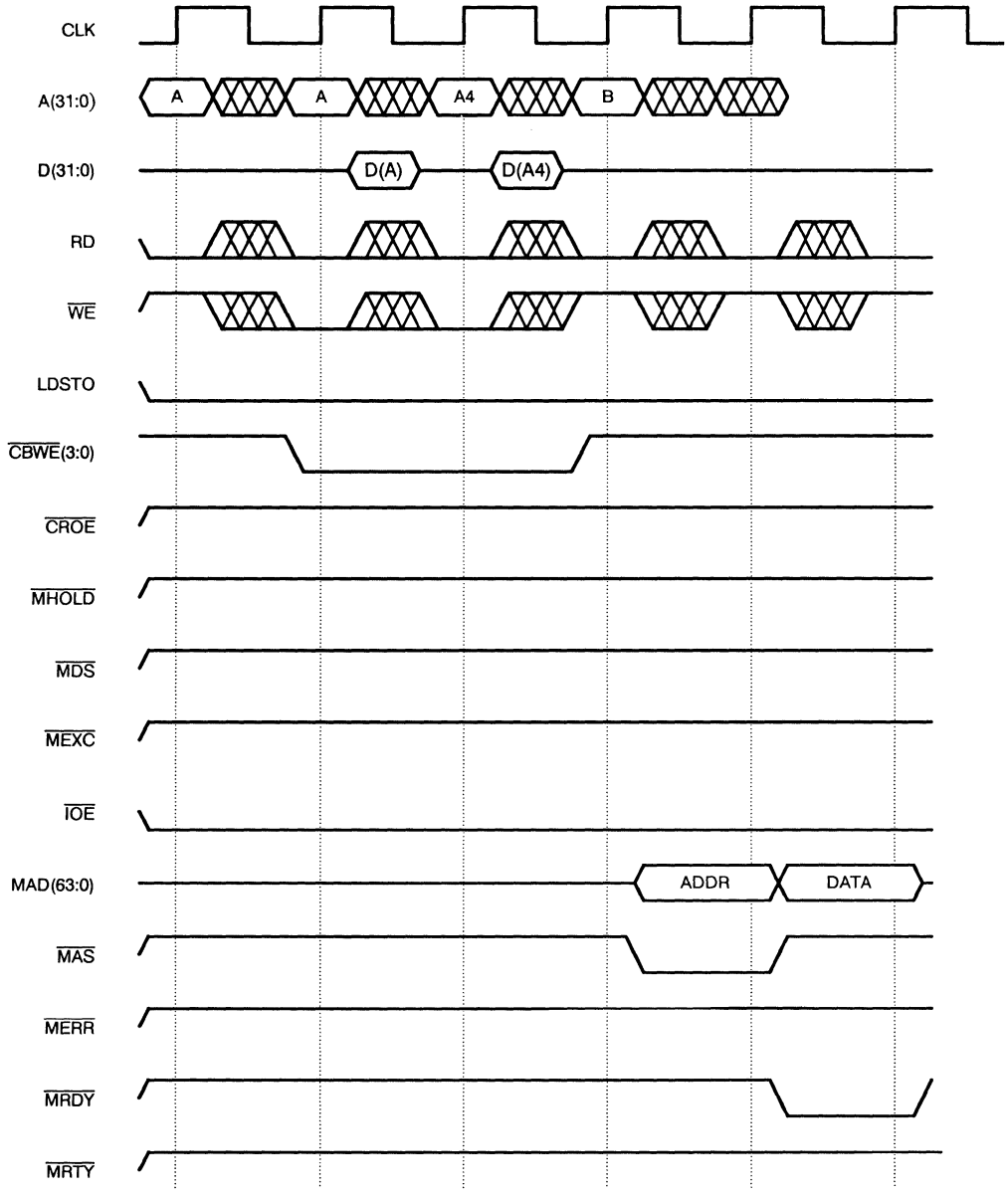
Copy-Back Write Cache Hit Timing Diagram



Write-Through Load Double Cache Hit Timing Diagram

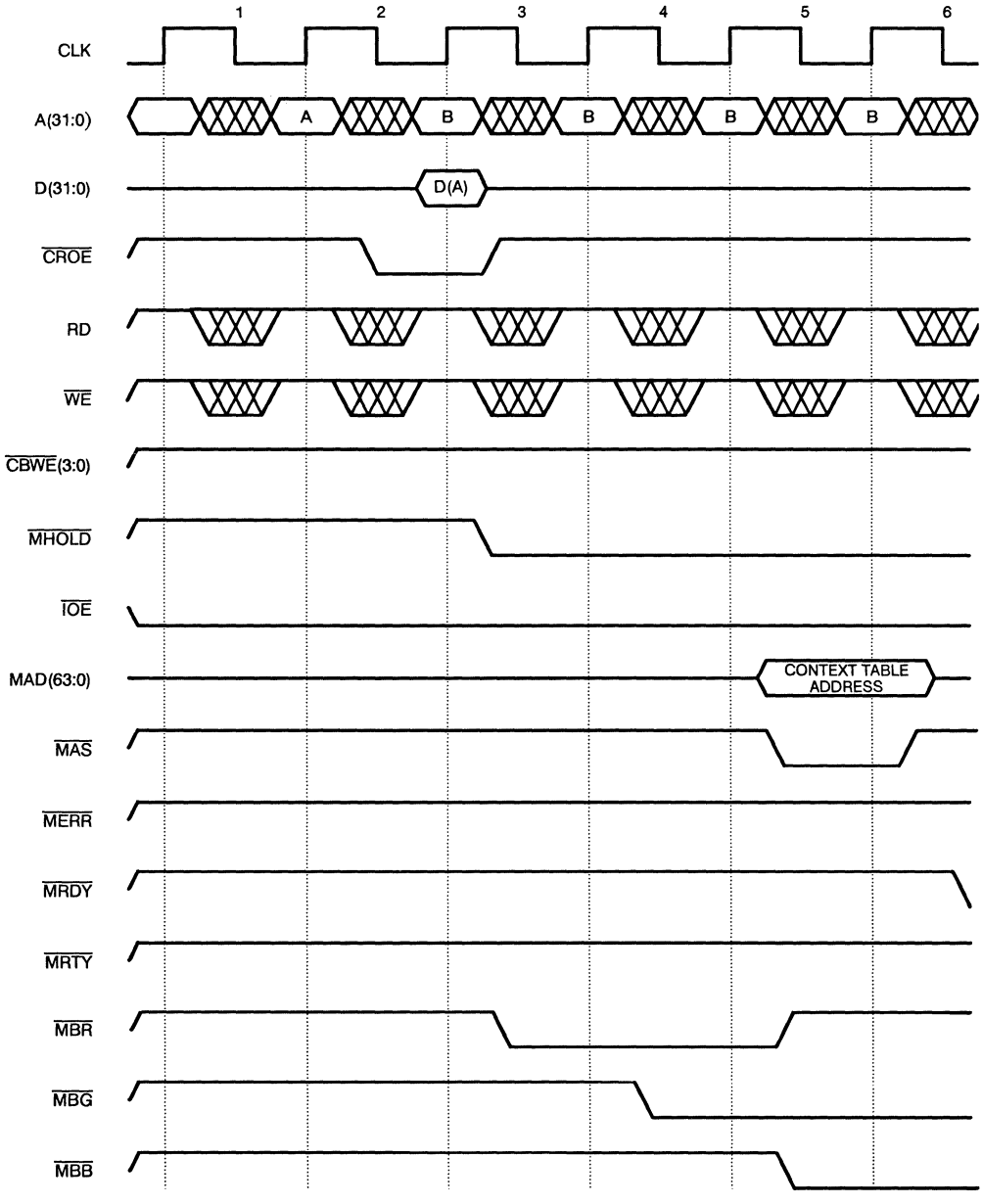


Write-Through Store Double Cache Hit Timing Diagram



Note: The Mbus cycle is not initiated until both 32-bit transfers of the double store are received.

Table Walk Timing Diagram* (with Modified Bit Update) (page 1 of 4)



* This table walk illustrates a cache read hit with TLB miss. This table walk updates the TLB and performs access protection checking.

Table Walk Timing Diagram (with Modified Bit Update) (page 2 of 4)

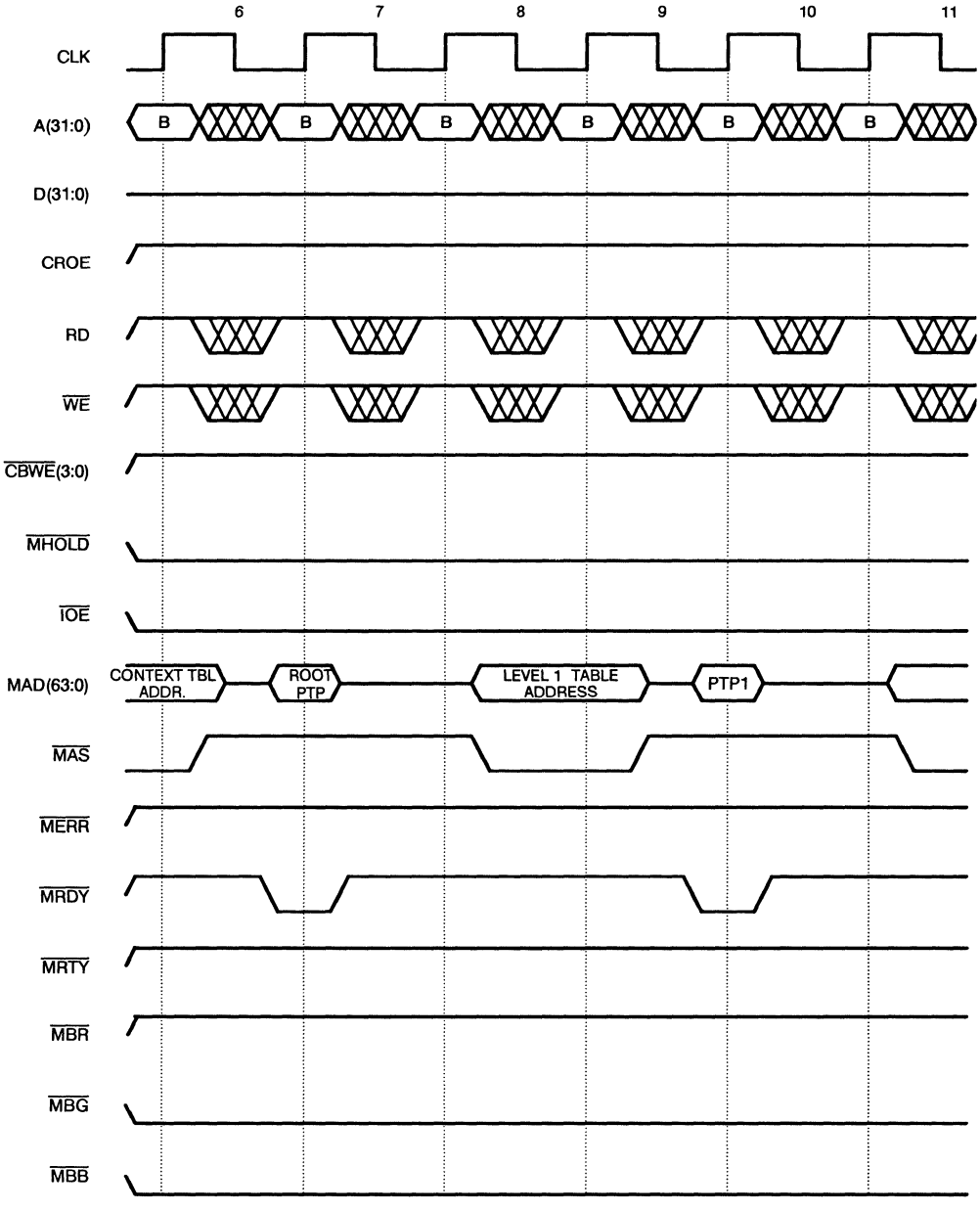


Table Walk Timing Diagram (with Modified Bit Update) (page 3 of 4)

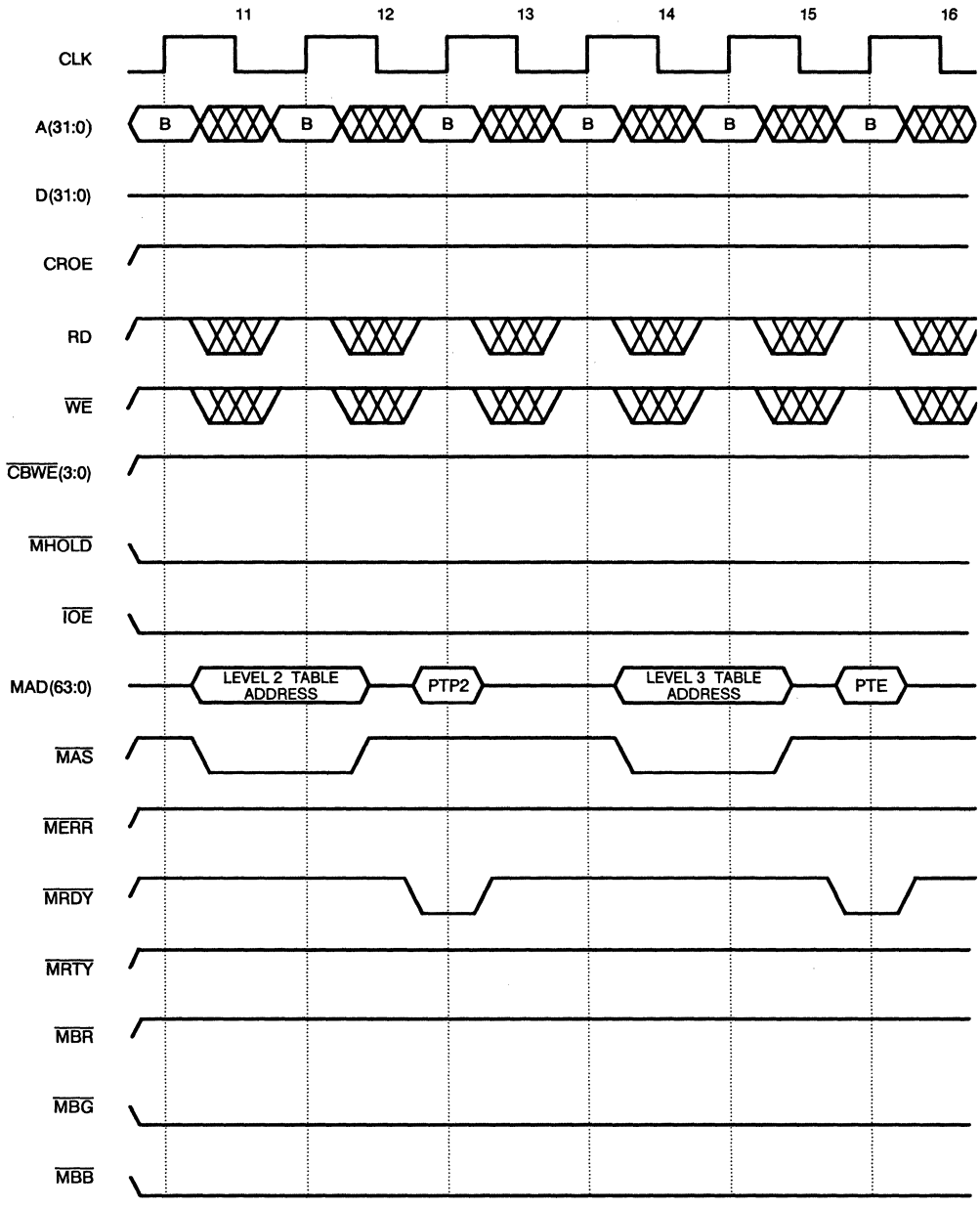
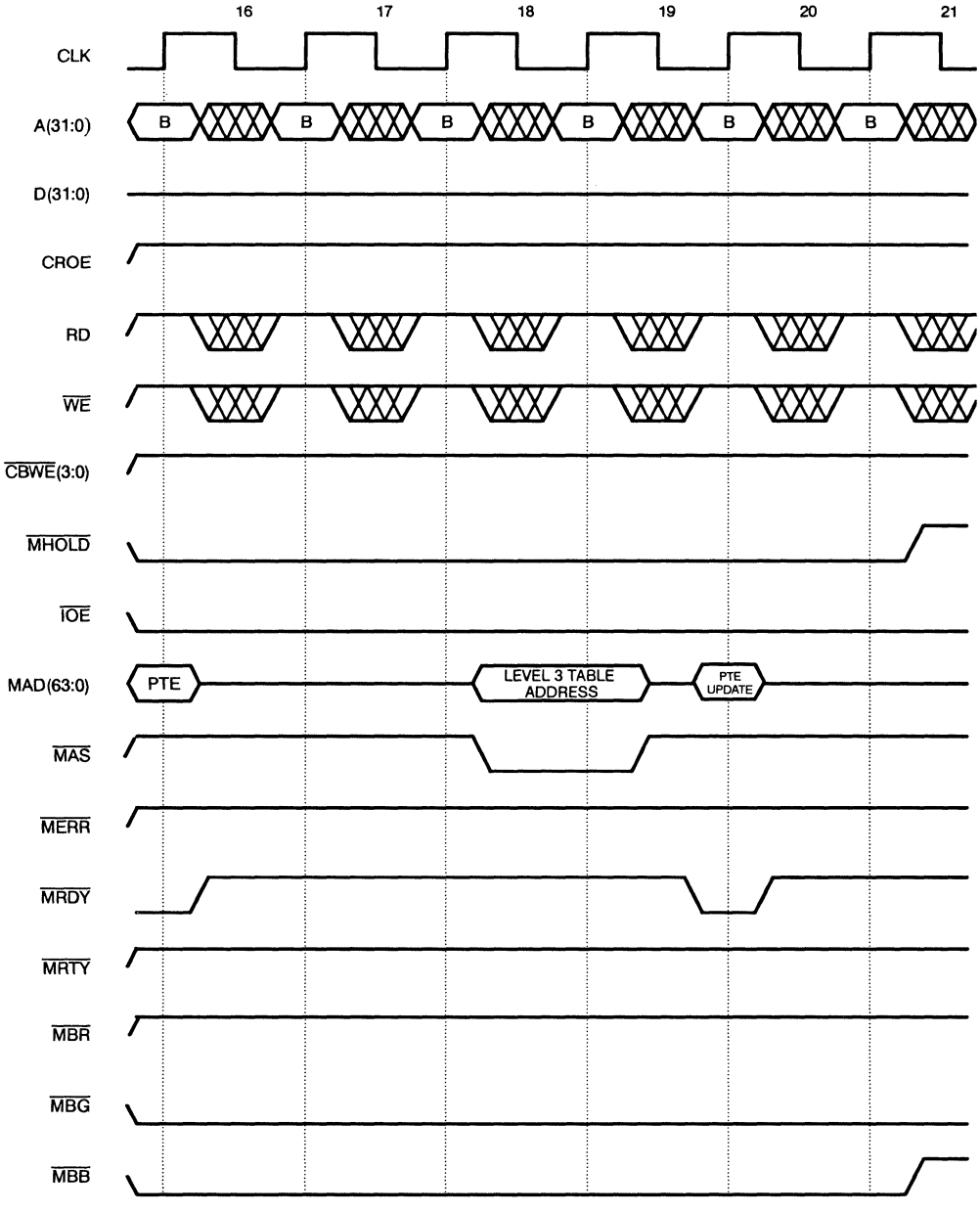
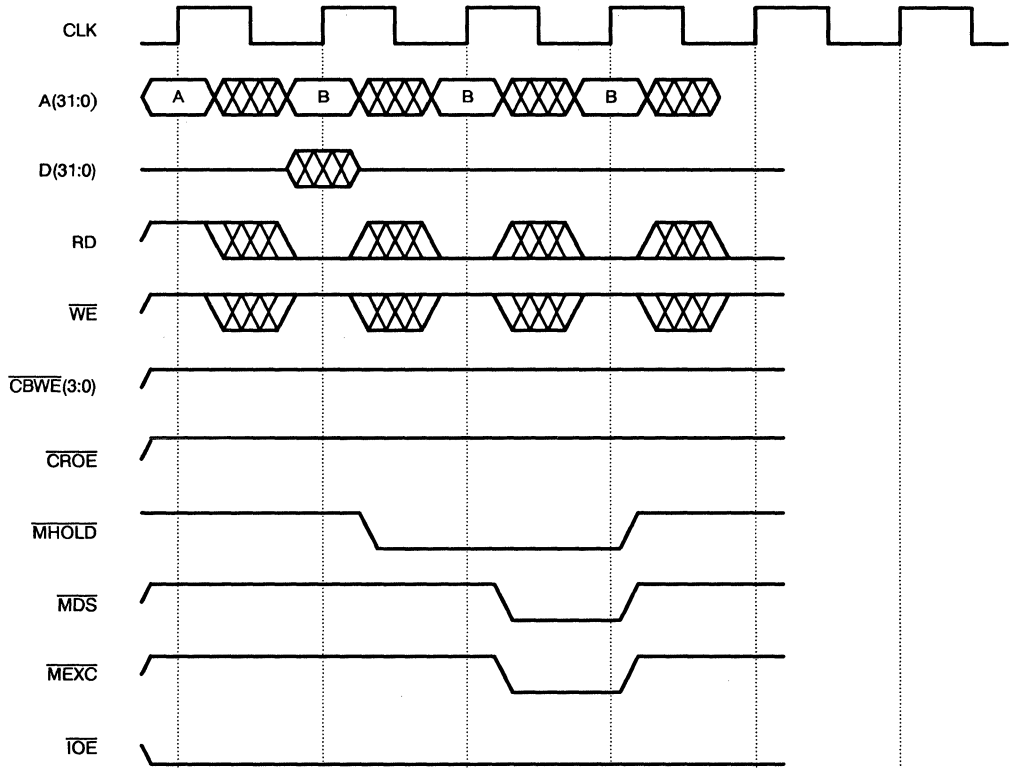


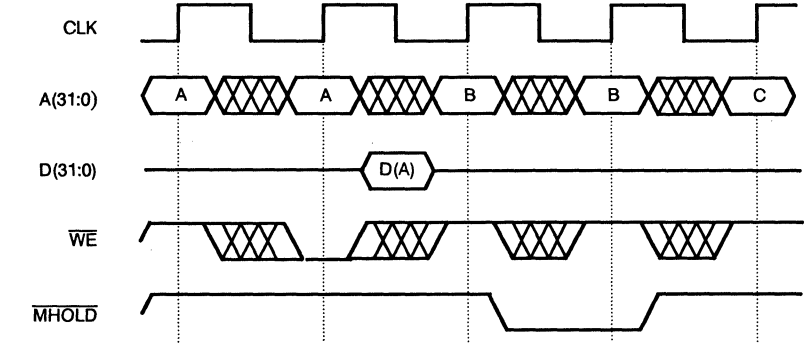
Table Walk Timing Diagram (with Modified Bit Update) (page 4 of 4)



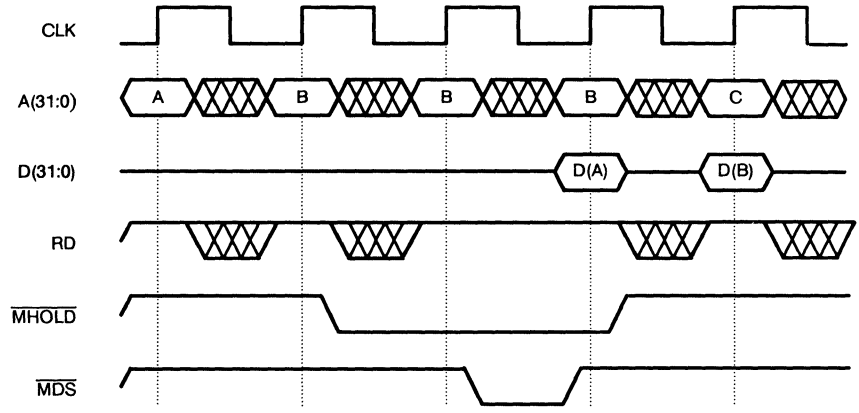
Read Access with Protection or Privilege Violation Timing Diagram



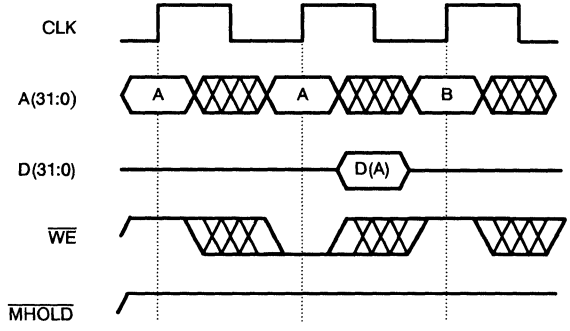
CY7C604/605 Diagnostic Cache Tag Write Access Timing Diagram



CY7C604/605 Register Read Timing Diagram

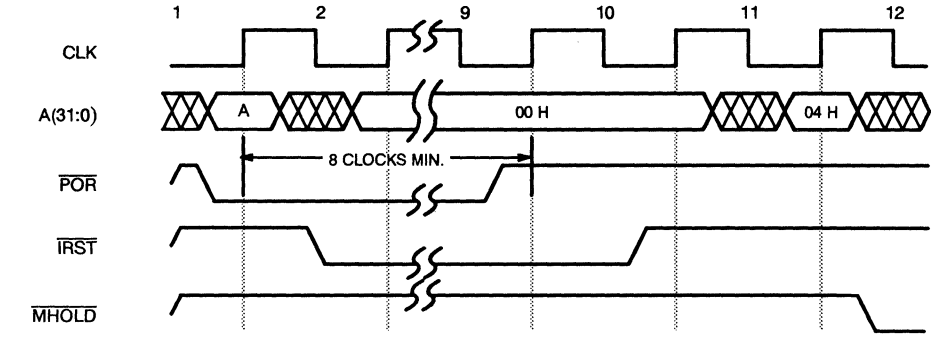


CY7C604/605 Register Write Timing Diagram

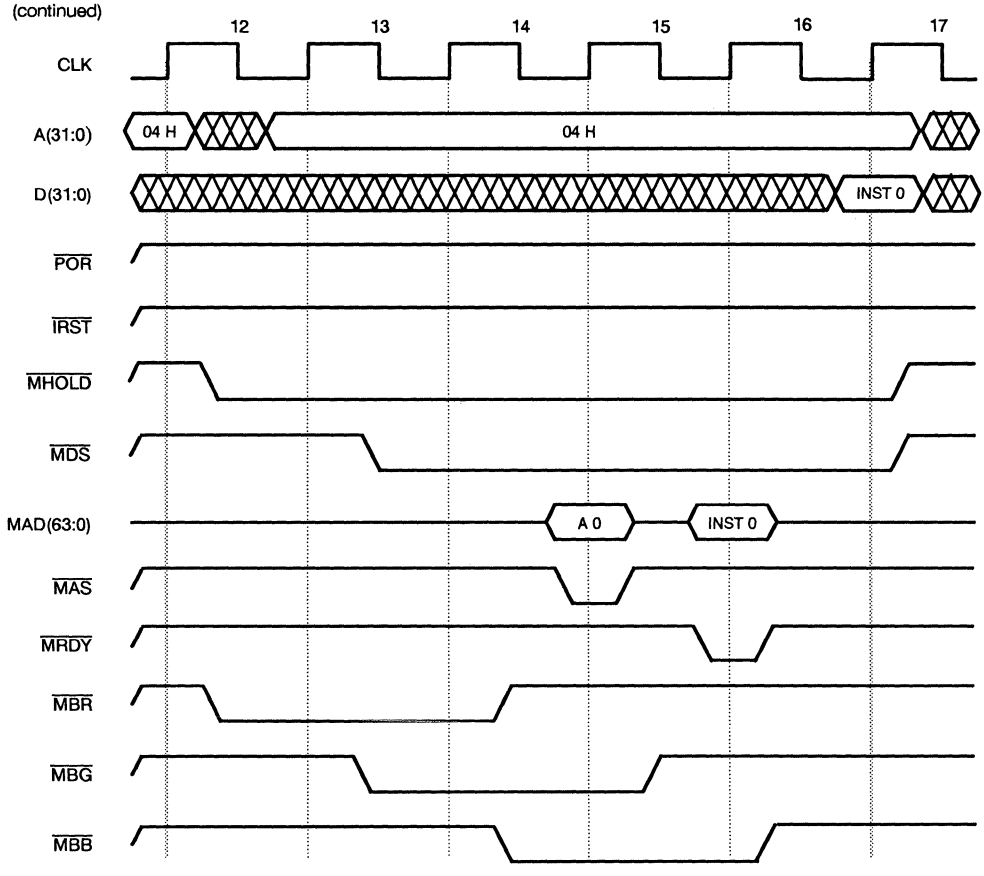


4

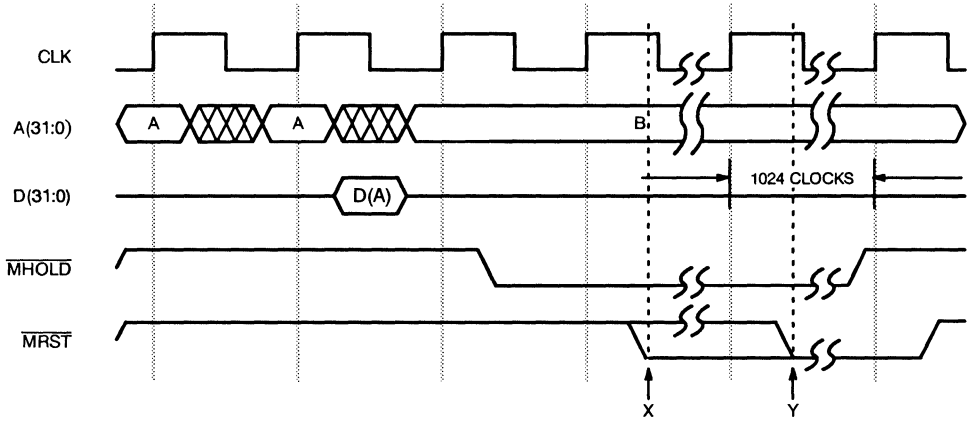
Power-On Reset Timing Diagram



(continued)



Software External Reset Timing Diagram

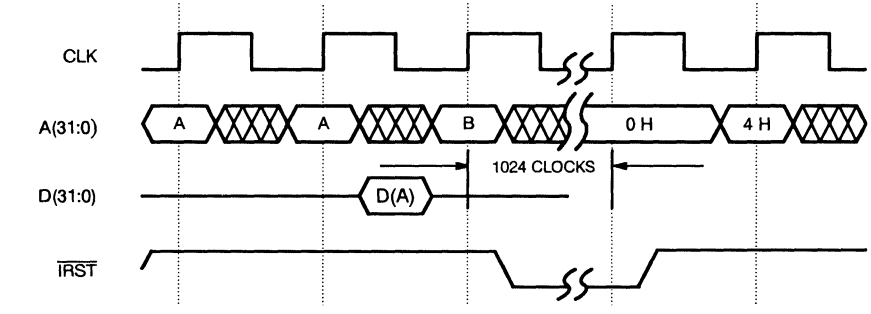


Notes:

1. Address A will be 00000700 H and ASI will be 04 H.
2. Data A will be 00000001 H.
3. $\overline{MRST}$ will not be asserted until the write buffers are empty. If empty, $\overline{MRST}$ will be asserted at point X. If not empty, $\overline{MRST}$ will be asserted at point Y (the rising clock following the final data phase of emptying the write buffer.) In either case, $\overline{MRST}$ will be asserted for 1024 clock cycles.

4

Software Internal Reset Timing Diagram



Notes:

1. Address A will be 00000700 H and ASI will be 4 H.
2. Data A will be 00000002 H.
3. $\overline{IRST}$ causes CY7C601/611 to place address 0 on address bus while asserted. CY7C601/611 continues with reset address sequence after $\overline{IRST}$ is deasserted.

4.12 Physical Bus (Mbus) Operation

The SPARC Mbus is a high-speed interface designed to connect SPARC processor modules to physical memory modules and I/O modules. The Mbus is a integrated circuit interface, and is not intended to operate as a general expansion bus across a system backplane. It is intended to operate as an interface between modules and interface circuitry located on a single printed circuit board. Modules consist of one or more integrated circuits that contain the Mbus interface. A CY7C600 CPU based upon the CY7C604/605 is an example of such a module.

Mbus is divided into two levels of implementation: level 1 and level 2. Level 1 (implemented on the CY7C604) includes the basic Mbus signals and transactions needed to support a uniprocessor system. Level 2 introduces additional signals and transactions needed to design a symmetric, cache-coherent, shared-memory multiprocessor system. Level 2 Mbus is supported by the CY7C605.

The *SPARC Mbus Interface Specification* (available from ROSS) provides further information on the Mbus from a system perspective. This section describes the Mbus as it pertains to signals specific to the operation of the CY7C604 and CY7C605. Additional Mbus signals not required for the operation of the CY7C604/605 are not explicitly described in this section.

4.12.1 Mbus Principles

- Fully synchronous bus
- Multiplexed 64-bit address/data bus
- 64 gigabytes of physical memory address space
- All signals are changed and sampled on the rising edge of clock
- Bus arbiter is a separate bus unit
- Peer level (multi-master) bus protocol
- Overlapped arbitration with bus "parking"
- Multiprocessor support signals and transactions (level 2)
- Write-invalidate type of cache-consistency protocol (level 2)

4.12.2 Mbus Level 1 Overview

Level 1 Mbus supports two transactions: Read and Write. These transactions simply read or write a specified SIZE of bytes from a specified physical address. These transactions are supported using a subset of the Mbus signals, namely a 64-bit multiplexed address/data bus (MAD(63:0), an address strobe signal ($\overline{\text{MAS}}$), and an encoded acknowledge on three signals ($\overline{\text{MRDY}}$, $\overline{\text{MRTY}}$, and $\overline{\text{MERR}}$). Additional level 1 signals support arbitration for modules ($\overline{\text{MBR}}$, $\overline{\text{MBG}}$, and $\overline{\text{MBB}}$), as well as the Mbus reset output ($\overline{\text{MRST}}$ on CY7C604, $\overline{\text{RSTOUT}}$ in the SPARC Mbus specification), and cache memory error ($\overline{\text{CMER}}$ on CY7C604, $\overline{\text{AERR}}$ in the SPARC Mbus specification). These signals are supported by the CY7C604 as part of its physical bus interface. Additional level 1 signals defined for Mbus but not used by the CY7C604 are interrupts (IRL(3:0)), module identification (ID(3:0)), and reset input ($\overline{\text{RSTIN}}$)(which corresponds to $\overline{\text{POR}}$ on the CY7C604/605). These signals are to be used by the processor, and are not specific to the CY7C604. The Mbus reference clock (CLK) completes the signal requirements for a level 1 system.

Mbus assumes that there are central functional elements to perform reset, arbitration, interrupt distribution, timeout, and Mbus clock generation. Refer to the *SPARC Mbus Interface Specification* for a detailed description of Mbus as defined for system implementation.

4.12.3 Mbus Level 2 Overview

The level 2 Mbus includes all level 1 transactions and signals and adds four transactions and two signals to support cache coherency. This is to facilitate the design of symmetric, shared memory, multiprocessor systems. In level 1, details of the cache operations inside modules are not visible to the Mbus transactions. This changes with level 2, where many aspects of the cache operation are assumed as part of the new Mbus transactions. To participate in cache-consistent sharing using level 2 transactions, a cache must have a copy-back with write-allocate policy and have a block size of 32 bytes. Cache lines

are assumed to be described as being in one of five states: *invalid, exclusive clean, exclusive modified, shared clean, and shared modified.*

The additional transactions present in level 2 systems are coherent read, coherent invalidate, coherent read and invalidate, and coherent write and invalidate. The two additional signals are Mbus Shared ($\overline{MSH}$) and Mbus Inhibit ($\overline{MIH}$). All coherent transactions have SIZE = 32 bytes. The cache coherency protocol is a "write invalidate" protocol, where the writing cache broadcasts a coherent invalidate if the cache line is not exclusive. This indicates to all caches that they should invalidate the cache line since it contains "stale data" after the write completes. All caches "snoop" coherent read transactions and assert $\overline{MSH}$ if the address of the transaction is present in their cache. By observing the $\overline{MSH}$ signal, other caches can update the state of the cache lines they hold. If a cache is the "owner," it asserts the signal $\overline{MIH}$ to tell memory not to send data. The cache then supplies the data to the requesting cache (referred to as direct data intervention). Coherent read and invalidate and coherent write and invalidate are simply the combination of a coherent invalidate and either a coherent read or a write. Their purpose is to reduce the quantity of Mbus transactions needed and thus conserve bandwidth. For more information, see Section 4.3.3.2.

Table 4–22. Mbus Signal Summary

Symbol	Description	Output	Input	Line Type	Signal Type
MAD(63:0)	Mbus Address/Data	Master/Slave	Master/Slave	bused	TS
$\overline{MAS}$	Mbus Address Strobe	Master	Slave	bused	TS
$\overline{MERR}$	Mbus Error	Slave	Master	bused	TS
$\overline{MRDY}$	Mbus Ready	Slave	Master	bused	TS
$\overline{MRTY}$	Mbus Retry	Slave	Master	bused	TS
$\overline{MBR}$	Mbus Bus Request	Master	Arbiter	dedicated	BS
$\overline{MBG}$	Mbus Bus Grant	Arbiter	Master	dedicated	BS
$\overline{MBB}$	Mbus Bus Busy	Master	Arbiter/Master	bused	TS
$\overline{MSH}$*	Mbus Shared	Bus Watcher	Master	bused	OD
$\overline{MIH}$*	Memory Inhibit	Bus Watcher	Master/Memory	bused	TS

TS: Three-state BS: Bi-state OD: Open Drain
*Level 2 (CY7C605) ONLY

4.12.4 Mbus Signal Summary

Table 4–22 summarizes the signals that comprise the Mbus interface. Bus agents (master, slave, arbiter, etc.) are listed in the output or input column of *Table 4–22* to denote whether the signal is an input or output for that bus agent. The "line type" column of *Table 4–22* lists signals as bused or dedicated. Bused signals are those driven or received by multiple bus agents, whereas dedicated signals are driven by one agent and received by only one other. For more details, refer to the CY7C604/605 Pin Definitions, Section 4.10, on page 4–55.

The Mbus is a 64-bit multiplexed address/data bus with three separate bus agents: master, slave, and arbiter. The bus arbiter is essentially a "traffic cop" for the Mbus. It is external to all bus masters or slaves, and is responsible for granting bus ownership to one of the various bus masters. The algorithm by which the arbiter assigns priority to the various bus masters is left to the system designer.

A bus master requests bus ownership by asserting its dedicated $\overline{MBR}$ signal. The arbiter grants bus ownership by asserting the dedicated $\overline{MBG}$ signal for that bus master. If the $\overline{MBB}$ (Mbus Bus Busy) signal is not asserted, the bus master asserts $\overline{MBB}$ and starts the bus transaction. If the $\overline{MBB}$ signal is asserted, the bus master must wait until is has been released. The bus master does not own the bus until it has asserted $\overline{MBB}$, and $\overline{MBB}$ cannot be asserted until it has been released by the previous bus master. This protocol allows the Mbus to support overlapped bus arbitration. Note that $\overline{MBG}$ should stay asserted until $\overline{MBB}$ has been released by the current bus master.

After $\overline{MBB}$ has been released by the current bus master, $\overline{MBG}$ may be deasserted at any time in response to other bus requests. If no further requests are made, the $\overline{MBG}$ should stay asserted. This is referred to as bus parking, and it allows

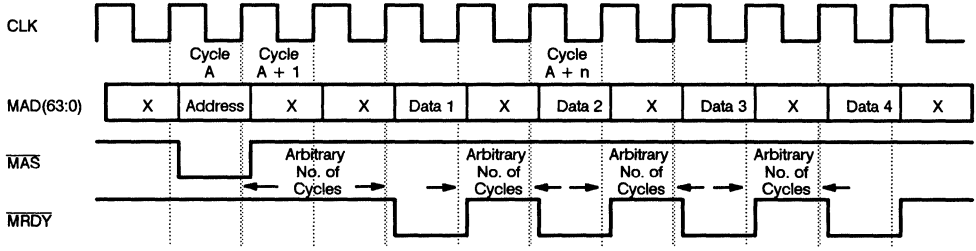


Figure 4–49. Mbus Burst Transaction Example

subsequent requests from the same bus master to be serviced without the delay of arbitrating the Mbus. If $\overline{MBG}$ for a particular bus master has already been asserted (i.e., the bus has been parked on that bus master), the bus master may assert $\overline{MBB}$ and claim the Mbus without first asserting $\overline{MBR}$.

The Mbus bus cycle consists of an address cycle followed by one or more data cycle(s). Transaction sizes supported by Mbus are: 1, 2, 4, 8, 16, 32, 64, and 128 bytes. A data transaction requiring more than one data cycle is referred to as a burst transaction.

Since the 64-bit Mbus can transfer eight bytes in a single data cycle, transactions greater than eight bytes are performed as burst transactions. Transactions less than or equal to eight bytes are performed as non-burst transactions. Non-burst transactions consist of a single address phase and a single data phase. *Figure 4–49* illustrates an example of a burst transaction. The CY7C604/605 supports 1, 2, 4, 8, and 32-byte transactions on the Mbus. The 32-byte cache line size is the only burst transaction supported by the CY7C604/605.

An Mbus cycle begins after the bus master has acquired the Mbus and asserted $\overline{MBB}$. The bus master supplies the address and strobes the Mbus Address Strobe ($\overline{MAS}$) for one clock period. The bus slave (usually the memory system) acknowledges the data transfer by strobing the $\overline{MRDY}$, $\overline{MERR}$, and $\overline{MRTY}$ signals. $\overline{MRDY}$ is strobed for each successful data cycle. Unsuccessful data cycles are acknowledged with other combinations of the $\overline{MRDY}$, $\overline{MERR}$, and $\overline{MRTY}$ signals. *Table 4–23* describes the decoding of the $\overline{MRDY}$, $\overline{MERR}$, and $\overline{MRTY}$ signals.

All Mbus transactions can be terminated by an error, which is reported by the state of the $\overline{MRDY}$, $\overline{MERR}$, and $\overline{MRTY}$ signals. These signals can be asserted during any data phase. All Mbus transactions can be suspended immediately by a *retry* or by a *relinquish and retry*, also signaled by the $\overline{MRDY}$, $\overline{MERR}$, and $\overline{MRTY}$ signals. If *retry* is signaled by the bus slave, the suspended transaction then restarts from the beginning with a new address phase. If *relinquish and retry* is signaled by the bus slave, the bus master must deassert $\overline{MBB}$ and re-arbitrate for Mbus ownership.

A special case occurs for the CY7C604/605 if a *relinquish and retry* is returned for an atomic load/store transaction. If the *relinquish and retry* occurs for the read section of the load/store transaction, the transaction is halted and $\overline{MBB}$ is deasserted. The entire transaction is repeated after re-arbitration (the normal case). If the read section has completed and the write section encounters a *relinquish and retry*, the transaction is halted and $\overline{MBB}$ is deasserted. However, in this case the transaction will retry with the write section and will not repeat the read section of the load/store transaction.

Table 4–23. Bus Status Encoding

$\overline{MERR}$	$\overline{MRDY}$	$\overline{MRTY}$	Action
H	H	H	Nothing
H	H	L	Relinquish and Retry
H	L	H	Data Strobe
H	L	L	Reserved
L	H	H	Bus Error
L	H	L	Time Out
L	L	H	Uncorrectable Error
L	L	L	Retry

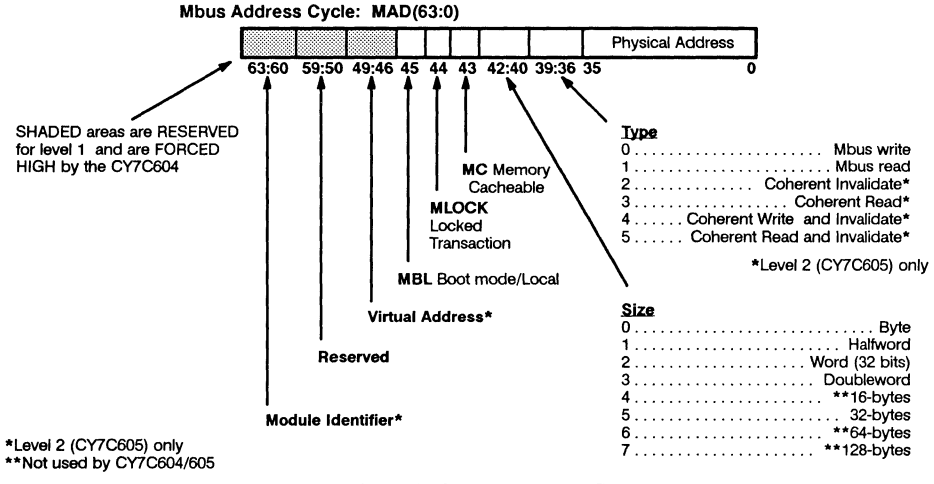


Figure 4–50. Mbus Address Cycle

The data transfer rate on the Mbus is controlled by the Mbus slave. All Mbus masters must be capable of accepting a burst transfer of the requested size at the maximum transfer rate supported by the bus. Bus slaves that cannot support the maximum transfer rate of the Mbus must insert wait states by delaying the $\overline{\text{MRDY}}$, $\overline{\text{MERR}}$, and $\overline{\text{MRTY}}$ signals until the data cycle is completed. After the Mbus transaction has finished, the bus master terminates the bus cycle by deasserting $\overline{\text{MBB}}$.

Level 2 requires two additional signals over level 1 in order to support cache coherency operations. $\overline{\text{MSH}}$ (memory shared) and $\overline{\text{MIH}}$ (memory inhibit) are asserted during Mbus coherent transactions to describe the shared and ownership status of a cache line whose address has been asserted on the Mbus. $\overline{\text{MSH}}$ is asserted by a CY7C605 in response to a bus snooping operation that discovers a Mbus transaction concerning a cache line which the CY7C605 has a copy. $\overline{\text{MIH}}$ is asserted by the CY7C605 in response to a coherent transaction on a cache line which the CY7C605 owns (i.e., has the most up-to-date copy). The $\overline{\text{MIH}}$ signal is used to inhibit the output of the memory system, and is asserted to indicate that the CY7C605 will respond to the memory request by supplying the data directly to the requesting cache.

4.12.5 Mbus Address Cycle

The address cycle of an Mbus transaction consists of a 36-bit physical address and 28 bits of control and transaction information. *Figure 4–50* illustrates the Mbus address cycle. The address fields of the Mbus address cycle are described below:

Module Identifier *MAD(63:60).* This field is defined by the module ID number field of the SCR. It is used by an Mbus agent issuing a *relinquish and retry* acknowledgement to identify the master to which to re-grant the bus.

Reserved *MAD(59:50).* This field is reserved for future expansion. The CY7C604/605 drives this field HIGH.

Virtual Address *MAD(49:46).* This field provides virtual address bits 19 through 16 for the virtually indexed cache.

Mbus Boot mode/Local *MAD(45.)* This bit indicates that the CY7C604/605 is in boot mode, or that the memory transaction has been made under local mode (ASI = 01 H).

Mbus Lock *MAD(44).* This bit indicates that the Mbus transaction is a "locked" transaction. This bit is useful to a slave with interfaces to both the Mbus and another interface external to the Mbus. It can be used by such a slave to lock the resource to the Mbus master. The locked state of the slave is released when the $\overline{\text{MBB}}$ signal for the transaction is deasserted.

Memory Cacheable *MAD(43).* This bit indicates the state of the cacheable bit for the memory address asserted.

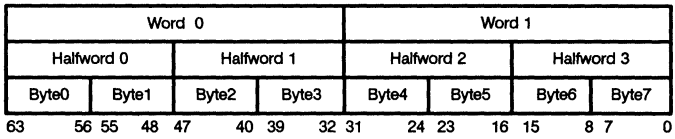


Figure 4–51. Mbus Data Ordering

Size *MAD(42:40).* This field describes the size of the Mbus transaction. Refer to *Figure 4–50* for the assignments for this field.

Type *MAD(39:36).* This field describes the transaction type. Refer to *Figure 4–50* for the assignments for this field.

Physical Address *MAD(35:0).* This field is the 36-bit physical address for the transaction.

4.12.6 Mbus Data Cycle

Mbus transactions consist of an address cycle followed by one or more data cycles. A single data cycle transaction is referred to as a non-burst transaction. Note that all non-cacheable transactions made by the CY7C604/605 are transferred as non-burst transactions. During non-burst read or write transactions, data appears in the byte locations of the Mbus as determined by the size (MAD(42:40)) and address bits MAD(2:0) (see *Figure 4–51*). The data on any unused Mbus lines is undefined.

Burst transactions are used by the CY7C604/605 for cache line transfers. Burst transactions made by the CY7C604/605 will always be on cache line boundaries (i.e., MAD<4:0> = 0 for the address cycle of a burst transaction). All burst transactions made by the CY7C604/605 are 32 bytes (one cache line) in length.

Note: The CY7C604/605 is designed to ensure one "implicit clock" after a Mbus read transaction before it will assert an address for the next Mbus transaction. This allows time for slow memory data buffers to release the Mbus.

4.12.7 Mbus Transactions

Two transactions are defined for level 1 Mbus: read and write. Level 2 defines four additional transactions: coherent read, coherent invalidate, coherent read and invalidate, and coherent write and invalidate. The following section describes these transaction types.

4.12.7.1 Read (CY7C604/605)

A read operation can be performed on any size of data transfer which is specified by the SIZE bits in the address cycle. Read transactions involving less than eight bytes will have undefined data on the unused bytes. The minimum Mbus read transaction takes two cycles (the minimum is three cycles if different masters are performing back-to-back reads). Note that the protocol requires a master to be able to receive data at the maximum rate of the Mbus for the entire transaction. *Figure 4–52* illustrates a read transaction.

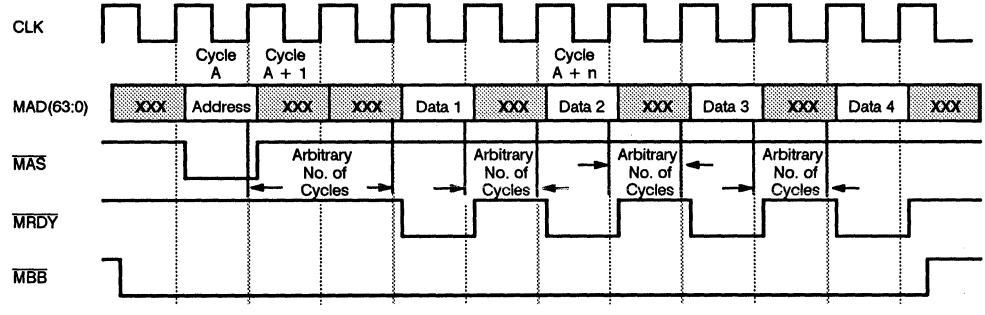


Figure 4–52. Mbus Read Transaction

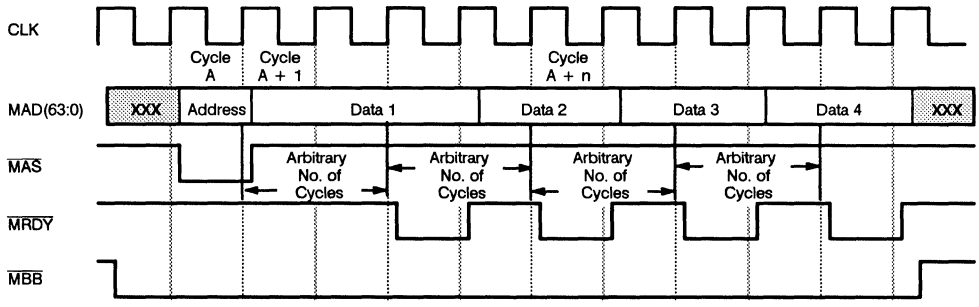


Figure 4–53. Mbus Write Transaction

4.12.7.2 Write (CY7C604/605)

A write operation can be performed on any size of data transfer specified by the SIZE bits in the Mbus address cycle. Write transactions involving less than eight bytes will have undefined data on the unused bytes. The bus master performing the write immediately drives the data in the period after the address phase of the transaction. The master releases the data immediately after receipt of each $\overline{\text{MRDY}}$ from the slave. Note that the protocol means that a master must be able to supply data at the maximum rate of the Mbus for the entire transaction. The minimum Mbus write operation takes two cycles (the minimum is three cycles if different masters are performing back-to-back writes).

4

4.12.7.3 Coherent Read (CY7C605 only)

A coherent read operation is a block read transaction that maintains cache consistency. The participants in the transaction are the requesting cache, the other caches performing bus snooping, and memory (or a second-level cache). There are three possible read scenarios for a multiprocessing system with snooping caches:

1. For a snooping cache that does not have a copy of the requested block, the cache simply ignores this transaction.
2. For a snooping cache that has a copy of the requested block but does not own it, the cache must assert $\overline{\text{MSH}}$ for one cycle during the cycle A+2. It will mark its copy as shared (if not already marked as such).
3. For a snooping cache which owns the requested block, the cache must assert both $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ signals for one cycle during the A+2 cycle. The cache supplies the requested data no sooner than cycle A+6 (four cycles after it issued $\overline{\text{MIH}}$). If the cache's own copy of the block was labeled exclusive, it will be changed to shared. Otherwise, no status change will take place for the cache's own copy.

Upon receiving the data block, the requesting master shall label the block exclusive if no one asserts $\overline{\text{MSH}}$ during the A+2 cycle or later. The requesting master shall label the block as shared if the $\overline{\text{MSH}}$ signal is asserted during the A+2 cycle or later.

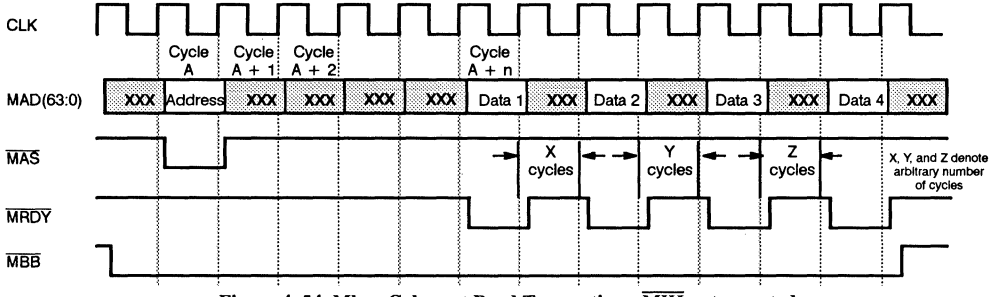


Figure 4–54. Mbus Coherent Read Transaction - $\overline{\text{MIH}}$ not asserted

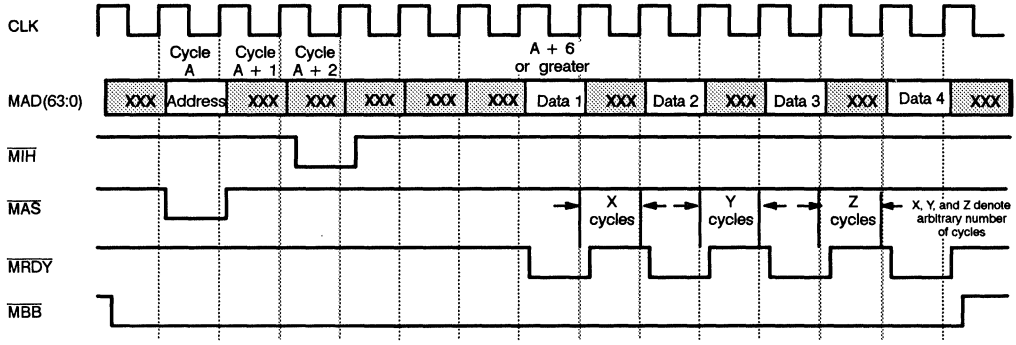


Figure 4–55. Mbus Coherent Read Transaction—$\overline{\text{MIH}}$ asserted

4.12.7.4 Coherent Invalidate (CY7C605 only)

An invalidate operation can only be performed on a cache-line basis. All invalidate operations are snooped. In an invalidate operation that hits in a cache, the cache line copy is invalidated immediately regardless of its state. Memory (or a second-level cache) is responsible for the acknowledgment of a coherent invalidate transaction on the A + 2 cycle or later. All acknowledgment types are possible. Memory will only issue normal acknowledgments to coherent invalidate transactions, but a second-level cache may issue the full range of acknowledgments. Memory (or second-level cache) designers should note that a coherent invalidate transaction has SIZE = 32 bytes during the address phase, but $\overline{\text{MRDY}}$ is only strobed once as acknowledgment. For a cache system that cannot guarantee to complete the invalidation before the A + 2 cycle, the memory controller for that system should delay the acknowledgment as required.

The coherent invalidate transaction is issued when a write is being performed on a shared cache line. Before the write can be performed, all other caches in the system must invalidate their copies (write-invalidate cache consistency protocol). Snooping caches need not assert $\overline{\text{MSH}}$ during the A + 2 cycle. The MAD(63:0) bus is undefined during the data cycles. *Figure 4–56* shows the basic coherent invalidate operation.

4.12.7.5 Coherent Read and Invalidate (CY7C605 only)

The coherent read and invalidate transaction combines a coherent read transaction with a coherent invalidate transaction. This transaction is included to reduce the number of Mbus coherent invalidate transactions. Caches performing coherent reads that intend to immediately modify the data can issue this transaction.

Each coherent read and invalidate transaction is snooped by all system caches. If the address hits in a cache but the cache does not own the block, then the cache invalidates its copy of this block. If the address hits in a cache and the cache owns the block, then it asserts $\overline{\text{MIH}}$ and supplies the data. When the data has been successfully supplied, the cache then invalidates its copy of the block. *Figure 4–57* and *Figure 4–58* show the coherent read and invalidate operation. Note that it is identical to the coherent read operation, except that the snooping caches invalidate their copy of the cache line upon a cache hit. All of the comments concerning $\overline{\text{MSH}}$ and $\overline{\text{MIH}}$ for the coherent read transaction apply to the coherent read and invalidate transaction.

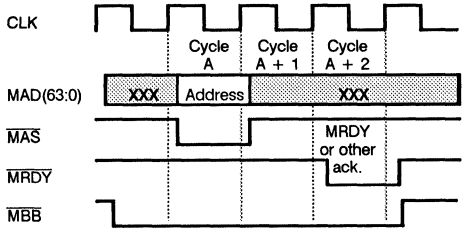


Figure 4–56. Mbus Coherent Invalidate Transaction

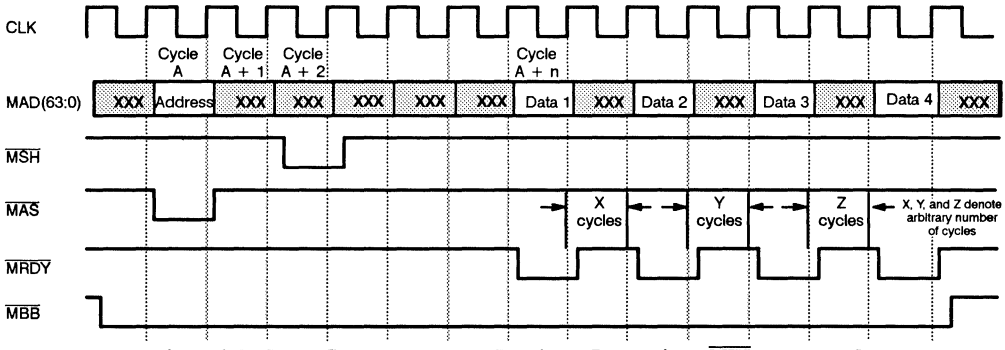


Figure 4–57. Mbus Coherent Read and Invalidate Transaction—$\overline{\text{MIH}}$ not asserted

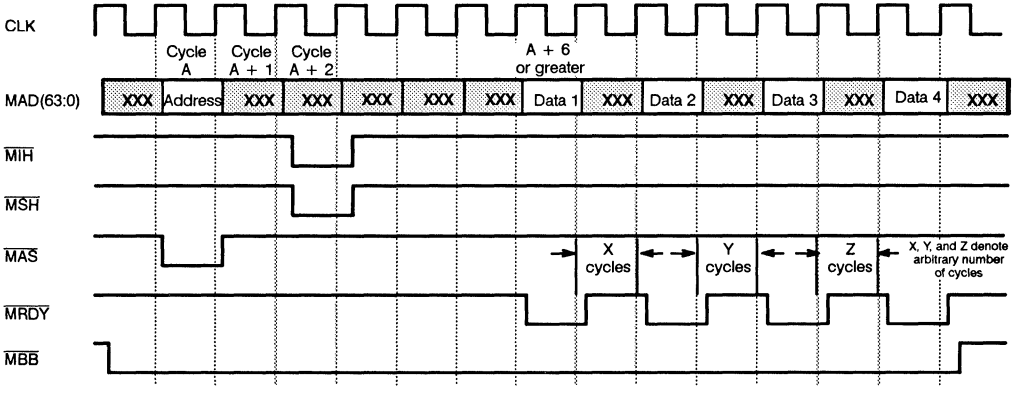


Figure 4–58. Mbus Coherent Read and Invalidate Transaction—$\overline{\text{MIH}}$ asserted

4

4.12.7.6 Coherent Write and Invalidate (CY7C605 only)

The coherent write and invalidate transaction combines a coherent write transaction with a coherent invalidate transaction. This transaction is included to reduce the number of Mbus coherent invalidate transactions.

Each coherent write and invalidate transaction is snooped by all system caches. If the address hits in a cache, then that cache invalidates its copy of the cache line. *Figure 4–59* illustrates the basic coherent write and invalidate operation. Note that this transaction is identical to the write operation, except that the snooping caches invalidate their block upon a cache hit. The SIZE for this transaction is always 32 bytes. Due to the nature of the cache coherency protocol, neither $\overline{\text{MIH}}$ or $\overline{\text{MSH}}$ need to be asserted.

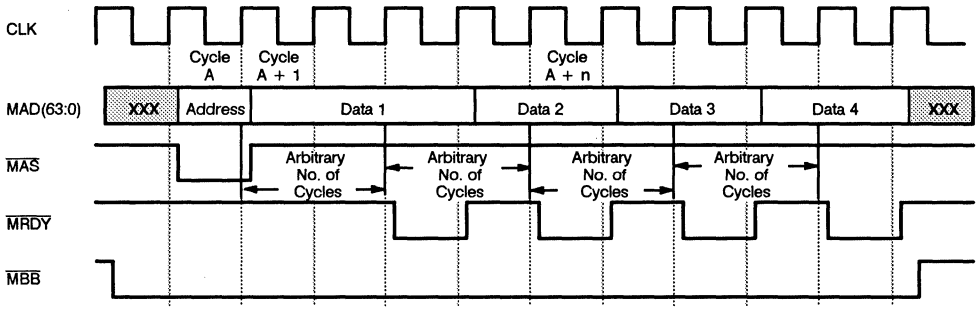


Figure 4–59. Mbus Coherent Write and Invalidate Transaction

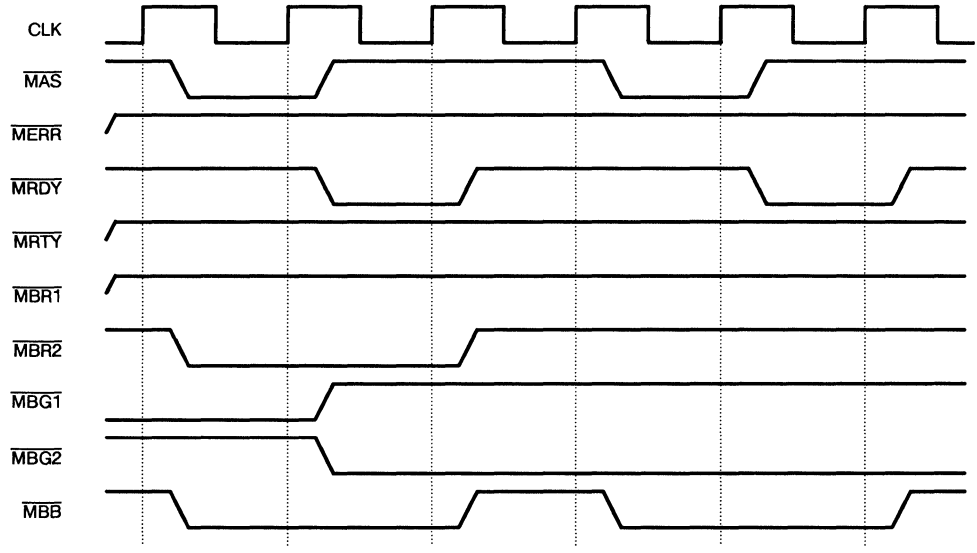
4.12.8 Mbus Transaction Timing

Page

Mbus Bus Mastership Transfer 4-93
Single-Cycle—Read Transaction 4-93
Single-Cycle—Write Transaction 4-94
Burst-Cycle—Read Transaction 4-94
Burst-Cycle—Read Transaction (Slow memory) 4-95
Burst-Cycle—Write Transaction 4-95
Burst-Cycle—Write Transaction (Slow memory) 4-96
Mbus Locked Transaction 4-96
Mbus Relinquish and Retry 4-97
Mbus Retry 4-97
Mbus Error 4-98
Mbus Coherent Read—Shared Data* 4-99
Mbus Coherent Read—Owned Data; Long Latency Memory* 4-101
Mbus Coherent Read—Owned Data; Fast Memory* 4-103
Mbus Coherent Write and Invalidate* 4-105
Mbus Coherent Invalidate* 4-107
Mbus Coherent Read and Invalidate; Shared Data* 4-108
Mbus Coherent Read and Invalidate; Owned Data* 4-110

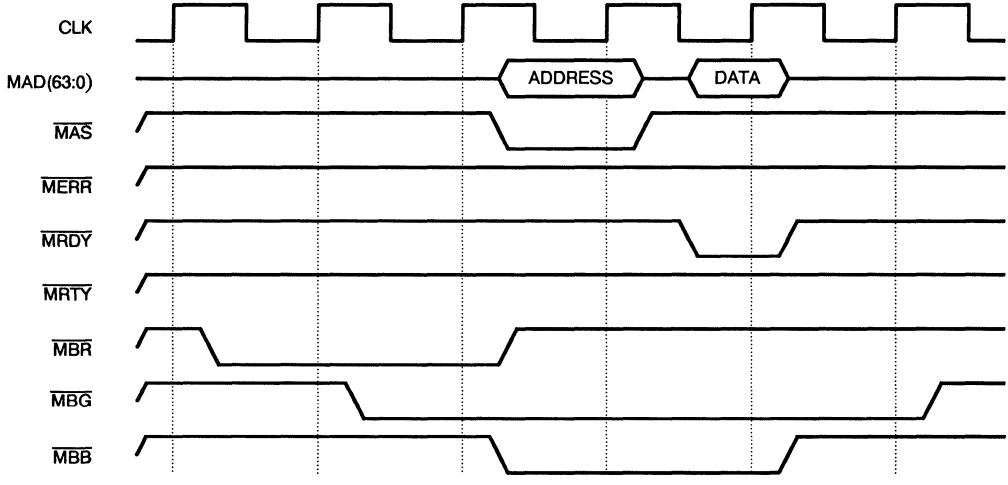
*Mbus level 2 (CY7C605) transaction only.

Mbus Bus Mastership Transfer

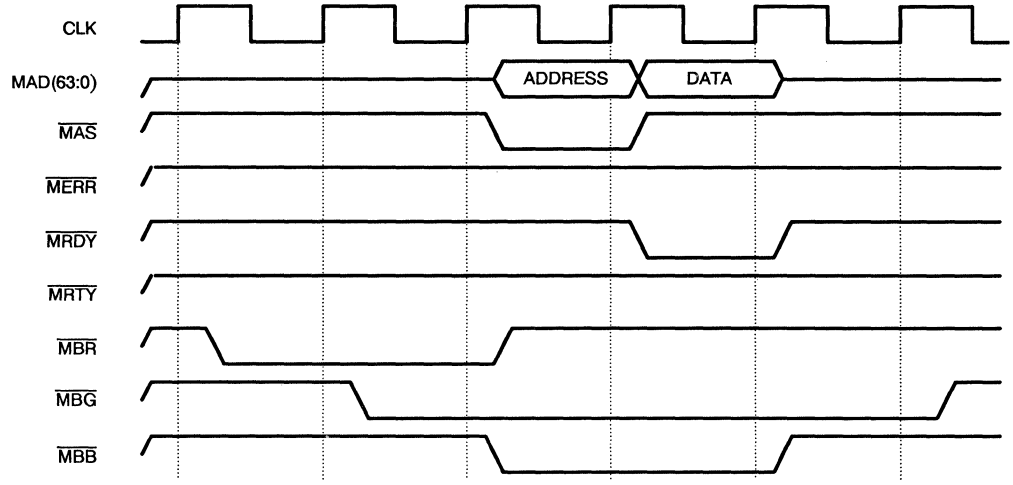


Note on arbitration: $\overline{\text{MBR2}}$ can appear anywhere and does not have to be granted immediately as shown above.

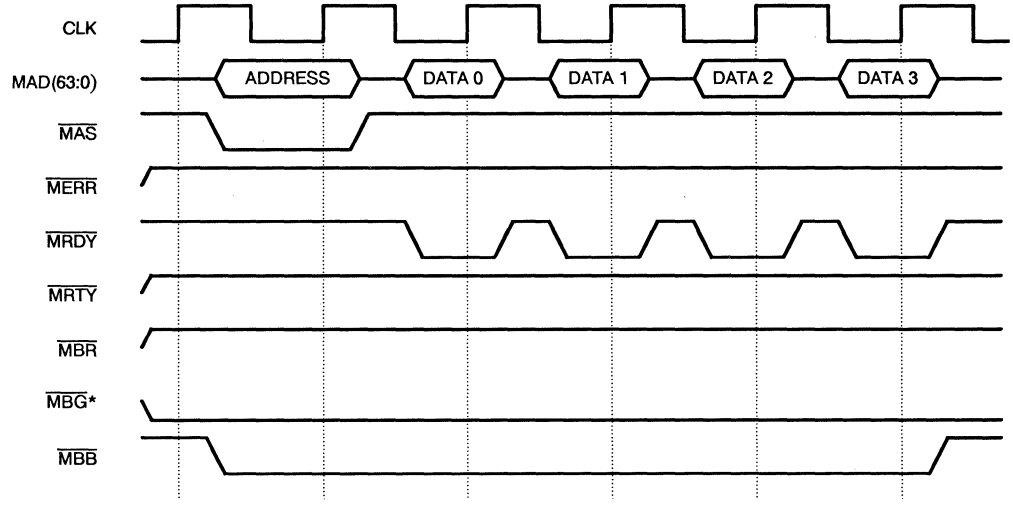
Mbus Single-Cycle Read Transaction



Mbus Single-Cycle Write Transaction

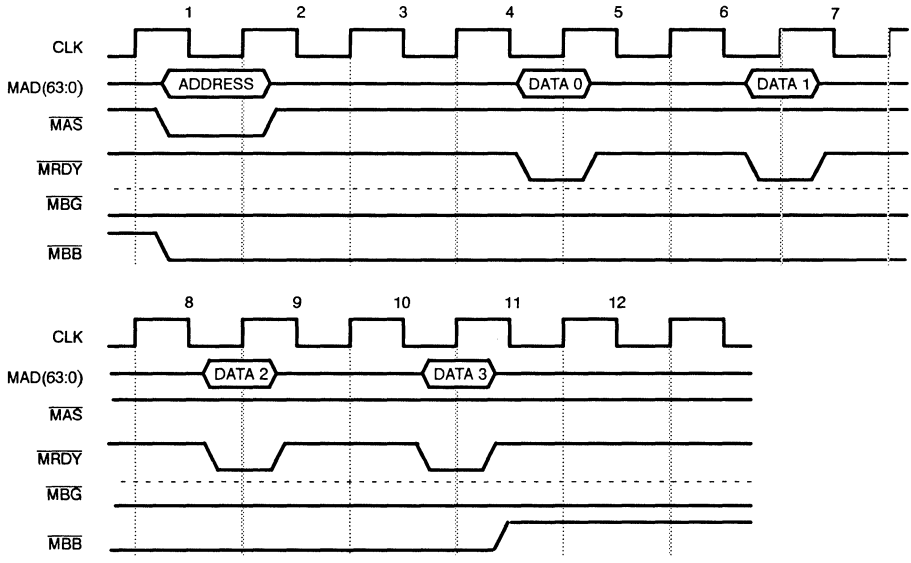


Mbus Burst-Cycle Read Transaction

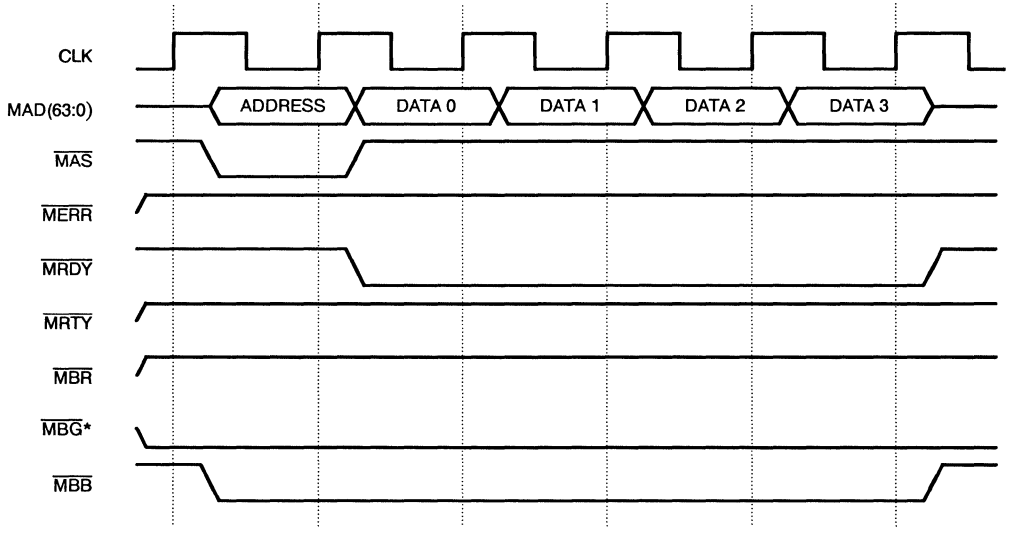


* This timing diagram illustrates a case of bus parking (i.e., Mbus granted to CY7C604/605 by default.)

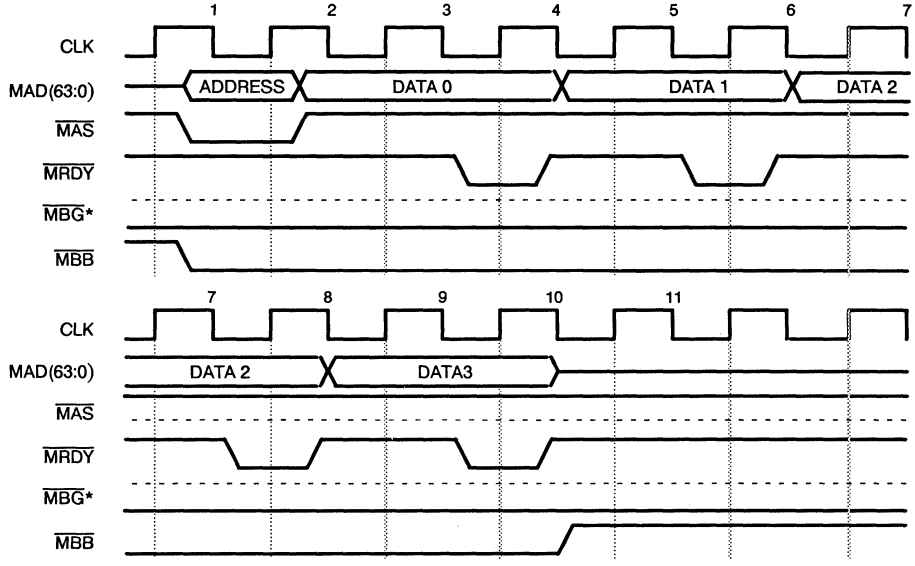
Mbus Burst-Cycle Read Transaction (Slow memory)



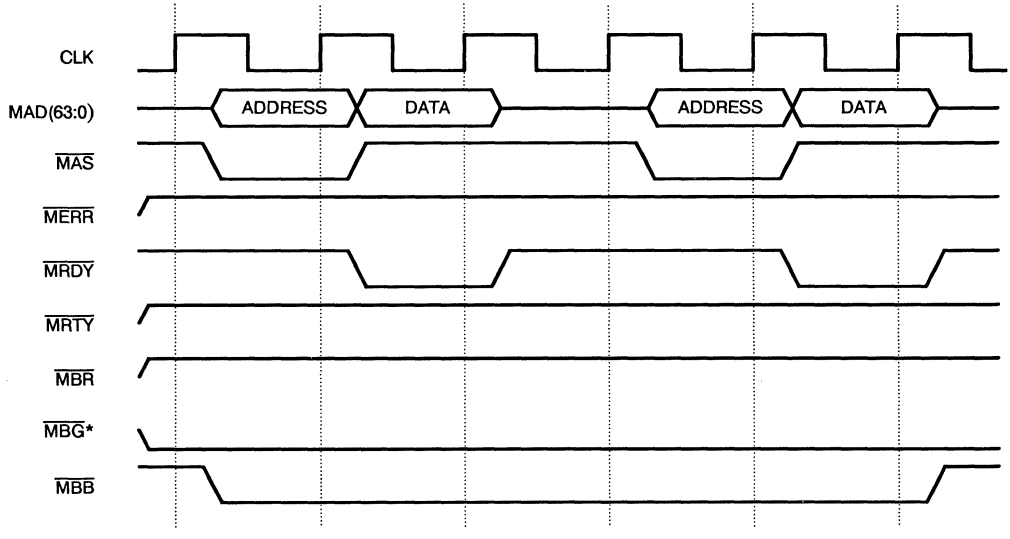
Mbus Burst-Cycle Write Transaction



Mbus Burst-Cycle Write Transaction (Slow memory)

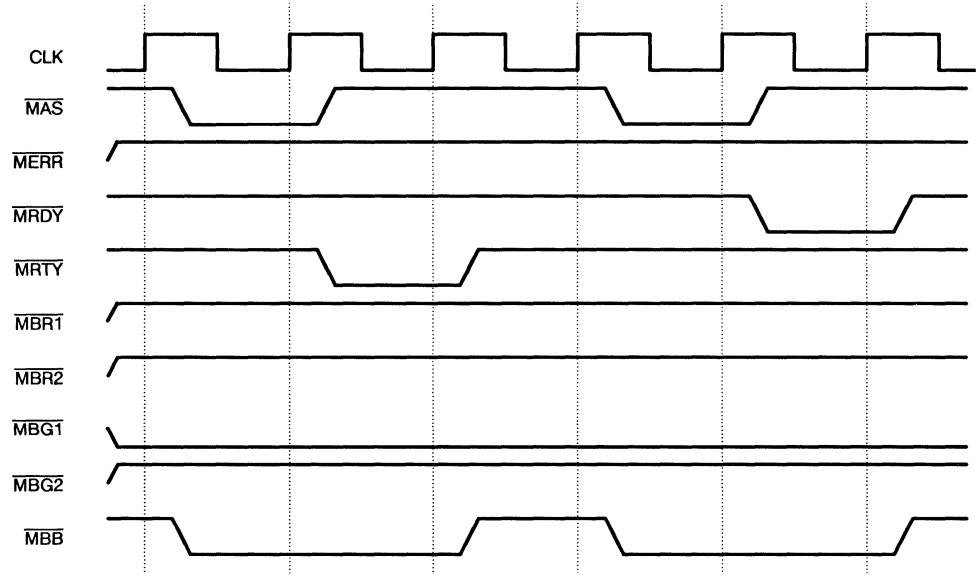


Mbus Locked Transaction

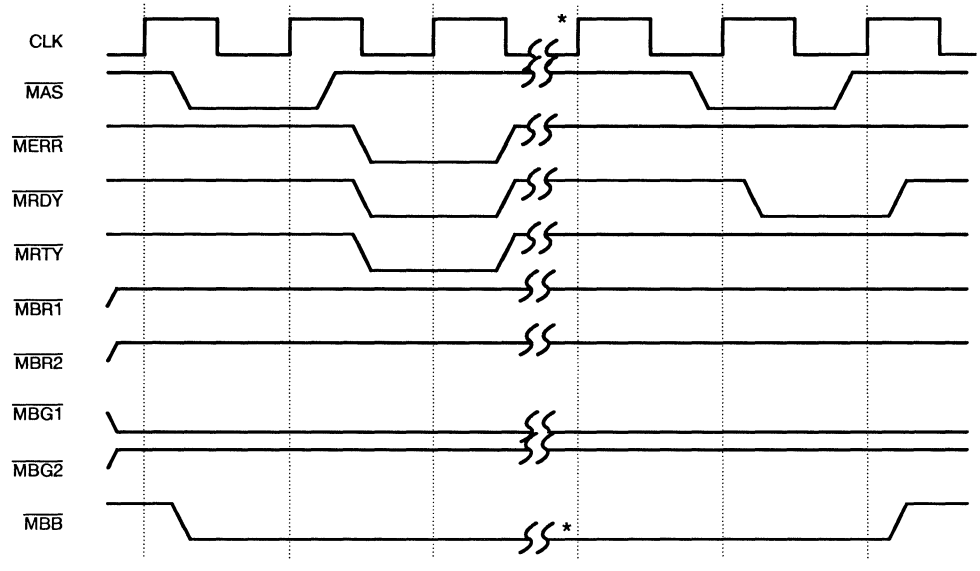


* This timing diagram illustrates a case of bus parking (i.e., Mbus granted to CY7C604/605 by default.)

Mbus Relinquish and Retry

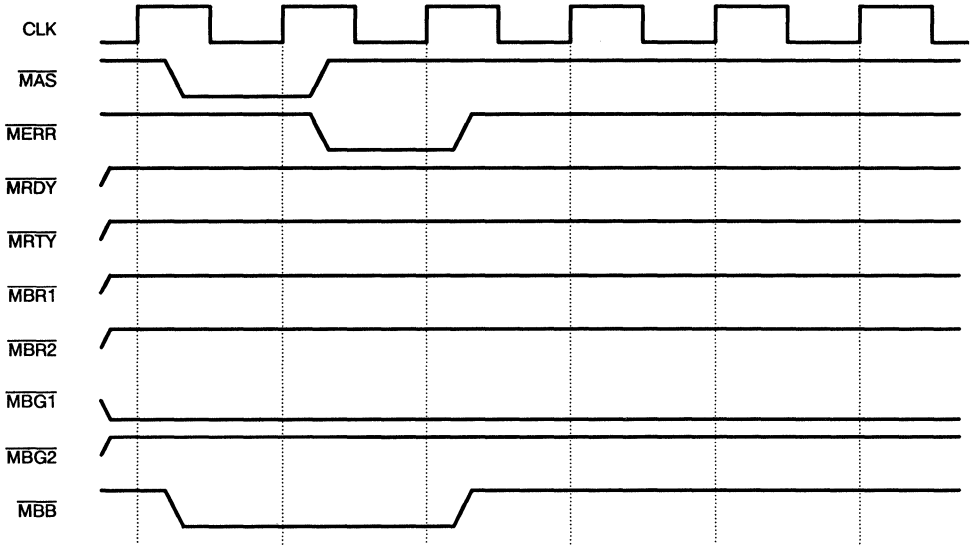


Mbus Retry

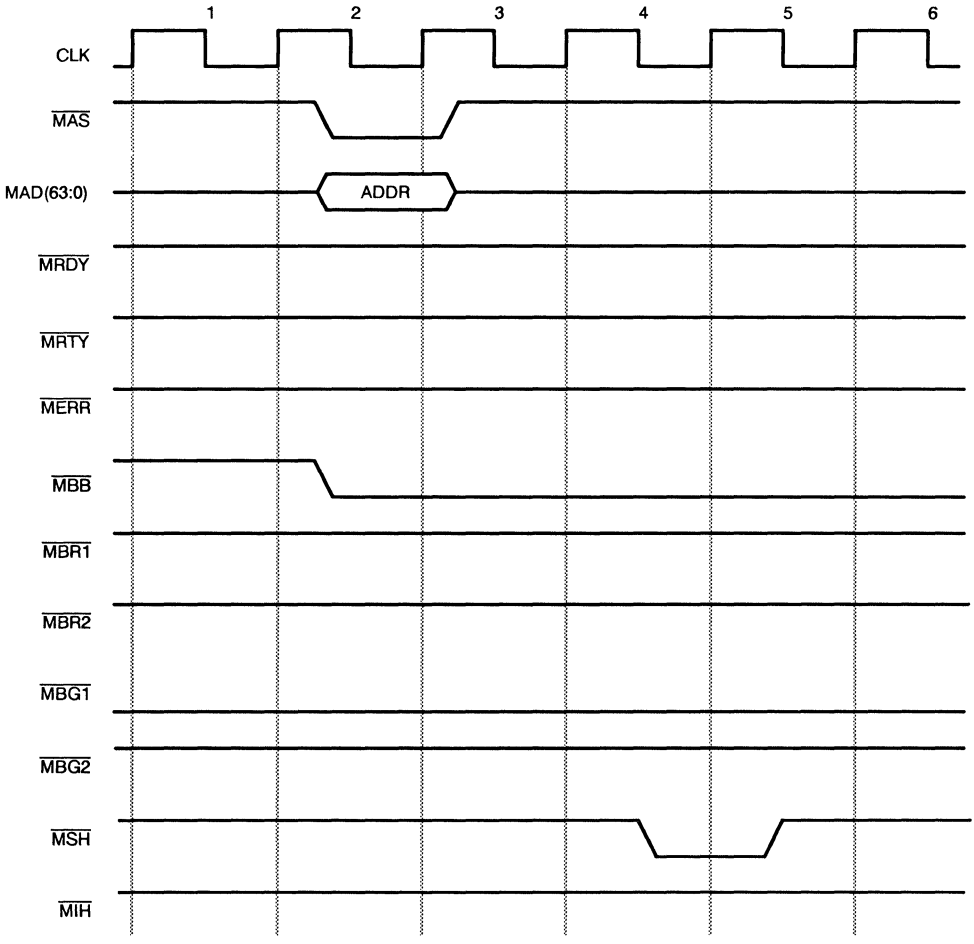


* Add one "dead clock" to this timing diagram in the case of a read access.

Mbus Error

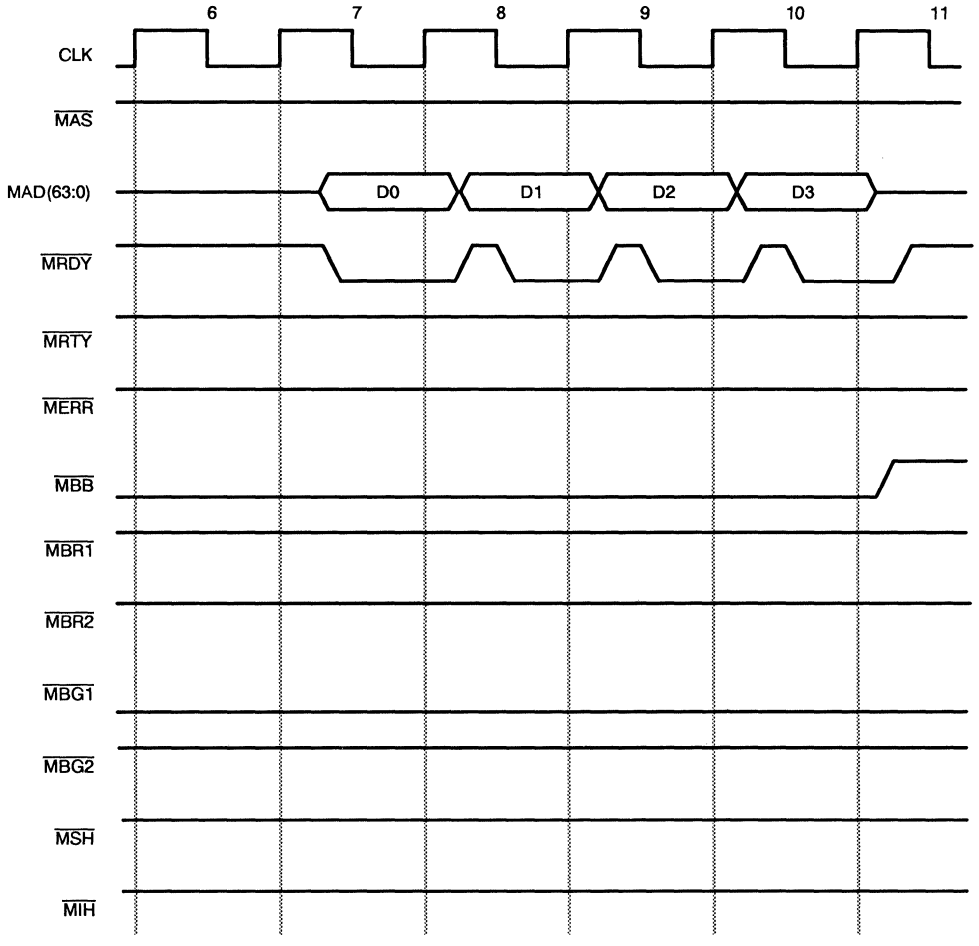


Mbus Coherent Read—Shared Data (CY7C605 only) (page 1 of 2)

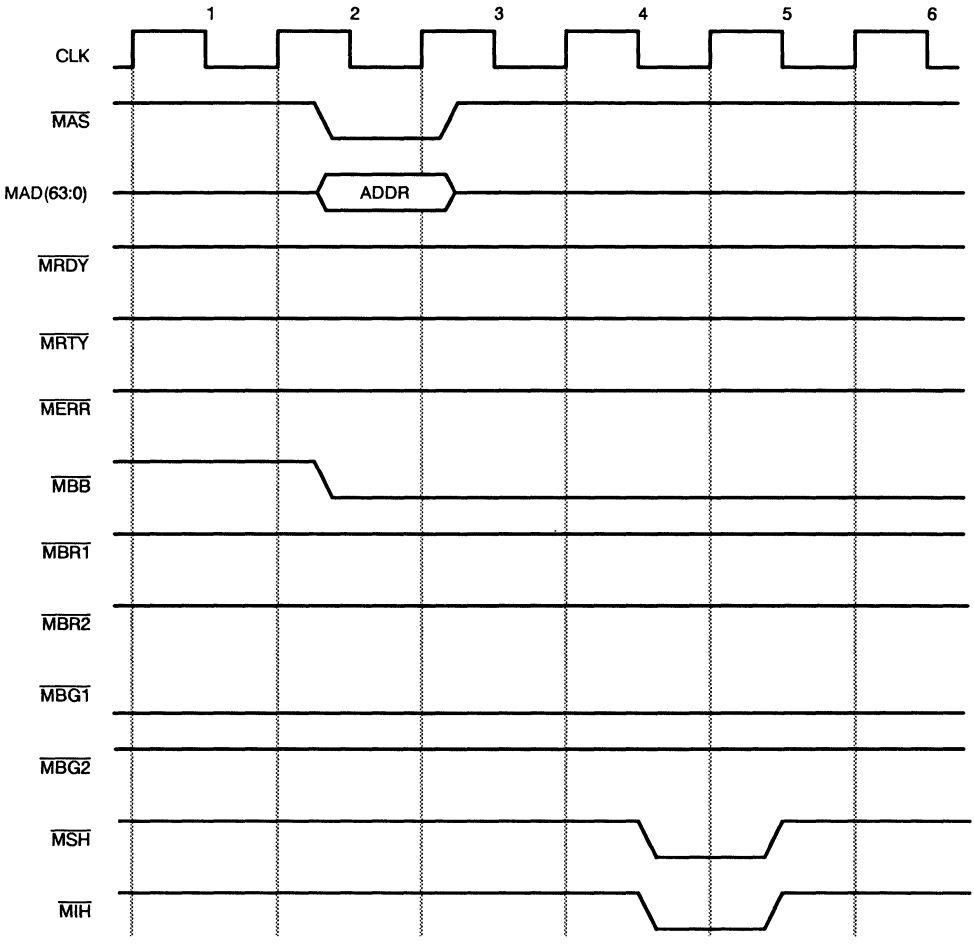


This timing diagram illustrates a coherent read in which the requested data exists in one or more caches in the system, but is not owned by any cache. These caches must assert MSH on cycle A+2 as shown.

Mbus Coherent Read—Shared Data (CY7C605 only) (page 2 of 2)

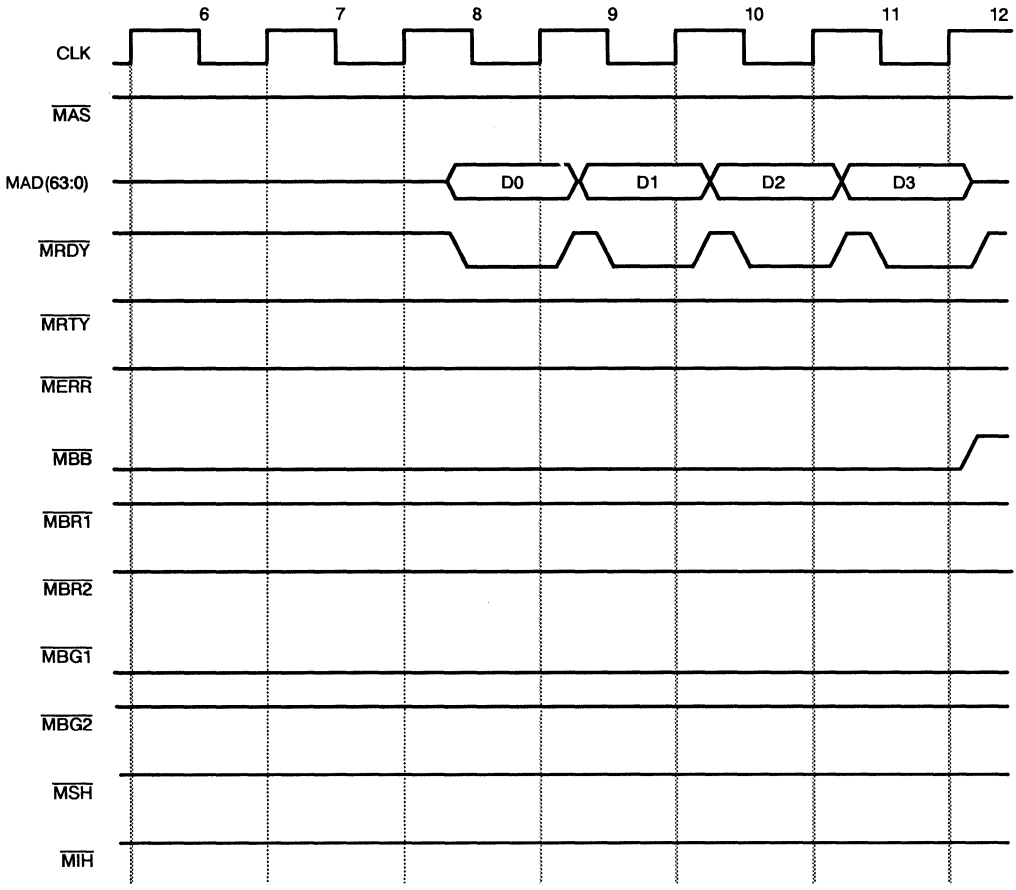


Mbus Coherent Read—Owned Data; Long Latency Memory (CY7C605 only) (page 1 of 2)

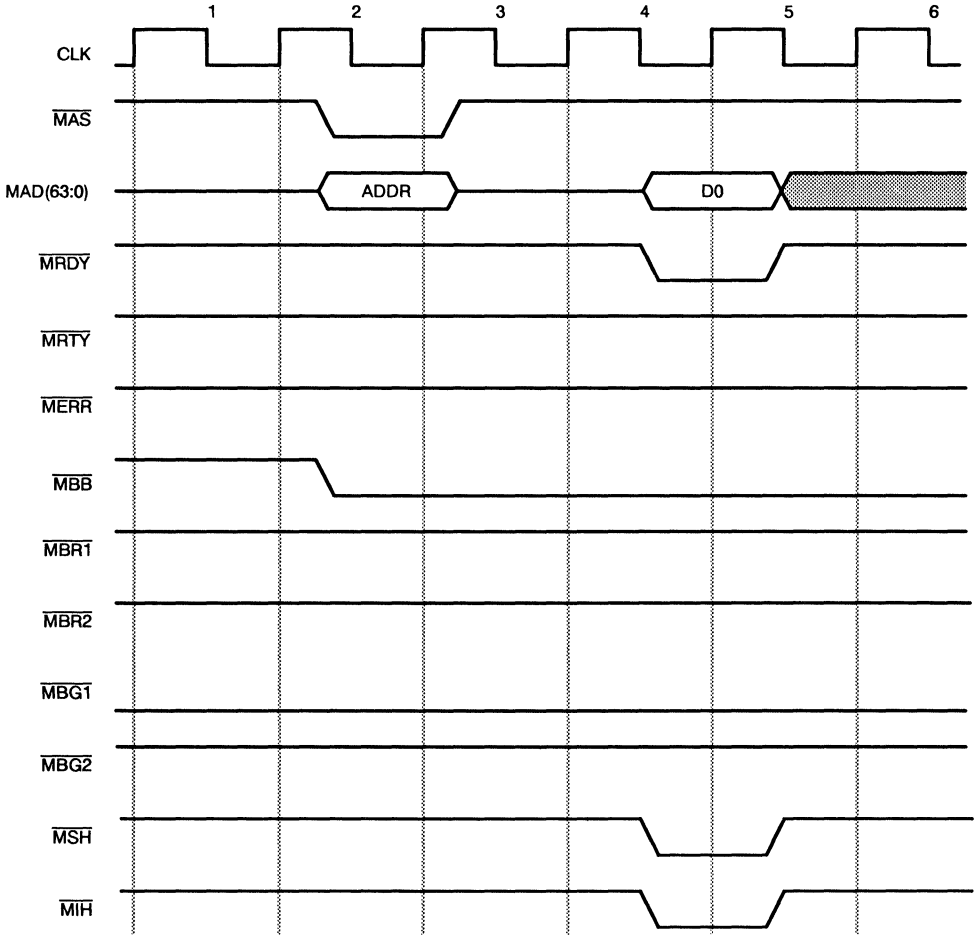


This timing diagram illustrates a coherent read in which the requested data exists in one or more caches in the system, and is owned by a cache. All caches with a copy of the requested data (including the owner) must assert $\overline{MSH}$. Only the owning cache will assert $\overline{MIH}$ on cycle A+2 and supply the data.

Mbus Coherent Read—Owned Data; Long Latency Memory (CY7C605 only) (page 2 of 2)

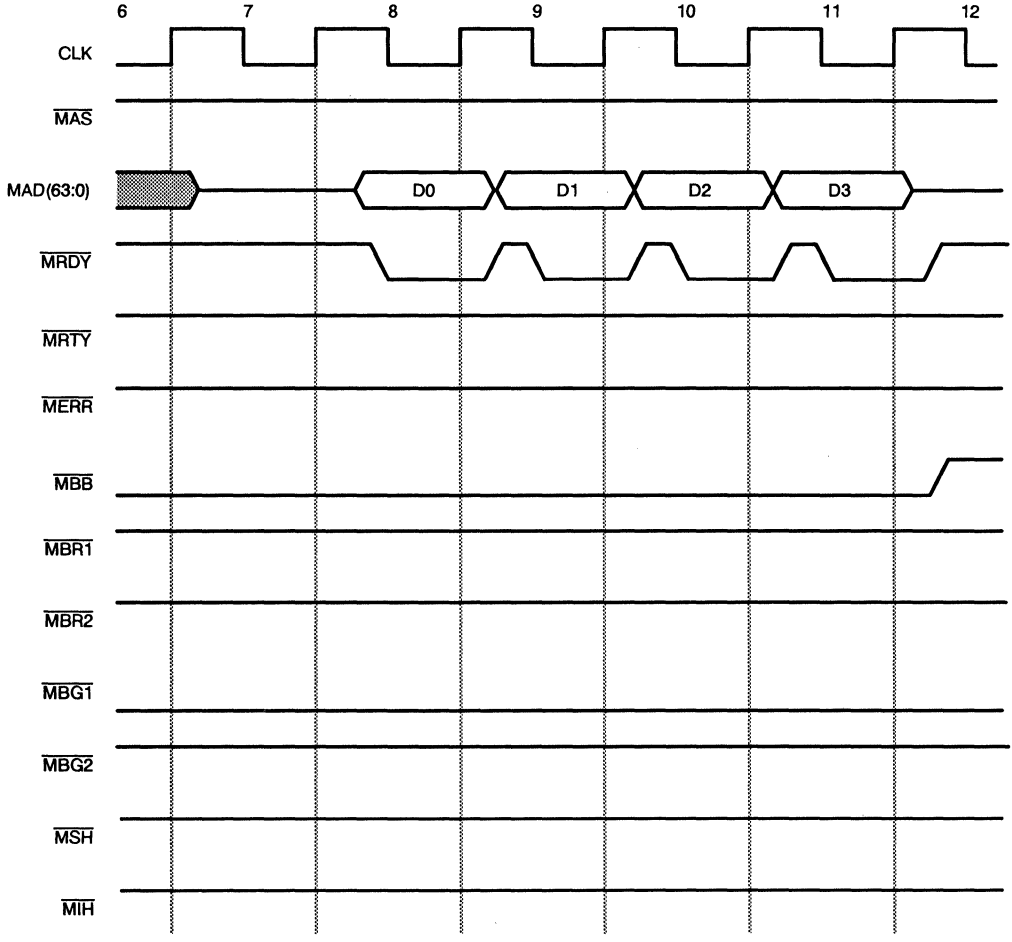


Mbus Coherent Read—Owned Data; Fast Memory (CY7C605 only) (page 1 of 2)

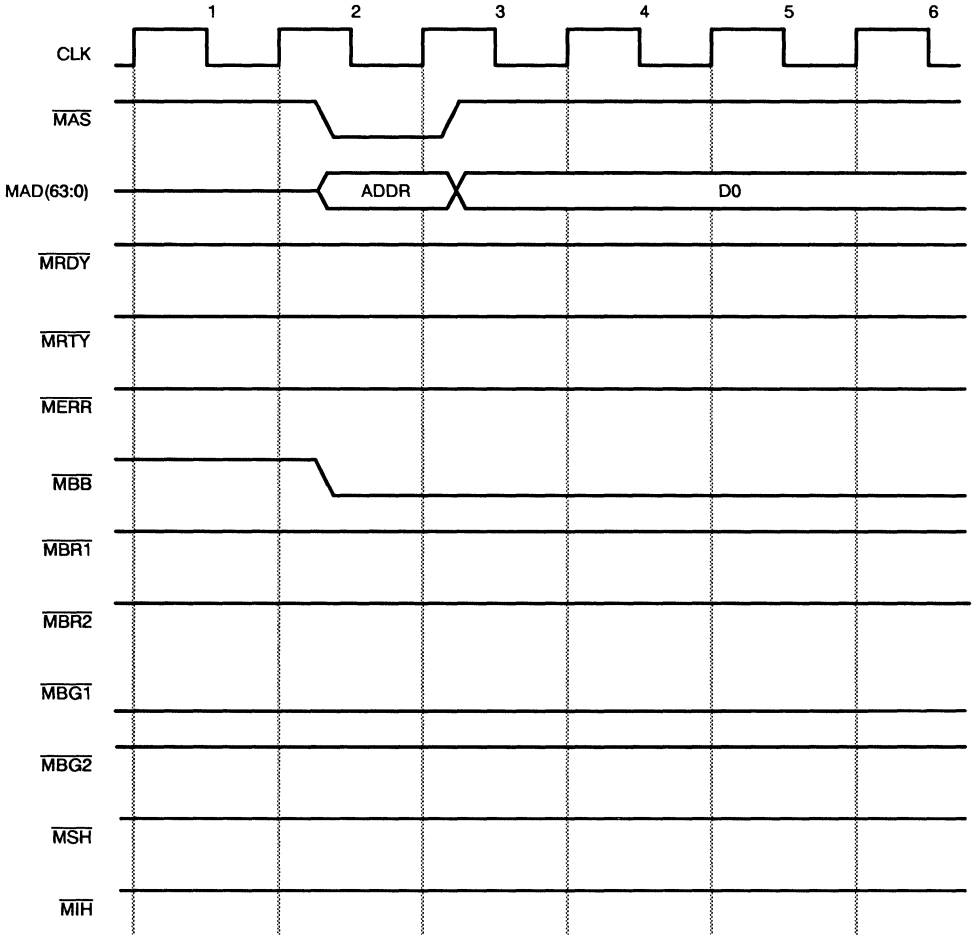


This timing diagram illustrates a coherent read in which the requested data exists in one or more caches in the system, and is owned by a cache. All caches with a copy of the requested data (including the owner) will assert $\overline{\text{MSH}}$. Only the owning cache asserts $\overline{\text{MIH}}$ on cycle A+2 and supplies the data. In this case, memory has already started to respond and thus must get off the bus immediately to allow the cache that owns the data to drive the bus.

Mbus Coherent Read—Owned Data; Fast Memory (CY7C605 only) (page 2 of 2)

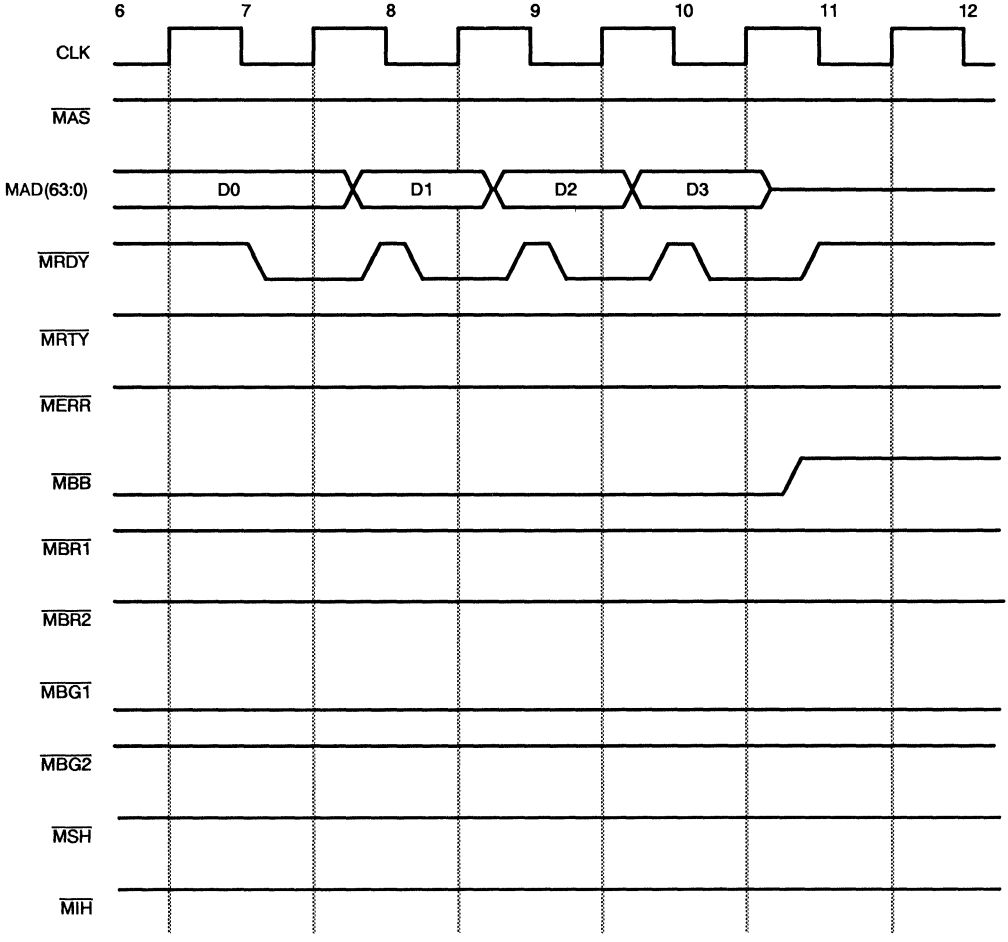


Mbus Coherent Write and Invalidate (CY7C605 only) (page 1 of 2)

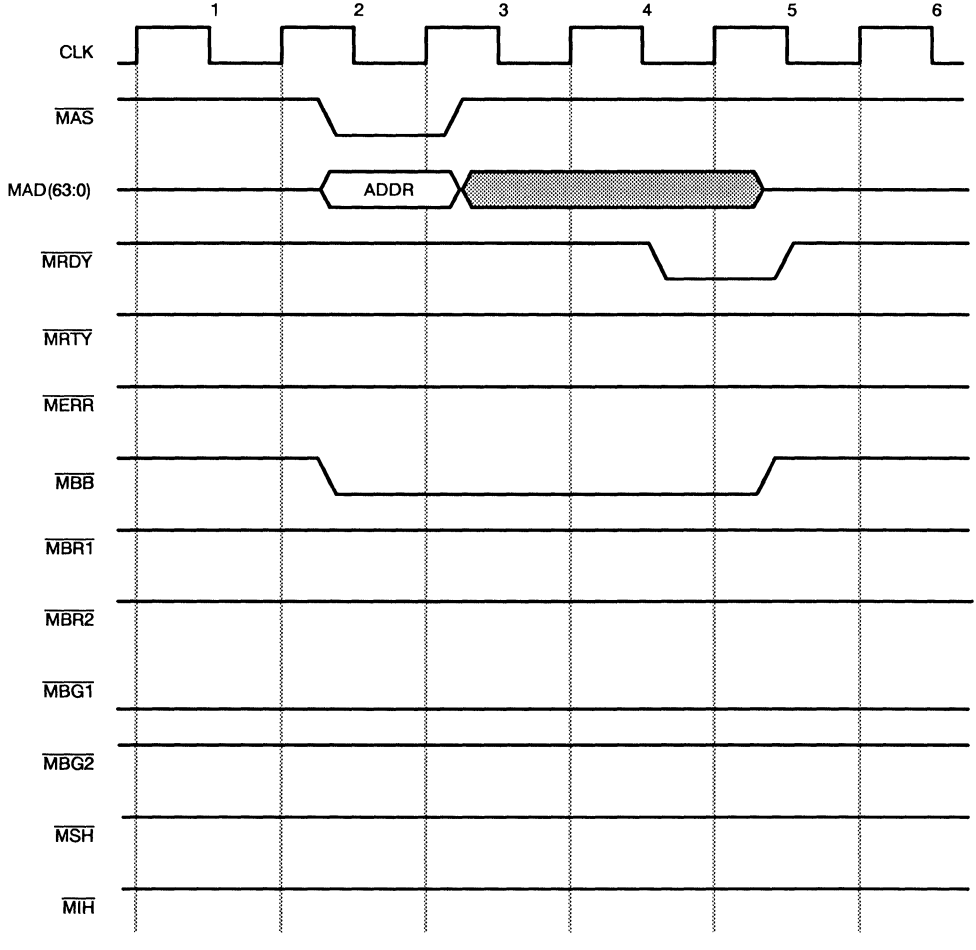


This timing diagram illustrates a coherent write and invalidate operation in which one or more other caches have a copy of the cache line. The other caches invalidate their copy of the cache line.

Mbus Coherent Write and Invalidate (CY7C605 only) (page 2 of 2)

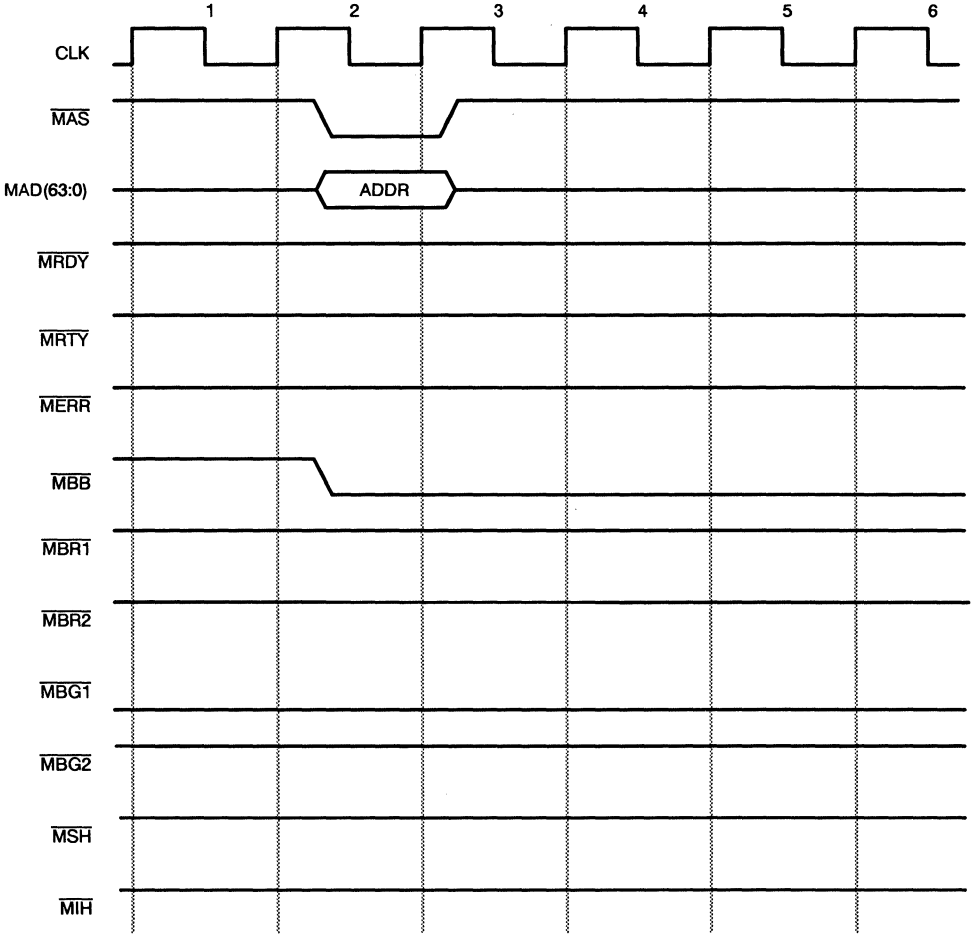


Mbus Coherent Invalidate (CY7C605 only)



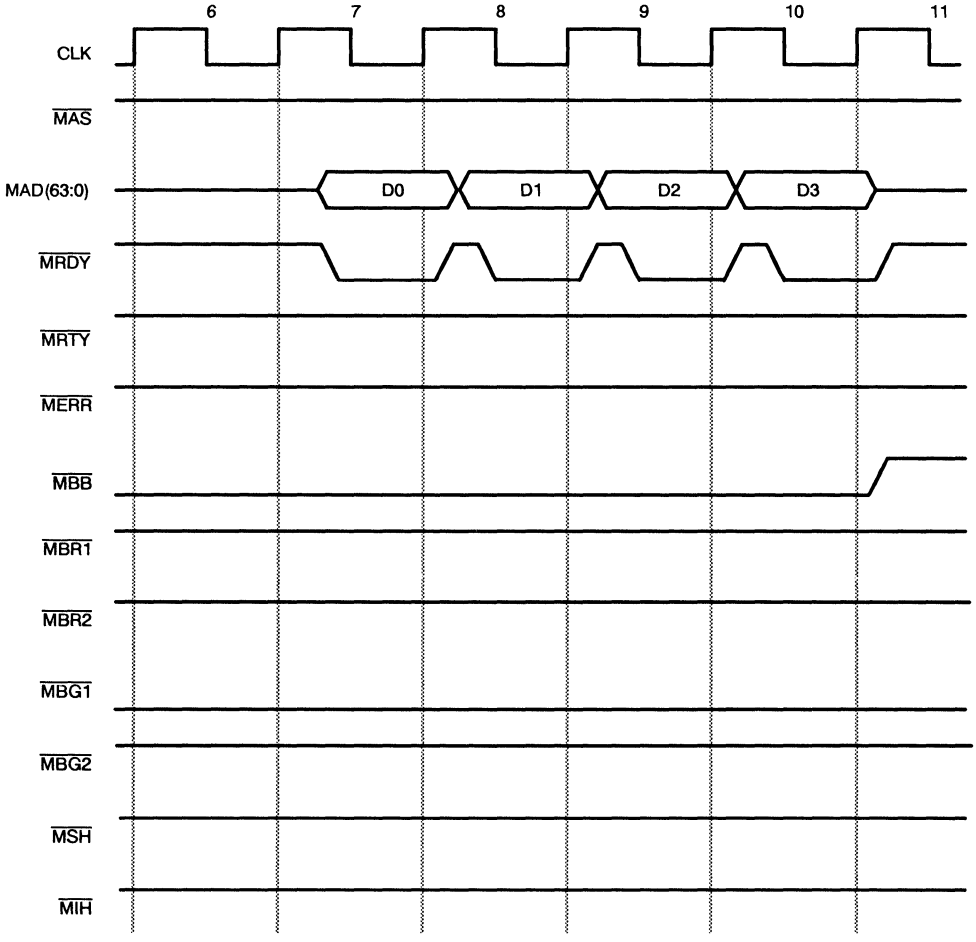
This timing diagram illustrates a coherent invalidate operation. Memory (or second-level cache) asserts $\overline{\text{MRDY}}$ during A + 2 (or later).

Mbus Coherent Read and Invalidate; Shared Data (CY7C605 only) (page 1 of 2)

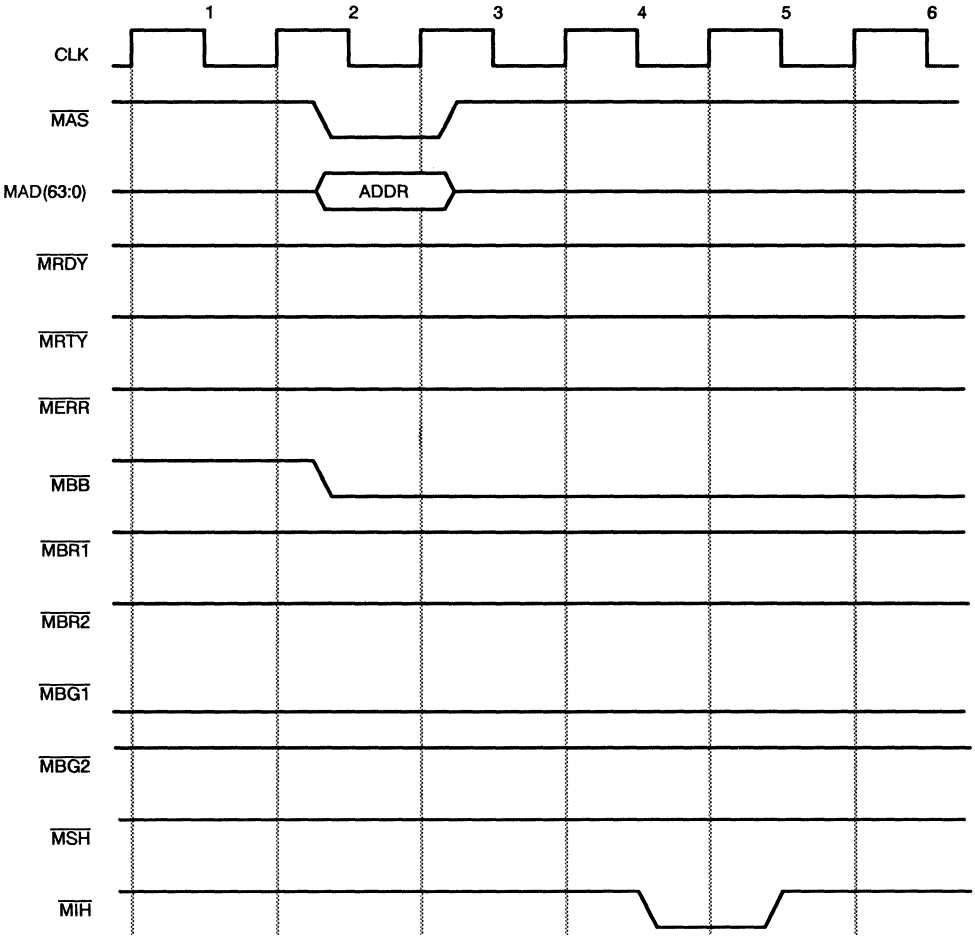


This timing diagram illustrates a coherent read and invalidate in which the requested data may exist in one or more caches in the system.

Mbus Coherent Read and Invalidate: Shared Data (CY7C605 only) (page 2 of 2)

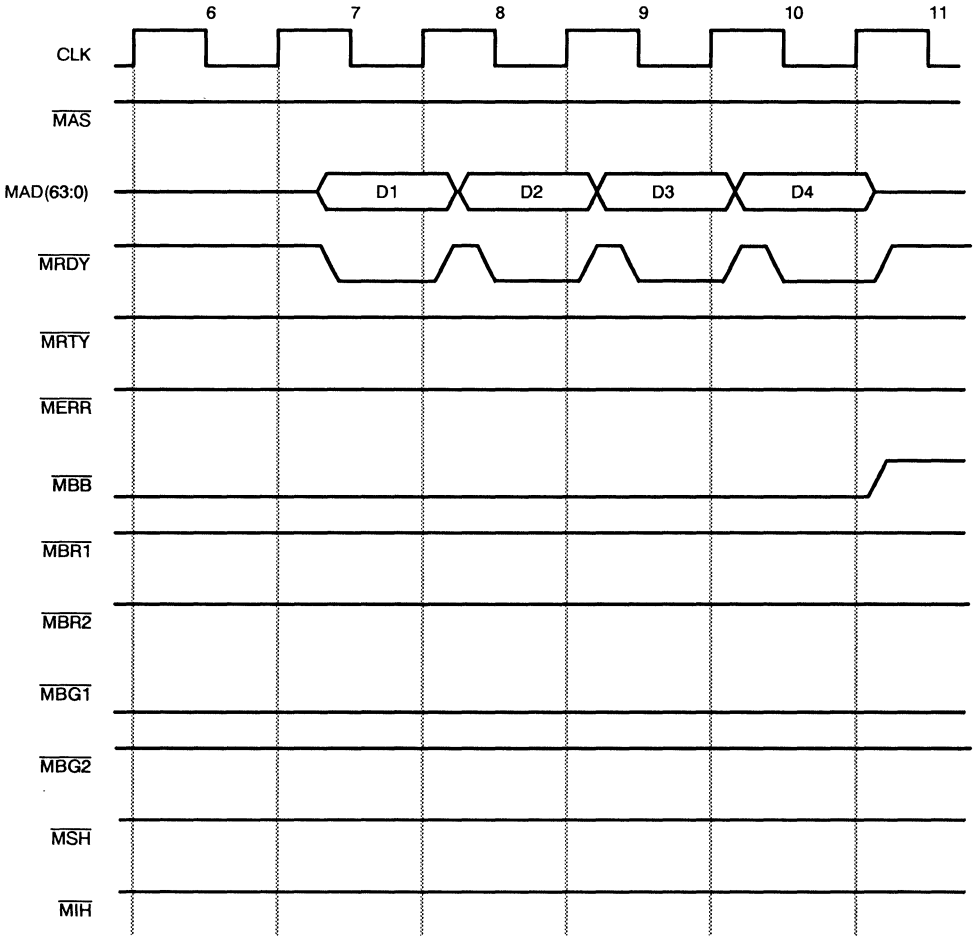


Mbus Coherent Read and Invalidate; Owned Data (CY7C605 only) (page 1 of 2)



This timing diagram illustrates a coherent read in which the requested data exists in one or more caches in the system and is owned by a cache. Only the owning cache asserts $\overline{\text{MIH}}$ on cycle A + 2 and supplies the data.

Mbus Coherent Read and Invalidate: Owned Data (CY7C605 only) (page 2 of 2)



Cache Data RAM

The CY7C157 is a high-performance CMOS static RAM organized as 16K x 16 bits. It is intended specifically for use as a high-speed cache memory for the CY7C600 family of SPARC devices. The CY7C157's 20-ns access time allows operation at processor clock speeds of up to 40 MHz.

The CY7C157 includes registered inputs as well as data-in and data-out latches. Because it was designed specifically for 7C600 family devices, the CY7C157 CRAM requires no glue logic to interface with the CY7C601, CY7C611, CY7C602, CY7C604, or CY7C605. All relevant pins on each device connect directly to one another. The combination of direct connection and on-chip latches and registers yields system designs requiring less board space at a lower cost and with increased reliability. In addition, the CY7C157's self-timed byte-write mechanism relieves the system of any write timing chores.

5.1 Description Of Part

The CY7C157 is organized as two arrays of 16-kbyte static memory. In order to minimize external timing and interface logic, the CY7C157 contains self-timed byte write logic, registered inputs, data-in and data-out latches, and output hold delay logic to control the data-out latches.

Reading the device is accomplished by deasserting $\overline{WE}$ HIGH and $\overline{OE}$ LOW. On the rising edge of CLOCK, addresses A(13:0) are loaded into the input registers. A memory access occurs, and data is held until the next rising edge of CLOCK in order to meet the hold time requirements of the CY7C601/611.

To write to the CY7C157, $\overline{OE}$ must be taken HIGH. If the falling edge of CLOCK samples either or both $\overline{WE}_0$ or $\overline{WE}_1$ LOW, a self-timed byte-write mechanism is triggered. Data is written from the data-in latch into the memory array at the corresponding address.

Note that the $\overline{OE}$ signal must be HIGH for a proper write, as the $\overline{WE}_0$ and $\overline{WE}_1$ signals do not three-state the outputs. A die coat insures alpha immunity.

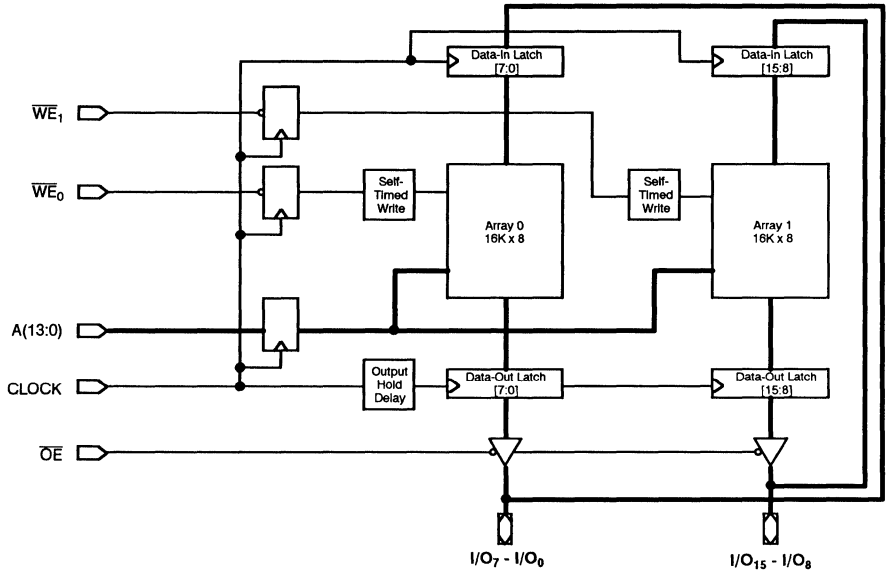


Figure 5–1. CY7C157 Block Diagram

5.2 Operation

Reading the device is accomplished by taking the appropriate WE HIGH and OE LOW. On the rising edge of CLOCK, addresses A_0 through A_{13} are loaded into the input registers. A memory access occurs, and data is held after a read cycle beyond the next rising edge of CLOCK to meet the hold time requirement of the microprocessor.

To write the device, OE must be taken HIGH. If the falling edge of CLOCK samples one or both of WE_0 or WE_1 LOW, a self-timed byte-write mechanism is triggered. Data is written from the data-in latch into the memory array at the corresponding address.

OE must be taken HIGH for a proper write because the write enables do not three-state the outputs.

5.3 Bus Timing

Timing parameters for the CY7C157 are given in Section 7.6, beginning on page 7–49.

5.4 Signal Descriptions

5.4.1 A(13:0)—Address Inputs

Addresses on inputs A<13:0> are loaded into the address registers on the rising edge of CLOCK.

5.4.2 I/O(15:0)—Data Inputs/Outputs

The 16 bidirectional data I/O pins are input signals during write accesses and output signals during read accesses. Data direction is controlled by the output enable pin, $\overline{OE}$.

5.4.3 $\overline{WE}$(1:0)—Write Enables

The write enables initiate the self-timed write mechanism when they are sampled LOW on the falling edge of CLOCK. $\overline{WE_0}$ controls byte writing on data lines I/O(7:0) and $\overline{WE_1}$ controls data lines I/O(15:8).

5.4.4 $\overline{OE}$—Output Enable

The output enable pin controls the output drivers of the bidirectional data lines. To begin a read access, $\overline{OE}$ is taken LOW to enable the output drivers. To begin a write access, $\overline{OE}$ is taken HIGH to three-state the output drivers.

5.4.5 CLOCK—Clock input

CLOCK is the system clock input and is the same signal used by the microprocessor.

6.1 Assembly Language Syntax

The notations given in this section are taken from Sun's SPARC Assembler and are used to describe the suggested assembly language syntax for the instruction definitions given in Section 6.2.

Understanding the use of type fonts is crucial to understanding the assembly language syntax in the instruction definitions. Items in `typewriter font` are literals, to be entered exactly as they appear. Items in *italic font* are metasymbols that are to be replaced by numeric or symbolic values when actual assembly language code is written. For example, *asi* would be replaced by a number in the range of 0 to 255 (the value of the bits in the binary instruction), or by a symbol that has been bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, *regrs2* is a *reg* (i.e., register name) whose binary value will end up in the *rs2* field of the resulting instruction.

6.1.1 Register Names

reg

A *reg* is an integer unit register. It can have a value of:

%0	through	%31	all integer registers
%g0	through	%g7	global registers—same as %0 through %7
%o0	through	%o7	out registers—same as %8 through %15
%l0	through	%l7	local registers—same as %16 through %23
%i0	through	%i7	in registers—same as %24 through %31

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

reg_{rs1}	—*rs1* field
reg_{rs2}	—*rs2* field
reg_{rd}	—*rd* field

freg

A *freg* is a floating-point register. It can have a value from %f0 through %f31. Subscripts further identify the placement of the operand in the binary instruction as one of the following:

$freg_{rs1}$	—*rs*1 field
$freg_{rs2}$	—*rs*2 field
$freg_{rd}$	—*rd* field

creg

A *creg* is a coprocessor register. It can have a value from %c0 through %c31. Subscripts further identify the placement of the operand in the binary instruction as one of the following:

$creg_{rs1}$	—*rs*1 field
$creg_{rs2}$	—*rs*2 field
$creg_{rd}$	—*rd* field

6.1.2 Special Symbol Names

Certain special symbols need to be written exactly as they appear in the syntax table. These appear in `typewriter font`, and are preceded by a percent sign (`%`). The percent sign is part of the symbol name; it must appear as part of the literal value.

The symbol names are:

`%psr`	Processor State Register
`%wim`	Window Invalid Mask register
`%tbr`	Trap Base Register
`%y`	Y register
`%fsr`	Floating-point State Register
`%csr`	Coprocessor State Register
`%fq`	Floating-point Queue
`%cq`	Coprocessor Queue
`%hi`	Unary operator that extracts high 22 bits of its operand
`%lo`	Unary operator that extracts low 10 bits of its operand

6.1.3 Values

Some instructions use operands comprising values as follows:

simm13—A signed immediate constant that fits in 13 bits
const22—A constant that fits in 22 bits
asi—An alternate address space identifier (0 to 255)

6.1.4 Label

A label is a sequence of characters comprised of alphabetic letters (a–z, A–Z (upper and lower case distinct)), underscore (_), dollar sign ($), period (.), and decimal digits (0–9), but which does not begin with a decimal digit.

Some instructions offer a choice of operands. These are grouped as follows:

regaddr:

reg_{rs1}
reg_{rs1} + reg_{rs2}

address:

reg_{rs1}
reg_{rs1} + reg_{rs2}
reg_{rs1} + *simm13*
reg_{rs1} - *simm13*
simm13
simm13 + reg_{rs1}

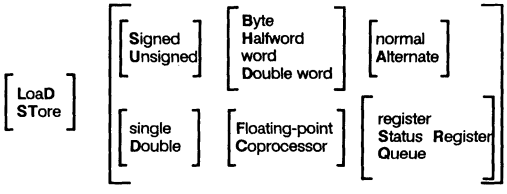
reg_or_imm:

reg_{rs2}
simm13

6.1.5 Instruction Mnemonics

Figure 6–1 illustrates the mnemonics used to describe the SPARC instruction set. Note that some combinations possible in *Figure 6–1* do not correspond to valid instructions (such as store signed or floating-point convert extended to extended). Refer to the instruction summary on page 6–6 for a list of valid SPARC instructions.

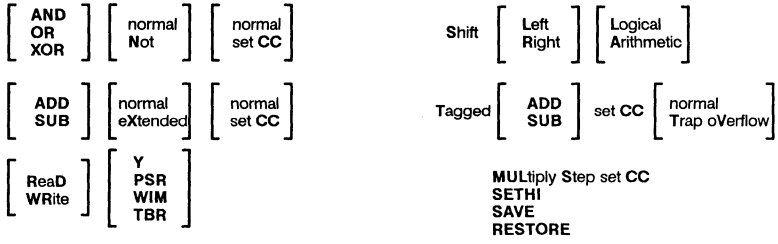
Data Transfer



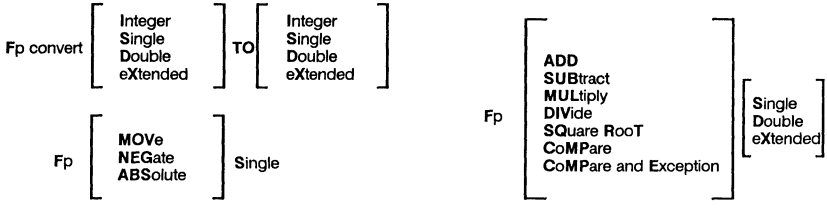
atomic **SWAP** word

atomic Load-Store Unsigned **Byte**

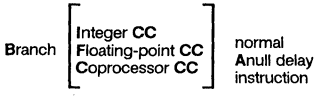
Integer Operations



Floating-Point Operations



Control Transfer



JuMP and Link
RETurn from Trap

CALL
Trap on Integer CC

Figure 6–1. SPARC Instruction Mnemonic Summary

6.2 Definitions

This section provides a detailed definition for each CY7C601 instruction. Each definition includes: the instruction operation; suggested assembly language syntax; a description of the salient features, restrictions and trap conditions; a list of synchronous or floating-point\coprocessor traps which can occur as a consequence of executing the instruction; and the instruction format and op codes. Instructions are defined in alphabetical order with the instruction mnemonic shown in large bold type at the top of the page for easy reference. The instruction set summary that precedes the definitions, (*Table 6–2*), groups the instructions by type.

Table 6–1 identifies the abbreviations and symbols used in the instruction definitions. An example of how some of the description notations are used is given below in *Figure 6–2*. Register names, labels and other aspects of the syntax used in these instructions are described in the previous section.

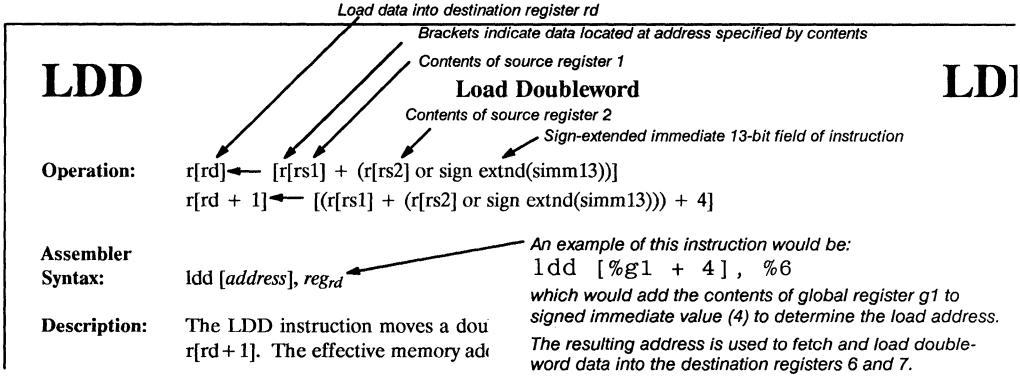


Figure 6–2. Instruction Description

Table 6–1. Instruction Description Notations

Symbol	Description
a	Instruction field that controls instruction annulling during control transfers
AND, OR XOR, etc.	AND, OR, XOR, etc operators
asi	Instruction field that identifies the load/store alternate address space
c	The icc carry bit
ccc	The coprocessor condition code field of the CSR
CONCAT	Concatenate
cond	Instruction field that selects the condition code test for branches
CQ.ADDR	The address portion of the Coprocessor Queue
CQ.INSTR	The instruction portion of the Coprocessor Queue
c[rd]	Depending on context, the coprocessor register (or its contents) specified by the instruction field, e.g., rd, rs1, rs2
CSR	Coprocessor State Register
CWP	PSR's Current Window Pointer field
disp22	Instruction field that contains the 22-bit sign-extended displacement for branches
disp30	Instruction field that contains the 30-bit word displacement for calls
dz	Floating-point exception:division by zero
EC	PSR's Enable Coprocessor bit
EF	PSR's Enable FPU bit

Table 6–1. Instruction Description Notations (continued)

Symbol	Description
ET	PSR's Enable Traps bit
fcc	The floating-point condition code field of the FSR
FQ.ADDR	The address portion of the Floating-point queue
FQ.INSTR	The instruction portion of the Floating-point queue
f[rd]s	The suffix (s, d, x) after the operand indicates the precision of the operand
f[rs1]	Depending on context, the floating-point register (or its contents) specified by the instruction field, e.g. , rd, rs1, rs2
FSR	Floating-point State Register
i	Instruction field that selects rs2 or sign extnd(simm13) as the second operand
icc	The integer condition code field of the PSR
imm22	Instruction field that contains the 22-bit constant used by SETHI
n	The icc negative bit
not	Logical complement operator
nPC	next Program Counter
nv	Floating-point exception:invalid
nx	Floating-point exception:inexact result
of	Floating-point exception:overflow
opc	Instruction field that specifies the count for Coprocessor-operate instructions
operand2	Either r[rs2] or sign extnd(simm13)
PC	Program Counter
pS	PSR's previous Supervisor bit
PSR	Processor State Register
r[15]	A directly addressed register (could be floating-point or coprocessor)
rd	Instruction field that specifies the destination register (except for store)
r[rd]	Depending on context, the integer register (or its contents) specified by the instruction field, e.g. , rd, rs1, rs2
r[rd]<31>	< > are used to specify bit fields of a particular register or I/O signal
[r[rs1] + r[rs2]]	The contents of the address specified by r[rs1] + r[rs2]
rs1	Instruction field that specifies the source 1 register
rs2	Instruction field that specifies the source 2 register
S	PSR's Supervisor bit
shcnt	Instruction field that specifies the count for shift instructions
sign extn(simm13)	Instruction field that contains the 13-bit, sign-extended immediate value
Symbol	Description
TBR	Trap Base Register
tt	TBR's trap type field
uf	Floating-point exception:underflow
v	The icc overflow bit
WIM	Window Invalid Mask register
Y	Y Register
z	The icc zero bit
-	Subtract
x	Multiply
/	Divide
<–	Replaced by
7FFFFF H	Hexadecimal number representation
+	Add

Table 6–2. Instruction Set Summary

	Name	Operation		Cycles
Load and Store Instructions	LDSB(LDSBA*)	Load Signed Byte	(from Alternate Space)	2
	LDSH(LDSHA*)	Load Signed Halfword	(from Alternate Space)	2
	LDUB(LDUBA*)	Load Unsigned Byte	(from Alternate Space)	2
	LDUH(LDUHA*)	Load Unsigned Halfword	(from Alternate Space)	2
	LD(LDA*)	Load Word	(from Alternate Space)	2
	LDD(LDDA*)	Load Doubleword	(from Alternate Space)	3
	LDF	Load Floating Point		2
	LDDF	Load Double Floating Point		3
	LDFSR	Load Floating Point State Register		2
	LDC	Load Coprocessor		2
	LDDC	Load Double Coprocessor		3
	LDCSR	Load Coprocessor State Register		2
	STB(STBA*)	Store Byte	(into Alternate Space)	3
	STH(STHA*)	Store Halfword	(into Alternate Space)	3
	ST(STA*)	Store Word	(into Alternate Space)	3
	STD(STDA*)	Store Doubleword	(into Alternate Space)	4
	STF	Store Floating Point		3
	STDF	Store Double Floating Point		4
	STFSR	Store Floating Point State Register		3
	STDFQ*	Store Double Floating Point Queue		4
	STC	Store Coprocessor		3
	STDC	Store Double Coprocessor		4
	STCSR	Store Coprocessor State Register		3
	STDCQ*	Store Double Coprocessor Queue		4
	LDSTUB(LDSTUBA*)	Atomic Load/Store Unsigned Byte	(in Alternate Space)	4
	SWAP(SWAPA*)	Swap r Register with Memory	(in Alternate Space)	4
Arithmetic/Logical/Shift	ADD(ADDcc)	Add	(and modify icc)	1
	ADDX(ADDXcc)	Add with Carry	(and modify icc)	1
	TADDcc(TADDccTV)	Tagged Add and modify icc	(and Trap on overflow)	1
	SUB(SUBcc)	Subtract	(and modify icc)	1
	SUBX(SUBXcc)	Subtract with Carry	(and modify icc)	1
	TSUBcc(TSUBccTV)	Tagged Subtract and modify icc	(and Trap on overflow)	1
	MULScc	Multiply Step and modify icc		1
	AND(ANDcc)	And	(and modify icc)	1
	ANDN(ANDNcc)	And Not	(and modify icc)	1
	OR(ORcc)	Inclusive Or	(and modify icc)	1
	ORN(ORNcc)	Inclusive Or Not	(and modify icc)	1
	XOR(XORcc)	Exclusive Or	(and modify icc)	1
	XNOR(XNORcc)	Exclusive Nor	(and modify icc)	1
	SLL	Shift Left Logical		1
	SRL	Shift Right Logical		1
	SRA	Shift Right Arithmetic		1
	SETHI	Set High 22 Bits of r Register		1
	SAVE	Save caller's window		1
	RESTORE	Restore caller's window		1
Control Transfer	Bicc	Branch on Integer Condition Codes		1**
	FBicc	Branch on Floating Point Condition Codes		1**
	CBccc	Branch on Coprocessor Condition Codes		1**
	CALL	Call		1**
	JMPL	Jump and Link		2**
	RETT	Return from Trap		2**
	Ticc	Trap on Integer Condition Codes		1 (4 if Taken)
Read/Write Control Registers	RDY	Read Y Register		1
	RDPSR*	Read Processor State Register		1
	RDWIM*	Read Window Invalid Mask		1
	RDTBR*	Read Trap Base Register		1
	WRY	Write Y Register		1
	WRPSR*	Write Processor State Register		1
	WRWIM*	Write Window Invalid Mask		1
	WRTBR*	Write Trap Base Register		1
	UNIMP	Unimplemented Instruction		1
	IFLUSH	Instruction Cache Flush		1
FP (CP) Ops	FPop	Floating Point Unit Operations		1 to Launch
	CPop	Coprocessor Operations		1 to Launch

* privileged instruction

** assuming delay slot is filled with useful instruction

ADD — Add — ADD

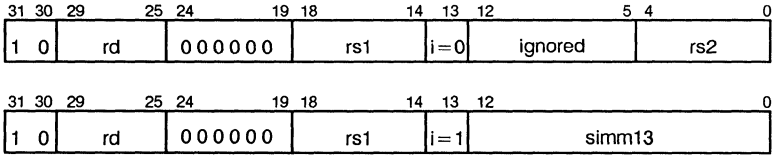
Operation: r[rd] ← r[rs1] + (r[rs2] or sign extnd(simm13))

Assembler Syntax: add *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: The ADD instruction adds the contents of the register named in the *rs1* field ,r[rs1], to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The result is placed in the register specified in the *rd* field.

Traps: none

Format:



6

ADDcc — Add and modify icc — ADDcc

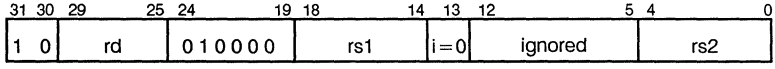
Operation: r[rd] ← r[rs1] + operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
c ← (r[rs1]<31> AND operand2<31>)
OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

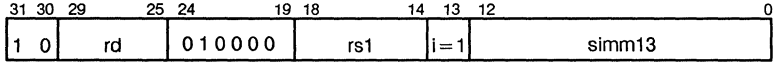
Assembler Syntax: addcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: ADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. The result is placed in the register specified in the *rd* field. In addition, ADDcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:





ADDX — Add with Carry — ADDX

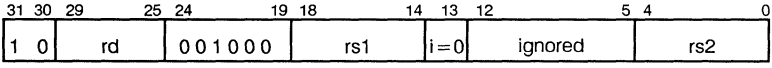
Operation: r[rd] ← r[rs1] + (r[rs2] or sign extnd(simm13)) + c

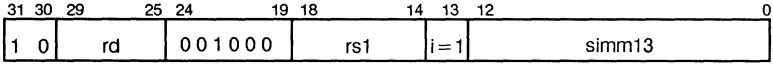
Assembler Syntax: addx *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: ADDX adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. It then adds the PSR's carry bit (*c*) to that result. The final result is placed in the register specified in the *rd* field.

Traps: none

Format:





6

ADDXcc — Add with Carry and modify icc — ADDXcc

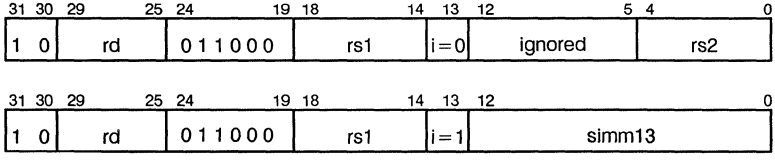
Operation: r[rd] ← r[rs1] + operand2 + c, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
c ← (r[rs1]<31> AND operand2<31>)
OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

Assembler Syntax: addxcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: ADDXcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. It then adds the PSR's carry bit (*c*) to that result. The final result is placed in the register specified in the *rd* field. ADDXcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



AND — And — AND

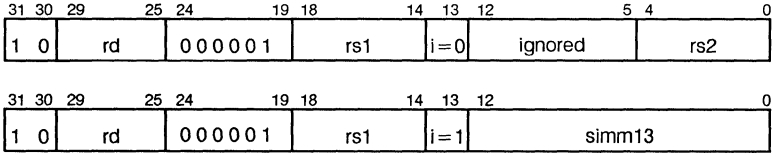
Operation: r[rd] ← r[rs1] AND (r[rs2] or sign extnd(simm13))

Assembler Syntax: and *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical AND of the contents of register r[rs1] with either the contents of r[rs2] (if if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if if bit field i = 1). The result is stored in register r[rd].

Traps: none

Format:



6

ANDcc — And and modify icc — ANDcc

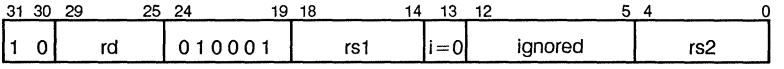
Operation: r[rd] ← r[rs1] AND (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← 0
c ← 0

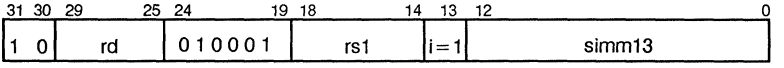
Assembler Syntax: andcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical AND of the contents of register r[rs1] with either the contents of r[rs2] (if if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if if bit field i = 1). The result is stored in register r[rd]. ANDcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:





ANDN — And Not — ANDN

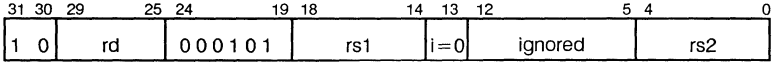
Operation: r[rd] ← r[rs1] AND $\overline{\text{(r[rs2] or sign extnd(simm13))}}$

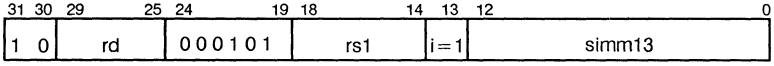
Assembler Syntax: andn *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: ANDN does a bitwise logical AND of the contents of register r[rs1] with the logical compliment (not) of either r[rs2] (if if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if if bit field i = 1). The result is stored in register r[rd].

Traps: none

Format:





6

ANDNcc — And Not and modify icc — ANDNcc

Operation:

r[rd] ← r[rs1] AND $\overline{\text{(r[rs2] or sign extnd(simm13))}}$
n ← r[rd] < 31 >
z ← if r[rd] = 0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax:

andncc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

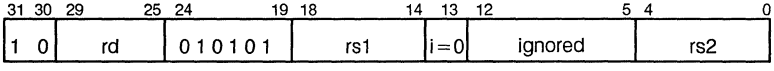
Description:

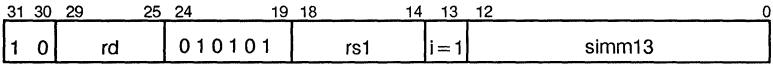
ANDNcc does a bitwise logical AND of the contents of register r[rs1] with the logical compliment (not) of either r[rs2] (if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i = 1). The result is stored in register r[rd]. ANDNcc also modifies all the integer condition codes in the manner described above.

Traps:

none

Format:





Bicc — Integer Conditional Branch — Bicc

Operation: PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
else nPC ← nPC + 4

Assembler Syntax:

ba{,a}	*label*	
bn{,a}	*label*	
bne{,a}	*label*	synonym: bnz
be{,a}	*label*	synonym: bz
bg{,a}	*label*	
ble{,a}	*label*	
bge{,a}	*label*	
bl{,a}	*label*	
bgu{,a}	*label*	
bleu{,a}	*label*	
bcc{,a}	*label*	synonym: bgeu
bcs{,a}	*label*	synonym: blu
bpos{,a}	*label*	
bneg{,a}	*label*	
bvc{,a}	*label*	
bvs{,a}	*label*	

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The Bicc instructions (except for BA and BN) evaluate specific integer condition code combinations (from the PSR's *icc* field) based on the branch type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. Refer to Section 2.3.3.3 for additional information on control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See Section 2.3.3.4 regarding delay-branch instructions.

Branch Never (BN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

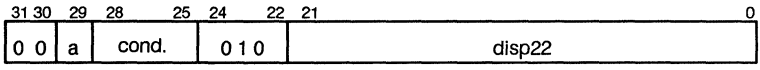
Branch Always (BA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if $a = 1$, the delay instruction is annulled; if $a = 0$, the delay instruction is executed.

The delay instruction following a Bicc (other than BA) should not be a delayed-control-transfer instruction. The results of following a Bicc with another delayed control transfer instruction are implementation-dependent and therefore unpredictable.

Traps: none

Mnemonic	Cond.	Operation	icc Test
BN	0000	Branch Never	No test
BE	0001	Branch on Equal	z
BLE	0010	Branch on Less or Equal	z OR (n XOR v)
BL	0011	Branch on Less	n XOR v
BLEU	0100	Branch on Less or Equal, Unsigned	c OR z
BCS	0101	Branch on Carry Set (Less than, Unsigned)	c
BNEG	0110	Branch on Negative	n
BVS	0111	Branch on oVerflow Set	v
BA	1000	Branch Always	No test
BNE	1001	Branch on Not Equal	not z
BG	1010	Branch on Greater	not(z OR (n XOR v))
BGE	1011	Branch on Greater or Equal	not(n XOR v)
BGU	1100	Branch on Greater, Unsigned	not(c OR z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not c
BPOS	1110	Branch on Positive	not n
BVC	1111	Branch on oVerflow Clear	not v

Format:



CALL — Call — CALL

Operation: r[15] ← PC
PC ← nPC
nPC ← PC + (disp30 x 4)

Assembler Syntax: call *label*

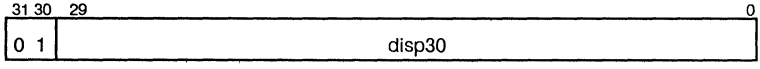
Description: The CALL instruction causes a delayed, unconditional, PC-relative control transfer to the address (PC + 4) + (disp30 x 4). The CALL instruction does not have an annul bit, therefore the delay slot instruction following the CALL instruction is always executed (See Section 2.3.3.4). CALL first writes its return address (PC) into the *outs* register, r[15], and then adds 4 to the PC. The 32-bit displacement which is added to the new PC is formed by appending two low-order zeros to the 30-bit word displacement contained in the instruction. Consequently, the target address can be anywhere in the CY7C601's user or supervisor address space.

If the instruction following a CALL uses register r[15] as a source operand, hardware interlocks add a one cycle delay.

Programming note: a register-indirect CALL can be constructed using a JMPL instruction with *rd* set to 15.

Traps: none

Format:



6

CBccc — Coprocessor Conditional Branch — CBccc

Operation: PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
else nPC ← nPC + 4

Assembler Syntax:

cba{,a}	*label*
cbn{,a}	*label*
cb3{,a}	*label*
cb2{,a}	*label*
cb23{,a}	*label*
cb1{,a}	*label*
cb13{,a}	*label*
cb12{,a}	*label*
cb123{,a}	*label*
cb0{,a}	*label*
cb03{,a}	*label*
cb02{,a}	*label*
cb023{,a}	*label*
cb01{,a}	*label*
cb013{,a}	*label*
cb012{,a}	*label*

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The CBccc instructions (except for CBA and CBN) evaluate specific coprocessor condition code combinations (from the CCC < 1:0 > inputs) based on the branch type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. See Section 2.3.3.3 regarding control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See Section 2.3.3.4 regarding delayed branching.

Branch Never (CBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch Always (CBA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if $a = 1$, the delay instruction is annulled; if $a = 0$, the delay instruction is executed.

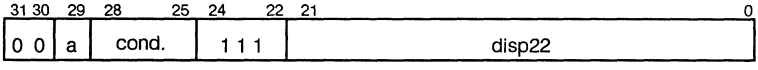
To prevent misapplication of the condition codes, a non-coprocessor instruction must immediately precede a CBccc instruction.

A CBccc instruction generates a cp_disabled trap (and does not branch or annul) if the PSR's EC bit is reset or if no coprocessor is present.

Traps: cp_disabled
cp_exception

Mnemonic	cond.	CCC<1:0> test
CBN	0000	Never
CB123	0001	1 or 2 or 3
CB12	0010	1 or 2
CB13	0011	1 or 3
CB1	0100	1
CB23	0101	2 or 3
CB2	0110	2
CB3	0111	3
CBA	1000	Always
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

Format:



6

CPop — Coprocessor Operate — CPop

Operation: Dependent on Coprocessor implementation

Assembler Syntax: Unspecified

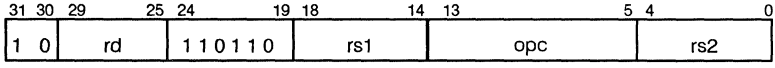
Description: CPop1 and CPop2 are the instruction formats for coprocessor operate instructions. The *op3* field for CPop1 is 110110; for CPop2 it's 110111. The coprocessor operations themselves are encoded in the *opc* field and are dependent on the coprocessor implementation. Note that this does not include load/store coprocessor instructions, which fall into the integer unit's load/store instruction category.

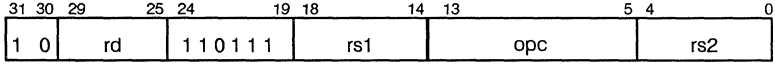
All CPop instructions take all operands from, and return all results to, the coprocessor's registers. The data types supported, how the operands are aligned, and whether a CPop generates a cp_exception trap are Coprocessor dependent.

A CPop instruction causes a cp_disabled trap if the PSR's EC bit is reset or if no coprocessor is present.

Traps: cp_disabled
cp_exception

Format:





FABSs — Absolute Value Single — FABSs

(CY7C602 Instruction Only)

Operation: f[rd]s ← f[rs2]s AND 7FFFFFFF H

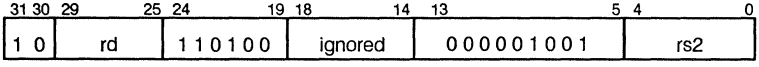
Assembler Syntax: fabss *freg$_{rs2}$*, *freg$_{rd}$*

Description: The FABSs instruction clears the sign bit of the word in f[rs2] and places the result in f[rd]. It does not round.

Since rs2 can be either an even or odd register, FABSs can also operate on the high-order words of double and extended operands, which accomplishes sign bit clear for these data types.

Traps: fp_disabled
fp_exception*

Format:



6

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FADDd

Add Double

FADDd

(CY7C602 Instruction Only)

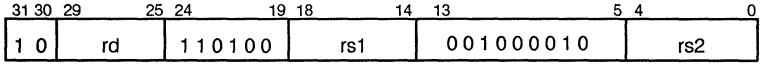
Operation: f[rd]d ← f[rs1]d + f[rs2]d

Assembler Syntax: faddd *freg$_{rs1}$*, *freg$_{rs2}$*, *freg$_{rd}$*

Description: The FADDd instruction adds the contents of f[rs1] CONCAT f[rs1 + 1] to the contents of f[rs2] CONCAT f[rs2 + 1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd + 1].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



FADDs — Add Single — FADDs

(CY7C602 Instruction Only)

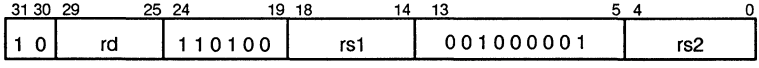
Operation: f[rd]s ← f[rs1]s + f[rs2]s

Assembler Syntax: fadds *freg$_{rs1}$*, *freg$_{rs2}$*, *freg$_{rd}$*

Description: The FADDs instruction adds the contents of f[rs1] to the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



6

FADDx — Add Extended — FADDx

(CY7C602 Instruction Only)

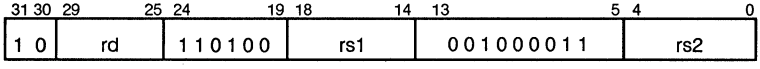
Operation: f[rd]x ← f[rs1]x + f[rs2]x

Assembler Syntax: faddx *freg$_{rs1}$*, *freg$_{rs2}$*, *freg$_{rd}$*

Description: The FADDx instruction adds the contents of f[rs1] CONCAT f[rs1 + 1] CONCAT f[rs1 + 2] to the contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd + 1], and f[rd + 2].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



FBfcc — Floating-Point Conditional Branch — FBfcc

Operation:

PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
else nPC ← nPC + 4

Assembler Syntax:

fba{,a}	*label*	
fbn{,a}	*label*	
fbu{,a}	*label*	
fbg{,a}	*label*	
fbug{,a}	*label*	
fbl{,a}	*label*	
fbul{,a}	*label*	
fblg{,a}	*label*	
fbne{,a}	*label*	synonym: fbnz
fbe{,a}	*label*	synonym: fbz
fbue{,a}	*label*	
fbge{,a}	*label*	
fbuge{,a}	*label*	
fble{,a}	*label*	
fbule{,a}	*label*	
fbo{,a}	*label*	

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description:

The FBfcc instructions (except for FBA and FBN) evaluate specific floating-point condition code combinations (from the FCC < 1:0 > inputs) based on the branch type, as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. See Section 2.3.3.3 for additional information on control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See Section 2.3.3.4 regarding delayed branch instructions.

Branch Never (FBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch Always (FBA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if $a = 1$, the delay instruction is annulled; if $a = 0$, the delay instruction is executed.

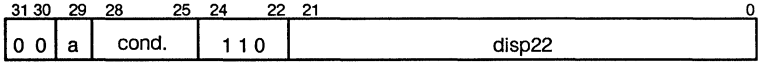
To prevent misapplication of the condition codes, a non-floating-point instruction must immediately precede an FBfcc instruction.

An FBfcc instruction generates an fp_disabled trap (and does not branch or annul) if the PSR's EF bit is reset or if no Floating-Point Unit is present.

Traps: fp_disabled
fp_exception*

Mnemonic	Cond.	Operation	fcc Test
FBN	0000	Branch Never	no test
FBNE	0001	Branch on Not Equal	U or L or G
FBLG	0010	Branch on Less or Greater	L or G
FBUL	0011	Branch on Unordered or Less	U or L
FBL	0100	Branch on Less	L
FBUG	0101	Branch on Unordered or Greater	U or G
FBG	0110	Branch on Greater	G
FBU	0111	Branch on Unordered	U
FBA	1000	Branch Always	no test
FBE	1001	Branch on Equal	E
FBUE	1010	Branch on Unordered or Equal	U or E
FBGE	1011	Branch on Greater or Equal	G or E
FBUGE	1100	Branch on Unordered or Greater or Equal	U or G or E
FBLE	1101	Branch on Less or Equal	L or E
FBULE	1110	Branch on Unordered or Less or Equal	U or L or E
FBO	1111	Branch on Ordered	L or G or E

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FCMPd — Compare Double — FCMPd

(CY7C602 Instruction Only)

Operation: fcc ← f[rs1]d COMPARE f[rs2]d

Assembler Syntax: fcmpd *freg$_{rs1}$, freg$_{rs2}$*

Description: FCMPd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

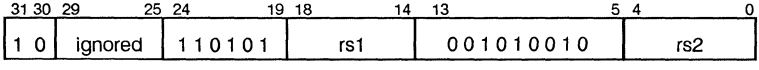
In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPd causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPEd Compare Double and Exception if Unordered FCMPEd

(CY7C602 Instruction Only)

Operation: fcc ← f[rs1]d COMPARE f[rs2]d

Assembler Syntax: fcmped *freg$_{rs1}$*, *freg$_{rs2}$*

Description: FCMPEd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

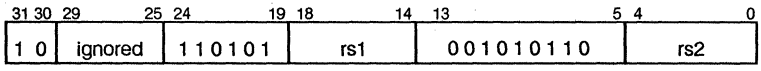
In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEd causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPEs Compare Single and Exception if Unordered FCMPEs

(CY7C602 Instruction Only)

Operation: fcc ← f[rs1]s COMPARE f[rs2]s

Assembler Syntax: fcmpes *freg$_{rs1}$*, *freg$_{rs2}$*

Description: FCMPEs subtracts the contents of f[rs2] from the contents of f[rs1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

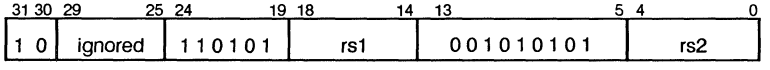
In this table, fs1 stands for the contents of f[rs1] and fs2 represents the contents of f[rs2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non–floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEs causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPEx Compare Extended and Exception if Unordered FCMPEx

(CY7C602 Instruction Only)

Operation: fcc ← f[rs1]x COMPARE f[rs2]x

Assembler Syntax: fcmpex *freg$_{rs1}$*, *freg$_{rs2}$*

Description: FCMPEx subtracts the contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] from the contents of f[rs1] CONCAT f[rs1 + 1] CONCAT f[rs1 + 2] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

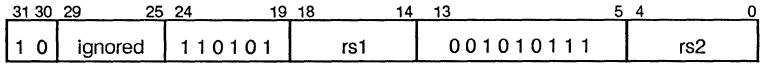
In this table, fs1 stands for the contents of f[rs1], f[rs1 + 1], f[rs1 + 2] and fs2 represents the contents of f[rs2], f[rs2 + 1], f[rs2 + 2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEx causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPs — Compare Single — FCMPs

(CY7C602 Instruction Only)

Operation: fcc ← f[rs1]s COMPARE f[rs2]s

Assembler Syntax: fcmps *freg$_{rs1}$*, *freg$_{rs2}$*

Description: FCMPs subtracts the contents of f[rs2] from the contents of f[rs1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

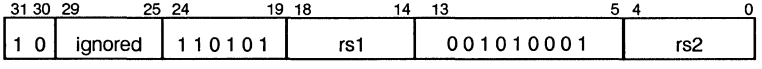
In this table, fs1 stands for the contents of f[rs1] and fs2 represents the contents of f[rs2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPs causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FCMPx — Compare Extended — FCMPx

(CY7C602 Instruction Only)

Operation: fcc ← f[rs1]x COMPARE f[rs2]x

Assembler Syntax: fcmpx *freg$_{rs1}$*, *freg$_{rs2}$*

Description: FCMPx subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

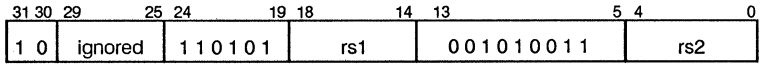
In this table, fs1 stands for the contents of f[rs1], f[rs1+1], f[rs1+2] and fs2 represents the contents of f[rs2], f[rs2+1], f[rs2+2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPx causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:



FDIVd

Divide Double

FDIVd

(CY7C602 Instruction Only)

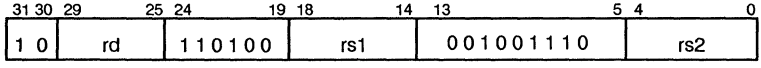
Operation: f[rd]d ← f[rs1]d / f[rs2]d

Assembler Syntax: fdivd *freg$_{rs1}$, freg$_{rs2}$, freg$_{rd}$*

Description: The FDIVd instruction divides the contents of f[rs1] CONCAT f[rs1 + 1] by the contents of f[rs2] CONCAT f[rs2 + 1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd + 1].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:



6

FDIVs — Divide Single — FDIVs

(CY7C602 Instruction Only)

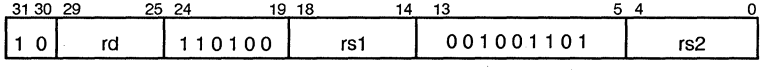
Operation: f[rd]s ← f[rs1]s / f[rs2]s

Assembler Syntax: fdivs *freg$_{rs1}$*, *freg$_{rs2}$*, *freg$_{rd}$*

Description: The FDIVs instruction divides the contents of f[rs1] by the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:



FDIVx

Divide Extended

FDIVx

(CY7C602 Instruction Only)

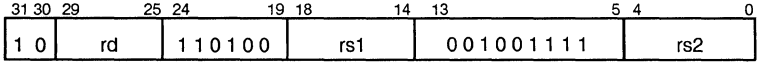
Operation: f[rd]x ← f[rs1]x / f[rs2]x

Assembler Syntax: fdivx *freg$_{rs1}$, freg$_{rs2}$, freg$_{rd}$*

Description: The FDIVx instruction divides the contents of f[rs1] CONCAT f[rs1 + 1] CONCAT f[rs1 + 2] by the contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd + 1], and f[rd + 2].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:



6

FdTOi

Convert Double to Integer

(CY7C602 Instruction Only)

FdTOi

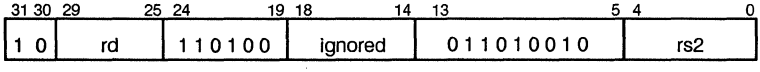
Operation: f[rd]i ← f[rs2]d

Assembler Syntax: fdtoi *freg$_{rs2}$, freg$_{rd}$*

Description: FdTOi converts the floating-point double contents of f[rs2] CONCAT f[rs2 + 1] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding direction field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



FdTOs

Convert Double to Single

(CY7C602 Instruction Only)

FdTOs

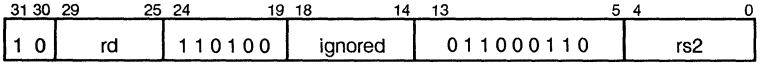
Operation: f[rd]s ← f[rs2]d

Assembler Syntax: fdtos *freg$_{rs2}$, freg$_{rd}$*

Description: FdTOs converts the floating-point double contents of f[rs2] CONCAT f[rs2 + 1] to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



6

FdTOx

Convert Double to Extended

(CY7C602 Instruction Only)

FdTOx

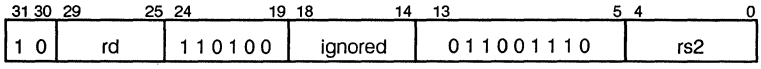
Operation: f[rd]x ← f[rs2]d

Assembler Syntax: fdtox *freg$_{rs2}$, freg$_{rd}$*

Description: FdTOx converts the floating-point double contents of f[rs2] CONCAT f[rs2 + 1] to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd + 1], and f[rd + 2]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:



FiTOd

Convert Integer to Double

(CY7C602 Instruction Only)

FiTOd

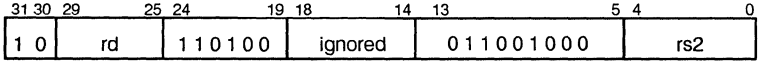
Operation: f[rd]d ← f[rs2]i

Assembler Syntax: fitod *freg$_{rs2}$*, *freg$_{rd}$*

Description: FiTOd converts the 32-bit, signed integer contents of f[rs2] to a floating-point, double-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd + 1].

Traps: fp_disabled
fp_exception*

Format:



6

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FiTOs

Convert Integer to Single

(CY7C602 Instruction Only)

FiTOs

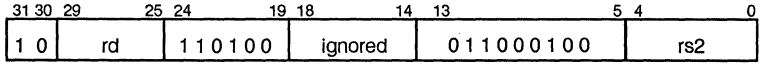
Operation: f[rd]s ← f[rs2]i

Assembler Syntax: fitos *freg$_{rs2}$*, *freg$_{rd}$*

Description: FiTOs converts the 32-bit, signed integer contents of f[rs2] to a floating-point, single-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field, *RD*.

Traps: fp_disabled
fp_exception (nx)

Format:



FiTOx

Convert Integer to Extended

(CY7C602 Instruction Only)

FiTOx

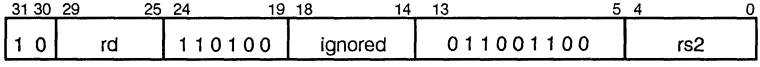
Operation: f[rd]x ← f[rs2]i

Assembler Syntax: fitox *freg$_{rs2}$*, *freg$_{rd}$*

Description: FiTOx converts the 32-bit, signed integer contents of f[rs2] to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], and f[rd+2].

Traps: fp_disabled
fp_exception*

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FMOVs — Move — FMOVs

(CY7C602 Instruction Only)

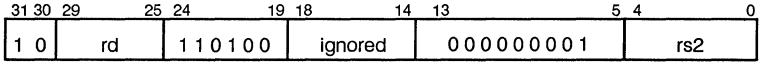
Operation: f[rd]s ← f[rs2]s

Assembler Syntax: fmovs *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FMOVs instruction moves the word content of register f[rs2] to the register f[rd]. Multiple FMOVs's are required to transfer multiple-precision numbers between *f* registers.

Traps: fp_disabled
fp_exception*

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FMULd

Multiply Double

FMULd

(CY7C602 Instruction Only)

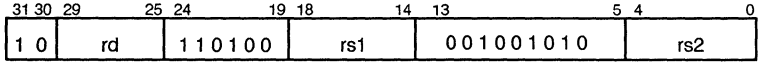
Operation: f[rd]d ← f[rs1]d x f[rs2]d

Assembler Syntax: fmuld *freg$_{rs1}$*, *freg$_{rs2}$*, *freg$_{rd}$*

Description: The FMULd instruction multiplies the contents of f[rs1] CONCAT f[rs1 + 1] by the contents of f[rs2] CONCAT f[rs2 + 1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd + 1].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



6

FMULs — Multiply Single — FMULs

(CY7C602 Instruction Only)

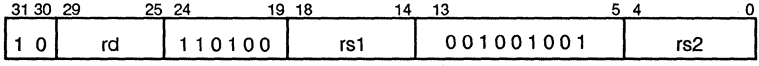
Operation: f[rd]s ← f[rs1]s x ([rs2]s

Assembler Syntax: fmuls *freg$_{rs1}$, freg$_{rs2}$, freg$_{rd}$*

Description: The FMULs instruction multiplies the contents of f[rs1] by the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



FMULx — Multiply Extended — FMULx

(CY7C602 Instruction Only)

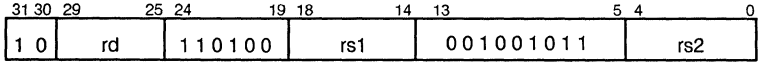
Operation: f[rd]x ← f[rs1]x x f[rs2]x

Assembler Syntax: fmulx *freg$_{rs1}$, freg$_{rs2}$, freg$_{rd}$*

Description: The FMULx instruction multiplies the contents of f[rs1] CONCAT f[rs1 + 1] CONCAT f[rs1 + 2] by the contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd + 1], and f[rd + 2].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



6

FNEGs — Negate — FNEGs

(CY7C602 Instruction Only)

Operation: f[rd]s ← f[rs2]s XOR 80000000 H

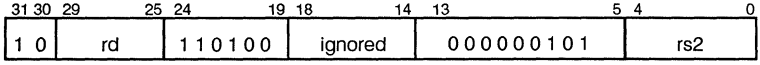
Assembler Syntax: fnegs *freg$_{rs2}$, freg$_{rd}$*

Description: The FNEGs instruction complements the sign bit of the word in f[rs2] and places the result in f[rd]. It does not round.

Since this FPop can address both even and odd *f* registers, FNEGs can also operate on the high-order words of double and extended operands, which accomplishes sign bit negation for these data types.

Traps: fp_disabled
fp_exception*

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FSQRTd

Square Root Double

FSQRTd

(CY7C602 Instruction Only)

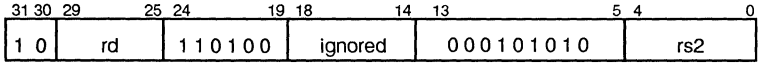
Operation: f[rd]d ← SQRT f[rs2]d

Assembler Syntax: fsqrtd *freg$_{rs2}$*, *freg$_{rd}$*

Description: FSQRTd generates the square root of the floating-point double contents of f[rs2] CONCAT f[rs2 + 1] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd + 1]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



6

FSQRTs — Square Root Single — FSQRTs

(CY7C602 Instruction Only)

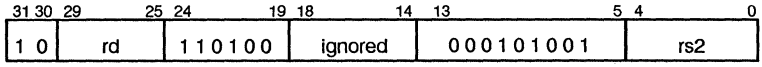
Operation: f[rd]s ← SQRT f[rs2]s

Assembler Syntax: fsqrts *freg$_{rs2}$, freg$_{rd}$*

Description: FSQRTs generates the square root of the floating-point single contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



FSQRTx — Square Root Extended — FSQRTx

(CY7C602 Instruction Only)

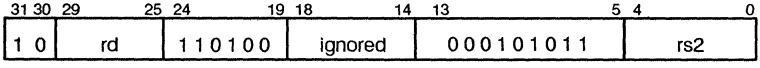
Operation: f[rd]x ← SQRT f[rs2]x

Assembler Syntax: fsqrtx *freg$_{rs2}$*, *freg$_{rd}$*

Description: FSQRTx generates the square root of the floating-point extended contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd + 1], and f[rd + 2]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



6

FsTOd

Convert Single to Double

(CY7C602 Instruction Only)

FsTOd

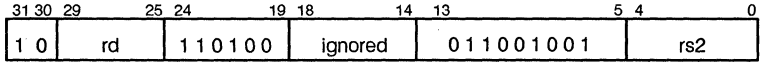
Operation: f[rd]d ← f[rs2]s

Assembler Syntax: fstod *freg$_{rs2}$*, *freg$_{rd}$*

Description: FsTOd converts the floating-point single contents of f[rs2] to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd + 1]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:



FsTOi

Convert Single to Integer

(CY7C602 Instruction Only)

FsTOi

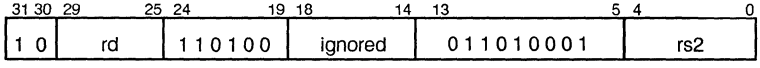
Operation: f[rd]i ← f[rs2]s

Assembler Syntax: fstoi *freg$_{rs2}$*, *freg$_{rd}$*

Description: FsTOi converts the floating-point single contents of f[rs2] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



6

FsTOx

Convert Single to Extended

(CY7C602 Instruction Only)

FsTOx

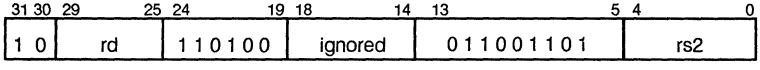
Operation: f[rd]x ← f[rs2]s

Assembler Syntax: fstox *freg$_{rs2}$*, *freg$_{rd}$*

Description: FsTOx converts the floating-point single contents of f[rs2] to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd + 1], and f[rd + 2]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:



FSUBd — Subtract Double — FSUBd

(CY7C602 Instruction Only)

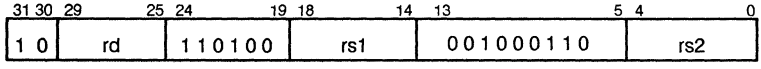
Operation: f[rd]d ← f[rs1]d - f[rs2]d

Assembler Syntax: fsubd *freg$_{rs1}$, freg$_{rs2}$, freg$_{rd}$*

Description: The FSUBd instruction subtracts the contents of f[rs2] CONCAT f[rs2 + 1] from the contents of f[rs1] CONCAT f[rs1 + 1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd + 1].

Traps: fp_disabled
fp_exception (of, uf, nx, nv)

Format:



6

FSUBs

Subtract Single

FSUBs

(CY7C602 Instruction Only)

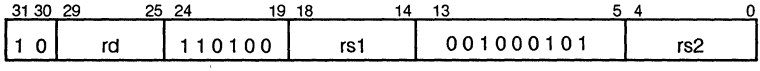
Operation: f[rd]s ← f[rs1]s – f[rs2]s

Assembler Syntax: fsubs *freg$_{rs1}$, freg$_{rs2}$, freg$_{rd}$*

Description: The FSUBs instruction subtracts the contents of f[rs2] from the contents of f[rs1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nx, nv)

Format:



FSUBx — Subtract Extended — FSUBx

(CY7C602 Instruction Only)

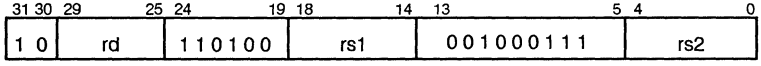
Operation: f[rd]x ← f[rs1]x − f[rs2]x

Assembler Syntax: fsubx *freg$_{rs1}$, freg$_{rs2}$, freg$_{rd}$*

Description: The FSUBx instruction subtracts the contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] from the contents of f[rs1] CONCAT f[rs1 + 1] CONCAT f[rs1 + 2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd + 1], and f[rd + 2].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



6

FxTOd

Convert Extended to Double

(CY7C602 Instruction Only)

FxTOd

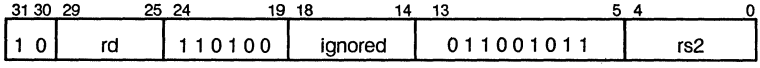
Operation: f[rd]d ← f[rs2]x

Assembler Syntax: fxtod *freg$_{rs2}$*, *freg$_{rd}$*

Description: FxTOd converts the floating-point extended contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd + 1]. Rounding is performed according to the rounding direction (*RD*) field of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



FxTOi

Convert Extended to Integer

(CY7C602 Instruction Only)

FxTOi

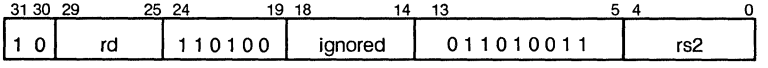
Operation: f[rd]i ← f[rs2]x

Assembler Syntax: fxtoi *freg$_{rs2}$*, *freg$_{rd}$*

Description: FxTOi converts the floating-point extended contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:



6

FxTOs — Convert Extended to Single — FxTOs

(CY7C602 Instruction Only)

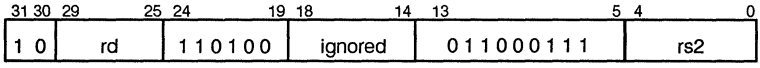
Operation: f[rd]s ← f[rs2]x

Assembler Syntax: fxtos *freg$_{rs2}$*, *freg$_{rd}$*

Description: FxTOs converts the floating-point extended contents of f[rs2] CONCAT f[rs2 + 1] CONCAT f[rs2 + 2] to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction (*RD*) field of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:



IFLUSH — Instruction Cache Flush — IFLUSH

Operation: FLUSH ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

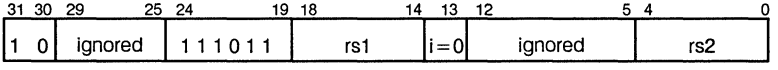
Assembler Syntax: iflush *address*

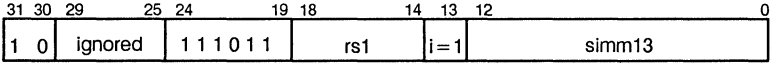
Description: The IFLUSH instruction causes a word to be flushed from an instruction cache which may be internal to the processor. The word to be flushed is at the address specified by the contents of r[rs1] plus either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

Since there is no internal instruction cache in the current CY7C600 family, the result of executing an IFLUSH instruction is dependent on the state of the input signal, Instruction Cache Flush Trap ($\overline{IFT}$). If $\overline{IFT}$ = 1, IFLUSH executes as a NOP, with no side effects. If $\overline{IFT}$ = 0, execution of IFLUSH causes an illegal_instruction trap.

Traps: illegal_instruction

Format:





6

JMPL — Jump and Link — JMPL

Operation: r[rd] ← PC
PC ← nPC
nPC ← r[rs1] + (r[rs2] or sign extnd(simm13))

Assembler Syntax: jmpl *address*, *reg$_{rd}$*

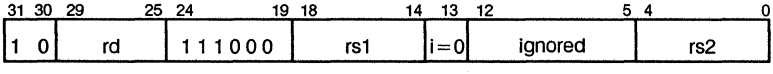
Description: JMPL first provides linkage by saving its return address into the register specified in the *rd* field. It then causes a register-indirect, delayed control transfer to an address specified by the sum of the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

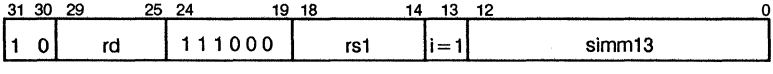
If either of the low-order two bits of the jump address is nonzero, a memory_address_not_aligned trap is generated.

Programming note: A register-indirect CALL can be constructed using a JMPL instruction with *rd* set to 15. JMPL can also be used to return from a CALL. In this case, *rd* is set to 0 and the return (jump) address would be equal to r[31] + 8.

Traps: memory_address_not_aligned

Format:





LD — Load Word — LD

Operation: r[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], *reg*$_{rd}$

Description: The LD instruction moves a word from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

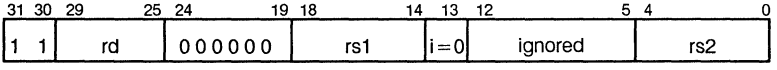
If LD takes a trap, the contents of the destination register remain unchanged.

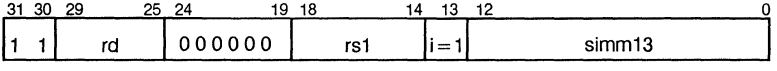
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:





6

LDA — Load Word from Alternate space — LDA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← [r[rs1] + r[rs2]]

Assembler Syntax: lda [*regaddr*] *asi*, *reg*$_{rd}$

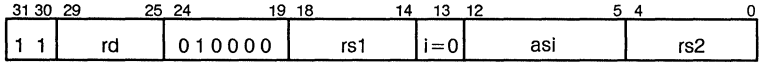
Description: The LDA instruction moves a word from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If LDA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



LDC — Load Coprocessor register — LDC

Operation: c[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], $creg_{rd}$

Description: The LDC instruction moves a word from memory into a coprocessor register, c[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

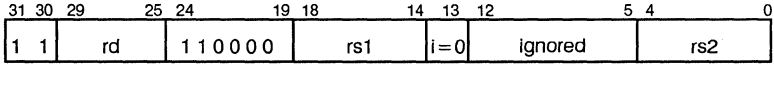
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDC takes a trap, the state of the coprocessor depends on the particular implementation.

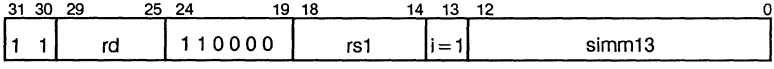
If the instruction following a coprocessor load uses the load's c[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





LDCSR — Load Coprocessor State Register — LDCSR

Operation: CSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], %csr

Description: The LDCSR instruction moves a word from memory into the Coprocessor State Register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

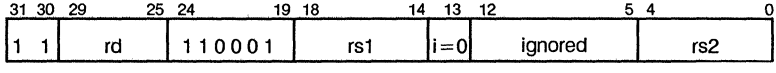
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDCSR takes a trap, the state of the coprocessor depends on the particular implementation.

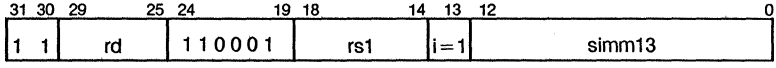
If the instruction following a LDCSR uses the CSR as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon implementation of the coprocessor.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





LDD — Load Doubleword — LDD

Operation: r[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
r[rd + 1] ← [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], *reg$_{rd}$*

Description: The LDD instruction moves a doubleword from memory into a destination register pair, r[rd] and r[rd + 1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section 2.2.5.1).

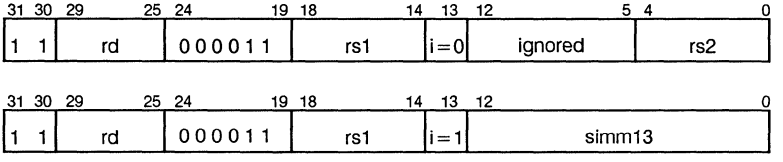
If a data_access_exception trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDD, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



LDDA — Load Doubleword from Alternate space — LDDA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← [r[rs1] + r[rs2]]
r[rd + 1] ← [r[rs1] + r[rs2] + 4]

Assembler Syntax: ldda [*regaddr*] *asi*, *reg*$_{rd}$

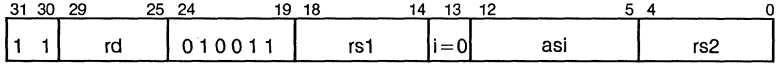
Description: The LDDA instruction moves a doubleword from memory into the destination registers, r[rd] and r[rd + 1]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section 2.2.5.1).

If a trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDA, this applies to both destination registers.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



LDDC — Load Doubleword Coprocessor — LDDC

Operation: c[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
c[rd + 1] ← [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], *creg*$_{rd}$

Description: The LDDC instruction moves a doubleword from memory into the coprocessor registers, c[rd] and c[rd + 1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section 2.2.5.1).

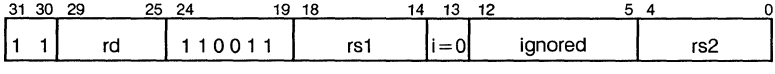
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDDC takes a trap, the state of the coprocessor depends on the particular implementation.

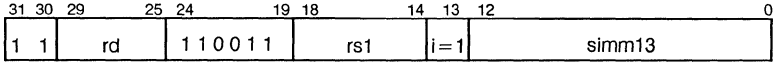
If the instruction following a coprocessor load uses the load's c[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem and coprocessor implementation. For an LDDC, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





LDDF — Load Doubleword Floating-Point — LDDF

Operation: f[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
f[rd + 1] ← [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], *freg*$_{rd}$

Description: The LDDF instruction moves a doubleword from memory into the floating-point registers, f[rd] and f[rd + 1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section 2.2.5.1).

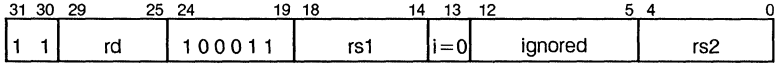
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place during the effective address memory access, the destination registers remain unchanged.

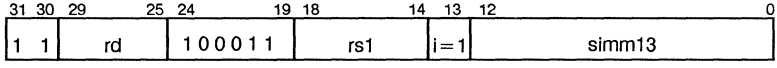
If the instruction following a floating-point load uses the load's f[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDF, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:





* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDF — Load Floating-Point register — LDF

Operation: f[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], *freg*$_{rd}$

Description: The LDF instruction moves a word from memory into a floating-point register, f[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

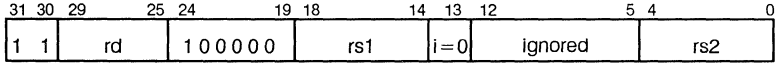
If the PSR's EF bit is set to zero or if no Floating-Point Unit is present, an fp_disabled trap will be generated. If LDF takes a trap, the contents of the destination register remain unchanged.

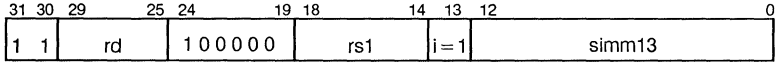
If the instruction following a floating-point load uses the load's f[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:





6

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDFSR — Load Floating-Point State Register — LDFSR

Operation: FSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], %fsr

Description: The LDFSR instruction moves a word from memory into the floating-point state register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. This instruction will wait for all pending FPops to complete execution before it loads the memory word into the FSR.

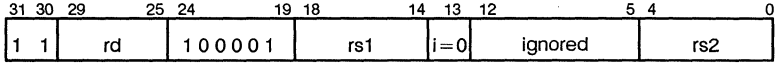
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If LDFSR takes a trap, the contents of the FSR remain unchanged.

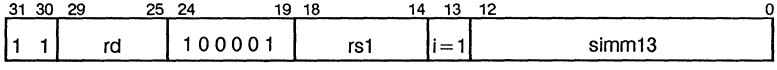
If the instruction following a LDFSR uses the FSR as a source operand, hardware interlocks add one or more cycle delay to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:





* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDSB — Load Signed Byte — LDSB

Operation: r[rd] ← sign extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldsb [*address*], *reg$_{rd}$*

Description: The LDSB instruction moves a signed byte from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and sign-extended in r[rd].

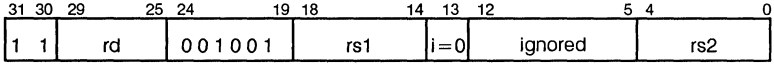
If LDSB takes a trap, the contents of the destination register remain unchanged.

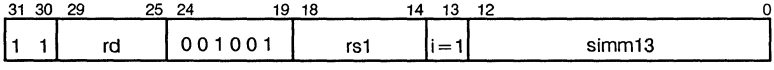
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:





6

LDSBA — Load Signed Byte from Alternate space — LDSBA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← sign extnd[r[rs1] + r[rs2]]

Assembler Syntax: ldsba [*regaddr*] *asi*, reg_{rd}

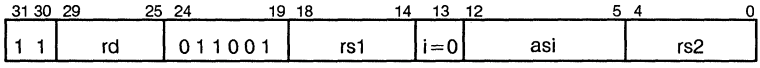
Description: The LDSBA instruction moves a signed byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and sign-extended in r[rd].

If LDSBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
data_access_exception

Format:



LDSH — Load Signed Halfword — LDSH

Operation: r[rd] ← sign extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldsh [*address*], *reg$_{rd}$*

Description: The LDSH instruction moves a signed halfword from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched halfword is right-justified and sign-extended in r[rd].

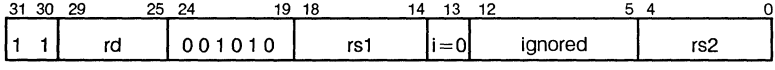
If LDSH takes a trap, the contents of the destination register remain unchanged.

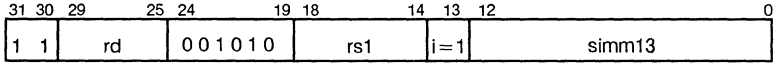
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:





6

LDSHA — Load Signed Halfword from Alternate space — LDSHA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← sign extnd[r[rs1] + r[rs2]]

Assembler Syntax: ldsha [reg_{addr}] *asi*, reg_{rd}

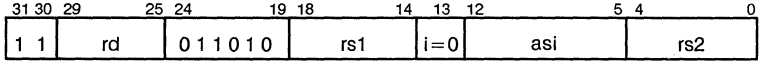
Description: The LDSHA instruction moves a signed halfword from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched halfword is right-justified and sign-extended in r[rd].

If LDSHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



LDSTUB — Atomic Load/Store Unsigned Byte — LDSTUB

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]
[r[rs1] + (r[rs2] or sign extnd(simm13))] ← FFFFFFFF H

Assembler Syntax: ldstub [*address*], *reg$_{rd}$*

Description: The LDSTUB instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic load/store instructions which address the same byte simultaneously are guaranteed to execute them serially, in some order.

The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and zero-extended in r[rd].

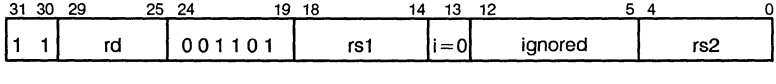
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

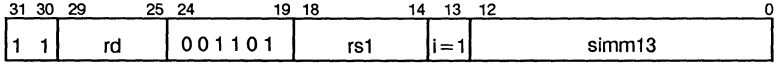
If LDSTUB takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:





LDSTUBA — Atomic Load/Store Unsigned Byte in Alternate space (Privileged Instruction) — LDSTUBA

Operation:
address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]
[r[rs1] + r[rs2]] ← FFFFFFFF H

Assembler Syntax:
ldstuba [reg_{addr}] *asi*, reg_{rd}

Description:
The LDSTUBA instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic load/store instructions which address the same byte simultaneously are guaranteed to execute them in some serial order.

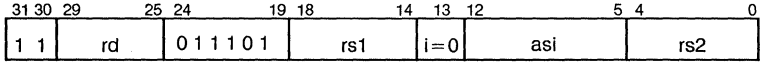
The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

If LDSTUBA takes a trap, the contents of the memory address remain unchanged.

Traps:
illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
data_access_exception

Format:



LDUB — Load Unsigned Byte — LDUB

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldub [*address*], *reg$_{rd}$*

Description: The LDUB instruction moves an unsigned byte from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and zero-extended in r[rd].

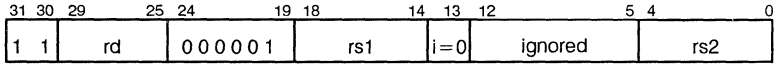
If LDUB takes a trap, the contents of the destination register remain unchanged.

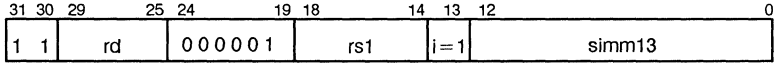
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:





6

LDUBA — Load Unsigned Byte from Alternate space — LDUBA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]

Assembler Syntax: lduba [reg_{addr}] *asi*, reg_{rd}

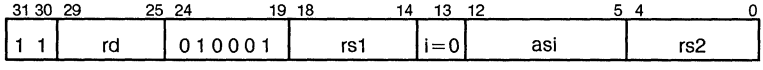
Description: The LDUBA instruction moves an unsigned byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

If LDUBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
data_access_exception

Format:



LDUH — Load Unsigned Halfword — LDUH

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: lduh [*address*], *reg*$_{rd}$

Description: The LDUH instruction moves an unsigned halfword from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched halfword is right-justified and zero-extended in r[rd].

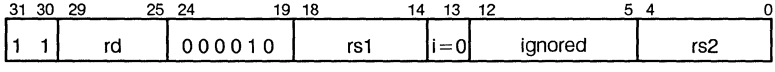
If LDUH takes a trap, the contents of the destination register remain unchanged.

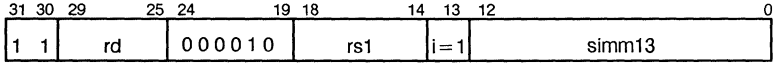
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:





6

LDUHA Load Unsigned Halfword from Alternate space LDUHA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]

Assembler Syntax: lduha [*regaddr*] *asi*, *reg$_{rd}$*

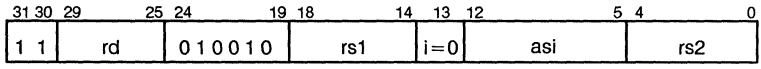
Description: The LDUHA instruction moves an unsigned halfword from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched halfword is right-justified and zero-extended in r[rd].

If LDUHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



MULScc — Multiply Step and modify icc — MULScc

Operation:

op1 = (n XOR v) CONCAT r[rs1]<31:1>
if (Y<0> = 0) op2 = 0, else op2 = r[rs2] or sign extnd(simm13)
r[rd] ← op1 + op2
Y ← r[rs1]<0> CONCAT Y<31:1>
n ← r[rd]<31>
z ← if [r[rd]]=0 then 1, else 0
v ← ((op1<31> AND op2<31> AND not r[rd]<31>)
OR (not op1<31> AND not op2<31> AND r[rd]<31>))
c ← ((op1<31> AND op2<31>)
OR (not r[rd] AND (op1<31> OR op2<31>))

Assembler Syntax:

mulscc *reg_{rs1}, reg_or_imm, reg_{rd}*

Description:

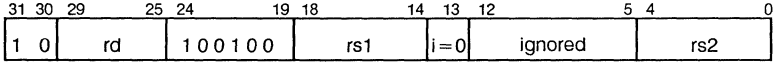
The multiply step instruction can be used to generate the 64-bit product of two signed or unsigned words. MULScc works as follows:

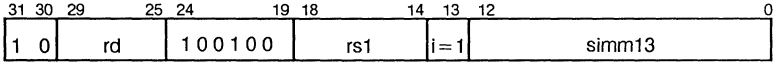
1. The "incoming partial product" in r[rs1] is shifted right by one bit and the high-order bit is replaced by the sign of the previous partial product (n XOR v). This is operand1.
2. If the least significant bit of the multiplier in the Y register equals zero, then operand2 is set to zero. If the LSB of the Y register equal one, then operand2 becomes the multiplicand, which is either the contents of r[rs2] if the instruction *i* field is zero, or sign extnd(simm13) if the *i* field is one. Operand2 is then added to operand1 and stored in r[rd] (the outgoing partial product).
3. The multiplier in the Y register is then shifted right by one bit and its high-order bit is replaced by the least significant bit of the incoming partial product in r[rs1].
4. The PSR's integer condition codes are updated according to the addition performed in step 2.

Traps:

none

Format:





6

OR — Inclusive-Or — OR

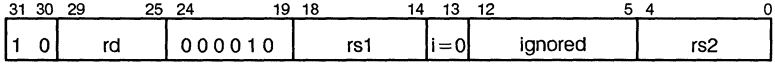
Operation: r[rd] ← r[rs1] OR (r[rs2] or sign extnd(simm13))

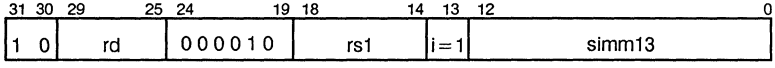
Assembler Syntax: or *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i = 1). The result is stored in register r[rd].

Traps: none

Format:





ORcc — Inclusive-Or and modify icc — ORcc

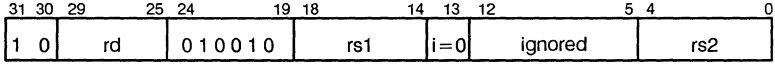
Operation: r[rd] ← r[rs1] OR (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if [r[rd]]=0 then 1, else 0
v ← 0
c ← 0

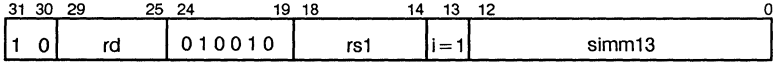
Assembler Syntax: orcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:





6

ORN — Inclusive-Or Not — ORN

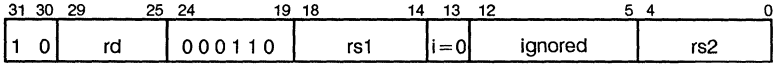
Operation: r[rd] ← r[rs1] OR not(operand2), where operand2 = (r[rs2] or sign extnd(simm13))

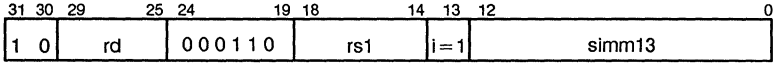
Assembler Syntax: orn *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i = 1). The result is stored in register r[rd].

Traps: none

Format:





ORNcc — Inclusive-Or Not and modify icc — ORNcc

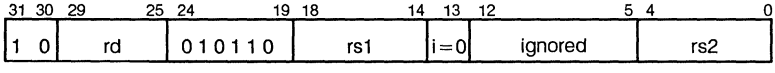
Operation: r[rd] ← r[rs1] OR not(operand2), where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if [r[rd]] = 0 then 1, else 0
v ← 0
c ← 0

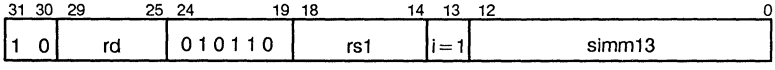
Assembler Syntax: orncc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i = 1). The result is stored in register r[rd]. ORNcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:





6

RDPSR — Read Processor State Register — RDPSR

(Privileged Instruction)

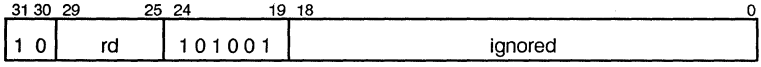
Operation: r[rd] ← PSR

Assembler Syntax: rd %psr, *reg$_{rd}$*

Description: RDPSR copies the contents of the PSR into the register specified by the *rd* field.

Traps: privileged-instruction (if S = 0)

Format:



RDTBR

Read Trap Base Register

(Privileged Instruction)

RDTBR

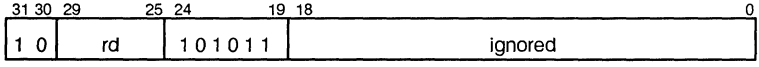
Operation: r[rd] ← TBR

Assembler Syntax: rd %tbr, *reg$_{rd}$*

Description: RDTBR copies the contents of the TBR into the register specified by the *rd* field.

Traps: privileged_instruction (if S = 0)

Format:



RDWIM — Read Window Invalid Mask register — RDWIM

(Privileged Instruction)

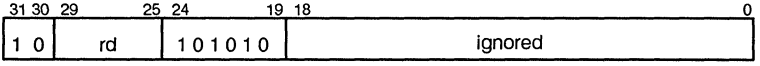
Operation: r[rd] ← WIM

Assembler Syntax: rd %wim, *reg$_{rd}$*

Description: RDWIM copies the contents of the WIM register into the register specified by the *rd* field.

Traps: privileged_instruction (if S = 0)

Format:



RDY — Read Y register — RDY

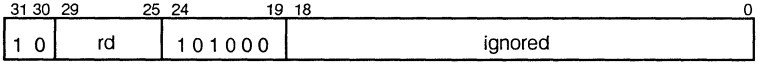
Operation: r[rd] ← Y

Assembler Syntax: rd %y, *reg$_{rd}$*

Description: RDY copies the contents of the Y register into the register specified by the *rd* field.

Traps: none

Format:



RESTORE — Restore caller's window — RESTORE

Operation:

ncwp ← CWP + 1
result ← r[rs1] + (r[rs2] or sign extnd(simm13))
CWP ← ncwp
r[rd] ← result
RESTORE does not affect condition codes

Assembler Syntax:

restore *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description:

RESTORE adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND $2^{ncwp} = 1$), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of r[rs1] is added to either the contents of r[rs2] (field bit $i = 1$) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit $i = 0$). Because the CWP has not been updated yet, r[rs1] and r[rs2] are read from the currently addressed window (the called window).

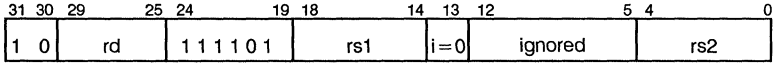
The new CWP value is written into the PSR, causing the previous window (the caller's window) to become the active window. The result of the addition is now written into the r[rd] register of the restored window.

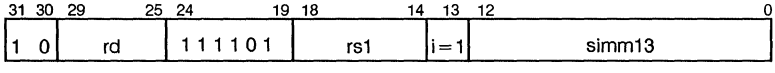
Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the CY7C601).

Traps:

window_underflow

Format:





RETT — Return from Trap — RETT

(Privileged Instruction)

Operation:

ncwp ← CWP + 1
ET ← 1
PC ← nPC
nPC ← r[rs1] + (r[rs2] or sign extnd(simm13))
CWP ← ncwp
S ← pS

Assembler Syntax:

rett *address*

Description:

RETT adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND $2^{ncwp} = 1$), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then RETT causes a delayed control transfer to the address derived by adding the contents of r[rs1] to either the contents of r[rs2] (field bit $i = 1$) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit $i = 0$).

Before the control transfer takes place, the new CWP value is written into the PSR, causing the previous window (the one in which the trap was taken) to become the active window. In addition, the PSR's ET bit is set to one (traps enabled) and the previous Supervisor bit (pS) is restored to the S field.

Although in theory RETT is a delayed control transfer instruction, in practice, RETT must always be immediately preceded by a JMPL instruction, creating a delayed control transfer couple (see Section 2.3.3.4.4). This has the effect of annulling the delay instruction.

If traps were already enabled before encountering the RETT instruction, an illegal_instruction trap is generated. If traps are not enabled (ET = 0) when the RETT is encountered, but (1) the processor is not in supervisor mode (S = 0), or (2) the window underflow condition described above occurs, or (3) if either of the two low-order bits of the target address are nonzero, then a reset trap occurs. If a reset trap does occur, the *tt* field of the TBR encodes the trap condition: privileged_instruction, window_underflow, or memory_address_not_aligned.

Programming note: To re-execute the trapping instruction when returning from a trap handler, use the following sequence:

```
jmpl    %17, %0          ! old PC
rett    %18              ! old nPC
```

Note that the CY7C601/611 saves the PC in r[17] (local 1) and the nPC in r[18] (local2) of the trap window upon entering a trap.

To return to the instruction after the trapping instruction (e.g., when the trapping instruction is emulated), use the sequence:

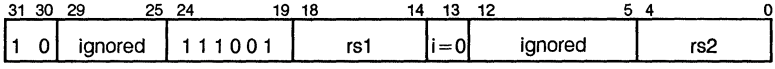
```
jmpl    %18,    %0       ! old nPC
rett    %18 + 4          ! old nPC + 4
```

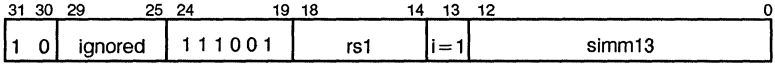
RETT — Return from Trap — RETT

(Privileged Instruction)

Traps: illegal_instruction
reset (privileged_instruction)
reset (memory_address_not_aligned)
reset (window_underflow)

Format:





SAVE — Save caller's window — SAVE

Operation:

ncwp ← CWP - 1
result ← r[rs1] + (r[rs2] or sign extnd(simm13))
CWP ← ncwp
r[rd] ← result
SAVE does not affect condition codes

Assembler Syntax:

save *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description:

SAVE subtracts one from the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND $2^{ncwp} = 1$), a window_overflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of r[rs1] is added to either the contents of r[rs2] (field bit $i = 1$) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit $i = 0$). Because the CWP has not been updated yet, r[rs1] and r[rs2] are read from the currently addressed window (the calling window).

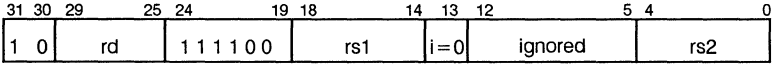
The new CWP value is written into the PSR, causing the active window to become the previous window, and the called window to become the active window. The result of the addition is now written into the r[rd] register of the new window.

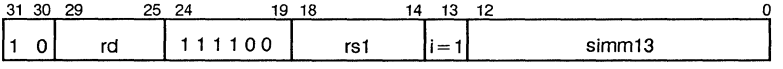
Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the CY7C601).

Traps:

window_overflow

Format:





6

SETHI — Set High 22 bits of *r register* — SETHI

Operation: r[rd]<31:10> ← imm22
r[rd]<9:0> ← 0

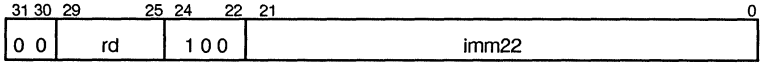
Assembler Syntax: sethi *const22, reg$_{rd}$*
sethi %hi *value, reg$_{rd}$*

Description: SETHI zeros the ten least significant bits of the contents of r[rd] and replaces its high-order 22 bits with *imm22*. The condition codes are not affected.

Programming note: SETHI 0, %0 is the preferred instruction to use as a NOP, because it will not increase execution time if it follows a load instruction.

Traps: none

Format:



SLL — Shift Left Logical — SLL

Operation: r[rd] ← r[rs1] SLL by (r[rs2] or shcnt)

Assembler Syntax: sll *reg_{rs1}, reg_or_imm, reg_{rd}*

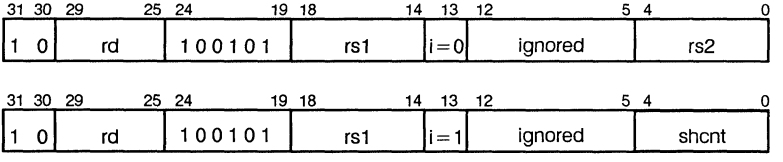
Description: SLL shifts the contents of r[rs1] left by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SLL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SLL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

Traps: none

Format:



6

SRA — Shift Right Arithmetic — SRA

Operation: r[rd] ← r[rs1] SRA by (r[rs2] or shcnt)

Assembler Syntax: sra *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: SRA shifts the contents of r[rs1] right by the number of bits specified by the shift count, filling the vacated positions with the MSB of r[rs1]. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

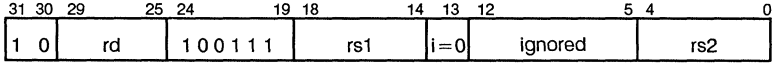
If the i bit field equals zero, the shift count for SRA is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SRA is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

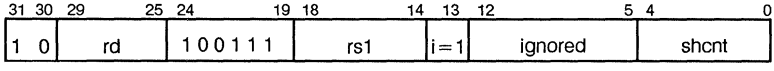
This instruction does *not* modify the condition codes.

Programming note: A "Shift Left Arithmetic by 1 (and calculate overflow)" can be implemented with an ADDcc instruction.

Traps: none

Format:





SRL — Shift Right Logical — SRL

Operation: r[rd] ← r[rs1] SRL by (r[rs2] or shcnt)

Assembler Syntax: srl *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

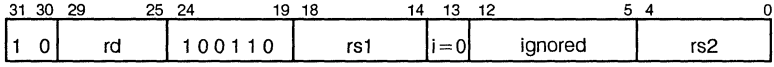
Description: SRL shifts the contents of r[rs1] right by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

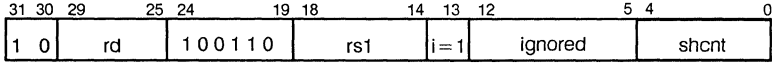
If the i bit field equals zero, the shift count for SRL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SRL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

Traps: none

Format:





6

ST — Store Word — ST

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

Assembler Syntax: st *reg$_{rd}$*, [*address*]

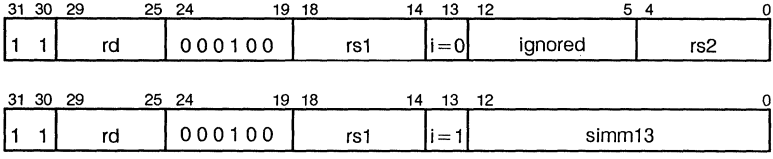
Description: The ST instruction moves a word from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If ST takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



STA — Store Word into Alternate space — STA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

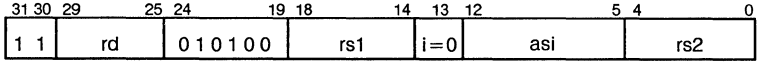
Assembler Syntax: sta *reg$_{rd}$*, [*regaddr*] *asi*

Description: The STA instruction moves a word from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



STB — Store Byte — STB

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

Assembler Syntax: stb *reg$_{rd}$*, [*address*]

synonyms: stub, stsb

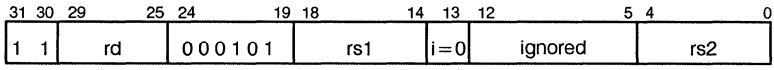
Description: The STB instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

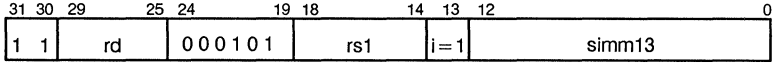
If STB takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: data_access_exception

Format:





STBA — Store Byte into Alternate space — STBA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

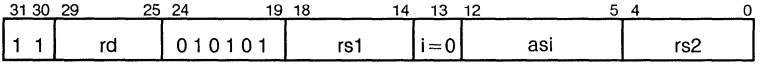
Assembler Syntax: stba *reg$_{rd}$*, [*regaddr*] *asi*
synonyms: stuba, stsba

Description: The STBA instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STBA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
data_access_exception

Format:



6

STC — Store Coprocessor register — STC

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← c[rd]

Assembler Syntax: st *creg$_{rd}$*, [*address*]

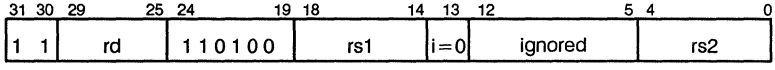
Description: The STC instruction moves a word from a coprocessor register, c[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

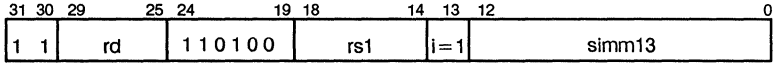
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If STC takes a trap, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





STCSR — Store Coprocessor State Register — STCSR

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← CSR

Assembler Syntax: st %csr, [*address*]

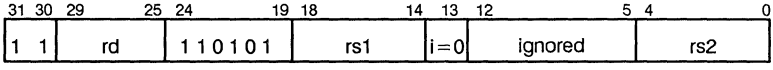
Description: The STCSR instruction moves the contents of the Coprocessor State Register into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

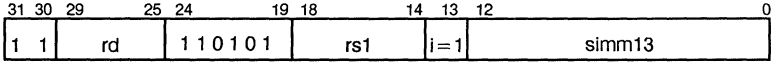
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If STCSR takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





6

STD — Store Doubleword — STD

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← r[rd + 1]

Assembler Syntax: std *reg$_{rd}$*, [*address*]

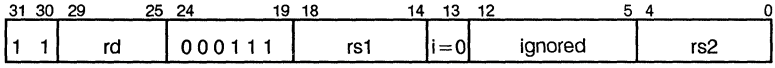
Description: The STD instruction moves a doubleword from the destination register pair, r[rd] and r[rd + 1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

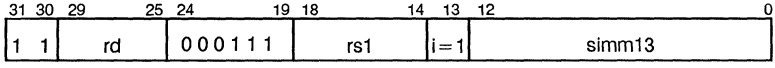
If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:





STDA — Store Doubleword into Alternate space — STDA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← r[rd + 1]

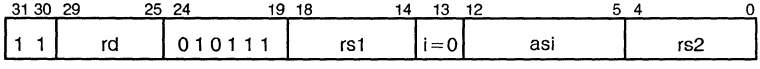
Assembler Syntax: stda *reg$_{rd}$*, [*regaddr*] *asi*

Description: The STDA instruction moves a doubleword from the destination register pair, r[rd] and r[rd + 1], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



STDC — Store Doubleword Coprocessor — STDC

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← c[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← c[rd + 1]

Assembler Syntax: std *creg$_{rd}$*, [*address*]

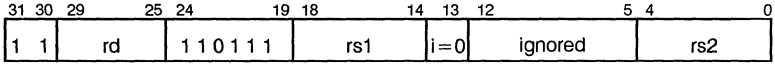
Description: The STDC instruction moves a doubleword from the coprocessor register pair, c[rd] and c[rd + 1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

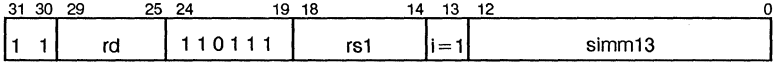
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





STDCQ — Store Doubleword Coprocessor Queue — STDCQ

(Privileged Instruction)

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← CQ.ADDR
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← CQ.INSTR

Assembler Syntax: std %cq, [*address*]

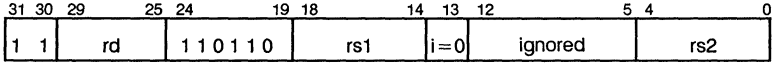
Description: The STDCQ instruction moves the front entry of the Coprocessor Queue into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4.

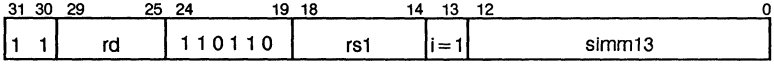
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:





STDF — Store Doubleword Floating-Point — STDF

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← f[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← f[rd + 1]

Assembler Syntax: std *freg$_{rd}$*, [*address*]

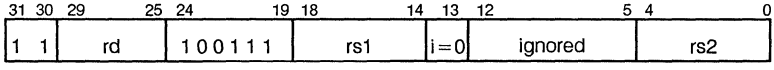
Description: The STDF instruction moves a doubleword from the floating-point register pair, f[rd] and f[rd + 1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

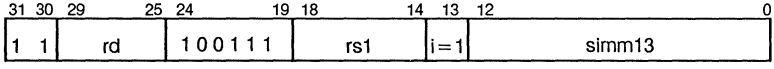
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:





* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STDFQ — Store Doubleword Floating-Point Queue — STDFQ

(Privileged Instruction)

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← FQ.ADDR
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← FQ.INSTR

Assembler Syntax: std %fq, [*address*]

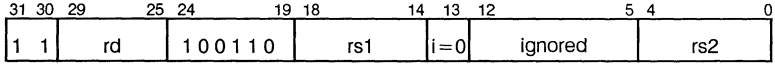
Description: The STDFQ instruction moves the front entry of the floating-point queue into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4. If the FPU is in exception mode, the queue is then advanced to the next entry, or it becomes empty (as indicated by the *qne* bit in the FSR).

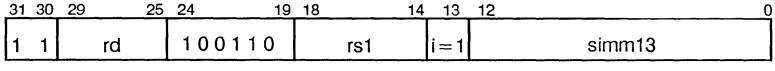
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: fp_disabled
fp_exception*
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:





* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STF — Store Floating-Point register — STF

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← f[rd]

Assembler Syntax: st *freg$_{rd}$*, [*address*]

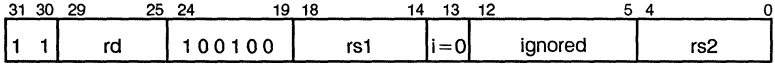
Description: The STF instruction moves a word from a floating-point register, f[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

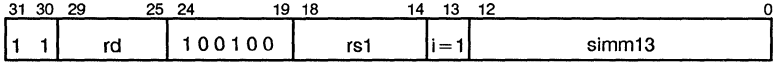
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STF takes a trap, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:





* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STFSR — Store Floating-Point State Register — STFSR

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← FSR

Assembler Syntax: st %fsr, [*address*]

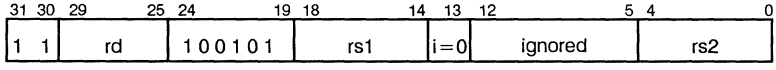
Description: The STFSR instruction moves the contents of the Floating-Point State Register into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. This instruction will wait for all pending FPops to complete execution before it writes the FSR into memory.

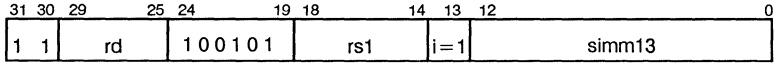
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STFSR takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:





6

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STH — Store Halfword — STH

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

Assembler Syntax: sth *reg$_{rd}$*, [*address*] synonyms: stuh, stsh

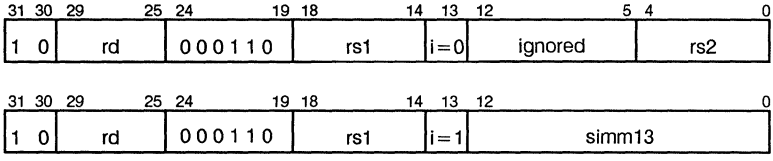
Description: The STH instruction moves the least significant halfword from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If STH takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



STHA

Store Halfword into Alternate space

(Privileged Instruction)

STHA

Operation: address space ← asi
[r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

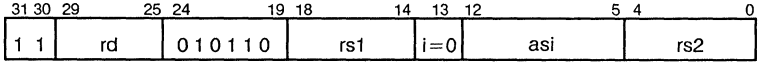
Assembler Syntax: stha *reg$_{rd}$*, [*address*]
synonyms: stuha, stsha

Description: The STHA instruction moves the least significant halfword from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STHA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



6

SUB — Subtract — SUB

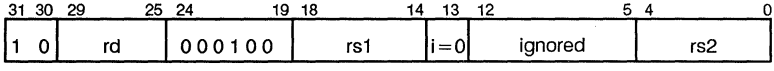
Operation: r[rd] ← r[rs1] - (r[rs2] or sign extnd(simm13))

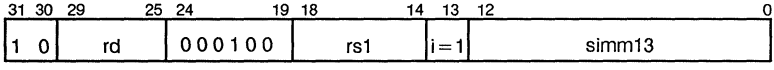
Assembler Syntax: sub *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: The SUB instruction subtracts either the contents of the register named in the *rs2* field, r[rs2], if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one, from register r[rs1]. The result is placed in the register specified in the *rd* field.

Traps: none

Format:





SUBcc — Subtract and modify icc — SUBcc

Operation:

r[rd] ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>)
c ← (not r[rs1]<31> AND operand2<31>)
OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))

Assembler Syntax:

subcc *regrs1, reg_or_imm, regrd*

Description:

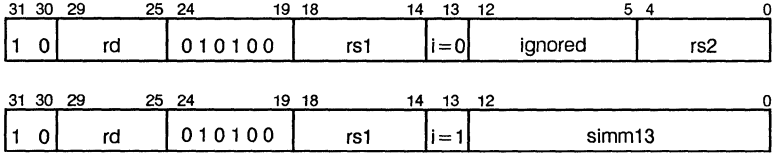
The SUBcc instruction subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. The result is placed in register r[rd]. In addition, SUBcc modifies all the integer condition codes in the manner described above.

Programming note: A SUBcc instruction with *rd* = 0 can be used for signed and unsigned integer comparison.

Traps:

none

Format:



6

SUBX — Subtract with Carry — SUBX

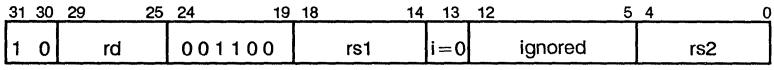
Operation: r[rd] ← r[rs1] - (r[rs2] or sign extnd(simm13)) - c

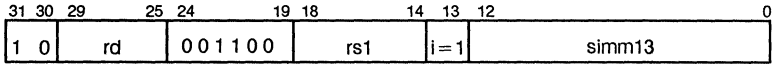
Assembler Syntax: subx *reg_{rs1}, reg_or_imm, reg_{rd}*

Description: SUBX subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. It then subtracts the PSR's carry bit (*c*) from that result. The final result is placed in the register specified in the *rd* field.

Traps: none

Format:





SUBXcc — Subtract with Carry and modify icc — SUBXcc

Operation:

r[rd] ← r[rs1] - operand2 - c, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>)
c ← (not r[rs1]<31> AND operand2<31>)
OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))

Assembler Syntax:

subxcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

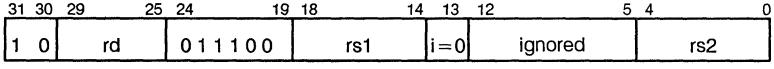
Description:

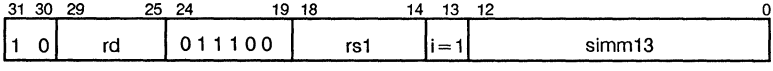
SUBXcc subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. It then subtracts the PSR's carry bit (*c*) from that result. The final result is placed in the register specified in the *rd* field. In addition, SUBXcc modifies all the integer condition codes in the manner described above.

Traps:

none

Format:





6

SWAP — Swap *r register* with memory — SWAP

Operation:
word ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
temp ← r[rd]
r[rd] ← word
r[rs1] + (r[rs2] or sign extnd(simm13)) ← temp

Assembler Syntax:
swap [*source*], reg_{rd}

Description:
SWAP atomically exchanges the contents of r[rd] with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAP instructions simultaneously are guaranteed to execute them serially, in some order.

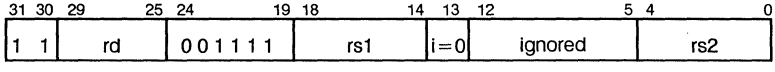
The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

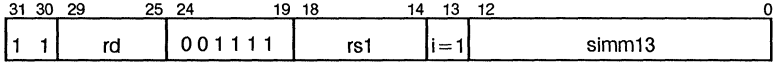
If SWAP takes a trap, the contents of the memory address and the destination register remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
memory_address_not_aligned
data_access_exception

Format:





SWAPA — Swap *r register* with memory in Alternate space — SWAPA

(Privileged Instruction)

Operation:
address space ← asi
word ← [r[rs1] + r[rs2]]
temp ← r[rd]
r[rd] ← word
[r[rs1] + r[rs2]] ← temp

Assembler Syntax:
swapa [*regsource*] *asi*, *reg$_{rd}$*

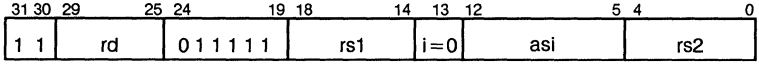
Description:
SWAPA atomically exchanges the contents of r[rd] with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAPA instructions simultaneously are guaranteed to execute them serially, in some order.

The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If SWAPA takes a trap, the contents of the memory address and the destination register remain unchanged.

Traps:
illegal_instruction (if i = 1)
privileged_instruction (if S = 0)
memory_address_not_aligned
data_access_exception

Format:



6

TADDcc — Tagged Add and modify icc — TADDcc

Operation:

r[rd] ← r[rs1] + operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
c ← (r[rs1]<31> AND operand2<31>
OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

Assembler Syntax:

taddcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

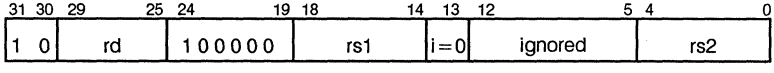
Description:

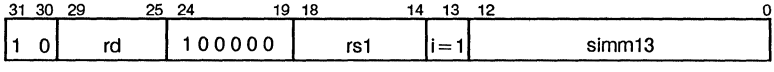
TADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. The result is placed in the register specified in the *rd* field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TADDcc modifies all the integer condition codes in the manner described above.

Traps:

none

Format:





TADDccTV Tagged Add (modify icc) Trap on Overflow TADDccTV

Operation:

result ← r[rs1] + operand2, where operand 2 = (r[rs2] or sign extnd(simm13))
tv ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
if tv = 1, then tag overflow trap; else
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← tv
c ← (r[rs1]<31> AND operand2<31>
OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))
r[rd] ← result

Assembler Syntax:

taddcctv *reg_{rs1}, reg_or_imm, reg_{rd}*

Description:

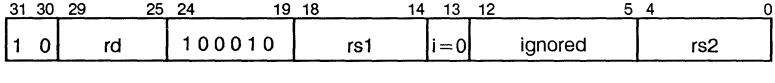
TADDccTV adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

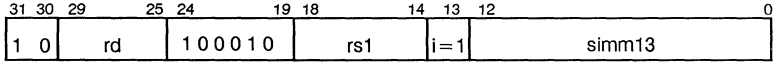
If TADDccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TADDccTV places the result in the register specified in the *rd* field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

Traps:

tag_overflow

Format:





Ticc — Trap on integer condition codes — Ticc

Operation: If condition true, then trap_instruction;
tt ← 128 + [r[rs1] + (r[rs2] or sign extnd(simm13))] < 6:0 >
else PC ← nPC
nPC ← nPC + 4

Assembler Syntax:

ta{,a}	*label*	
tn{,a}	*label*	
tne{,a}	*label*	synonym: tnz
te{,a}	*label*	synonym: tz
tg{,a}	*label*	
tle{,a}	*label*	
tge{,a}	*label*	
tl{,a}	*label*	
tgu{,a}	*label*	
tleu{,a}	*label*	
tcc{,a}	*label*	synonym: tgeu
tcs{,a}	*label*	synonym: tlu
tpos{,a}	*label*	
tneg{,a}	*label*	
tvc{,a}	*label*	
tvs{,a}	*label*	

Description: A Ticc instruction evaluates specific integer condition code combinations (from the PSR's *icc* field) based on the trap type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, and there are no higher-priority traps pending, then a trap_instruction trap is generated. If the condition codes evaluate as false, the trap is not generated.

If a trap_instruction trap is generated, the *tt* field of the Trap Base Register (TBR) is written with 128 plus the least significant seven bits of r[rs1] plus either r[rs2] (bit field $i = 0$) or the 13-bit sign-extended immediate value contained in the instruction (bit field $i = 1$). See Section 2.7 for the complete definition of a trap.

Traps: trap_instruction

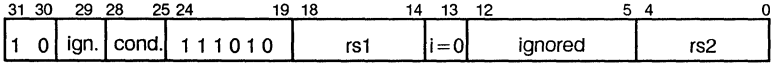
Ticc

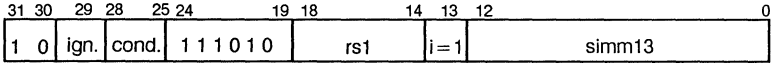
Trap on integer condition codes

Ticc

Mnemonic	Cond.	Operation	icc Test
TN	0000	Trap Never	No test
TE	0001	Trap on Equal	z
TLE	0010	Trap on Less or Equal	z OR (n XOR v)
TL	0011	Trap on Less	n XOR v
TLEU	0100	Trap on Less or Equal, Unsigned	c OR z
TCS	0101	Trap on Carry Set (Less then, Unsigned)	c
TNEG	0110	Trap on Negative	n
TVS	0111	Trap on oVerflow Set	v
TA	1000	Trap Always	No test
TNE	1001	Trap on Not Equal	not z
TG	1010	Trap on Greater	not(z OR (n XOR v))
TGE	1011	Trap on Greater or Equal	not(n XOR v)
TGU	1100	Trap on Greater, Unsigned	not(c OR z)
TCC	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	not c
TPOS	1110	Trap on Positive	not n
TVC	1111	Trap on oVerflow Clear	not v

Format:





ign. = ignored
cond. = condition

6

TSUBcc — Tagged Subtract and modify icc — TSUBcc

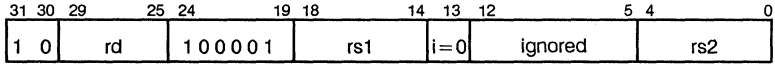
Operation: r[rd] ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>) OR (not r[rs1]<31>
AND operand2<31> AND r[rd]<31>) OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
c ← (not r[rs1]<31> AND operand2<31>
OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))

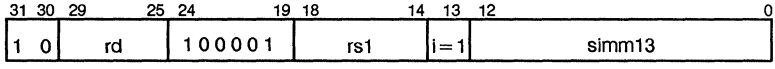
Assembler Syntax: tsubcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: TSUBcc subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. The result is placed in the register specified in the *rd* field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TSUBcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:





TSUBccTV Tagged Subtract (modify icc) TSUBccTV

Trap on Overflow

Operation:

result ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))
tv ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>) OR (not r[rs1]<31>
AND operand2<31> AND r[rd]<31>)
OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
if tv = 1, then tag overflow trap; else
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← tv
c ← (not(r[rs1]<31>) AND operand2<31> OR
(r[rd]<31> AND (not(r[rs1]<31>) OR operand2<31>))
r[rd] ← result

Assembler Syntax:

tsubcctv *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description:

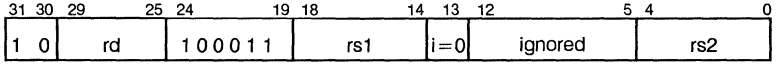
TSUBccTV subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

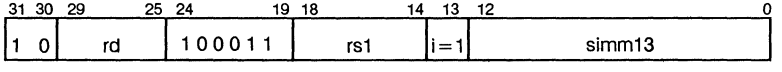
If TSUBccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TSUBccTV places the result in the register specified in the *rd* field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

Traps:

tag_overflow

Format:





UNIMP Unimplemented instruction UNIMP

Operation: illegal instruction trap

Assembler Syntax: unimp *const22*

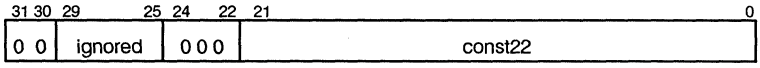
Description: Executing the UNIMP instruction causes an immediate illegal_instruction trap. The value in the const22 field is ignored.

Programming note: UNIMP can be used as part of the protocol for calling a function that is expected to return an aggregate value, such as a C-language structure.

1. An UNIMP instruction is placed after (not in) the delay slot after the CALL instruction in the calling function.
2. If the called function is expecting to return a structure, it will find the size of the structure that the caller expects to be returned as the const22 operand of the UNIMP instruction. The called function can check the opcode to make sure it is indeed UNIMP.
3. If the function is not going to return a structure, upon returning, it attempts to execute UNIMP rather than skipping over it as it should. This causes the program to terminate. The behavior adds some run-time checking to an interface that cannot be checked properly at compile time.

Traps: illegal_instruction

Format:



WRPSR — Write Processor State Register — WRPSR

(Privileged Instruction)

Operation: PSR ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg$_{rs1}$*, *reg_or_imm*, %psr

Description: WRPSR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i = 0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i = 1). The result is written into the writable subfields of the PSR. However, if the result's CWP field would point to an unimplemented window, an illegal_instruction trap is generated and the PSR remains unchanged.

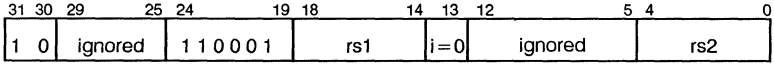
WRPSR is a delayed-write instruction:

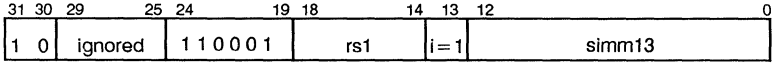
1. If any of the three instructions following a WRPSR uses any PSR field that WRPSR modified, the value of that field is unpredictable. Note that any instruction which references a non-global register makes use of the CWP, so following WRPSR with three NOPs would be the safest course.
2. If a WRPSR instruction is updating the PSR's Processor Interrupt Level (PIL) to a new value and is simultaneously setting Enable Traps (ET) to one, this could result in an interrupt trap at a level equal to the old PIL value.
3. If any of the three instructions after a WRPSR instruction reads the modified PSR, the value read is unpredictable.
4. If any of the three instructions after a WRPSR is trapped, a subsequent RDPSR in the trap handler will get the register's new value.

Programming note: Two WRPSR instructions should be used when enabling traps and changing the PIL value. The first WRPSR should specify ET = 0 with the new PIL value, and the second should specify ET = 1 with the new PIL value.

Traps: illegal_instruction
privileged_instruction (if S = 0)

Format:





6

WRTBR — Write Trap Base Register — WRTBR

(Privileged Instruction)

Operation: TBR ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg$_{rs1}$*, *reg_or_imm*, %tbr

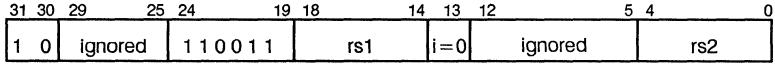
Description: WRTBR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i = 0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i = 1). The result is written into the Trap Base Address field of the TBR.

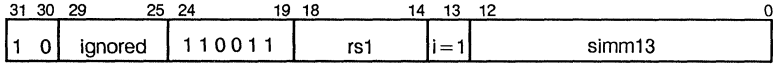
WRTBR is a delayed-write instruction:

1. If any of the three instructions following a WRTBR causes a trap, the TBA used may be either the old or the new value.
2. If any of the three instructions after a WRTBR is trapped, a subsequent RDTBR in the trap handler will get the register's new TBA value.

Traps: privileged_instruction (if S = 0)

Format:





WRWIM — Write Window Invalid Mask register — WRWIM

(Privileged Instruction)

Operation: WIM ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg$_{rs1}$, reg_or_imm*, %wim

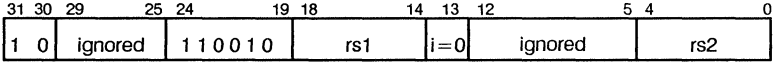
Description: WRWIM does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i = 1). The result is written into the writable bits of the WIM register.

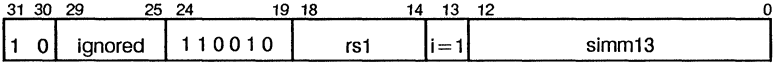
WRWIM is a delayed-write instruction:

1. If any of the three instructions following a WRWIM is a SAVE, RESTORE, or RETT, the occurrence of window_overflow and window_underflow is unpredictable.
2. If any of the three instructions after a WRWIM instruction reads the modified WIM, the value read is unpredictable.
3. If any of the three instructions after a WRWIM is trapped, a subsequent RDWIM in the trap handler will get the register's new value.

Traps: privileged_instruction (if S = 0)

Format:





6

WRY — Write Y register — WRY

Operation: Y ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg$_{rs1}$, reg_or_imm*, %y

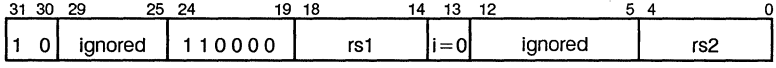
Description: WRY does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i = 1). The result is written into the Y register.

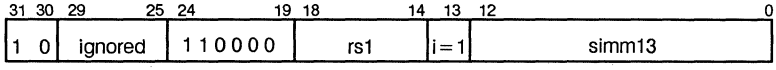
WRY is a delayed-write instruction:

1. If any of the three instructions following a WRY is a MULScc or a RDY, the value of Y used is unpredictable.
2. If any of the three instructions after a WRY instruction reads the modified Y register, the value read is unpredictable.
3. If any of the three instructions after a WRY is trapped, a subsequent RDY in the trap handler will get the register's new value.

Traps: none

Format:





XNOR — Exclusive-Nor — XNOR

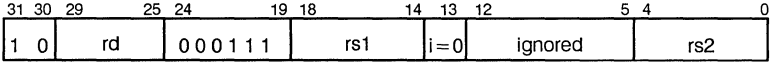
Operation: r[rd] ← r[rs1] XOR not(r[rs2] or sign extnd(simm13))

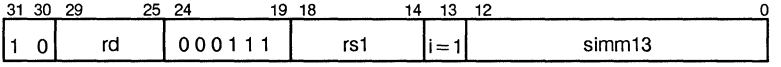
Assembler Syntax: xnor *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i = 0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i = 1). The result is stored in register r[rd].

Traps: none

Format:





6

XNORcc — Exclusive-Nor and modify icc — XNORcc

Operation:

r[rd] ← r[rs1] XOR not(r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax:

xnorcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

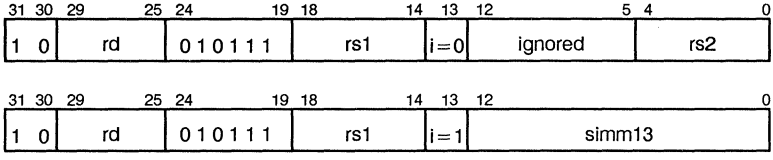
Description:

This instruction does a bitwise logical XOR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i = 0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i = 1). The result is stored in register r[rd]. XNORcc also modifies all the integer condition codes in the manner described above.

Traps:

none

Format:



XOR — Exclusive-Or — XOR

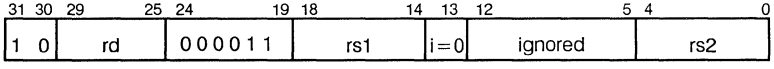
Operation: r[rd] ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

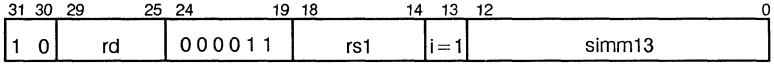
Assembler Syntax: xor *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





6

XORcc — Exclusive-Or and modify icc — XORcc

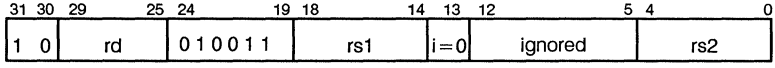
Operation: r[rd] ← r[rs1] XOR (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← 0
c ← 0

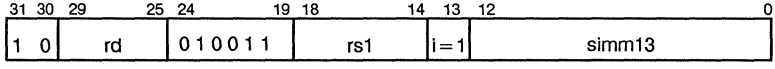
Assembler Syntax: xorcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. XORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:





CYPRESS SEMICONDUCTOR

Chapter 7

CY7C600 Electrical and Mechanical Characteristics

7.1 CY7C601 Electrical and Mechanical Characteristics

7.1.1 CY7C601 Maximum Ratings

Storage Temperature ..... −65° C to +150° C
Ambient Temperature with Power Applied ..... −55° C to +125° C
Supply Voltage to Ground Potential [1] ..... −0.5 V to +7.0 V
DC Voltage Applied to Outputs in High Z State ..... −0.5 V to +7.0 V
DC Input Voltage ..... −3.0 V to +7.0 V
Output Low Sink Current ..... 30 mA

7.1.2 CY7C601 Operating Range

Range	Ambient Temperature [2]	Vcc
Commercial	0° C to 70° C	5V ± 10%
Military	−55° C to +125° C	5V ± 10%

7.1.3 CY7C601 DC Characteristics Over the Operating Range

Parameters	Description	Test Conditions	Min.	Max.	Units
V_{OH}	Output HIGH Voltage	V_{CC} = Min., I_{OH} = −2.0 mA	2.4		V
V_{OL}	Output LOW Voltage	V_{CC} = Min., I_{OL} = 8.0 mA		0.5	V
V_{IH}	Input HIGH Voltage		2.1	V_{CC}	V
V_{IL}	Input LOW Voltage		−3.0	0.8	V
I_{IH}	Input HIGH Current	V_{CC} = Max., V_{IN} = V_{CC}		10	µA
I_{IL}	Input LOW Current	V_{CC} = Max., V_{IN} = V_{SS}		−10	µA
I_{OZ}	Output Leakage Current	V_{CC} = Max., $V_{SS} \leq V_{OUT} \leq V_{CC}$	−40	40	µA
I_{SC}	Output Short Circuit Current	V_{CC} = Max., V_{OUT} = 0V	−30	−180	mA
I_{CCQ}	Quiescent Supply Current	$V_{SS} \leq V_{IN} \leq V_{IL}$ or $V_{IH} \leq V_{IN} \leq V_{CC}$		150	mA
I_{CC}	Supply Current (All outputs loaded to 80 pF)	V_{CC} = Max., f = 40 MHz		675	mA
		V_{CC} = Max., f = 33 MHz		600	mA
		V_{CC} = Max., f = 25 MHz		600	mA
I_{CCF}	Supply Current (outputs floating)	V_{CC} = Max., f = 40 MHz		400	mA
		V_{CC} = Max., f = 33 MHz		350	mA
		V_{CC} = Max., f = 25 MHz		350	mA

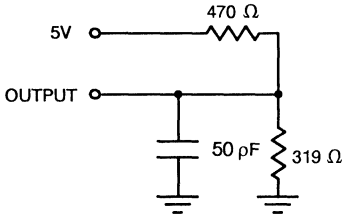
7.1.4 CY7C601 Capacitance [3]

Parameters	Description	Test Conditions	Max.	Units
C_{IN}	Input Capacitance	V_{CC} = 5.0V, T_a = 25°C, f = 1 MHz	10	pF
C_{OUT}	Output Capacitance	V_{CC} = 5.0V, T_a = 25°C, f = 1 MHz	12	pF
C_{IO}	I/O Bus Capacitance	V_{CC} = 5.0V, T_a = 25°C, f = 1 MHz	15	pF

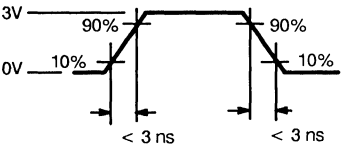
7.1.5 CY7C601 AC Characteristics [4]

Parameter		Description	Reference Edge	CY7C601-25		CY7C601-33		CY7C601-40		Units
				Min	Max.	Min.	Max.	Min	Max.	
1	t_{CY}	Clock Cycle		40	1000	30	1000	25	1000	ns
2	t_{CHL}	Clock High and Low		18	990	13	990	11	990	ns
3	t_{CRF}	Clock Rise and Fall		1		1		1		V/ns
4	t_{AD}	Address/Control Output Delay[5]	CLK+		33		24		20	ns
5	t_{AH}	Address/Control Output Valid[5]	CLK+	7		7		7		ns
6	t_{DOD}	D(31:0) Output Delay	CLK–		20		15		13	ns
7	t_{DOH}	D(31:0) Output Valid	CLK–	4		4		4		ns
8	t_{DIS}	D(31:0) Input Set-Up	CLK+	3		2		2		ns
9	t_{DIH}	D(31:0) Input Hold	CLK+	5		5		4		ns
10	t_{MAD}	MAO Asserted to Address/Control Output Delay	MAO+		19		14		12	ns
11	t_{MAH}	MAO Deasserted to Address/Control Output Valid	MAO–	2		2		2		ns
12	t_{MES}	$\overline{CEXC}$, $\overline{FEXC}$, $\overline{MEXC}$ Input Set-Up	CLK+	15		11		10		ns
13	t_{MEH}	$\overline{CEXC}$, $\overline{FEXC}$, $\overline{MEXC}$ Input Hold	CLK+	2		1		1		ns
14	t_{HS}	$\overline{XHOLD}$ Input Set-Up[6]	CLK–	7		4		3		ns
15	t_{HH}	$\overline{XHOLD}$ Input Hold	CLK–	6		5		4.5		ns
16	t_{HOD}	$\overline{XHOLD}$ to Address/Control Output Delay	$\overline{XHOLD}$–		22		15		12	ns
17	t_{HOH}	$\overline{XHOLD}$ to Address/Control Output Valid	$\overline{XHOLD}$+	0		0		0		ns
18	t_{OE}	$\overline{AOE}$, $\overline{COE}$, $\overline{DOE}$ to Output Enable Delay	$\overline{XOE}$–		15		11		9	ns
19	t_{OD}	$\overline{AOE}$, $\overline{COE}$, $\overline{DOE}$ to Output Disable Delay	$\overline{XOE}$+		15		11		9	ns
20	t_{TOE}	$\overline{TOE}$ Asserted to Output Enable Delay	$\overline{TOE}$–		21		19		17	ns
21	t_{TOD}	$\overline{TOE}$ Deasserted to Output Disable Delay	$\overline{TOE}$+		21		19		17	ns
22	t_{SSD}	INST, FXACK, CXACK, INTACK, $\overline{ERROR}$ Output Delay	CLK+		20		15		13	ns
23	t_{SSH}	INST, FXACK, CXACK, INTACK, $\overline{ERROR}$ Output Valid	CLK+	3		3		3		ns
24	t_{RS}	$\overline{RESET}$ Input Set-Up	CLK+	15		10		8		ns
25	t_{RH}	$\overline{RESET}$ Input Hold	CLK+	3		3		2		ns
26	t_{FD}	FINS(1:0), CINS(1:0) Output Delay	CLK+		27		18		15	ns
27	t_{FH}	FINS(1:0), CINS(1:0) Output Valid	CLK+	3.5		3.5		3.5		ns
28	t_{FIS}	FCC(1:0), CCC(1:0) Input Set-Up	CLK+	10		8		5		ns
29	t_{FIH}	FCC(1:0), CCC(1:0) Input Hold	CLK+	4		3		2		ns
30	t_{DXD}	DXFER Output Delay	CLK+		28		23		19	ns
31	t_{DXH}	DXFER Ouput Valid	CLK+	2		2		2		ns
32	t_{HDXD}	$\overline{XHOLD}$ Asserted to DXFER Output Delay[6]	$\overline{XHOLD}$–		20		15		12	ns
33	t_{HDXH}	$\overline{XHOLD}$ Deasserted to DXFER Output Valid	$\overline{XHOLD}$+	0		0		0		ns
34	t_{NUD}	INULL Output Delay	CLK+		20		13		11	ns
35	t_{NUH}	INULL Output Valid	CLK+	3		3		3		ns
36	t_{MDS}	$\overline{MDS}$ Input Set-Up	CLK–	5		4		3		ns
37	t_{MDH}	$\overline{MDS}$ Input Hold	CLK–	6		5		4.5		ns
38	t_{FLS}	FLUSH Output Delay	CLK+		15		13		11	ns
39	t_{FLH}	FLUSH Output Valid	CLK+	3		3		3		ns
40	t_{CCVS}	FCCV, CCCV Input Set-Up	CLK–	7		4		3		ns
41	t_{CCVH}	FCCV, CCCV Input Hold	CLK–	6		5		4.5		ns

7.1.6 CY7C601 AC Test Loads and Waveforms



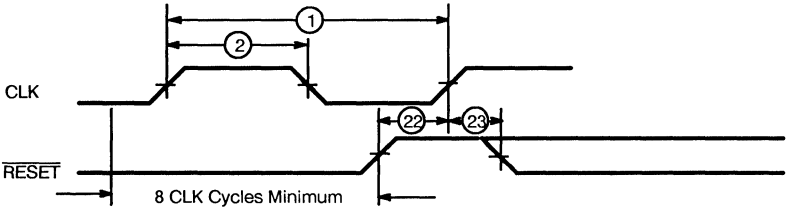
Test Load



Waveform

7.1.7 CY7C601 AC Waveforms

Clock and Reset Timing



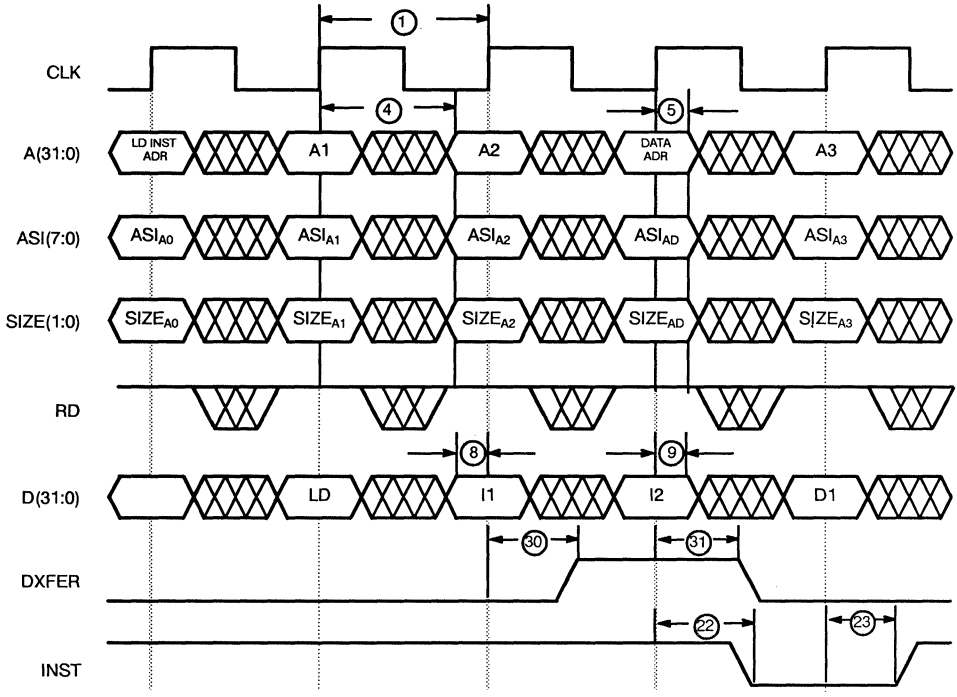
Reset need be synchronized with CLK only if the processor must be in step with other devices in the system.

Notes:

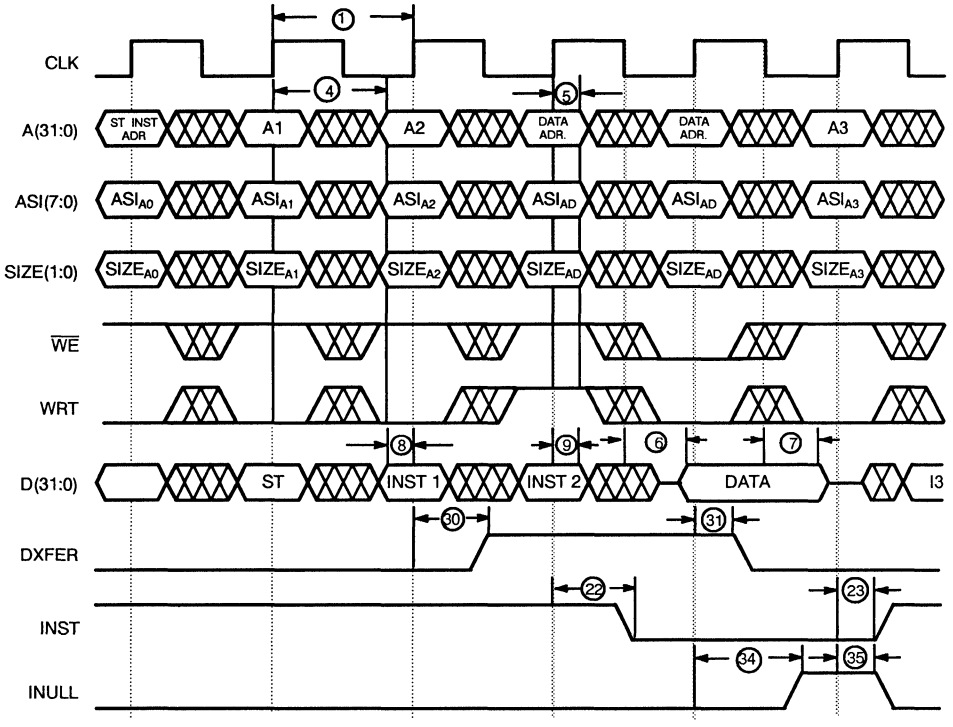
1. All power and ground pins must be connected before power is applied.
2. Ambient temperature is defined as the 'instant on' case temperature.
3. Tested initially and after any design or process changes that may affect these parameters.
4. Test conditions assume signal transition times of 3 ns or less, a timing reference level of 1.5V, input levels of 0 to 3.0V, and output loading of 50 pF.
5. Address/Control signals include: A(31:0), ASI(7:0), SIZE(1:0), RD, WRT, $\overline{WE}$, LOCK, and LDSTO.
6. $\overline{XHOLD}$ includes $\overline{BHOLD}$, $\overline{MHOLDA}$, $\overline{MHOLDB}$, $\overline{FHOLD}$, and $\overline{CHOLD}$.

7

Load Timing

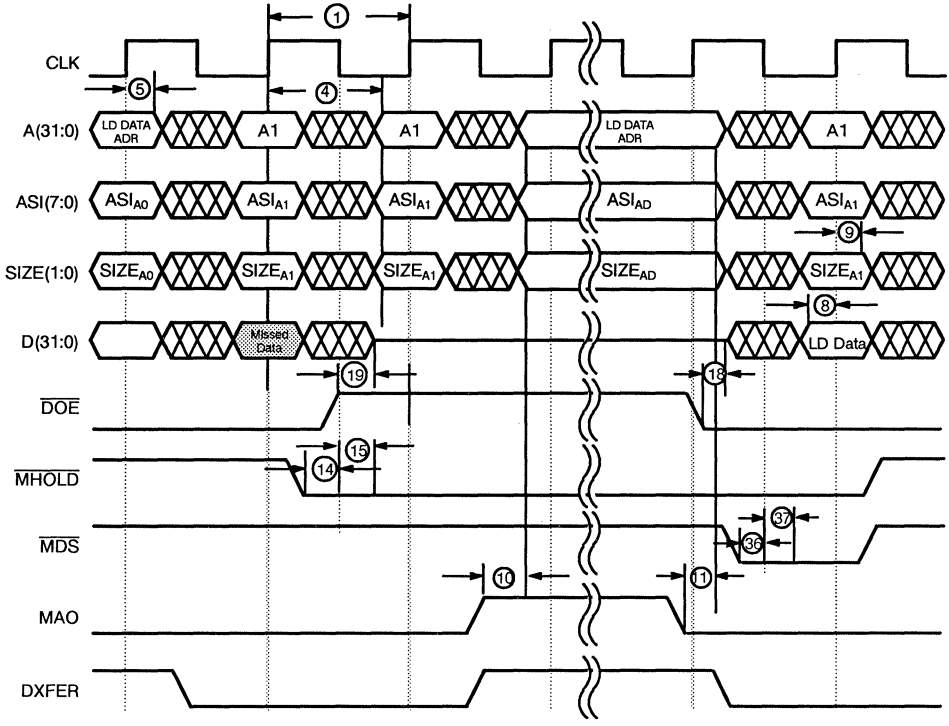

CLK
A(31:0)
ASI(7:0)
SIZE(1:0)
RD
D(31:0)
DXFER
INST
LD INST ADR
A1
A2
DATA ADR
A3
ASIA0
ASIA1
ASIA2
ASIAD
ASIA3
SIZEA0
SIZEA1
SIZEA2
SIZEAD
SIZEA3
LD
I1
I2
D1
1
4
5
8
9
30
31
22
23

Store Timing

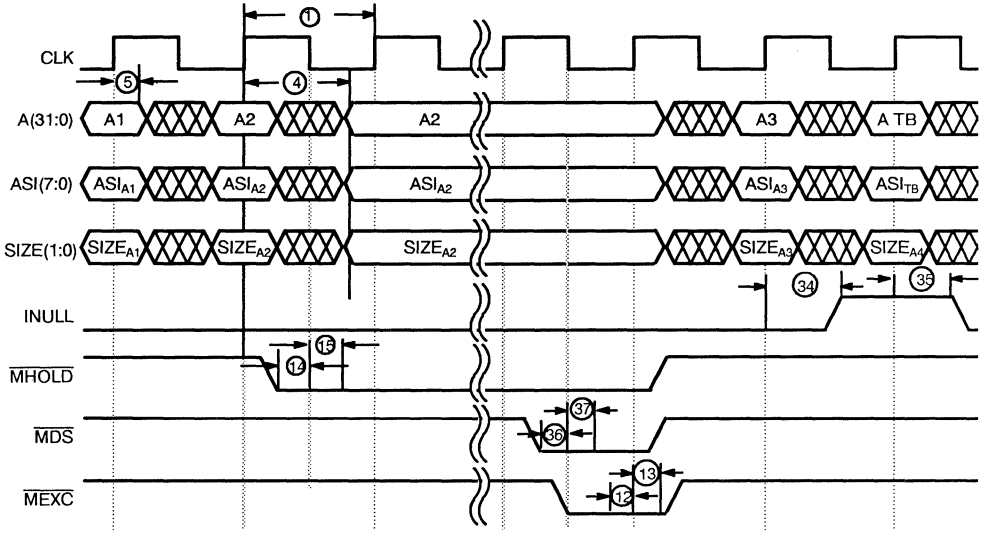


7

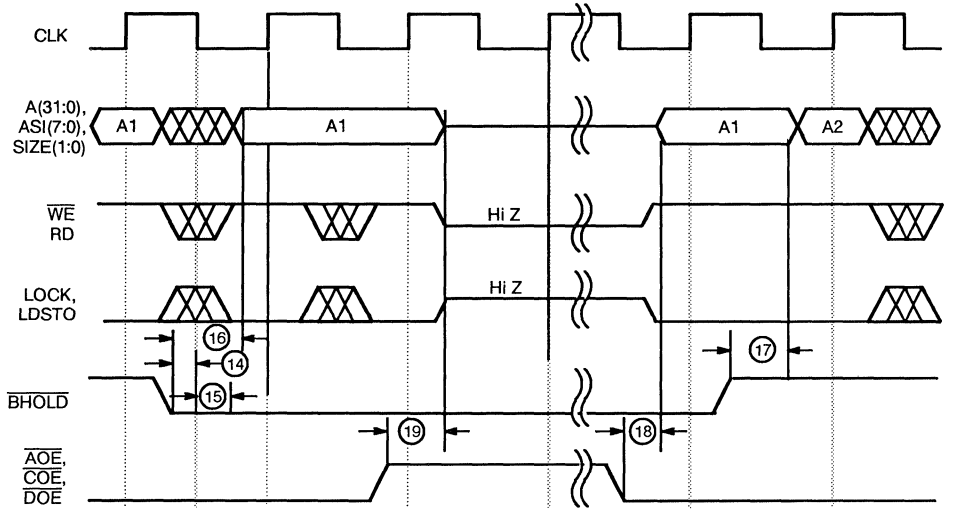
Load with Cache Miss



Memory Exception Timing

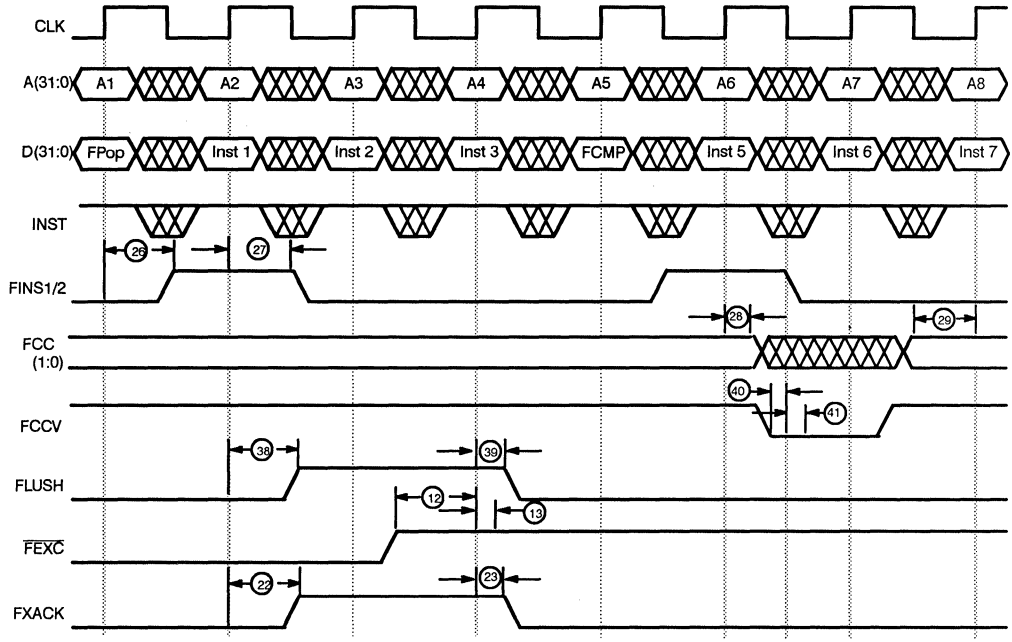


Bus Arbitration Timing

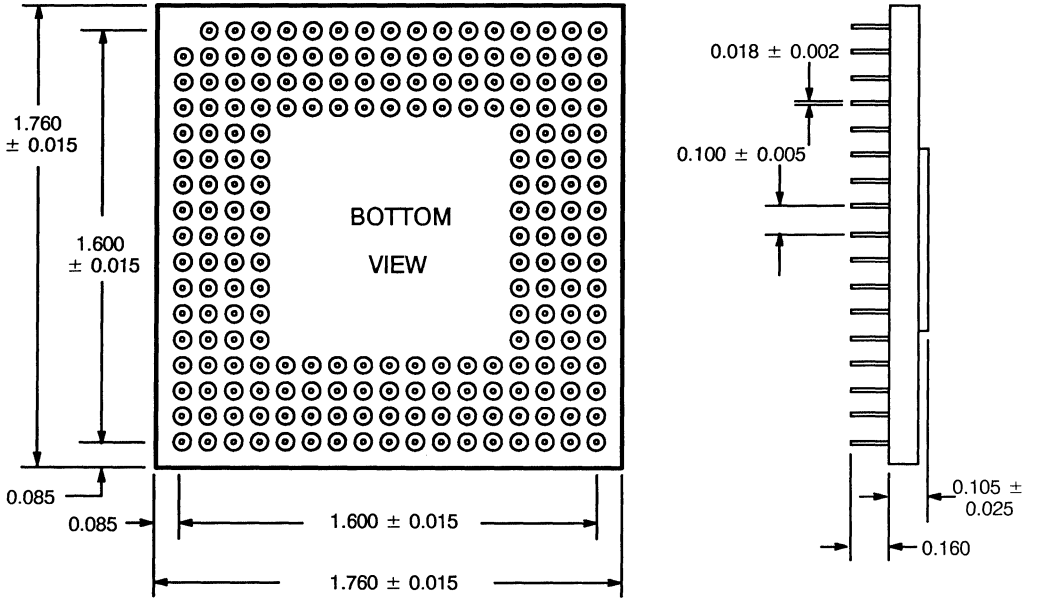


7

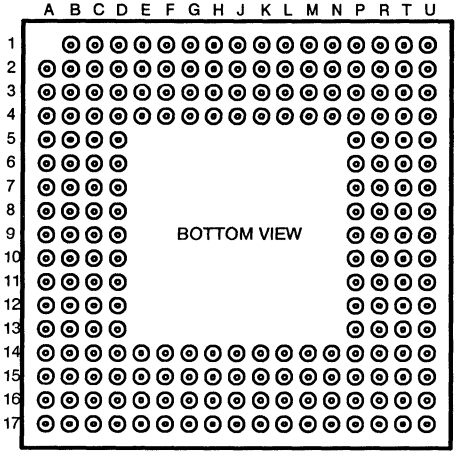
Floating-Point Timing



7.1.8 CY7C601 PGA Package Dimensions



7.1.9 CY7C601 PGA Pin Assignments

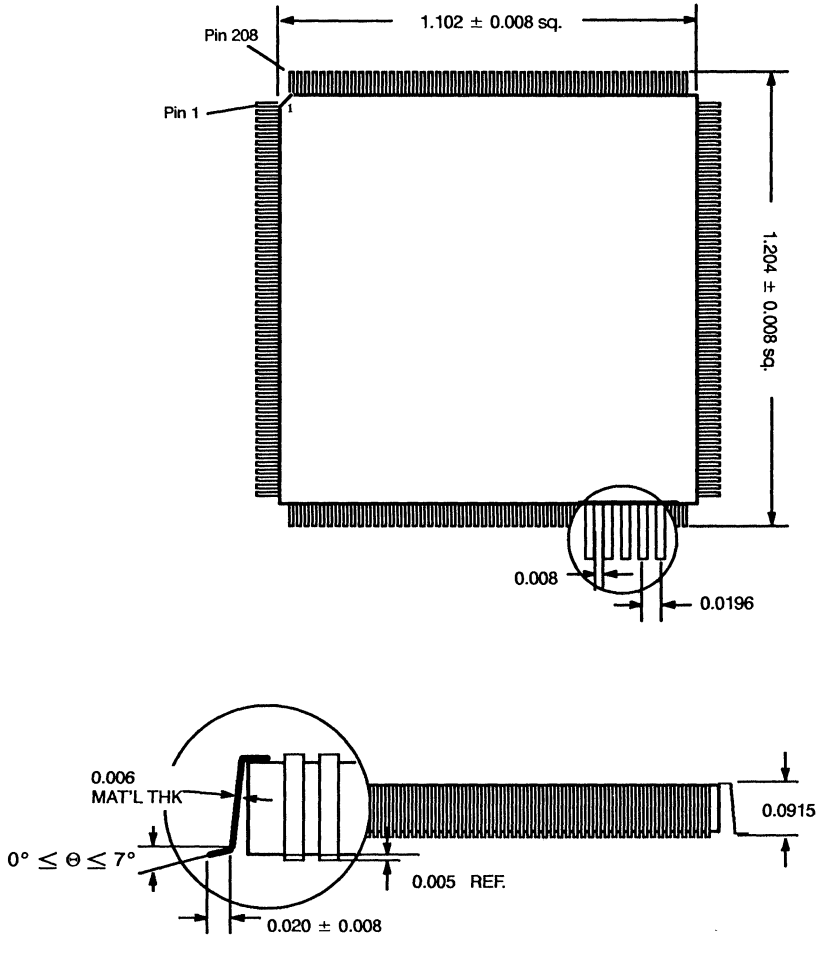


7

7.1.9 CY7C601 PGA Pin Assignments (continued)

Pin Name	Pin Number	Pin Name	Pin Number	Pin Name	Pin Number
A0	K2	ASI0	F3	VSSO	B16 H4 T16
A1	K1	ASI1	F2		B17 J2 T17
A2	L3	ASI2	G3		C3 K14 U16
A3	L1	ASI3	G2		C4 N14 U17
A4	L2	ASI4	G1		D6 P4
A5	M2	ASI5	H2		D14 P6
A6	N2	ASI6	H1		F1 P11
A7	M1	ASI7	J1		F4 P14
A8	M3	SIZE0	E2		F14 R5
A9	P1	SIZE1	D2		F17 R14
A10	P2	$\overline{\text{MEXC}}$	D8	VCCO	A15 L4
A11	N1	$\overline{\text{MHOLDA}}$	C8		A16 M14
A12	N3	$\overline{\text{MHOLDB}}$	B8		A17 N4
A13	R3	$\overline{\text{BHOLD}}$	A7		D1 P8
A14	R2	$\overline{\text{AOE}}$	P3		D12 P12
A15	R4	$\overline{\text{COE}}$	C2		D17 P16
A16	T4	$\overline{\text{DOE}}$	N17		E1 P17
A17	T5	$\overline{\text{MDS}}$	B7		G4 R16
A18	R6	$\overline{\text{MAO}}$	E3		K4 R17
A19	T6	$\overline{\text{IFT}}$	C14		K15
A20	U5	$\overline{\text{RD}}$	A4	VSSI	A3 J3 U2
A21	U6	$\overline{\text{WE}}$	B4		A14 L14 U10
A22	U7	LDSTO	C5		B2 M4
A23	T7	INULL	B5		B3 P5
A24	U8	LOCK	D4		B9 P7
A25	T8	DXFER	D3		C1 R1
A26	U9	WRT	E4		C16 R11
A27	R8	$\overline{\text{FP}}$	C7		D13 T1
A28	T9	FCC0	A11		E15 T15
A29	R9	FCC1	B11		H14 U1
A30	T10	$\overline{\text{FCCV}}$	C10	VCCI	A2 R12
A31	U11	$\overline{\text{FHOLD}}$	A8		B1 T2
D0	R10	$\overline{\text{FEXC}}$	A5		D7 T3
D1	T11	$\overline{\text{CP}}$	B6		E14 U3
D2	U12	CCC0	A12		E16 U4
D3	T12	CCC1	B13		G14
D4	U13	$\overline{\text{CCCV}}$	B10		H3
D5	T13	$\overline{\text{CHOLD}}$	C9		J15
D6	T14	$\overline{\text{CEXC}}$	A6		P10
D7	R13	INST	C6		R7
D8	U14	FLUSH	B14	VSST	D9 J4 J14
D9	U15	FINS1	E17		P9
D10	R15	FINS2	D16	VCCT	D5 P13
D11	P15	FXACK	D11		
D12	N15	CINS1	D15		
D13	M15	CINS2	C17		
D14	M16	CXACK	C13		
D15	N16	IRL0	A10		
D16	L15	IRL1	C11		
D17	M17	IRL2	D10		
D18	L16	IRL3	B12		
D19	L17	$\overline{\text{INTACK}}$	A13		
D20	K16	$\overline{\text{RESET}}$	A9		
D21	K17	$\overline{\text{ERROR}}$	B15		
D22	J16	$\overline{\text{TOE}}$	C15		
D23	J17	FPSYN	C12		
D24	H17	CLK	K3		
D25	H15				
D26	G17				
D27	H16				
D28	G16				
D29	F16				
D30	F15				
D31	G15				

7.1.10 CY7C601 QFP Package Dimensions



208-pin EIAJ standard QFP
All dimensions in inches

7

7.1.11 CY7C601 QFP Pin Assignments

Pin #	Function	Pin #	Function	Pin #	Function	Pin #	Function
1	VCCO	53	VSSO	105	VCCO	157	VSSO
2	VCCO	54	VSSO	106	VCCO	158	VSSO
3	VCCI	55	VSSO	107	VCCI	159	VSSO
4	LOCK	56	VCCO	108	VCCI	160	VSSI
5	COE	57	VSSI	109	D10	161	VCCO
6	WRT	58	A15	110	D11	162	IFT
7	DXFER	59	A16	111	VCCO	163	FLUSH
8	MAO	60	A17	112	D12	164	ERROR
9	SIZE1	61	VSSO	113	D13	165	INTACK
10	SIZE0	62	A18	114	VSSO	166	CXACK
11	VSSO	63	A19	115	D14	167	FXACK
12	ASI0	64	VCCI	116	D15	168	CCC1
13	ASI1	65	A20	117	VSSI	169	CCC0
14	VCCO	66	A21	118	DOE	170	FP SYN
15	ASI2	67	VSSI	119	D16	171	FCC1
16	VSSI	68	A22	120	D17	172	VSSI
17	ASI3	69	A23	121	VSSO	173	FCC0
18	VSSO	70	VCCO	122	D18	174	IRL3
19	ASI4	71	A24	123	D19	175	IRL2
20	VCCI	72	A25	124	VCCO	176	IRL1
21	ASI5	73	VCCO	125	D20	177	IRL0
22	VSSO	74	A26	126	D21	178	CCCV
23	ASI6	75	A27	127	VCCI	179	VCCI
24	ASI7	76	VSSO	128	D22	180	FCCV
25	VSST	77	A28	129	D23	181	VSST
26	CLK	78	A29	130	VSST	182	RESET
27	VSSI	79	VSSI	131	VSSI	183	VSSI
28	VSSI	80	VSSI	132	VSSI	184	VSSI
29	A0	81	VSST	133	D24	185	CHOLD
30	A1	82	A30	134	D25	186	FHOLD
31	VCCI	83	A31	135	VSSO	187	BHOLD
32	A2	84	VCCI	136	D26	188	MHOLDB
33	A3	85	D0	137	D27	189	MHOLDA
34	VCCO	86	D1	138	VCCO	190	MDS
35	A4	87	VCCO	139	D28	191	FP
36	A5	88	D2	140	D29	192	CEXC
37	VSSO	89	D3	141	VSSI	193	MEXC
38	A6	90	VSSO	142	D30	194	FEXC
39	A7	91	VSSI	143	D31	195	VSSI
40	VCCO	92	D4	144	VCCI	196	INST
41	A8	93	D5	145	VCCI	197	VCCI
42	A9	94	VCCI	146	VSSO	198	RD
43	VSSI	95	D6	147	FINS1	199	VSSO
44	A10	96	D7	148	FINS2	200	LDSTO
45	A11	97	VCCO	149	VSSI	201	CP
46	AOE	98	D8	150	CINS1	202	WE
47	A12	99	D9	151	CINS2	203	INULL
48	A13	100	VCCT	152	TOE	204	VCCT
49	A14	101	VSSI	153	VSSI	205	VSSI
50	VCCI	102	VSSO	154	VCCI	206	VSSO
51	VCCI	103	VSSO	155	VCCI	207	VSSO
52	VCCI	104	VSSO	156	VCCI	208	VSSO

7.1.12 CY7C601 Military Specifications—Group A Subgroup Testing

7.1.12.1 CY7C601 DC Characteristics

Parameter	Subgroups	Parameter	Subgroups
V_{OH}	1,2,3	I_{OH}	1,2,3
V_{OL}	1,2,3	I_{OL}	1,2,3
V_{IH}	1,2,3	I_{OZ}	1,2,3
V_{IL}	1,2,3	I_{SC}	1,2,3
I_{IH}	1,2,3	I_{CCQ}	1,2,3
I_{IL}	1,2,3	I_{CC}	1,2,3

7.1.12.2 CY7C601 AC Characteristics

Parameter		Subgroups	Parameter		Subgroups
1	t_{CY}	7,8,9,10,11	22	t_{SSD}	7,8,9,10,11
2	t_{CHL}	7,8,9,10,11	23	t_{SSH}	7,8,9,10,11
4	t_{AD}	7,8,9,10,11	24	t_{RS}	7,8,9,10,11
5	t_{AH}	7,8,9,10,11	26	t_{FD}	7,8,9,10,11
6	t_{DOD}	7,8,9,10,11	27	t_{FH}	7,8,9,10,11
7	t_{DOH}	7,8,9,10,11	28	t_{FIS}	7,8,9,10,11
8	t_{DIS}	7,8,9,10,11	29	t_{FIH}	7,8,9,10,11
9	t_{DIH}	7,8,9,10,11	30	t_{DXD}	7,8,9,10,11
10	t_{MAD}	7,8,9,10,11	31	t_{DXH}	7,8,9,10,11
11	t_{MAH}	7,8,9,10,11	32	t_{HDXD}	7,8,9,10,11
12	t_{MES}	7,8,9,10,11	33	t_{HDXH}	7,8,9,10,11
13	t_{MEH}	7,8,9,10,11	34	t_{NUD}	7,8,9,10,11
14	t_{HS}	7,8,9,10,11	35	t_{NUH}	7,8,9,10,11
15	t_{HH}	7,8,9,10,11	36	t_{MDS}	7,8,9,10,11
16	t_{HOD}	7,8,9,10,11	37	t_{MDH}	7,8,9,10,11
17	t_{HOH}	7,8,9,10,11	38	t_{FLS}	7,8,9,10,11
18	t_{OE}	7,8,9,10,11	39	t_{FLH}	7,8,9,10,11
19	t_{OD}	7,8,9,10,11	40	t_{CCVS}	7,8,9,10,11
20	t_{TOE}	7,8,9,10,11	41	t_{CCVH}	7,8,9,10,11
21	t_{TOD}	7,8,9,10,11			

7.2 CY7C611 Electrical and Mechanical Characteristics

7.2.1 CY7C611 Maximum Ratings

Storage Temperature ..... -65° C to +150° C
Ambient Temperature with Power Applied ..... -55° C to +125° C
Supply Voltage to Ground Potential [1] ..... -0.5 V to +7.0 V
DC Voltage Applied to Outputs in High Z State ..... -0.5 V to +7.0 V
DC Input Voltage ..... -3.0 V to +7.0 V
Output Low Sink Current ..... 30 mA

7.2.2 CY7C611 Operating Range

Range	Ambient Temperature [2]	Vcc
Commercial	0° C to 70° C	5V ± 10%
Military	-55° C to +125° C	5V ± 10%

7.2.3 CY7C611 DC Characteristics Over the Operating Range

Parameters	Description	Test Conditions	Min.	Max.	Units
V_{OH}	Output HIGH Voltage	V_{CC} = Min., I_{OH} = -2.0 mA	2.4		V
V_{OL}	Output LOW Voltage	V_{CC} = Min., I_{OL} = 8.0 mA		0.5	V
V_{IH}	Input HIGH Voltage		2.1	V_{CC}	V
V_{IL}	Input LOW Voltage		-3.0	0.8	V
I_{IH}	Input HIGH Current	V_{CC} = Max., V_{IN} = V_{CC}		10	µA
I_{IL}	Input LOW Current	V_{CC} = Max., V_{IN} = V_{SS}		-10	µA
I_{OZ}	Output Leakage Current	V_{CC} = Max., $V_{SS} \leq V_{OUT} \leq V_{CC}$	-40	40	µA
I_{SC}	Output Short Circuit Current	V_{CC} = Max., V_{OUT} = 0V	-30	-180	mA
I_{CCQ}	Quiescent Supply Current	$V_{SS} \leq V_{IN} \leq V_{IL}$ or $V_{IH} \leq V_{IN} \leq V_{CC}$		150	mA
I_{CC}	Supply Current (All outputs loaded to 80 pF)	V_{CC} = Max., f = 25 MHz		600	mA
I_{CCF}	Supply Current (outputs floating)	V_{CC} = Max., f = 25 MHz		350	mA

7.2.4 CY7C611 Capacitance [3]

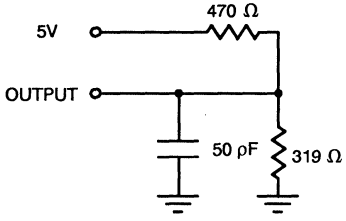
Parameters	Description	Test Conditions	Max.	Units
C_{IN}	Input Capacitance	V_{CC} = 5.0V, T_a = 25°C, f = 1 MHz	10	pF
C_{OUT}	Output Capacitance	V_{CC} = 5.0V, T_a = 25°C, f = 1 MHz	12	pF
C_{IO}	I/O Bus Capacitance	V_{CC} = 5.0V, T_a = 25°C, f = 1 MHz	15	pF

7.2.5 CY7C611 AC Characteristics [4]

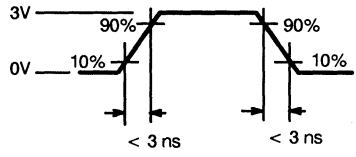
Parameter		Description	Reference Edge	CY7C611-25 Min.	CY7C611-25 Max.	Units
1	t_{CY}	Clock Cycle		40	1000	ns
2	t_{CHL}	Clock High and Low		18	990	ns
3	t_{CRF}	Clock Rise and Fall		1		V/ns
4	t_{AD}	Address/Control Output Delay[5]	CLK+		33	ns
5	t_{AH}	Address/Control Output Valid	CLK+	7		ns
6	t_{DOD}	D(31:0) Output Delay	CLK–		20	ns
7	t_{DOH}	D(31:0) Output Valid	CLK–	4		ns
8	t_{DIS}	D(31:0) Input Set-Up	CLK+	3		ns
9	t_{DIH}	D(31:0) Input Hold	CLK+	5		ns
10	t_{MAD}	MAO Asserted to Address/Control Output Delay	MAO+		19	ns
11	t_{MAH}	MAO Deasserted to Address/Control Output Valid	MAO–	2		ns
12	t_{MES}	$\overline{FEXC}$, $\overline{MEXC}$ Input Set-Up	CLK+	15		ns
13	t_{MEH}	$\overline{FEXC}$, $\overline{MEXC}$ Input Hold	CLK+	2		ns
14	t_{HS}	$\overline{XHOLD}$ Input Set-Up[6]	CLK–	7		ns
15	t_{HH}	$\overline{XHOLD}$ Input Hold	CLK–	6		ns
16	t_{HOD}	$\overline{XHOLD}$ to Address/Control Output Delay	$\overline{XHOLD}$–		22	ns
17	t_{HOH}	$\overline{XHOLD}$ to Address/Control Output Valid	$\overline{XHOLD}$+	0		ns
20	t_{TOE}	$\overline{TOE}$ Asserted to Output Enable Delay	$\overline{TOE}$–		21	ns
21	t_{TOD}	$\overline{TOE}$ Deasserted to Output Disable Delay	$\overline{TOE}$+		21	ns
22	t_{SSD}	INST, FXACK, INTACK, $\overline{ERROR}$ Output Delay	CLK+		20	ns
23	t_{SSH}	INST, FXACK, INTACK, $\overline{ERROR}$ Output Valid	CLK+	3		ns
24	t_{RS}	$\overline{RESET}$ Input Set-Up	CLK+	15		ns
25	t_{RH}	$\overline{RESET}$ Input Hold	CLK+	3		ns
26	t_{FD}	FINS(1:0) Output Delay	CLK+		27	ns
27	t_{FH}	FINS(1:0) Output Valid	CLK+	3.5		ns
28	t_{FIS}	FCC(1:0) Input Set-Up	CLK+	10		ns
29	t_{FIH}	FCC(1:0) Input Hold	CLK+	4		ns
34	t_{NUD}	INULL Output Delay	CLK+		20	ns
35	t_{NUH}	INULL Output Valid	CLK+	3		ns
36	t_{MDS}	$\overline{MDS}$ Input Set-Up	CLK–	5		ns
37	t_{MDH}	$\overline{MDS}$ Input Hold	CLK–	6		ns
38	t_{FLS}	FLUSH Output Delay	CLK+		15	ns
39	t_{FLH}	FLUSH Output Valid	CLK+	3		ns
40	t_{CCVS}	FCCV Input Set-Up	CLK–	7		ns
41	t_{CCVH}	FCCV Input Hold	CLK–	6		ns

Notes:
1. All power and ground pins must be connected before power is applied.
2. Ambient temperature is defined as the 'instant on' case temperature.
3. Tested initially and after any design or process changes that may affect these parameters.
4. Test conditions assume signal transition times of 3 ns or less, a timing reference level of 1.5V, input levels of 0 to 3.0V, and output loading of 50 pF.
5. Address/Control signals include: A(23:0), ASI(2:0), SIZE(1:0), RD, WRT, $\overline{WE}$, LOCK, and LDSTO.
6. $\overline{XHOLD}$ includes $\overline{BHOLD}$, $\overline{MHOLDA}$, $\overline{MHOLDB}$, and $\overline{FHOLD}$.

7.2.6 CY7C611 AC Test Loads and Waveforms



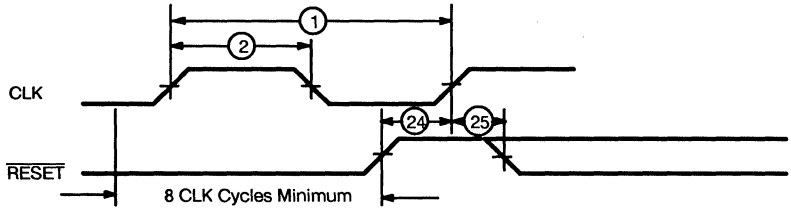
Test Load



Waveform

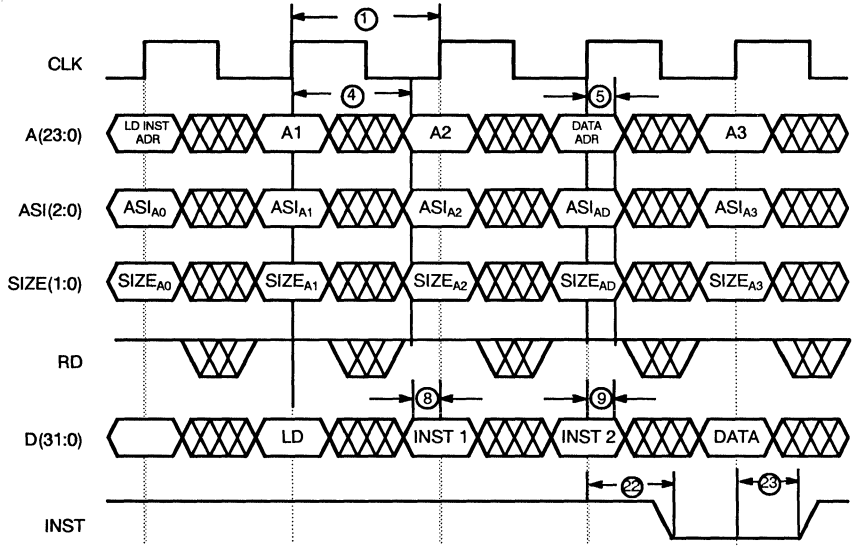
7.2.7 CY7C611 AC Waveforms

Clock and Reset Timing

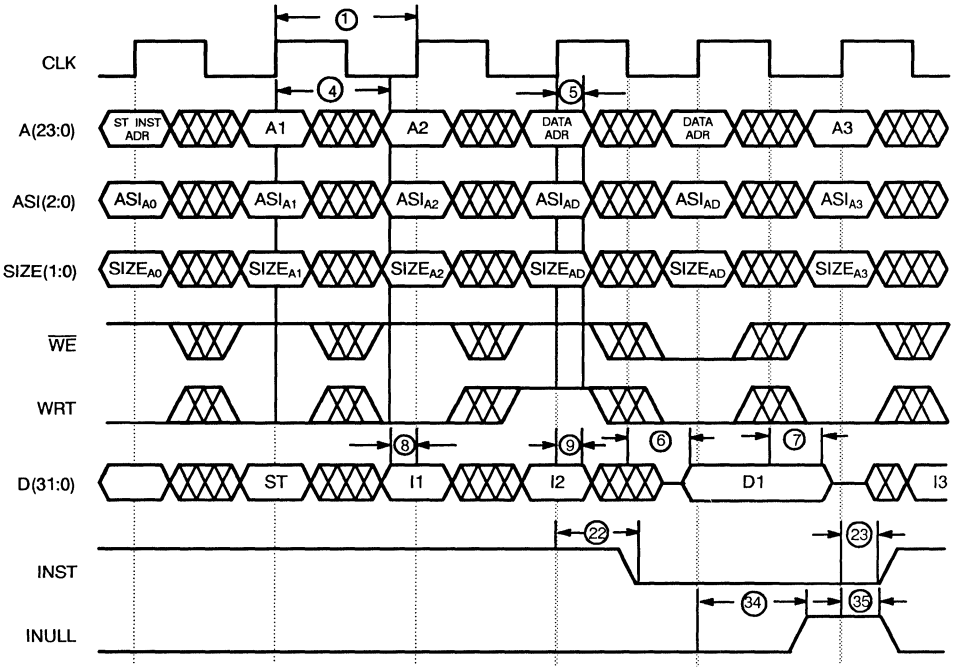


Reset needs to be synchronized with CLK only if the processor must be in step with other devices in the system.

Load Timing

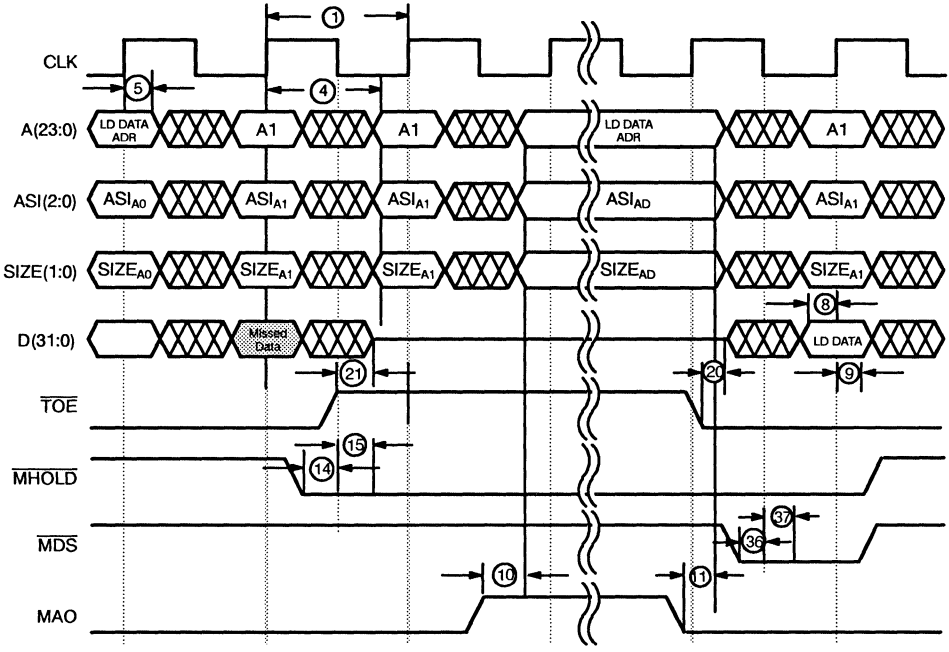


Store Timing

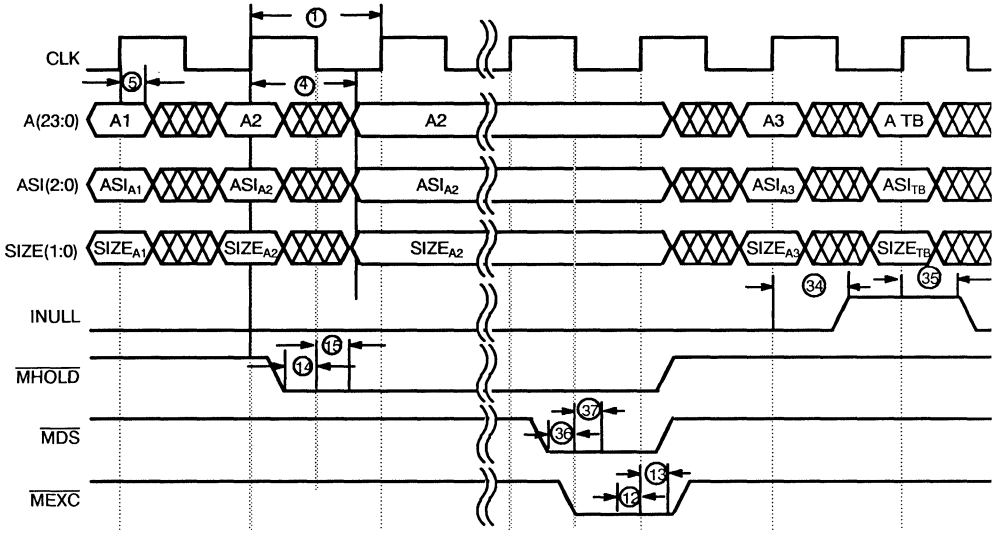


7

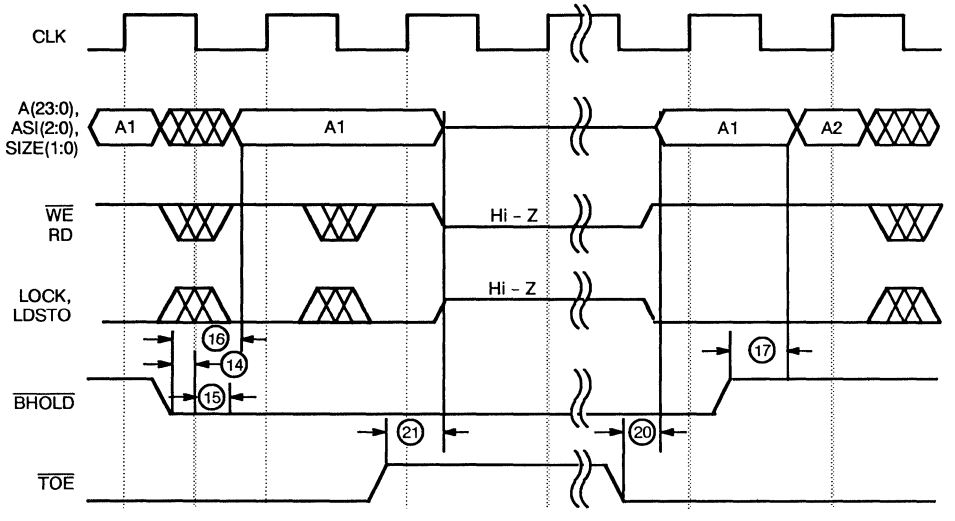
Load with Cache Miss



Memory Exception Timing

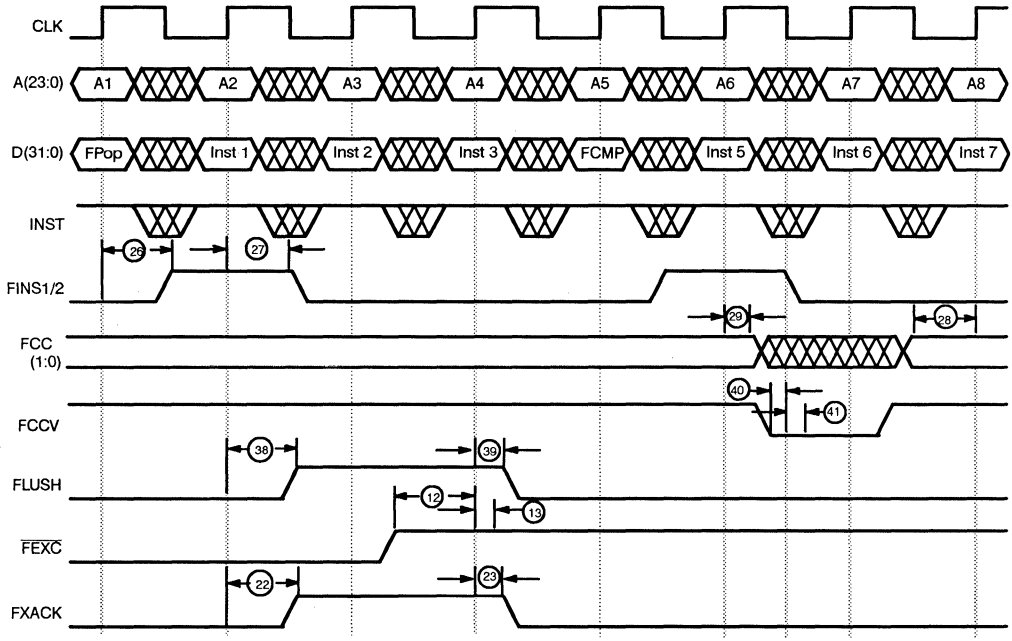


Bus Arbitration Timing

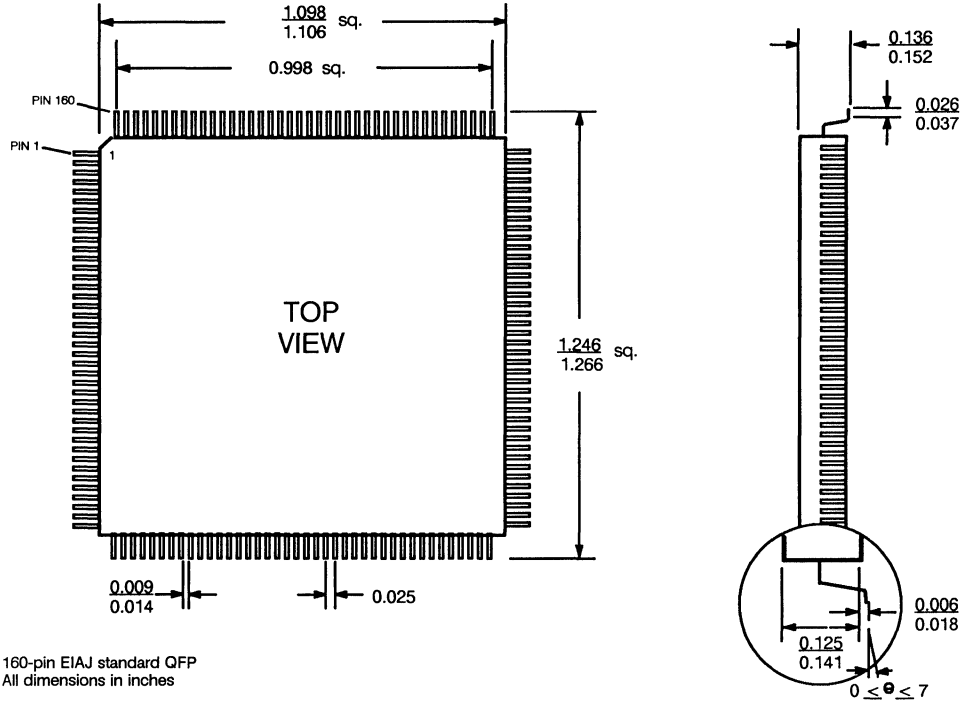


7

Floating-Point Timing



7.2.8 CY7C611 PQFP Package Dimensions



160-pin EIAJ standard QFP
All dimensions in inches

160–Pin Quad Flat Package

7

7.2.9 CY7C611 PQFP Pin Assignments

Function	Pin #	Function	Pin #	Function	Pin #	Function	Pin #
VCCO	1	VSSO	41	VCCO	81	VSSO	121
VCCI	2	VSSO	42	VCCI	82	VSSO	122
LOCK	3	VCCO	43	D10	83	VSSI	123
WRT	4	VSSI	44	D11	84	VCCO	124
MAO	5	A15	45	D12	85	FLUSH	125
SIZE1	6	A16	46	D13	86	$\overline{\text{ERROR}}$	126
SIZE0	7	A17	47	VSSO	87	INTACK	127
VSSO	8	VSSO	48	D14	88	FXACK	128
ASI0	9	A18	49	D15	89	FPSYN	129
ASI1	10	A19	50	VSSI	90	FCC1	130
VCCO	11	VCCI	51	D16	91	VSSI	131
ASI2	12	A20	52	D17	92	FCC0	132
VSSI	13	A21	53	VSSO	93	IRL3	133
VSSO	14	VSSI	54	D18	94	IRL2	134
VCCI	15	A22	55	D19	95	IRL1	135
VSSO	16	A23	56	VCCO	96	IRL0	136
VSST	17	VCCO	57	D20	97	VCCI	137
CLK	18	VCCO	58	D21	98	FCCV	138
VSSI	19	VSSO	59	VCCI	99	VSST	139
A0	20	VSSI	60	D22	100	$\overline{\text{RESET}}$	140
A1	21	VSST	61	D23	101	VSSI	141
VCCI	22	VCCI	62	VSST	102	$\overline{\text{FHOLD}}$	142
A2	23	D0	63	VSSI	103	$\overline{\text{BHOLD}}$	143
A3	24	D1	64	D24	104	$\overline{\text{MHOLDB}}$	144
VCCO	25	VCCO	65	D25	105	$\overline{\text{MHOLDA}}$	145
A4	26	D2	66	VSSO	106	$\overline{\text{MDS}}$	146
A5	27	D3	67	D26	107	$\overline{\text{FP}}$	147
VSSO	28	VSSO	68	D27	108	$\overline{\text{MEXC}}$	148
A6	29	VSSI	69	VCCO	109	$\overline{\text{FEXC}}$	149
A7	30	D4	70	D28	110	VSSI	150
VCCO	31	D5	71	D29	111	INST	151
A8	32	VCCI	72	VSSI	112	VCCI	152
A9	33	D6	73	D30	113	RD	153
VSSI	34	D7	74	D31	114	VSSO	154
A10	35	VCCO	75	VCCI	115	LDSTO	155
A11	36	D8	76	VSSO	116	$\overline{\text{WE}}$	156
A12	37	D9	77	FINS1	117	INULL	157
A13	38	VCCT	78	FINS2	118	VCCT	158
A14	39	VSSI	79	$\overline{\text{TOE}}$	119	VSSI	159
VCCI	40	VSSO	80	VCCI	120	VSS0	160

7.3 CY7C602 Electrical and Mechanical Characteristics

7.3.1 CY7C602 Maximum Ratings

Storage Temperature ..... -65° C to +150° C
Ambient Temperature with Power Applied ..... -55° C to +125° C
Supply Voltage to Ground Potential [1] ..... -0.5V to +7.0V
DC Voltage Applied to Outputs in High Z State ..... -0.5V to +7.0V
DC Input Voltage ..... -3.0V to +7.0V
Output Low Sink Current ..... 4.0 mA

7.3.2 CY7C602 Operating Range

Range	Ambient Temperature[2]	V_{CC}
Commercial	0° C to 85° C	5V ± 10%

7.3.3 CY7C602 DC Characteristics Over the Operating Range

Parameters	Description	Test Conditions	Min.	Max.	Units
V_{OH}	Output HIGH Voltage	V_{CC} = Min., I_{OH} = -2.0 mA	2.4		V
V_{OL}	Output LOW Voltage	V_{CC} = Min., I_{OL} = 8.0 mA		0.5	V
V_{IH}	Input HIGH Voltage		2.1		V
V_{IL}	Input LOW Voltage		-3.0	0.8	V
I_{IH}	Input HIGH Current	V_{CC} = Max., V_{IN} = V_{CC}	-10	10	µA
I_{IL}	Input LOW Current	V_{CC} = Max., V_{IN} = V_{SS}	-10	10	µA
I_{OZ}	Output Leakage Current	V_{CC} = Max., $V_{SS} \leq V_{OUT} \leq V_{CC}$	-10	10	µA
I_{CCQ}	Quiescent Supply Current	$V_{SS} \leq V_{IN} \leq V_{IL}$ or $V_{IH} \leq V_{IN} \leq V_{CC}$		150	mA
I_{CC}	Supply Current, Commercial	V_{CC} = Max., f = 40 MHz		450	mA
		V_{CC} = Max., f = 33 MHz		400	
		V_{CC} = Max., f = 25 MHz		350	

7.3.4 CY7C602 Capacitance [3]

Parameters	Description	Test Conditions	Max.	Units
C_{IN}	Input Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	15	pF
C_{OUT}	Output Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	20	pF
C_{IO}	I/O Bus Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	15	pF
C_{DOE}	$\overline{DOE}$ Input Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	30	pF
C_{CLK}	CLK Input Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	25	pF

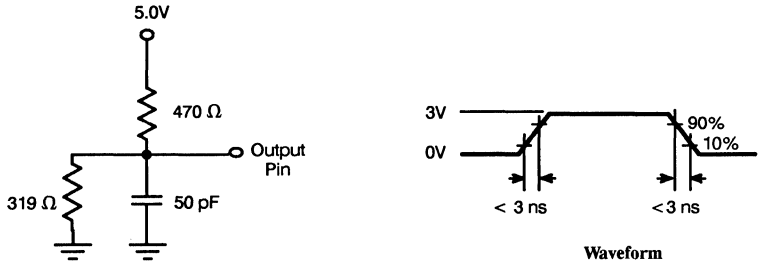
Notes:

1. All power and ground pins must be connected to the other pins of same type before any power is applied to the part.
2. Ambient temperature is the 'instant on' case temperature.
3. Tested initially and after any design or process changes that may affect these parameters.

7.3.5 CY7C602 AC Characteristics

Parameter	Description	Reference Edge	CY7C602-25		CY7C602-33		CY7C602-40		Unit
			Min.	Max.	Min.	Max.	Min.	Max.	
1	Clock Cycle		40		30		25		ns
2	Clock High and Low		18		13		11		ns
3	A(31:2) Set-Up	CLK+	3		3		2		ns
4	A(31:2) Hold	CLK+	6		6		6		ns
5	D(31:0) Input Set-Up	CLK+	3		2		2		ns
6	D(31:0) Input Hold	CLK+	5		5		4		ns
7	D(31:0) Output Delay	CLK-		20		15		13	ns
8	D(31:0) Data Valid	CLK-	4		4		4		ns
9	D(31:0) Output Turn-Off	FLUSH		31		22		18	ns
10	D(31:0) Output Valid	FLUSH	0		0		0		ns
11	D(31:0) Output Turn-Off	$\overline{\text{DOE}}$+		15		11		9	ns
12	D(31:0) Output Turn-On	$\overline{\text{DOE}}$-		15		11		9	ns
13	D(31:0) Output Valid	$\overline{\text{DOE}}$-	0		0		0		ns
14	FINS1/2 Set-Up	CLK+	9		9		7		ns
15	FINS1/2 Hold	CLK+	2.5		2.5		2.5		ns
16	INST Setup	CLK+	16		12		9		ns
17	INST Hold	CLK+	2		2		2		ns
18	FXACK Set-Up	CLK+	16		12		9		ns
19	FXACK Hold	CLK+	2		2		2		ns
20	FLUSH Set-Up	CLK+	21		14		11		ns
21	FLUSH Hold	CLK+	2		2		2		ns
22	$\overline{\text{RESET}}$ Set-Up	CLK+	15		10		8		ns
23	$\overline{\text{RESET}}$ Hold	CLK+	3		3		2		ns
24	$\overline{\text{MHOLD}}$ Set-Up	CLK-	7		4		3		ns
25	$\overline{\text{MHOLD}}$ Hold	CLK-	6		5		4.5		ns
26	$\overline{\text{MDS}}$ Set-Up	CLK-	5		4		3		ns
27	$\overline{\text{MDS}}$ Hold	CLK-	6		5		4.5		ns
28	$\overline{\text{FHOLD}}$ Delay	CLK-		29		23		19	ns
29	$\overline{\text{FHOLD}}$ Valid	CLK-	6		6		5.5		ns
30	$\overline{\text{FHOLD}}$ Delay	FINS1/2		16		15		12	ns
31	$\overline{\text{FHOLD}}$ Delay	FLUSH		28		20		16	ns
32	$\overline{\text{FHOLD}}$ Delay	$\overline{\text{MHOLD}}$-		36		27		22	ns
33	FCCV Delay	CLK-		29		23		19	ns
34	FCCV Valid	CLK-	8		6		5.5		ns
35	FCCV Delay	FLUSH		28		20		16	ns
36	FCCV Delay	$\overline{\text{MHOLD}}$-		36		27		22	ns
37	FCC(1:0) Delay	CLK+	26		19		17		ns
38	FCC(1:0) Valid	CLK+	5		4		3		ns
39	$\overline{\text{FEXC}}$ Delay	CLK+		26		19		17	ns
40	$\overline{\text{FEXC}}$ Valid	CLK+	5		4		3		ns
41	FNULL Delay	CLK+		20		13		11	ns
42	FNULL Valid	CLK+	3		3		3		ns

7.3.6 CY7C602 AC Test Loads and Waveforms

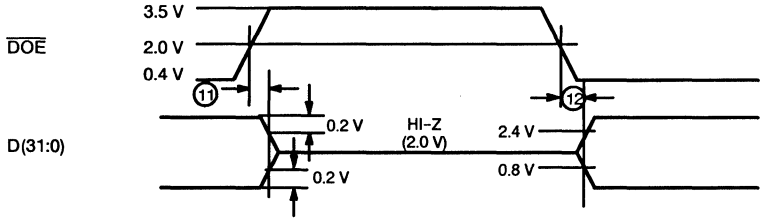


Test Load

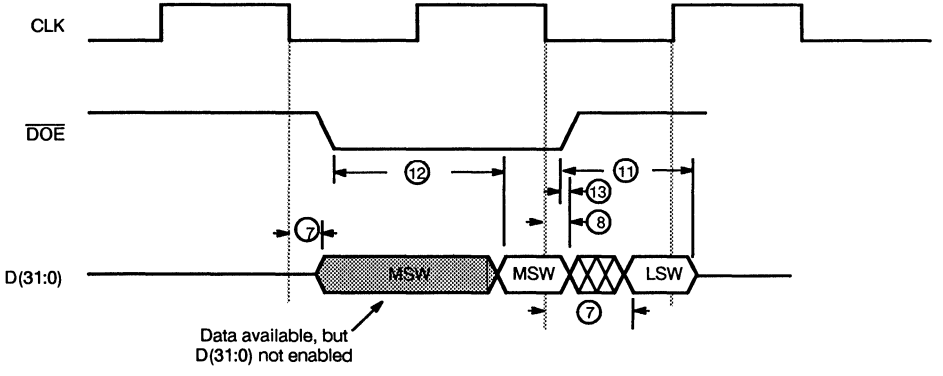
Waveform

7.3.7 CY7C602 AC Waveforms

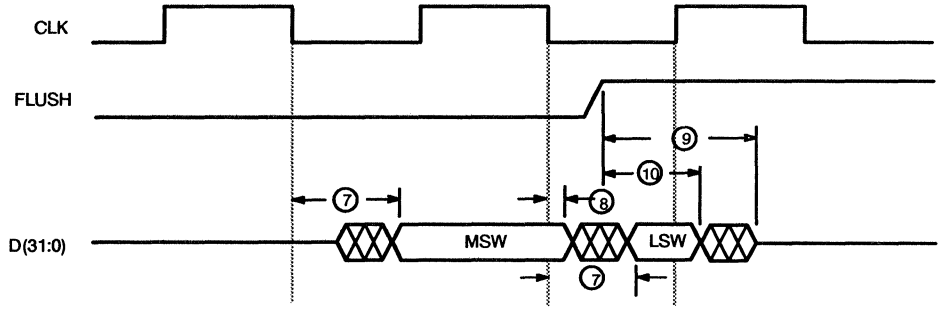
Three–State Timing



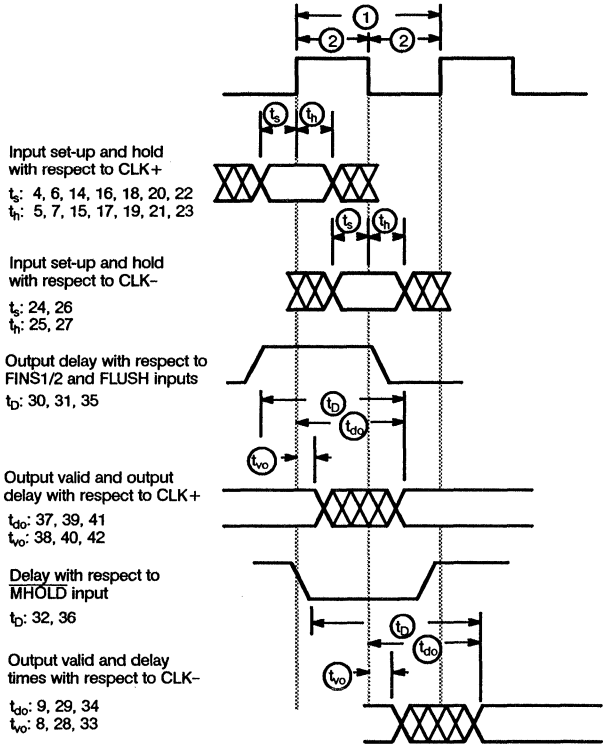
Asynchronous Store Timing



Effect of FLUSH on Store Timing



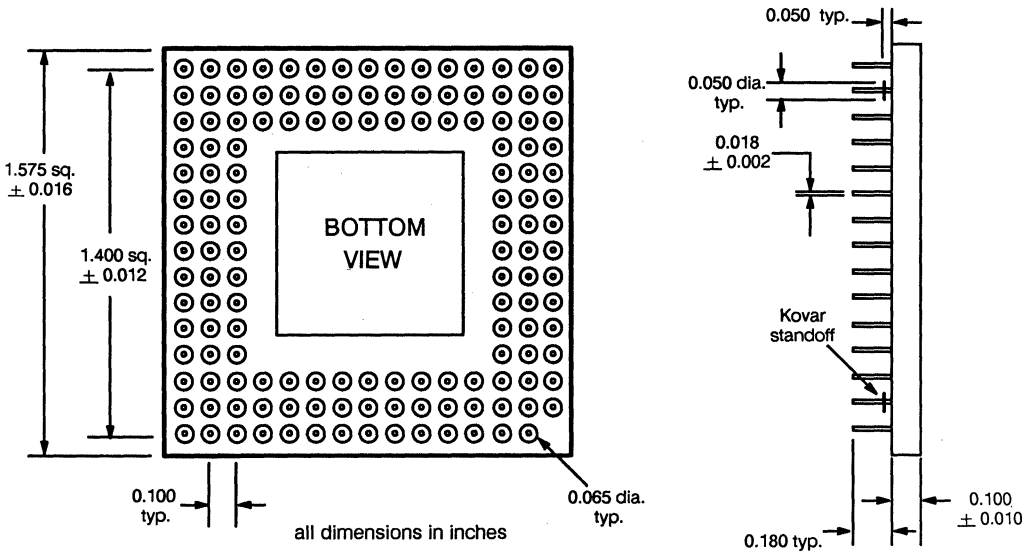
General Timing Parameters



7.3.8 CYC7602 Pin Assignments

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A		D22	A22	D24	A24	A25	D26	A26	A27	A28	A29	A30	A31	D31	GND	A
B	D21	VCC	VCC	A23	D23	VCC	D25	VCC	D27	D28	D29	D30	VCC	VCC	VCC	B
C	D20	A21	GND	GND	VCC	GND	NC	VCC	GND	GND	GND	GND	GND	VCC	FCCV	C
D	D19	VCC	GND										GND	GND	FCC1	D
E	A18	A19	A20										CCCV	FCC0	FXACK	E
F	A16	D17	D18										$\overline{\text{RESET}}$	GND	$\overline{\text{FEXC}}$	F
G	D16	A17	GND				CY7C602						CLK	GND	FNULL	G
H	A0	A1	D0				144-PIN PGA						GND	$\overline{\text{CHOLD}}$	$\overline{\text{FHOLD}}$	H
J	D1	$\overline{\text{DOE}}$	NC				TOP VIEW						VCC	$\overline{\text{MHOLDA}}$	$\overline{\text{BHOLD}}$	J
K	D2	VCC	GND				(cavity down)						VCC	$\overline{\text{MDS}}$	$\overline{\text{MHOLDB}}$	K
L	A2	D3	GND										FLUSH	VCC	VCC	L
M	A3	VCC	D5										GND	FINS1	INST	M
N	D4	VCC	GND	GND	GND	D8	GND	D10	NC	GND	D14	GND	GND	VCC	FINS2	N
P	A4	VCC	GND	A6	VCC	A8	VCC	A11	D12	VCC	VCC	VCC	D15	VCC	VCC	P
R	A5	VCC	D6	A7	D7	A9	D9	A10	D11	A12	A13	D13	A14	A15	$\overline{\text{FP}}$	R
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	

7.3.9 CY7C602 Package Diagrams



7.4 CY7C604 Electrical and Mechanical Characteristics

7.4.1 CY7C604 Maximum Ratings

Storage Temperature ..... -65° C to +150° C
Ambient Temperature with Power Applied ..... -55° C to +125° C
Supply Voltage to Ground Potential [1] ..... -0.5V to +7.0V
DC Voltage Applied to Outputs in High Z State ..... -0.5V to +7.0V
DC Input Voltage ..... -3.0V to +7.0V
Output Low Sink Current ..... 30 mA

7.4.2 CY7C604 Operating Range

Range	Ambient Temperature	V_{CC}
Commercial	0° C to 70° C	5V ± 10%
Military[2]	-55° C to +125° C	5V ± 10%

7.4.3 CY7C604 DC Characteristics Over the Operating Range [3]

Parameters	Description	Test Conditions	Min.	Max.	Units
V_{OH}	Output HIGH Voltage	V_{CC} = Min., I_{OH} = -2.0 mA	2.4		V
V_{OL}	Output LOW Voltage	V_{CC} = Min., I_{OL} = 8.0 mA		0.5	V
V_{IH}	Input HIGH Voltage		2.1	V_{CC}	V
V_{IL}	Input LOW Voltage		-3.0	0.8	V
I_{IH}	Input HIGH Current	V_{CC} = Max., V_{IN} = V_{CC}	-10	10	µA
I_{IL}	Input LOW Current	V_{CC} = Max., V_{IN} = V_{SS}	-10	10	µA
I_{SC}	Output Short Circuit Current [4]	V_{CC} = Max., V_{OUT} = 0V	-30	-180	mA
I_{OZ}	Output Leakage Current	V_{CC} = Max., $V_{SS} \leq V_{OUT} \leq V_{CC}$	-40	40	µA
I_{CCQ}	Quiescent Supply Current	$V_{SS} \leq V_{IN} \leq V_{IL}$ or $V_{IH} \leq V_{IN} \leq V_{CC}$		400	mA
I_{CC}	Supply Current, Commercial	V_{CC} = Max., f = 40 MHz V_{CC} = Max., f = 33 MHz V_{CC} = Max., f = 25 MHz		650 600 600	mA
	Supply Current, Military	V_{CC} = Max., f = 25 MHz		650	mA

7.4.4 CY7C604 Capacitance [5]

Parameters	Description	Test Conditions	Max.	Units
C_{IN}	Input Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	10	pF
C_{OUT}	Output Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	12	pF
C_{IO}	I/O Bus Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	15	pF

Notes:

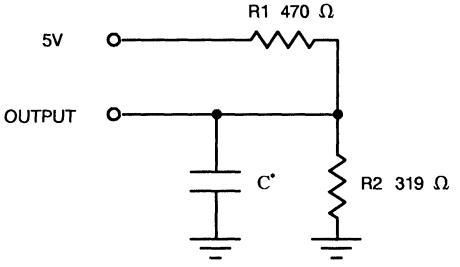
1. All power and ground pins must be connected to the other pins of same type before any power is applied to the part.
2. See last page of this document for Group A subgroup testing information.
3. Ambient temperature is the 'instant on' case temperature.
4. Not more than one output should be tested at a time. Duration of the short circuit should not be more than one second.
5. Tested initially and after any design or process changes that may affect these parameters.

7.4.5 CY7C604 AC Characteristics Over the Operating Range [6, 7]

Parameter	Description	Reference Edge	CY7C604-25		CY7C604-33		CY7C604-40		Units
			Min.	Max.	Min.	Max.	Min.	Max.	
1	Clock Cycle		40	1000	30	1000	25	1000	ns
2	Clock High and Low		18	990	13	990	10	990	ns
3	A(31:0) Output Delay[10]	CLK+		33		24		20	ns
4	A(31:0) Output Hold[10]	CLK+	7		7		7		ns
5	Address/Control [12] Input Set-Up	CLK+	3		3		2		ns
6	Address/Control [12] Input Hold	CLK+	6		6		6		ns
7	D(31:0) Output Delay [10,11]	CLK- /+		23(31)		18(24)		15(19)	ns
8	D(31:0) Output Hold [10,11]	CLK- /+	3(6)		3(6)		3(6)		ns
9	D(31:0) Input Set-Up [11]	CLK+/-	3(7)		2(6)		2(6)		ns
10	D(31:0) Input Hold [11]	CLK+/-	5(3)		5(3)		5(3)		ns
11	$\overline{\text{MDS}}$, $\overline{\text{MHOLD}}$ Output Delay	CLK-		29		23		19	ns
12	$\overline{\text{MDS}}$, $\overline{\text{MHOLD}}$ Output Hold	CLK-	7		7		7		ns
13	$\overline{\text{CBWE}}$ Output Delay	CLK-		33		25		20	ns
14	$\overline{\text{CBWE}}$ Output Hold	CLK-	7		7		7		ns
15	$\overline{\text{CROE}}$ Output Delay	CLK+		15		13		10	ns
16	$\overline{\text{CROE}}$ Output Hold	CLK+	2		2		2		ns
17	INULL/FNULL Input Set-Up	CLK+	16		14		11		ns
18	INULL/FNULL Input Hold	CLK+	2		2		2		ns
19	$\overline{\text{MEXC}}$ Output Delay	CLK+		21		16		12	ns
20	$\overline{\text{MEXC}}$ Output Hold	CLK+	3		3		3		ns
21	$\overline{\text{IOE}}$ Output Delay	CLK+		18		15		12	ns
22	$\overline{\text{IOE}}$ Output Hold	CLK+	2		2		2		ns
23	$\overline{\text{ERROR}}$ Input Set-Up	CLK+	5		4		4		ns
24	$\overline{\text{ERROR}}$ Input Hold	CLK+	2		2		2		ns
25	$\overline{\text{IRST}}$ Output Delay	CLK+		21		17		14	ns
26	$\overline{\text{IRST}}$ Output Hold	CLK+	4		4		4		ns
27	$\overline{\text{POR}}$ Input Set-Up	CLK+	15		10		8		ns
28	$\overline{\text{POR}}$ Input Hold	CLK+	3		3		3		ns
29	$\overline{\text{SNULL}}$ Input Set-Up	CLK-	7		4		3		ns
30	$\overline{\text{SNULL}}$ Input Hold	CLK-	6		5		4.5		ns
31	MAD(63:0) Output Delay [9]	CLK+		26		20		18	ns
32	MAD(63:0) Output Hold [9]	CLK+	4		4		4		ns
33	MAD(63:0) Input Set-Up	CLK+	5		4		3		ns
34	MAD(63:0) Input Hold	CLK+	2		2		2		ns
35	Mbus Control Output Delay [8,9,10]	CLK+		24		18		16	ns
36	Mbus Control Output Hold [8,9,10]	CLK+	4		4		4		ns
37	Mbus Control Input Set-Up [8]	CLK+	5		4		3		ns
38	Mbus Control Input Hold [8]	CLK+	2		2		2		ns
39	$\overline{\text{CSEL}}$ Setup upon $\overline{\text{POR}}$ Deassertion*	CLK+	8		7		6		ns
40	$\overline{\text{CSEL}}$ Hold upon $\overline{\text{POR}}$ Deassertion*	CLK+	6		6		6		ns
41	$\overline{\text{TOE}}$ Assertion to Output Disable	$\overline{\text{TOE}}$+		21		19		17	ns
42	$\overline{\text{TOE}}$ Assertion to Output Enable	$\overline{\text{TOE}}$-		21		19		17	ns

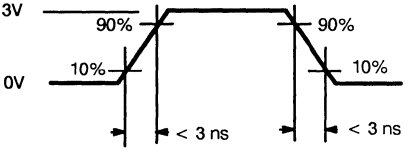
* Refer to Power-On Reset timing diagram

7.4.6 CY7C604 AC Test Loads and Waveforms



* See notes 6, 9, and 10

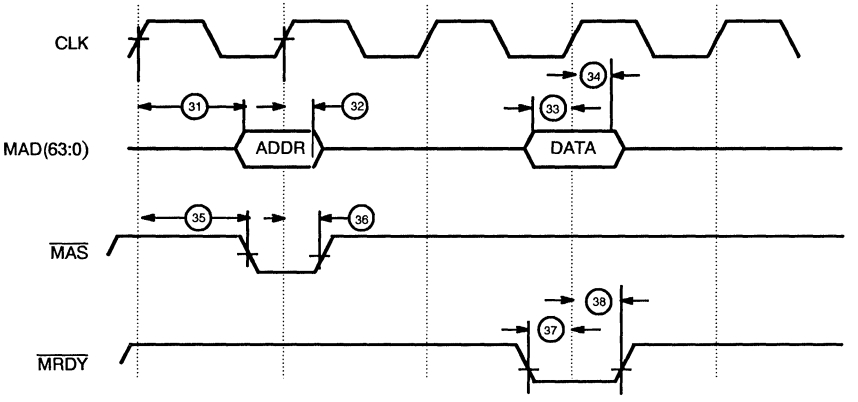
Test Load



Waveform

7.4.7 CY7C604 AC Waveforms

Mbus Timing Diagram (Single Read Transaction)



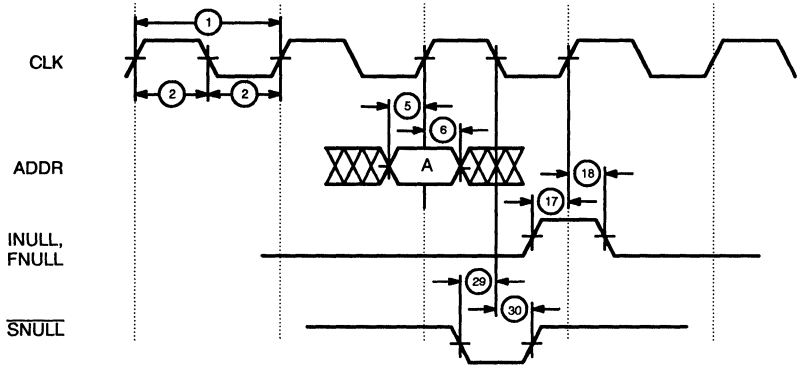
$\overline{\text{MAS}}$ timing is representative of all Mbus output signals from the CY7C604.

$\overline{\text{MRDY}}$ timing is representative of all Mbus input signals to the CY7C604.

Notes:

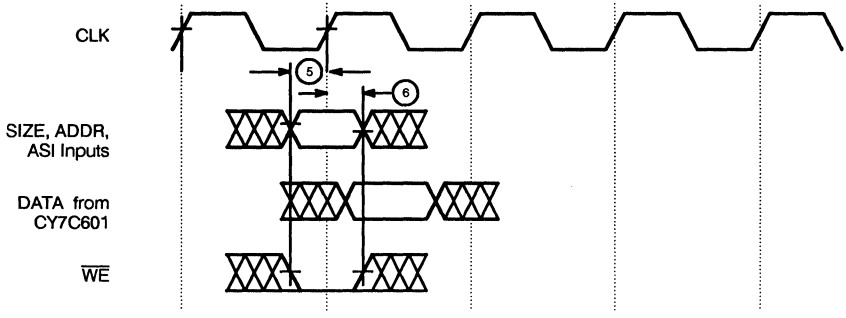
6. Test conditions assume signal transition times of 3 ns or less, a timing reference level of 1.5V, input levels of 0 to 3.0V, and output loading of 50 pF capacitance.
7. See the last page of this specification for Group A subgroup testing information.
8. Mbus Control signals include: $\overline{\text{MAS}}$, $\overline{\text{MERR}}$, $\overline{\text{MRTY}}$, $\overline{\text{MRDY}}$, $\overline{\text{MBR}}$, $\overline{\text{MBG}}$, $\overline{\text{MBB}}$, $\overline{\text{MRST}}$, and $\overline{\text{CMER}}$.
9. MAD(63:0), $\overline{\text{MAS}}$, $\overline{\text{MBB}}$, $\overline{\text{MBR}}$, and $\overline{\text{MRST}}$ timing specifications are tested using an output loading of 100 pF.
10. CMER, CSTA, A(15:2), and D(31:0) timing specifications are tested using an output loading of 80 pF.
11. First number applies to transactions with the CY7C157 CRAM. Second number applies to transactions with the CY7C601.
12. Address/Control signals include: A(31:0), ASI(5:0), SIZE(1:0), RD, $\overline{\text{WE}}$, and LDSTO.

Clock and Null Signal Timing Diagram

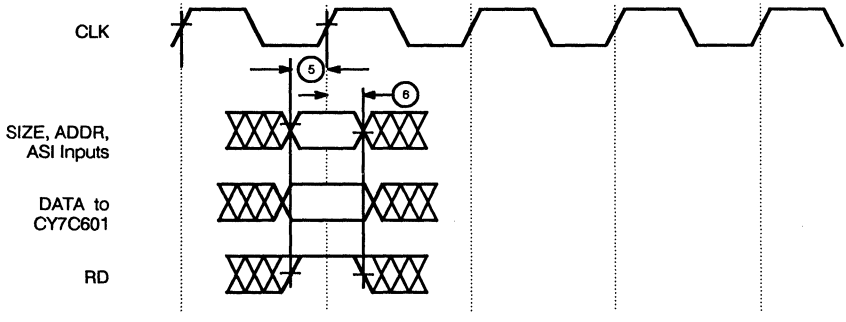


These nullification signals nullify address A. Address A is the current address of the address cycle.

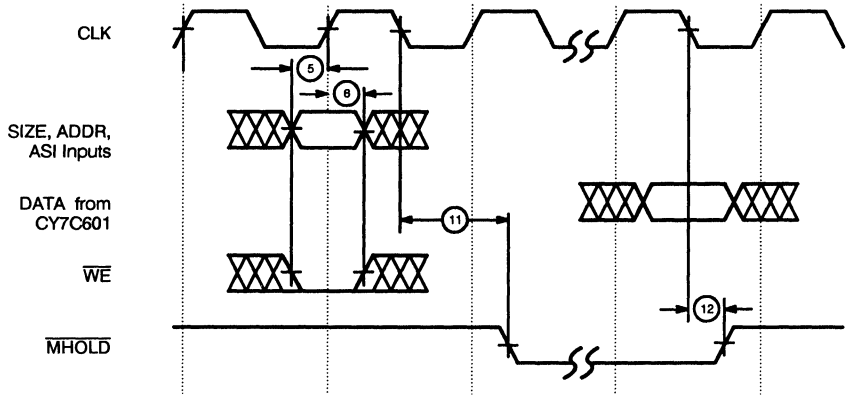
Store Timing Diagram



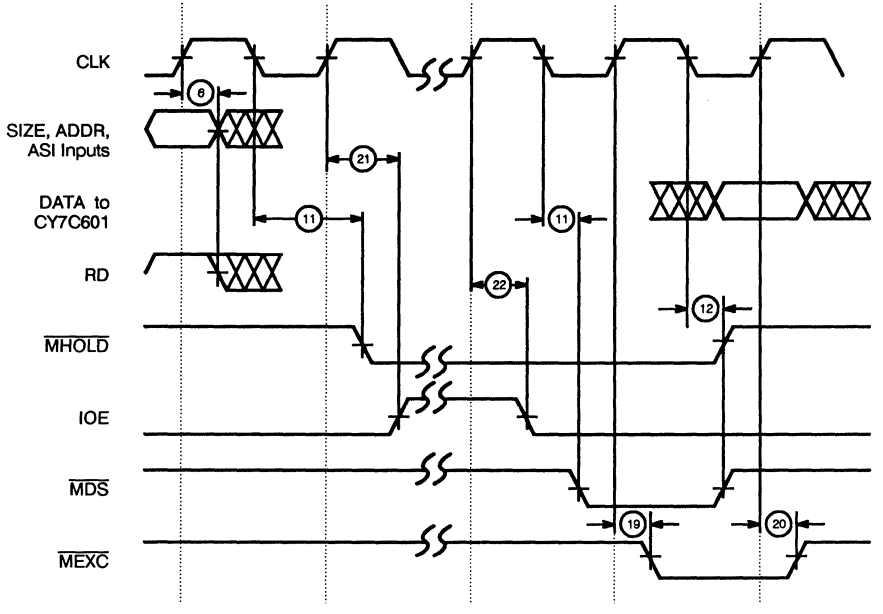
Load Timing Diagram



Store with Miss Timing

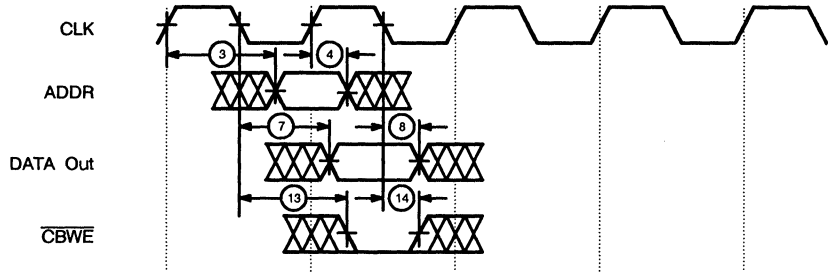


Load with Miss Timing

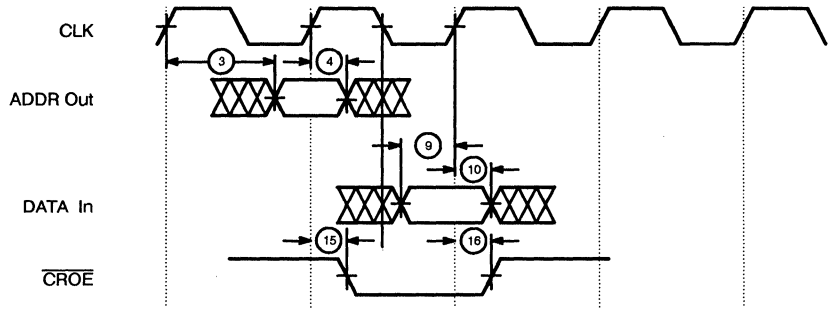


7

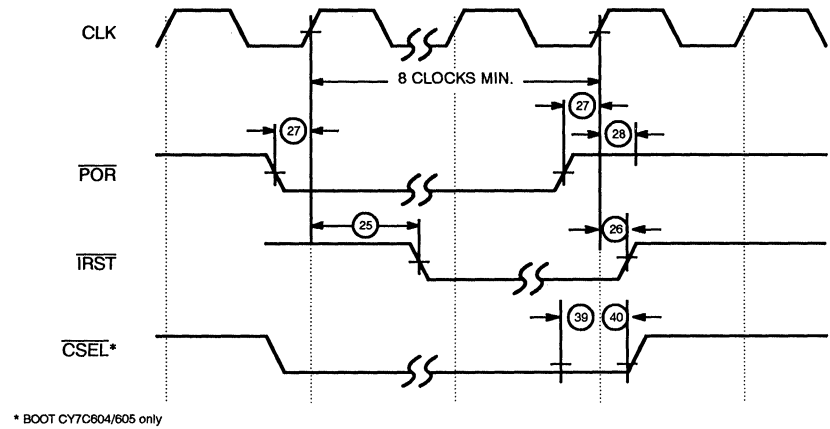
Write to CY7C157 CRAM



Read from CY7C157 CRAM

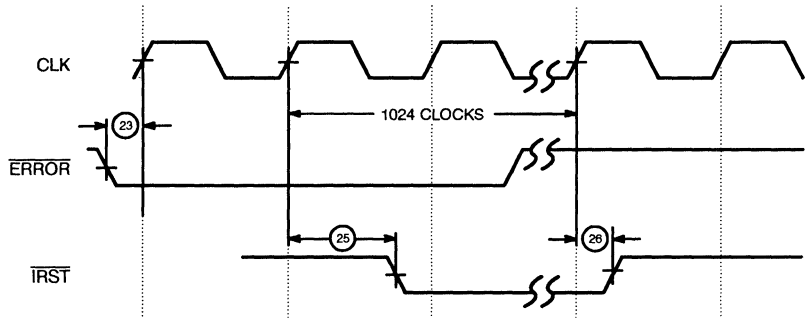


Power-On Reset Timing Diagram

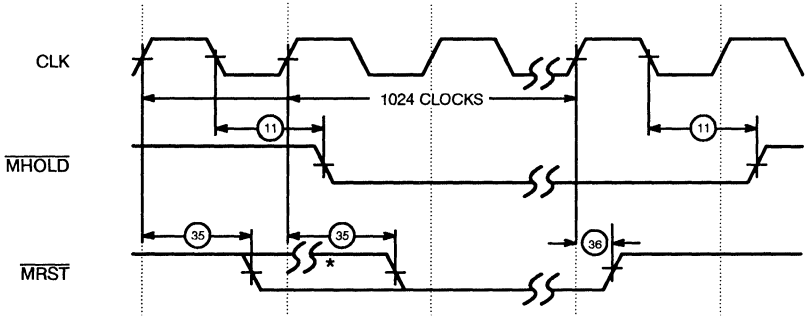


* BOOT CY7C604/605 only

Watchdog Reset Timing Diagram

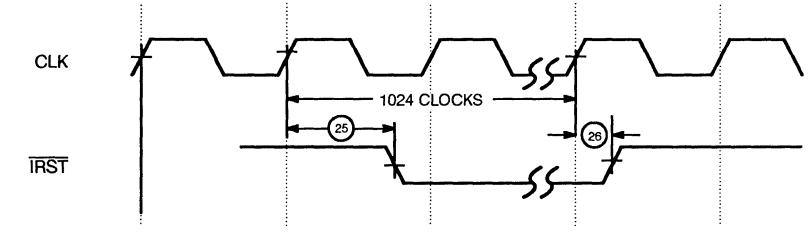


*Software External Reset Timing Diagram**



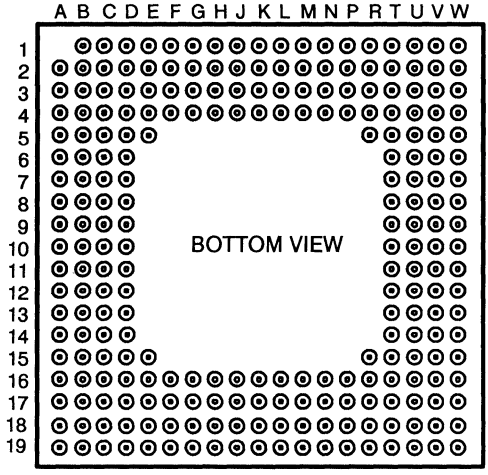
*Refer to page 4-83.

*Software Internal Reset Timing Diagram**



*Refer to page 4-83.

7.4.8 CY7C604 Pin Configuration



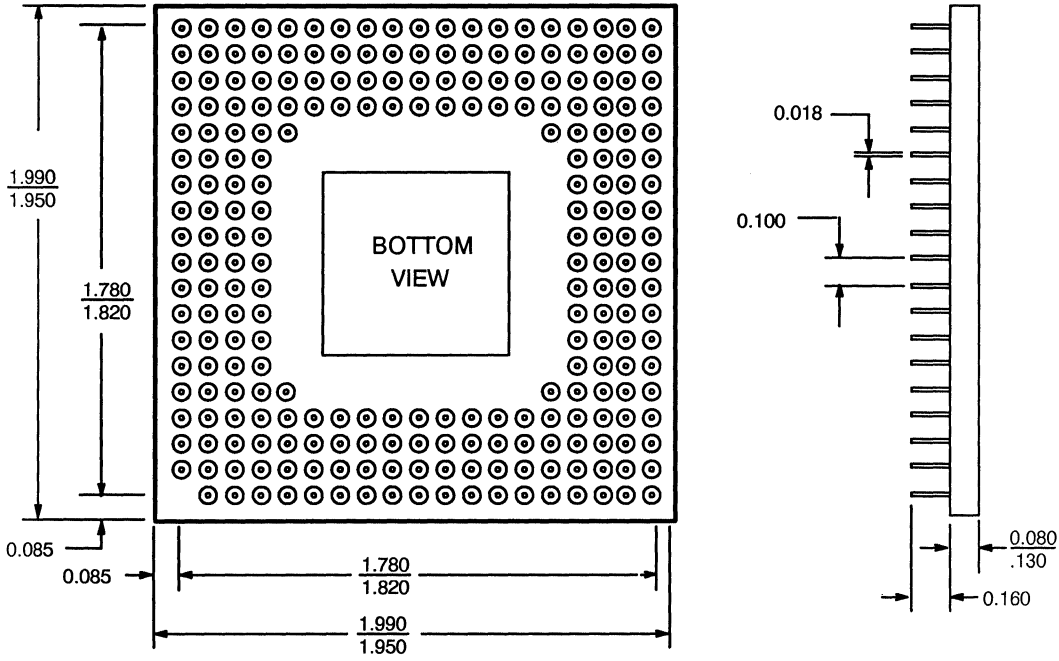
243-Pin Grid Array Package

Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #
A0	C3	A19	A9	D6	C16	D25	J18	ASI2	C1	$\overline{\text{MBR}}$	T3
A1	B3	A20	B10	D7	A17	D26	H17	ASI3	F3	$\overline{\text{MAS}}$	R1
A2	A2	A21	C10	D8	B17	D27	G19	ASI4	D2	$\overline{\text{TOE}}$	P1
A3	B4	A22	A10	D9	C17	D28	K18	ASI5	F1	$\overline{\text{CBWE0}}$	N1
A4	C4	A23	B11	D10	B18	D29	H19	$\overline{\text{IOE}}$	J2	$\overline{\text{CBWE1}}$	K2
A5	A3	A24	C11	D11	A19	D30	J19	$\overline{\text{MHOLD}}$	F2	$\overline{\text{CBWE2}}$	M3
A6	B5	A25	B12	D12	C18	D31	K17	$\overline{\text{MDS}}$	H1	$\overline{\text{CBWE3}}$	L2
A7	C5	A26	A11	D13	B19	$\overline{\text{POR}}$	C2	$\overline{\text{MEXC}}$	J1	$\overline{\text{CMER}}$	M2
A8	B6	A27	A12	D14	D18	$\overline{\text{ERROR}}$	B2	CSTA	N3	$\overline{\text{CROE}}$	M1
A9	A4	A28	A13	D15	C19	SIZE0	J3	LDSTO	L1	MAD0	U3
A10	A5	A29	B13	D16	E18	SIZE1	K1	$\overline{\text{IRST}}$	D3	MAD1	T2
A11	A6	A30	C13	D17	D19	RD	K3	CLK	L3	MAD2	U4
A12	B7	A31	B14	D18	F18	$\overline{\text{WE}}$	H2	$\overline{\text{MRST}}$	P3	MAD3	U2
A13	C7	D0	A14	D19	F17	INULL	E2	$\overline{\text{MERR}}$	N2	MAD4	W3
A14	B8	D1	B15	D20	G18	FNULL	G3	$\overline{\text{MRDY}}$	T1	MAD5	V2
A15	A7	D2	A15	D21	E19	$\overline{\text{SNULL}}$	G1	$\overline{\text{MRTY}}$	P2	MAD6	U5
A16	B9	D3	C15	D22	H18	$\overline{\text{CSEL}}$	G2	$\overline{\text{MBG}}$	U1	MAD7	W4
A17	C8	D4	A16	D23	G17	ASI0	D1	$\overline{\text{MBB}}$	R2	MAD8	V3
A18	A8	D5	B16	D24	F19	ASI1	E1			MAD9	W5

Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin Numbers					
MAD10	U6	MAD28	V11	MAD47	U18	V_{DDO}	H3	D4	R4	D11	T11	T13
MAD11	V4	MAD29	W12	MAD48	W19		D14	U14	T15	E16	G16	J16
MAD12	W6	MAD30	V12	MAD49	T18		N16	R16	L17			
MAD13	U7	MAD31	W13	MAD50	U19	V_{SSO}	R3	E4	F4	K4	M4	T4
MAD14	V5	MAD32	V13	MAD51	T19		D5	R5	T5	C6	T6	C9
MAD15	W7	MAD33	U13	MAD52	R19		D9	U9	D10	T10	C12	U12
MAD16	V6	MAD34	W14	MAD53	R18		D13	C14	T14	E15	R15	D16
MAD17	U8	MAD35	V14	MAD54	P19		H16	K16	M16	T16	D17	P17
MAD18	V7	MAD36	W15	MAD55	P18		T17					
MAD19	W8	MAD37	U15	MAD56	N19	V_{DDI}	G4	J4	L4	N4	D6	D8
MAD20	W9	MAD38	V15	MAD57	N18		T7	T9	L16	P16	E17	J17
MAD21	V8	MAD39	W16	MAD58	M17		R17					
MAD22	V9	MAD40	V16	MAD59	M19	V_{SSI}	B1	W2	E3	H4	P4	E5
MAD23	U10	MAD41	U16	MAD60	L19		D7	T8	D12	T12	D15	F16
MAD24	W10	MAD42	V17	MAD61	M18		N17	A18	V19			
MAD25	W11	MAD43	W17	MAD62	K19							
MAD26	V10	MAD44	W18	MAD63	L18							
MAD27	U11	MAD45	V18									
		MAD46	U17									

7

7.4.9 CY7C604 Package Diagrams



7.5 CY7C605 Electrical and Mechanical Characteristics

7.5.1 CY7C605 Maximum Ratings

Storage Temperature ..... -65° C to +150° C
Ambient Temperature with Power Applied ..... -55° C to +125° C
Supply Voltage to Ground Potential [1] ..... -0.5V to +7.0V
DC Voltage Applied to Outputs in High Z State ..... -0.5V to +7.0V
DC Input Voltage ..... -3.0V to +7.0V
Output Low Sink Current ..... 30 mA

7.5.2 CY7C605 Operating Range

Range	Ambient Temperature	V_{CC}
Commercial	0° C to 70° C	5V ± 10%
Military[2]	-55° C to +125° C	5V ± 10%

7.5.3 CY7C605 DC Characteristics Over the Operating Range [3]

Parameters	Description	Test Conditions	Min.	Max.	Units
V_{OH}	Output HIGH Voltage	V_{CC} = Min., I_{OH} = -2.0 mA	2.4		V
V_{OL}	Output LOW Voltage	V_{CC} = Min., I_{OL} = 8.0 mA		0.5	V
V_{IH}	Input HIGH Voltage		2.1	V_{CC}	V
V_{IL}	Input LOW Voltage		-3.0	0.8	V
I_{IH}	Input HIGH Current	V_{CC} = Max., V_{IN} = V_{CC}	-10	10	µA
I_{IL}	Input LOW Current	V_{CC} = Max., V_{IN} = V_{SS}	-10	10	µA
I_{SC}	Output Short Circuit Current [4]	V_{CC} = Max., V_{OUT} = 0V	-30	-180	mA
I_{OZ}	Output Leakage Current	V_{CC} = Max., $V_{SS} \leq V_{OUT} \leq V_{CC}$	-40	40	µA
I_{CCQ}	Quiescent Supply Current	$V_{SS} \leq V_{IN} \leq V_{IL}$ or $V_{IH} \leq V_{IN} \leq V_{CC}$		400	mA
I_{CC}	Supply Current, Commercial	V_{CC} = Max., f = 40 MHz V_{CC} = Max., f = 33 MHz V_{CC} = Max., f = 25 MHz		650 600 600	mA
	Supply Current, Military	V_{CC} = Max., f = 25 MHz		650	mA

7.5.4 CY7C605 Capacitance [5]

Parameters	Description	Test Conditions	Max.	Units
C_{IN}	Input Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	10	pF
C_{OUT}	Output Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	12	pF
C_{IO}	I/O Bus Capacitance	V_{CC} = 5.0 V, T_A = 25° C, f = 1 MHz	15	pF

Notes:

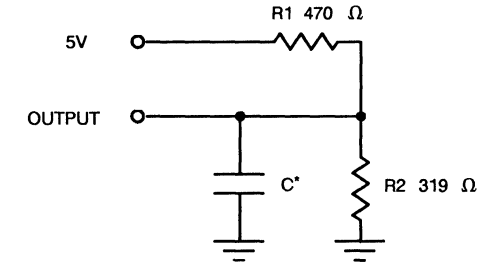
1. All power and ground pins must be connected to the other pins of same type before any power is applied to the part.
2. See last page of this document for Group A subgroup testing information
3. Ambient temperature is the 'instant on' case temperature.
4. Not more than one output should be tested at a time. Duration of the short circuit should not be more than one second.
5. Tested initially and after any design or process changes that may affect these parameters.

7.5.5 CY7C605 AC Characteristics Over the Operating Range [6, 7]

Parameter	Description	Reference Edge	CY7C605–25		CY7C605–33		CY7C605–40		Unit
			Min.	Max.	Min.	Max.	Min.	Max.	
1	Clock Cycle		40	1000	30	1000	25	1000	ns
2	Clock High and Low		18	990	13	990	10	990	ns
3	A(31:0) Output Delay[10]	CLK+		33		24		20	ns
4	A(31:0) Output Hold[10]	CLK+	7		7		7		ns
5	Address/Control Input Set-Up[12]	CLK+	3		3		2		ns
6	Address/Control Input Hold[12]	CLK+	6		6		6		ns
7	D(31:0) Output Delay [10,11]	CLK-/+		23(31)		18(24)		15(19)	ns
8	D(31:0) Output Hold [10,11]	CLK-/+	3(6)		3(6)		3(6)		ns
9	D(31:0) Input Set-Up [11]	CLK+/-	3(7)		2(6)		2(6)		ns
10	D(31:0) Input Hold [11]	CLK+/-	5(3)		5(3)		5(3)		ns
11	$\overline{\text{MDS}}$, $\overline{\text{MHOLD}}$ Output Delay	CLK-		29		23		19	ns
12	$\overline{\text{MDS}}$, $\overline{\text{MHOLD}}$ Output Hold	CLK-	7		7		7		ns
13	$\overline{\text{CBWE}}$ Output Delay	CLK-		33		25		20	ns
14	$\overline{\text{CBWE}}$ Output Hold	CLK-	7		7		7		ns
15	$\overline{\text{CROE}}$ Output Delay	CLK+		15		13		10	ns
16	$\overline{\text{CROE}}$ Output Hold	CLK+	2		2		2		ns
17	INULL/FNULL Input Set-Up	CLK+	16		14		11		ns
18	INULL/FNULL Input Hold	CLK+	2		2		2		ns
19	$\overline{\text{MEXC}}$ Output Delay	CLK+		21		16		12	ns
20	$\overline{\text{MEXC}}$ Output Hold	CLK+	3		3		3		ns
21	$\overline{\text{IOE}}$ Output Delay	CLK+		18		15		12	ns
22	$\overline{\text{IOE}}$ Output Hold	CLK+	2		2		2		ns
23	$\overline{\text{ERROR}}$ Input Set-Up	CLK+	5		4		4		ns
24	$\overline{\text{ERROR}}$ Input Hold	CLK+	2		2		2		ns
25	$\overline{\text{IRST}}$ Output Delay	CLK+		21		17		14	ns
26	$\overline{\text{IRST}}$ Output Hold	CLK+	4		4		4		ns
27	$\overline{\text{POR}}$ Input Set-Up	CLK+	15		10		8		ns
28	$\overline{\text{POR}}$ Input Hold	CLK+	3		3		3		ns
29	$\overline{\text{SNULL}}$ Input Set-Up	CLK-	7		4		3		ns
30	$\overline{\text{SNULL}}$ Input Hold	CLK-	6		5		4.5		ns
31	MAD(63:0) Output Delay [9]	CLK+		26		20		18	ns
32	MAD(63:0) Output Hold [9]	CLK+	4		4		4		ns
33	MAD(63:0) Input Set-Up	CLK+	5		4		3		ns
34	MAD(63:0) Input Hold	CLK+	2		2		2		ns
35	Mbus Control Output Delay [8,9,10]	CLK+		24		18		16	ns
36	Mbus Control Output Hold [8,9,10]	CLK+	4		4		4		ns
37	Mbus Control Input Set-Up [8]	CLK+	5		4		3		ns
38	Mbus Control Input Hold [8]	CLK+	2		2		2		ns
39	$\overline{\text{CSEL}}$ Setup upon $\overline{\text{POR}}$ Deassertion*	CLK+	8		7		6		ns
40	$\overline{\text{CSEL}}$ Hold upon $\overline{\text{POR}}$ Deassertion*	CLK+	6		6		6		ns
41	$\overline{\text{TOE}}$ Assertion to Output Disable	$\overline{\text{TOE}}$+		21		19		17	ns
42	$\overline{\text{TOE}}$ Assertion to Output Enable	$\overline{\text{TOE}}$-		21		19		17	ns

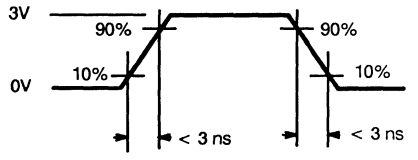
*Refer to Power-On Reset timing diagram

7.5.6 CY7C605 AC Test Loads and Waveforms



* See notes 6, 9, and 10 above

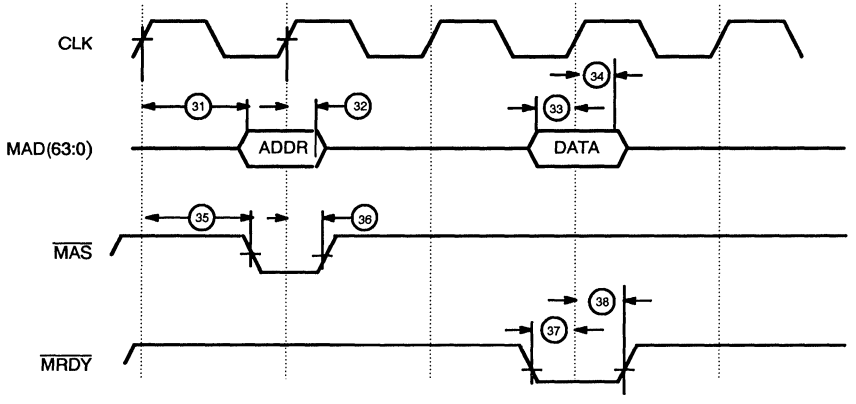
Test Load



Waveform

7.5.7 CY7C605 AC Waveforms

Mbus Timing Diagram (Single Read Transaction)



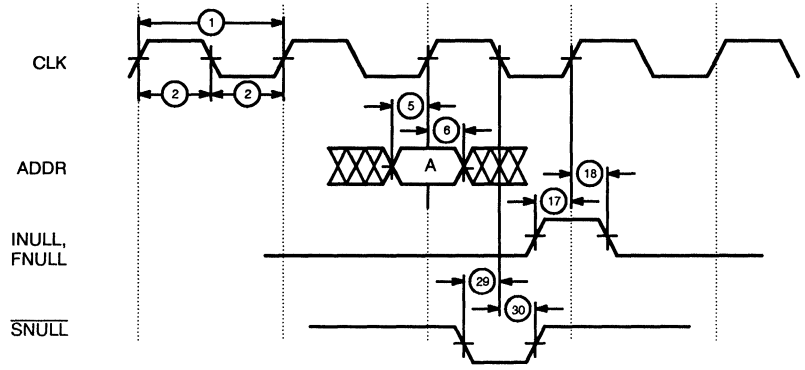
$\overline{MAS}$ timing is representative of all Mbus output signals from the CY7C605

$\overline{MRDY}$ timing is representative of all Mbus input signals to the CY7C605

Notes:

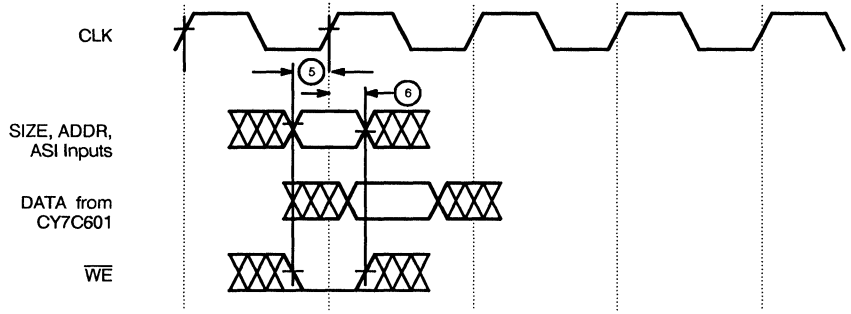
6. Test conditions assume signal transition times of 3 ns or less, a timing reference level of 1.5V, input levels of 0 to 3.0V, and output loading of 50 pF capacitance.
7. See the last page of this specification for Group A subgroup testing information.
8. Mbus Control signals include: $\overline{MAS}$, $\overline{MERR}$, $\overline{MRTY}$, $\overline{MRDY}$, $\overline{MBR}$, $\overline{MBG}$, $\overline{MBB}$, $\overline{MRST}$, $\overline{MIH}$, $\overline{MSH}$, and $\overline{CMER}$.
9. MAD(63:0), $\overline{MAS}$, $\overline{MBB}$, $\overline{MBR}$, $\overline{MIH}$, $\overline{MSH}$, and $\overline{MRST}$ timing specifications are tested using an output loading of 100 pF.
10. CMER, CSTA, A(15:2), and D(31:0) timing specifications are tested using an output loading of 80 pF.
11. First number applies to transactions with the CY7C157 CRAM. Second number applies to transactions with the CY7C601.
12. Address/Control signals include: A(31:0), ASI(5:0), SIZE(1:0), RD, $\overline{WE}$, and LDSTO.

Clock and Null Signal Timing Diagram

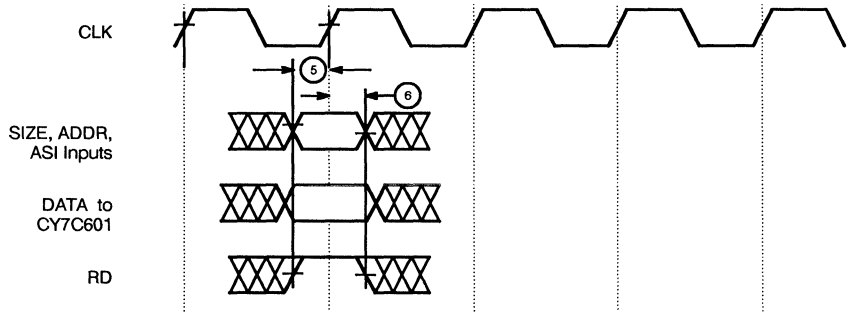


These nullification signals nullify address A. Address A is the current address of the address cycle.

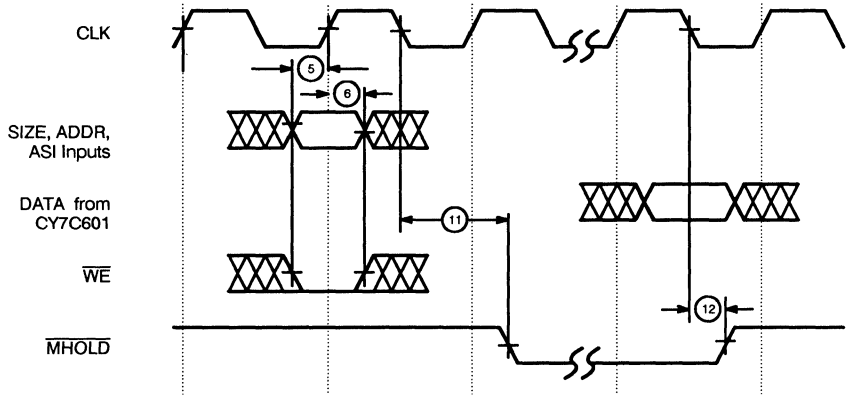
Store Timing Diagram



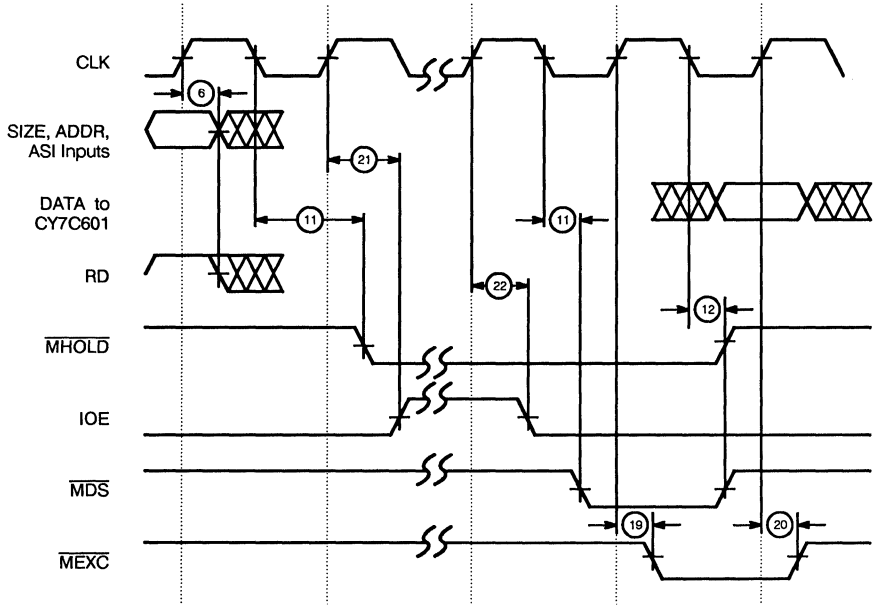
Load Timing Diagram



Store with Miss Timing

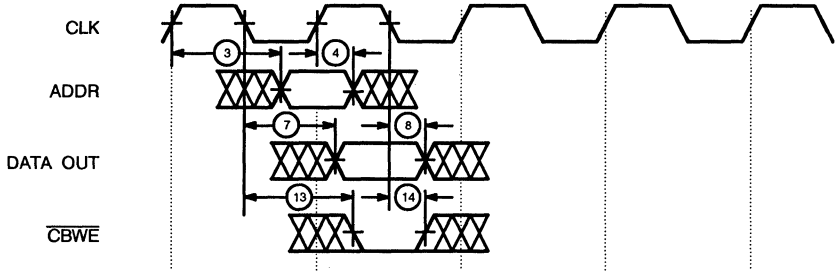


Load with Miss Timing

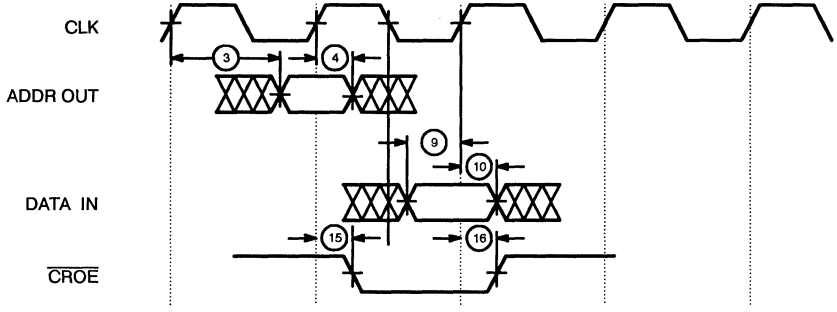


7

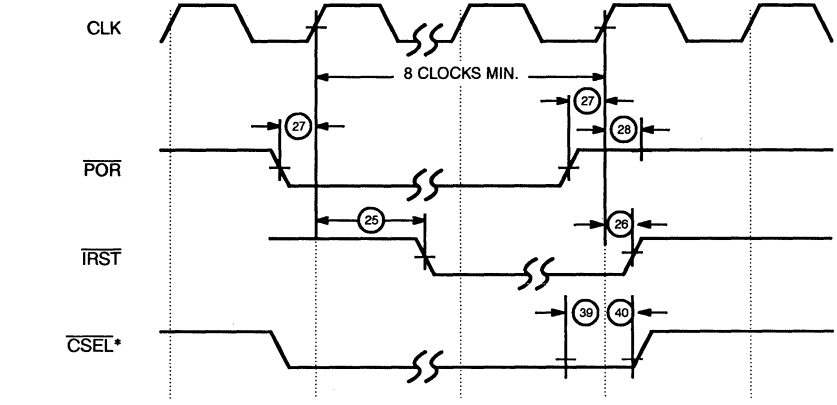
Write to CY7C157 CRAM



Read from CY7C157 CRAM

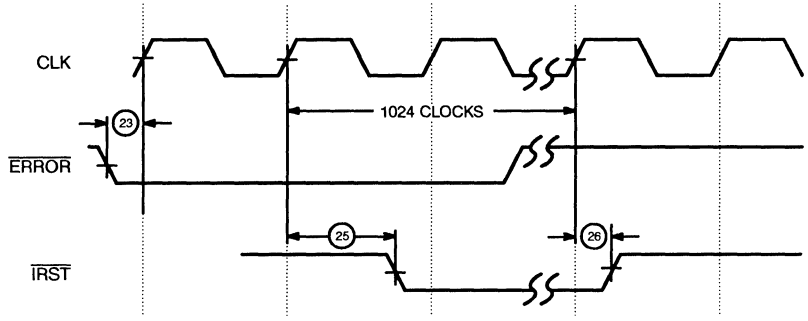


Power-On Reset Timing Diagram

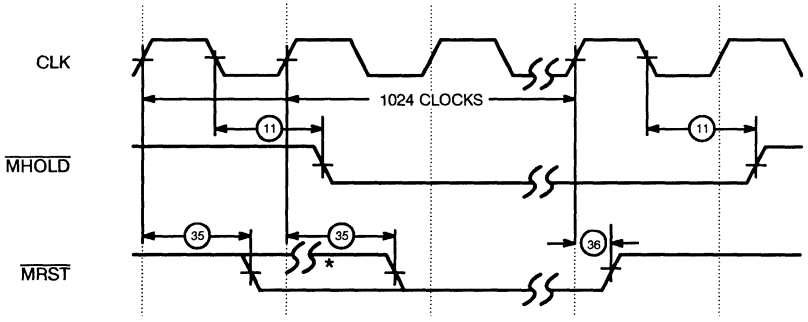


* BOOT CY7C604/605 Only

Watchdog Reset Timing Diagram

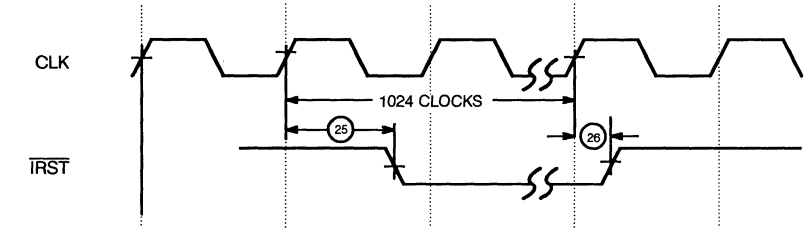


*Software External Reset Timing Diagram**



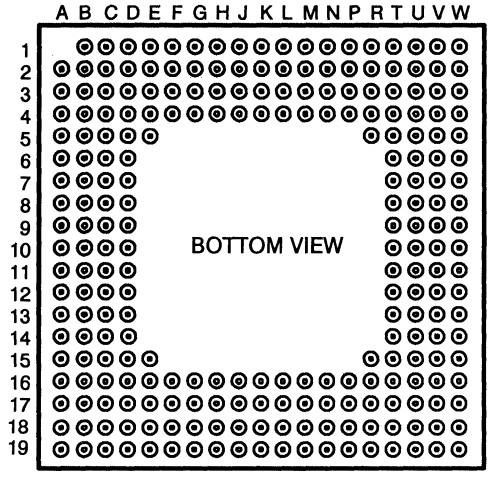
*Refer to page 4–83.

*Software Internal Reset Timing Diagram**



*Refer to page 4–83.

7.5.8 CY7C605 Pin Configuration



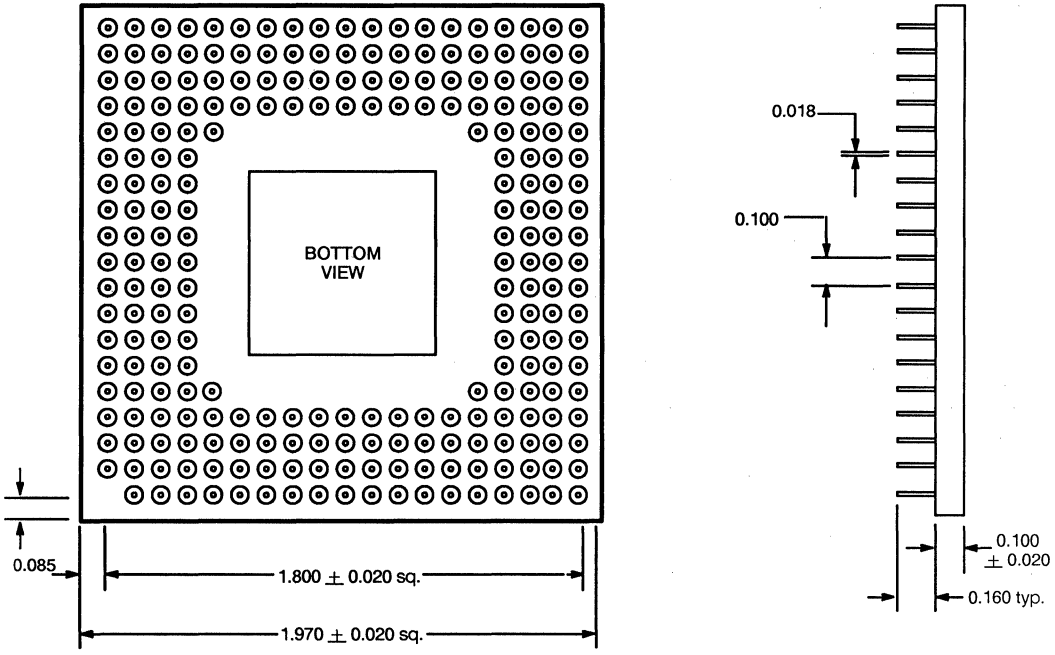
243-Pin Grid Array Package

Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #
A0	C3	A19	A9	D6	C16	D25	J18	ASI2	C1	$\overline{\text{MBR}}$	T3
A1	B3	A20	B10	D7	A17	D26	H17	ASI3	F3	$\overline{\text{MAS}}$	R1
A2	A2	A21	C10	D8	B17	D27	G19	ASI4	D2	$\overline{\text{TOE}}$	P1
A3	B4	A22	A10	D9	C17	D28	K18	ASI5	F1	$\overline{\text{CBWE0}}$	N1
A4	C4	A23	B11	D10	B18	D29	H19	$\overline{\text{IOE}}$	J2	$\overline{\text{CBWE1}}$	K2
A5	A3	A24	C11	D11	A19	D30	J19	$\overline{\text{MHOLD}}$	F2	$\overline{\text{CBWE2}}$	M3
A6	B5	A25	B12	D12	C18	D31	K17	$\overline{\text{MDS}}$	H1	$\overline{\text{CBWE3}}$	L2
A7	C5	A26	A11	D13	B19	$\overline{\text{POR}}$	C2	$\overline{\text{MEXC}}$	J1	$\overline{\text{CMER}}$	M2
A8	B6	A27	A12	D14	D18	$\overline{\text{ERROR}}$	B2	N.C.**	N3	$\overline{\text{CROE}}$	M1
A9	A4	A28	A13	D15	C19	SIZE0	J3	LDSTO	L1	MAD0	U3
A10	A5	A29	B13	D16	E18	SIZE1	K1	$\overline{\text{IRST}}$	D3	MAD1	T2
A11	A6	A30	C13	D17	D19	RD	K3	CLK	L3	MAD2	U4
A12	B7	A31	B14	D18	F18	$\overline{\text{WE}}$	H2	$\overline{\text{MRST}}$	P3	MAD3	U2
A13	C7	D0	A14	D19	F17	INULL	E2	$\overline{\text{MERR}}$	N2	MAD4	W3
A14	B8	D1	B15	D20	G18	FNULL	G3	$\overline{\text{MRDY}}$	T1	MAD5	V2
A15	A7	D2	A15	D21	E19	$\overline{\text{SNULL}}$	G1	$\overline{\text{MRTY}}$	P2	MAD6	U5
A16	B9	D3	C15	D22	H18	$\overline{\text{CSEL}}$	G2	$\overline{\text{MBG}}$	U1	MAD7	W4
A17	C8	D4	A16	D23	G17	ASI0	D1	$\overline{\text{MBB}}$	R2	MAD8	V3
A18	A8	D5	B16	D24	F19	ASI1	E1			MAD9	W5

** N.C. is a no connect (CSTA on CY7C604)

Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin #	Pin Name	Pin Numbers					
MAD10	U6	MAD28	V11	MAD47	U18	V_{DDO}	H3	D4	R4	D11	T11	T13
MAD11	V4	MAD29	W12	MAD48	W19		D14	U14	T15	E16	G16	J16
MAD12	W6	MAD30	V12	MAD49	T18		N16	R16	L17			
MAD13	U7	MAD31	W13	MAD50	U19	V_{SSO}	R3	E4	F4	K4	M4	T4
MAD14	V5	MAD32	V13	MAD51	T19		D5	R5	T5	C6	T6	C9
MAD15	W7	MAD33	U13	MAD52	R19		D9	U9	D10	T10	C12	U12
MAD16	V6	MAD34	W14	MAD53	R18		D13	C14	T14	E15	R15	D16
MAD17	U8	MAD35	V14	MAD54	P19		H16	K16	M16	T16	D17	P17
MAD18	V7	MAD36	W15	MAD55	P18		T17					
MAD19	W8	MAD37	U15	MAD56	N19	V_{DDI}	G4	J4	L4	N4	D6	D8
MAD20	W9	MAD38	V15	MAD57	N18		T7	T9	L16	P16	E17	J17
MAD21	V8	MAD39	W16	MAD58	M17		R17					
MAD22	V9	MAD40	V16	MAD59	M19	V_{SSI}	B1	W2	E3	H4	P4	E5
MAD23	U10	MAD41	U16	MAD60	L19		D7	T8	D12	T12	D15	F16
MAD24	W10	MAD42	V17	MAD61	M18		N17	A18	V19			
MAD25	W11	MAD43	W17	MAD62	K19							
MAD26	V10	MAD44	W18	MAD63	L18							
MAD27	U11	MAD45	V18	$\overline{MIH}$	W1							
		MAD46	U17	$\overline{MSH}$	V1							

7.5.9 CY7C605 CPGA Package Diagram



7.6 CY7C157 Electrical and Mechanical Characteristics

7.6.1 CY7C157 Maximum Rating

(Above which the useful life may be impaired. For user guidelines, not tested.)

Storage Temperature ..... -65°C to +150°C
Ambient Temperature with Power Applied ..... -55°C to +125°C
Supply Voltage to Ground Potential ..... -0.5V to +7.0V
DC Voltage Applied to Outputs in High Z State ..... -0.5V to +7.0V
DC Input Voltage ..... -3.0V to + 7.0V
Output Current into Outputs (Low) ..... 50 mA
Static Discharge Voltage (per MIL-STD-883, Method 3015) ..... >2001V
Latch-Up Current ..... >200 mA

7.6.2 CY7C157 Operating Range

Range	Ambient Temperature	V_{CC}
Commercial	0°C to + 70°C	5V ± 10%
Military[1]	- 55°C to + 125°C	5V ± 10%

7.6.3 CY7C157 DC Characteristics Over the Operating Range[2]

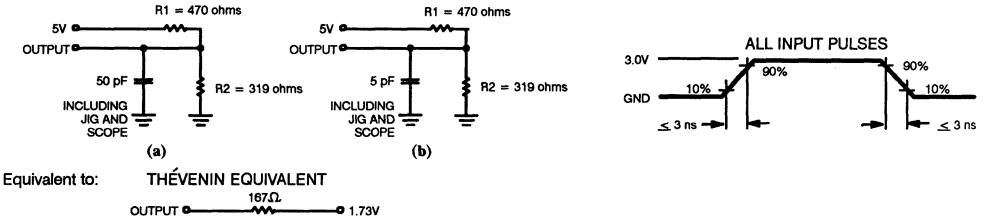
Parameters	Description	Test Conditions		7C157-20 Min.	7C157-20 Max.	7C157-24 Min.	7C157-24 Max.	7C157-33 Min.	7C157-33 Max.	Units
V_{OH}	Output HIGH Voltage	V_{CC} = Min., I_{OH} = -4.0 mA		2.4		2.4		2.4		V
V_{OL}	Output LOW Current	V_{CC} = Min., I_{OL} = 8.0 mA			0.5		0.5		0.5	V
V_{IH}	Input HIGH Voltage			2.1	V_{CC}	2.1	V_{CC}	2.1	V_{CC}	V
V_{IL}	Input LOW Voltage			-3.0	0.8	-3.0	0.8	-3.0	0.8	V
I_{IX}	Input Load Current	GND < V_I < V_{CC}		-10	+10	-10	+10	-10	+10	µA
I_{OZ}	Output Leakage Current	GND < V_O < V_{CC}, Output Disabled		-50	+50	-50	+50	-50	+50	µA
I_{OS}	Output Short Circuit Current[3]	V_{CC} = Max., V_{OUT} = GND			-350		-350		-350	mA
I_{CC}	V_{CC} Operating Supply Current	V_{CC} = Max,	Commercial		300		250		250	mA
		I_{OUT} = 0 mA	Military				300		300	

7.6.4 CY7C157 Capacitance[4]

Parameters	Description	Test Conditions	Max.	Units
C_{IN}	Input Capacitance	T_A = 25°C, f = 1 MHz, V_{CC} = 5.0 V	5	pF
C_{OUT}	Output Capacitance		8	pF

Notes:
1. T_A is the "instant on" case temperature.
2. See the last page of this specification for Group A subgroup testing information.
3. Not more than 1 output should be shorted at a time. Duration of the short circuit should not exceed 30 seconds.
4. Tested initially and after any design or process changes that may affect these parameters.

7.6.5 CY7C157 AC Test Loads and Waveforms



7.6.6 CY7C157 AC Characteristics Over the Operating Range[2, 5]

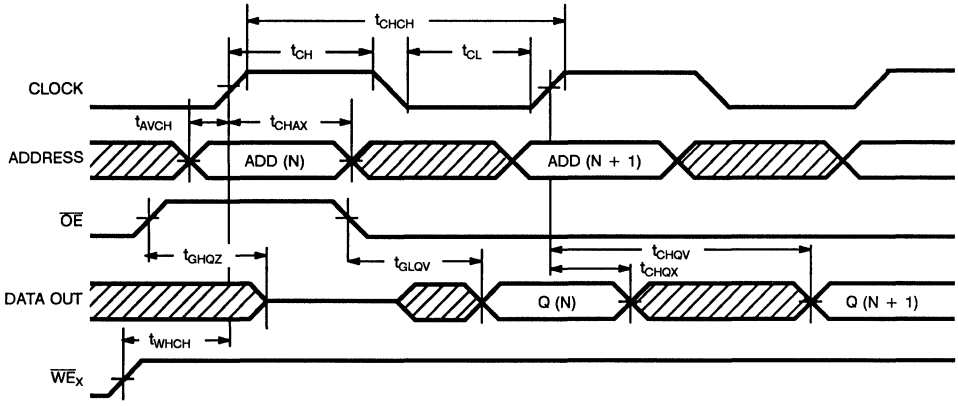
Parameters	Description	CY7C157-20[6]		CY7C157-24[6]		CY7C157-33		Units
		Min.	Max.	Min.	Max.	Min.	Max.	
READ CYCLE[7, 8]								
t_{CHCH}	Clock Cycle Time	25		30		40		ns
t_{CH}	Clock HIGH Time	11		13		18		ns
t_{CL}	Clock LOW Time	11		13		18		ns
t_{CHQV}	Clock HIGH to Output Valid		20		24		33	ns
t_{CHQX}	Output Data Hold	5		5		5		ns
t_{WHCH}	$\overline{WE}_X$ HIGH to Next Clock HIGH	2		2		3		ns
t_{GLQV}	$\overline{OE}$ LOW to Output Valid	0	8	0	10	0	15	ns
t_{GHQZ}	$\overline{OE}$ HIGH to Output Three-state	0	8	0	10	0	15	ns
t_{GHCH}	$\overline{OE}$ HIGH to Next Clock HIGH	7		7		7		ns
t_{AVCH}	Address Set-Up	2		2		3		ns
t_{CHAX}	Address Hold	6		6		6		ns
WRITE CYCLE[9]								
t_{CHCH}	Clock Cycle Time[10]	25		30		40		ns
t_{CH}	Clock HIGH Time	11		13		18		ns
t_{CL}	Clock LOW Time	11		13		18		ns
t_{GHQZ}	$\overline{OE}$ HIGH to Output Three-state	0	8	0	10	0	15	ns
t_{GHCH}	$\overline{OE}$ HIGH to Next Clock HIGH	7		7		7		ns
t_{DVCL}	Data in Set-Up to Clock	6		6		7		ns
t_{CLDX}	Data in Hold from Clock	2		2		2		ns
t_{WLCL}	$\overline{WE}_X$ LOW to Clock LOW[11, 12]	2		2		3		ns
t_{CLWH}	Clock LOW to $\overline{WE}_X$ HIGH[11,12]	6		6		7		ns
t_{AVCH}	Address Set-Up	2		2		3		ns
t_{CHAX}	Address Hold	6		6		6		ns

Notes:

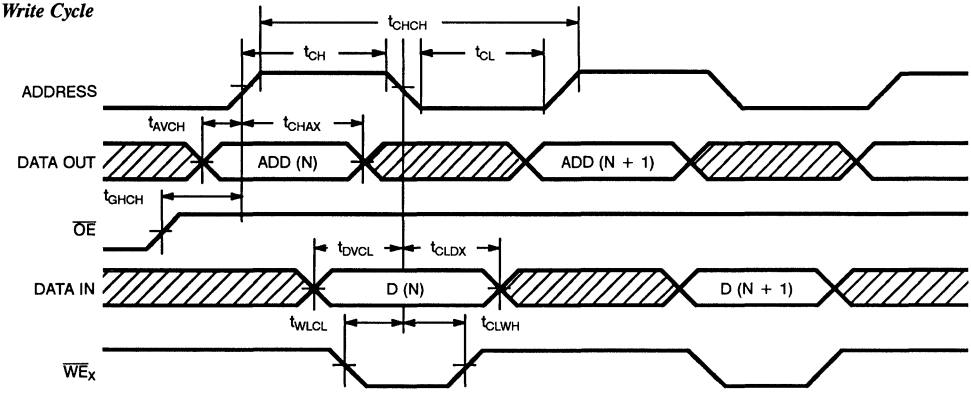
5. Test conditions assume signal transition times of 5 ns or less, timing referenece levels of 1.5V, input pulse levels of 0 to 3.0V, and output loading of the specified I_{OL}/I_{OH} and 50-pF load capacitance.
6. Surface mount package only.
7. $\overline{WE}$ is HIGH for read cycle.
8. $\overline{OE}$ is selected (LOW).
9. $\overline{OE}$ must be HIGH for data-in to propagate to latch.
10. t_{GHQZ} is tested with C_L = 5 pF as in part (b) of AC Test Loads. Transition is measured ±500 mV from steady state voltage.
11. Self-Timed Write is triggered on falling edge of registered $\overline{WE}_0$ or $\overline{WE}_1$ signals.
12. X = 0 or 1 for low byte and high byte, respectively.

7.6.7 CY7C157 AC Waveforms

Read Cycle



Write Cycle



7

7.6.8 CY7C157 Truth Table

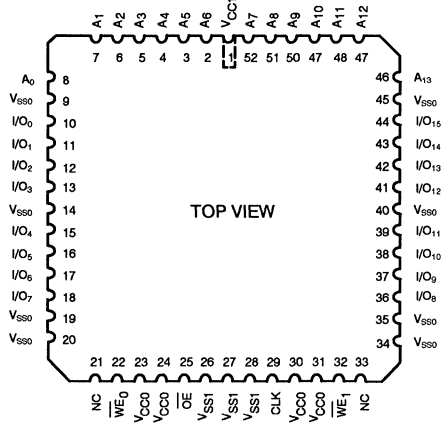
Inputs			
$\overline{OE}$	$\overline{WE_0}$ (↓CLOCK)	$\overline{WE_1}$ (↓CLOCK)	Outputs
X	X	X	High Z
H	H	H	High Z
L	H	H	$I/O_0 - I/O_{15}$
H	L	H	$I/O_0 - I/O_7$
H	H	L	$I/O_8 - I/O_{15}$
H	L	L	$I/O_0 - I/O_{15}$

7.6.9 CY7C157 Pin Timing Cross Reference

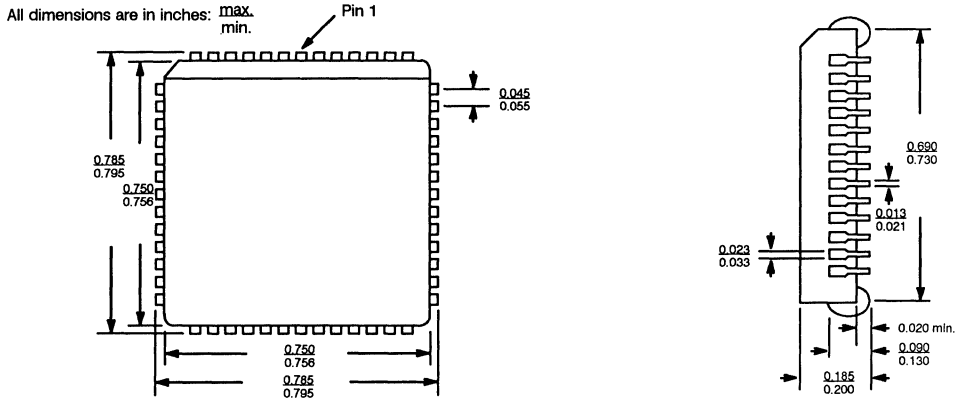
Pin Name	Timing Reference	Description
Clock	C	Clock Inputs
$A_0 - A_{13}$	A	Address Inputs
$I/O_0 - I/O_{15}$ (Input)	D	Data Inputs
$I/O_0 - I/O_{15}$ (Output)	Q	Data Outputs
$\overline{WE_0}$, $\overline{WE_1}$, $\overline{WE_X}$	W	Write Enable
$\overline{OE}$	G	Output Enable

7.6.10 CY7C157 Pin Assignments

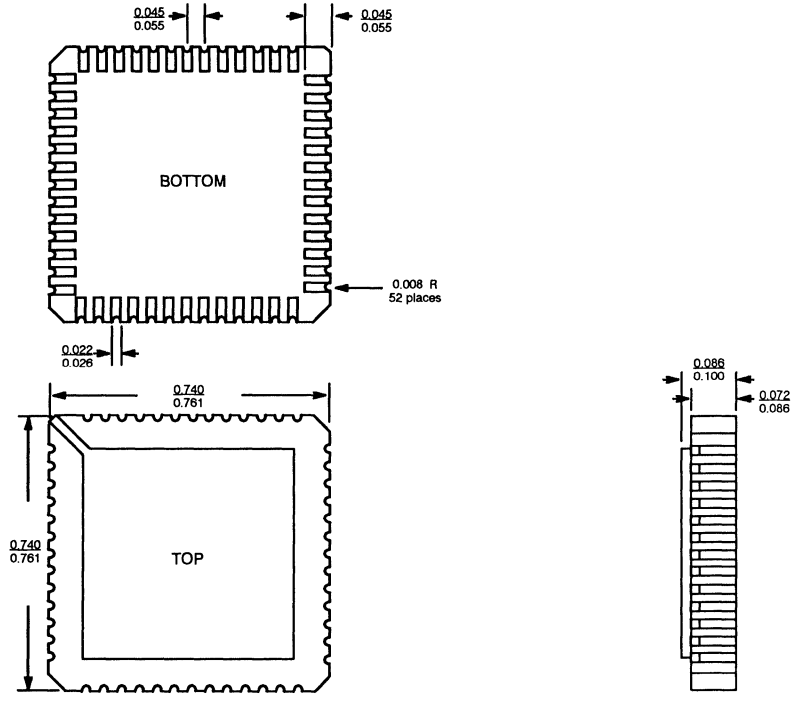
LCC and PLCC



7.6.11 CY7C157 Package Diagrams



52-Lead Plastic Leadless Chip Carrier J69



52-Pin Square Leadless Chip Carrier L69

CYPRESS SEMICONDUCTOR CY7C600 Ordering Information

8.1 CY7C601 Ordering Information

Clock Frequency (MHz)	Ordering Code	Package Type	Operating Range
40	CY7C601-40GC	G208	Commercial
	CY7C601-40FC	CQFP-208	
33	CY7C601-33GC	G208	Commercial
	CY7C601-33FC	CQFP-208	
25	CY7C601-25GC	G208	Commercial
	CY7C601-25FC	CQFP-208	
25	CY7C601-25GMB	G208	Military
	CY7C601-25FMB	CQFP-208	

8.2 CY7C611 Ordering Information

Clock Frequency (MHz)	Ordering Code	Package Type	Operating Range
25	CY7C611-25KC	PQFP-208	Commercial

8.3 CY7C602 Ordering Information

Clock Frequency (MHz)	Ordering Code	Package Type	Operating Range
40	CY7C602-40GC	G144	Commercial
33	CY7C602-33GC	G144	Commercial
25	CY7C602-25GC	G144	Commercail

Contact your local Cypress sales office for up-to-date ordering and availability information.

8.4 CY7C604 Ordering Information

Clock Frequency (MHz)	Ordering Code	Package Type	Operating Range
40	CY7C604-40GC	G243	Commercial
33	CY7C604-33GC	G243	Commercial
25	CY7C604-25GC	G243	Commercial

8.5 CY7C605 Ordering Information

Clock Frequency (MHz)	Ordering Code	Package Type	Operating Range
40	CY7C605-40GC	G243	Commercial
33	CY7C605-33GC	G243	Commercial
25	CY7C605-25GC	G243	Commercial

8.6 CY7C157 Ordering Information

Speed (ns)	Ordering Code	Package Type	Operating Range
20	CY7C157-20LC	L69	Commercial
	CY7C157-20JC	J69	
24	CY7C157-24LC	L69	Commercial
	CY7C157-24JC	J69	
33	CY7C157-33LC	L69	Commercial
	CY7C157-33JC	J69	

A.1 Uni-Module Board Hardware Description

A.1.1 Introduction

The SPARC Uni-Module Board is a single-board computer utilizing the complete SPARC chip set (Integer Unit, Floating Point Unit, Cache Memory Management Unit, and Cache RAMs) operating at maximum speed. It contains a 64-kbyte, direct-mapped, virtual cache (1 CMU and 2 CRAMs). The PC board size is 3.3" x 7.25" and it has an Mbus interface via a 100-pin connector. The Uni-Module is described in this section as an example of a CY7C600 processor node and to demonstrate how to interconnect the CY7C600 chip set.

A.1.2 Features

1. CY7C601 SPARC Integer Unit
2. CY7C602 Floating-Point Unit
3. CY7C604 Cache Controller and Memory Management Unit
4. 64 kbytes of direct-mapped cache using two CY7C157 CRAMs (0 Wait States on Virtual bus)
5. Operates over a frequency range of 10 to 40 MHz at ambient temperature and nominal + 5V
6. The board requires approximately 2A @ + 5V via the Mbus connector

A.1.3 Basic Mbus Operation and Timing

The Mbus is a fully synchronous (same clock as IU and CMU), multiplexed (address and data), 64-bit bus. A cycle is started when $\overline{\text{MAS}}$ (Mbus Address Strobe) is asserted via the CMU and is completed successfully upon the assertion of $\overline{\text{MRDY}}$ alone, or unsuccessfully with the assertion of various combinations of $\overline{\text{MERR}}$ (Mbus ERRor), $\overline{\text{MRDY}}$ (Mbus ReaDY) or $\overline{\text{MRTY}}$ (Mbus ReTrY) from the Mbus.

The Mbus allows multiple masters via an external arbiter. An Mbus master can request the Mbus by asserting its $\overline{\text{MBR}}$ (Mbus Bus Request) line to the arbiter and the arbiter can grant the bus by asserting the $\overline{\text{MBG}}$ (Mbus Bus Grant) line to the requester. Each potential bus master monitors the $\overline{\text{MBB}}$ (Mbus Bus Busy) line and, after receiving its bus grant and observing that $\overline{\text{MBB}}$ has been deasserted, will synchronously assert $\overline{\text{MBB}}$ on the next clock and keep it asserted until its access is finished.

A Power-On Reset signal is generated to the CMU from the Mbus. Reset is asserted on the Mbus via the $\overline{\text{MRST}}$ (Mbus ReSeT) line from the CMU.

Level sensitive interrupts (15 max.) are generated to the CY7C601 IU via the IRL(3:0) lines from the Mbus. A value of 0000b means that there is no interrupt, whereas a value of 1111b means an NMI is being asserted.

Basic Mbus timing is as follows:

1. The CY7C604, running at 40 MHz, makes address, data and status signals available 18 ns after the clock rising edge (RE) which gives 7-ns set-up time before the following clock RE to latch them. It also holds these signals until 4 ns after the next clock RE. For control signals the respective timings are 16 ns after the clock RE (gives 9-ns set-up time) and holds the signals for 4 ns after the clock RE.
2. The CMU requires that data, control, and status signals be valid no later than 3 ns before the clock RE and that it be held for 2 ns after the clock RE.

A.1.3.1 Board Detailed Description

The first figure is a block diagram of the SPARC Uni-Module Board. For a more complete understanding, consult the Uni-Module Board schematic diagrams that follow. The logic can be broken down as in the following sections.

A.1.3.1.1 Computing Cluster (IU, CMU, CRAMs and FPU)

The socketed computing cluster consists of the CY7C601 Integer Unit in a 207-pin PGA package, the CY7C602 Floating-Point Unit in a 143-pin PGA package, the CY7C604 Cache Controller and Memory Management Unit in a 243-pin PGA package, and two CY7C157 Cache RAMs in 52-pin PLCC packages. These chips are connected together in a tightly coupled configuration to provide integer, floating-point, and memory management capabilities as well as 64 kbytes of direct-mapped, virtual, cache. The board/Mbus does not make use of any coprocessor signals nor the $\overline{\text{BHOLD}}$ or MAO signals.

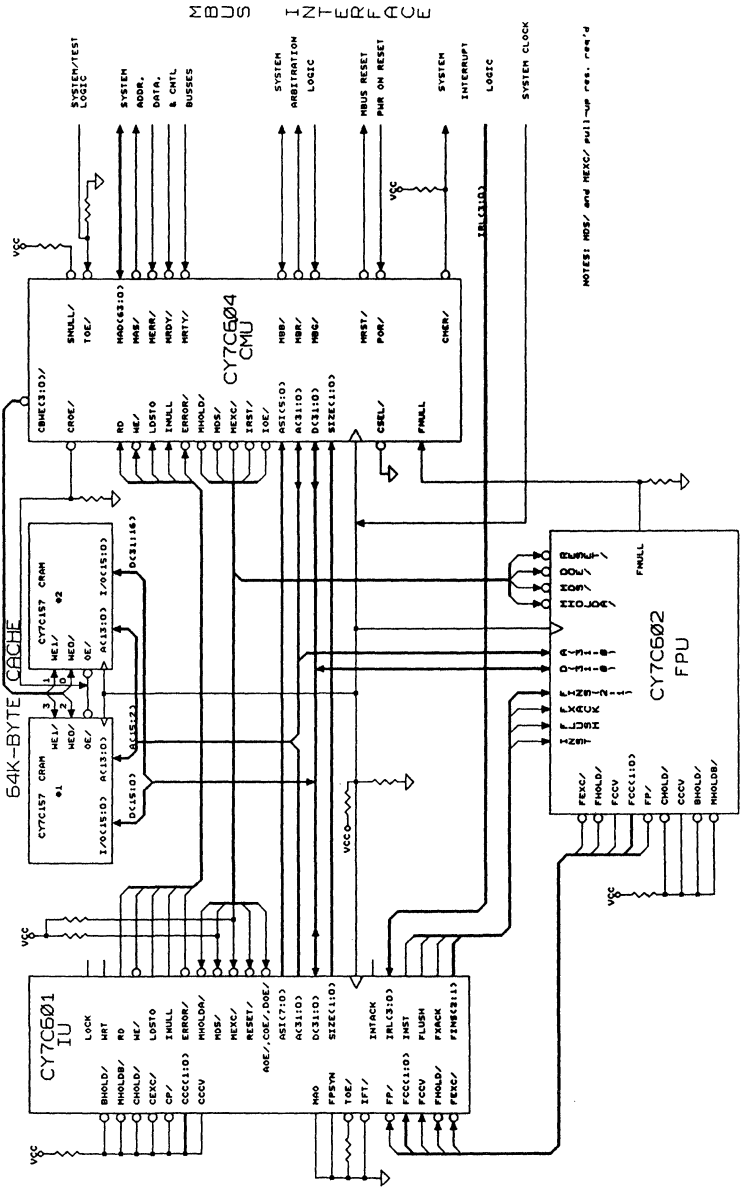
A.1.3.1.2 Board Decoupling and Signal Termination

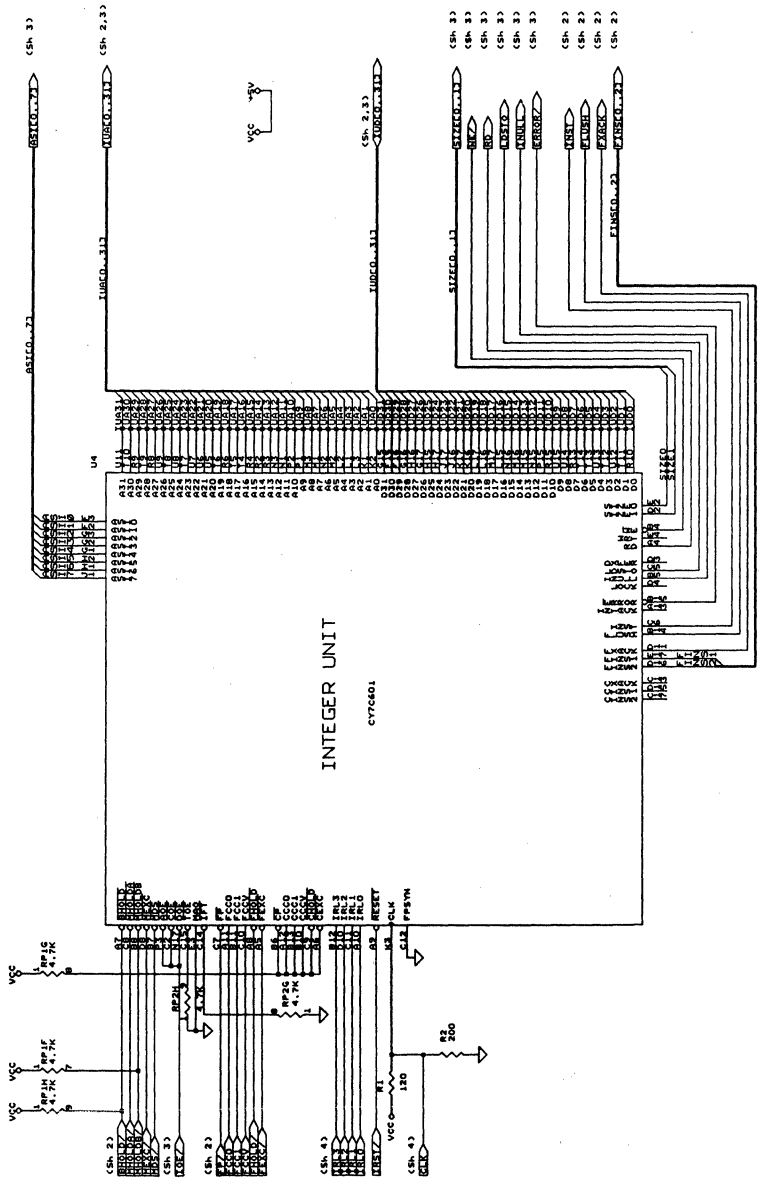
There are various pull-up and pull-down resistors on the Uni-Module board in order to improve operation, testability, and to allow the removal of the FPU. Multiple 0.1-μF ceramic decoupling capacitors are placed around each chip to provide power for instantaneous, high-frequency current requirements. Multiple 22-μF tantalum decoupling capacitors are placed near the Mbus connector and at the board edges to help provide a stable, low-frequency, low-impedance power source.

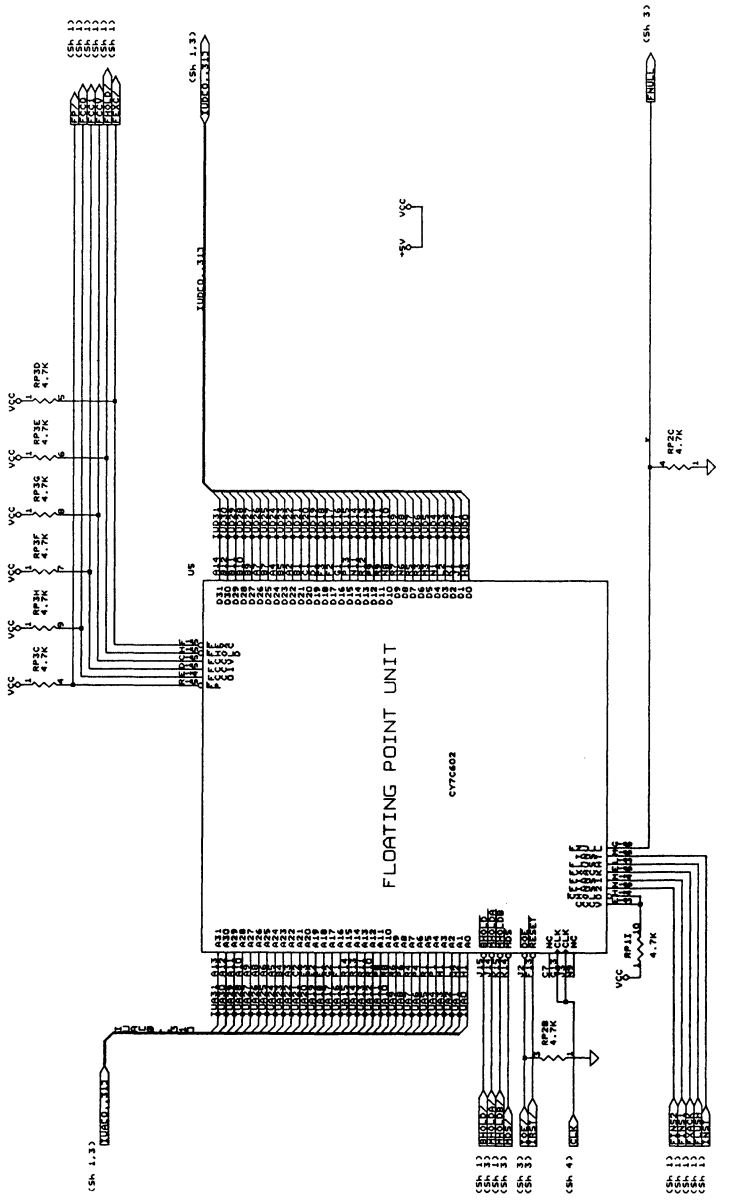
$\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ lines have pull-up resistors on them since these are three-state lines driven by the CMU. The CLK line from the Mbus connector is parallel terminated at its end (FPU) by a Thevinin equivalent of 75 ohms, since this is the design impedance of the board.

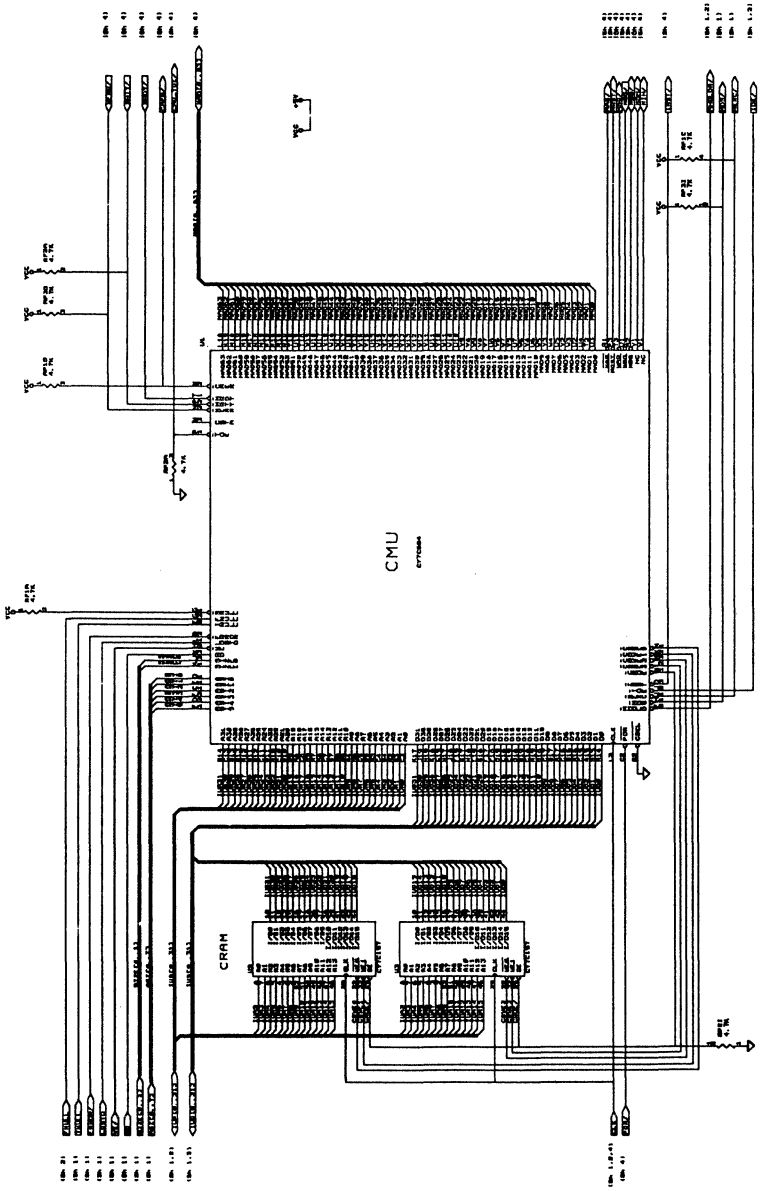
There is a pull-down resistor on the $\overline{\text{TOE}}$ pin of the IU ($\overline{\text{DOE}}$ pin of the FPU, $\overline{\text{OE}}$ pin of the CRAMs) so that the outputs are always enabled except when three-stated by the CMU via the $\overline{\text{IOE}}$ ($\overline{\text{CROE}}$) signal. There is a similar pull-down resistor on the $\overline{\text{TOE}}$ pin of the CMU so that during board test these IC's can be three-stated. The FNULL line from the FPU has a pull-down resistor on it so that if an FPU is not present, the IU and CMU will still operate correctly. The $\overline{\text{IFT}}$ pin of the IU has a pull-down resistor on it so that the execution of an IFLUSH instruction will cause an illegal_instruction trap. There are pull-up resistors on all of the coprocessor lines ($\overline{\text{MHOLDB}}$, $\overline{\text{BHOLD}}$, $\overline{\text{SNULL}}$), the floating point lines from the FPU to the IU, and on the $\overline{\text{CMER}}$, $\overline{\text{MERR}}$ and $\overline{\text{MRTY}}$ lines of the CMU.

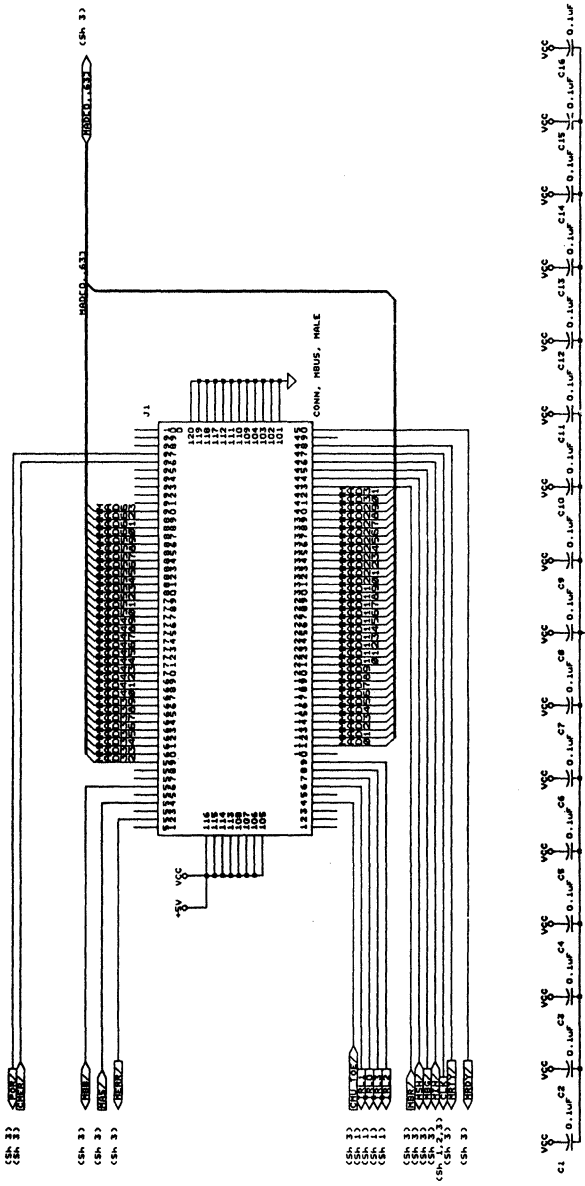
The CMU is always selected because its chip select pin is tied to GND. The MAO pin of the IU is similarly grounded to prevent falsely switching the internal source address mux of the IU.


MBUS INTERFACE
SYSTEM/TEST LOGIC
SYSTEM ADDR, DATA, & CNTL BUSSES
SYSTEM ARBITRATION LOGIC
MBUS RESET PWR ON RESET
SYSTEM INTERRUPT LOGIC
SYSTEM CLOCK
NOTES: MDS/ and MEXC/ pull-up res. req'd
CY7C604 CMU
CY7C601 IU
CY7C602 FPU
64K-BYTE CACHE
CY7C157 CRAM #1
CY7C157 CRAM #2
VCC


INTEGER UNIT
CY7C601
U4
VCC
3V
RP1G 4.7K
RP1F 4.7K
RP1H 4.7K
RP2A 2.2K
RP2G 4.7K
R1 120
R2 200


FLOATING POINT UNIT
CY7C602
U5
RP3D 4.7K
RP3E 4.7K
RP3G 4.7K
RP3F 4.7K
RP3H 4.7K
RP3C 4.7K
RP2C 4.7K
RP11 4.7K
RP28 4.7K
VCC
5V


CMU
CRAM


CONN, MBUS, MALE
J1
MADR[0..6]
MDATA[0..7]
VSS
VCC
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10
C11
C12
C13
C14
C15
C16
C17
C18
C19
C20
C21
0.1uF
22uF

Glossary

Address Translation Cache (ATC): The ATC is a cache of address translation entries used by an MMU to translate virtual addresses to physical addresses. The CY7C604/605 uses an ATC for address translation, but the more familiar term translation lookaside buffer (TLB) is used throughout the text.

Aliasing: Mapping two or more virtual addresses to the same physical address. SPARC software conventions permit the use of aliases in address spaces that are modulo with respect to the system's underlying cache size.

Annul bit: This bit is used in the SPARC architecture to allow the designer or compiler to decide whether or not the delay slot instruction of a delay control transfer instruction will be executed if the conditional branch is taken. See Section 2.3.3.4 for further information.

Cache controller: Provides cache memory access control for a 64-kbyte direct-mapped virtual cache.

Cache lock: A mechanism that allows the system to lock all entries in the cache, supported by the CY7C604. This feature allows deterministic response times for real-time systems.

Content addressable memory (CAM): A memory that is accessed by supplying the value to be compared to the memory contents. When accessed, the CAM returns the location of the memory where the value is stored, or returns a no-match signal if the memory does not contain the value. In the case of the CY7C604/605 MMU, the value returned by the CAM array is used to address a value in the TLB RAM array, which in turn provides the physical translation value to be used by the MMU.

Copy-back mode: A style of cache management in which write accesses are written to the cache only, not to main memory.

Current window: The block of 24 *r* registers pointed to by the current window pointer.

Current window pointer (CWP): Selects the current register window.

Delay instruction: The instruction immediately following a control transfer instruction. This instruction is always fetched, and is either executed or annulled before the control transfer takes place.

Double-precision floating point: A data type consisting of 64 bits.

Doubleword: A data type consisting of two 32-bit words used as a single 64-bit operand. A doubleword is always aligned with the most significant word at an even word boundary (bits 2–0 equal to zero). The subsequent least significant word is on an odd word boundary (bit 2 equal to one, bits 1–0 equal to zero).

Extended-precision floating point: A data type consisting of 128 bits.

***f* register:** One of the FPU's 32 working registers.

Floating-point unit (FPU): The coprocessor that performs floating-point calculations.

Floating-point operate (FPop) instruction: Instructions that perform floating-point calculations. This category does not include loads and stores between the memory and the FPU.

Floating-point queue (FQ): A three-deep storage area for FPop instructions and their addresses while they are being executed in the FPU. Floating-point exception traps occur sometime after the floating-point instruction is issued, asynchronously to the IU and its pipeline. The queue supplies instruction/address pair information to the IU for the FPop that caused the exception.

Frame pointer: The pointer to the beginning of a memory stack. The frame pointer is often specific to a window, and is set from the stack pointer of the previous window.

Global registers: A block of eight registers within the register file that are always available to the IU regardless of the value of the CWP.

Halfword: A data type consisting of 16 bits.

Integer unit (IU): The main computing engine. It fetches all instructions and executes all but the FPop and CPop instructions.

Mbus: The interface between a SPARC processing module and the memory subsystem.

Load/Store: The class of instructions that are either load or store instructions.

Load-Store: The class of instructions that are atomic (indivisible or locked) load THEN store. These instructions are typically used for the manipulation of multiprocessor semaphores or any other process where interruption during the process of loading a variable and storing a new value for that variable could be disastrous. The SPARC load-store instructions are: SWAP, SWAPA, LDSTUB, and LDSTUBA.

Next program counter (nPC): Contains the address of the next instruction to be executed, assuming no trap occurs.

Processor state register (PSR): The IU's status register.

Program counter: Contains the address of the current instruction being executed by the IU.

***r* register**: A global register or a register in the current window of the register file.

Register window: A group of 24 working registers from the set of window r registers (128 window registers or eight windows are available on the CY7C601/611). Register windows overlap by eight registers, causing three types of window registers: *ins, outs, and locals.* Ins are the window registers that were the outs for the previous window. Locals are specific to the register window, and are not shared. See Section 2.2 for further information.

rd, rs1, and rs2: Instruction format fields which specify the register operands of an instruction. rd is the destination register and rs1 and rs2 are the source registers.

RISC: An acronym that stands for Reduced Instruction Set Computer.

r[rd], r[rs1], and r[rs2]: The actual *r* registers specified by rd, rs1, and rs2.

Page table entry (PTE): An address mapping for a single 4-kbyte page, a 256-kbyte region, a 16-Mbyte region, or a 4-Gbyte region.

Page table pointer (PTP): The address pointer used to identify the beginning of a page table in memory.

Page table pointer cache (PTPC): The cache of page table pointers stored by the CY7C604/605 in order to minimize the levels of table walks required for a TLB miss. See Section 4.1 for further details.

SPARC: An acronym that stands for Scalable Processor ARChitecture.

Stack pointer: The pointer to the next address in memory that registers are temporarily stored, typically in response to a procedure call or trap routine.

Table walk: The process of accessing levels of tables in memory to find a page table entry for a particular virtual address. Each level of the table either has a pointer to the next level of table, or has the page table entry. Upon finding a page table entry, the table walk is terminated by the MMU.

Translation lookaside buffer (TLB): Acts as a cache for address mapping entries used by the MMU to map a virtual address to a physical address.

Virtual cache: Refers to the direct addressing of the cache by the integer unit using the virtual address bus.

Word: A data type consisting of 32 bits.

Write-through mode: A style of cache management that causes write accesses to the cache to be written through to main memory upon each write access.

Index

A

ADD, 6–7

ADDcc, 6–8

ADDX, 6–9

ADDXcc, 6–10

AFAR (asynch. fault addr. reg.), 4–40

AFSR (asynch. fault status reg.), 4–40

AND, 6–11

ANDcc, 6–12

ANDN, 6–13

ANDNcc, 6–14

annul bit, 2–22, 2–26, 2–56

ASI
- CY7C601/611, 2–19
- CY7C604/605 asi mapping, 4–46 to 4–47
- CY7C604/605 signal, 4–55
- CY7C611 ASI, 2–87
- signal description, 2–45
- Use of in instructions, 2–19

assembly language, 6–1 to 6–3

B

BHOLD, 2–46, 2–58, 2–67, 3–11, 3–24

Bicc, 2–22, 2–26, 2–56, 6–15

big endian, 2–14

C

cache controller, 4–15 to 4–34
- cache flushing, 4–32
- cacheable/non-cacheable, 4–33
- control signals, 4–30 to 4–31
- CY7C604, 4–16 to 4–19
  - aliasing, 4–17 to 4–19
  - cache locking, 4–19
  - cache tag, 4–17

cache controller (continued)
- CY7C605, 4–20 to 4–30
  - aliasing, 4–29 to 4–30
  - bus snooping, 4–29
  - cache state transitions, 4–23 to 4–29
  - cache tag, 4–21 to 4–22
    - MPTAG, 4–21
    - PVTAG, 4–21
  - multiprocessing support, 4–22 to 4–23
- LDSTO cycles, 4–34
- MC (Mbus cacheable bit), 4–33
- modes, 4–15
- read buffer, 4–32
- write buffer, 4–31

CALL, 2–8, 2–15, 2–18, 2–22, 2–25, 2–56, 6–17

CBccc, 2–22, 2–26, 2–56, 6–18

CEXC, 2–49, 2–78

CHOLD, 2–49, 2–84, 3–11, 3–24

context switching, 2–6

control registers, 2–8

coprocessor interface, 2–84

CPop, 2–30, 2–66, 6–20

CTPR (context table pointer reg.), 4–37

current window pointer. *See* CWP

CWP, 2–3 to 2–4, 2–5, 2–10, 2–11, 2–23

CXR (context register), 4–37

CY7C601/611 registers, 2–2 to 2–15
- control/status registers, 2–8 to 2–12
- PC and nPC, 2–8
- PSR, 2–9 to 2–10
- r registers, 2–2 to 2–8
- TBR, 2–11
- WIM, 2–11
- Y register, 2–11

CY7C602 registers, 3–12 to 3–15
- f registers, 3–12
- FP queue, 3–13
- FSR, 3–14 to 3–15

CY7C604/605 diagnostics
- cache data entries, 4–44
- cache tag entries, 4–44
- TLB entries, 4–43

CY7C604/605 Multichip, 4–41 to 4–43

CY7C604/605 registers, 4–35 to 4–40
'604 system control register, 4–35 to 4–36
'605 system control register, 4–36
asynchronous fault address (AFAR), 4–40
asynchronous fault status (AFSR), 4–40
context register (CXR), 4–37
context table pointer (CTPR), 4–37
data access PTP (DPTP), 4–38
index tag register (ITR), 4–38
instruction access PTP (IPTP), 4–38
reset register (RR), 4–37
root pointer register (RPR), 4–38
synchronous fault address (SFAR), 4–40
synchronous fault status (SFSR), 4–39
TLB replacement control (TRCR), 4–39

CY7C604/605 reset, 4–45 to 4–46
Power-on reset, 4–45
Software External reset, 4–45
Software Internal reset, 4–45
Watch-dog reset, 4–45

CY7C604/605 synchronous faults, 4–47 to 4–54

D

delayed control transfer, 2–25, 2–56

delayed control transfer couples, 2–27 to 2–29

DPTP (data PTP), 4–38

E

ERROR
signal, 2–8, 2–51, 4–56
state, 2–8
timing, 2–76 to 2–77

F

f registers, 3–12

FABSs, 6–21

FADDd, 6–22

FADDs, 6–23

FADDx, 6–24

FBfcc, 2–22, 2–26, 2–56, 6–25

FCMP, 2–22, 3–9

FCMPd, 6–27

FCMPE, 2–22

FCMPEd, 6–28

FCMPEs, 6–29

FCMPEx, 6–30

FCMPs, 6–31

FCMPx, 6–32

FDIVd, 6–33

FDIVs, 6–34

FDIVx, 6–35

FdTOi, 6–36

FdTOs, 6–37

FdTOx, 6–38

FEXC, 2–50, 2–75, 2–78, 3–23

FHOLD, 2–50, 3–11, 3–23

FINS1/2, 3–7, 3–23

FiTOd, 6–39

FiTOs, 6–40

FiTOx, 6–41

floating-point
double-precision, 2–12, 3–20
exceptions, 3–17, 3–22
extended-precision, 2–14, 3–21
interface, 3–4
operate instr., 3–16
queue, 3–9, 3–13
single-precision, 2–12, 3–20
status register (FSR), 3–14

FLUSH, 2–51, 3–9, 3–23

FMOVs, 6–42

FMULd, 6–43

FMULs, 6–44

FMULx, 6–45

FNEGs, 6–46

FNULL, 3–12, 3–24, 4–56

FP Queue, 3–13

FPop, 2–30, 2–66

FPops, 3–16

frame pointer, 2–4, G-2

FSQRTd, 6–47

FSQRTs, 6–48

FSQRTx, 6–49

FSR (FP status register), 3–14 to 3–15

FsTOd, 6–50

FsTOi, 6–51

FsTOx, 6–52
FSUBd, 6–53
FSUBs, 6–54
FSUBx, 6–55
FXACK, 2–51, 2–75, 3–23
FxTOd, 6–56
FxTOi, 6–57
FxTOs, 6–58

H

hardware interlocks, 2–56

I

IFLUSH, 2–30, 6–59
INST, 3–23
instruction
  arithmetic/logical/shift, 2–20
  control transfer, 2–22
  delay, 2–26, 2–56
  delayed control transfer, 2–25, 2–56
  fetch, 2–61
  floating-point, 3–16
  formats, 2–15
  FP inst. fetch, 3–6
  load, 2–61, 2–62
  load/store, 2–19
  load-store, 2–20
  mnemonics, 6–3
  multiprocessing, 2–20
  op codes, 2–31 to 2–45
  pipeline, 2–52 to 2–53, 3–5
  store, 2–63, 2–64
  summary table, 6–6
  types, 2–19 to 2–30
integer condition codes (icc), 2–9
INULL, 2–47, 4–56
IOP, 2–53
IPTP (instruction PTP reg.), 4–38
ITR (index tag reg.), 4–38

J

JMPL, 2–4, 2–8, 2–22, 2–53, 2–56, 6–60
JMPL, RETT, 2–8, 2–23, 2–27, 6–91

L

LD, 6–61
LDA, 6–62
LDC, 6–63
LDCSR, 6–64
LDD, 6–65
LDDA, 6–66
LDDC, 6–67
LDDF, 6–68
LDF, 6–69
LDFSR, 6–70
LDSB, 6–71
LDSBA, 6–72
LDSH, 6–73
LDSHA, 6–74
LDSTO
  '604/'605 operation, 4–34
  CY7C604/605 signal, 4–56
  signal, 2–47
  timing, 2–65
LDSTO instructions, 2–20
LDSTUB, 2–20, 6–75
LDSTUBA, 6–76
LDUB, 6–77
LDUBA, 6–78
LDUH, 6–79
LDUHA, 6–80
load-store. *See* instruction, load–store; LDSTO

M

Mbus, 4–84 to 4–112
  address cycle, 4–87 to 4–88
  burst transactions, 4–86
  data cycle(s), 4–88
  Level 1, 4–84
  Level 2, 4–84 to 4–85
  MAD bus, 4–57
  MAS signal, 4–57
  MBB signal, 4–57
  MBG signal, 4–58
  MBR signal, 4–58
  MERR signal, 4–58
  MIH signal, 4–58
  MRDY signal, 4–58

Mbus, (continued)
MRST signal, 4–58
MRTY signal, 4–58
MSH signal, 4–58
non-burst transactions, 4–86
relinquish and retry, 4–86
retry, 4–86
signal summary, 4–85 to 4–87
transactions, 4–88 to 4–92
coherent invalidate, 4–90
coherent read, 4–89 to 4–90
coherent read and invalidate, 4–90 to 4–91
coherent write and invalidate, 4–91 to 4–112
read, 4–88 to 4–89
write, 4–89
MDS, 2–47, 3–24, 4–56
memory stack, 2–4
MEXC, 2–47, 2–71, 2–78, 4–56
MHOLD, 2–48, 2–58, 2–68 to 2–74, 3–11, 3–24, 4–56
MMU, 4–3 to 4–12
flush, 4–14 to 4–15
operation modes, 4–13 to 4–15
probe, 4–15
MULScc, 2–11, 6–81

O

OR, 6–82
ORcc, 6–83
ORN, 6–84
ORNcc, 6–85

P

page table entry. *See* PTE
page table pointer. *See* PTP
page table pointer cache. *See* PTPC
PC and nPC, 2–8, 2–26
POR, 4–58
processor interrupt level (PIL), 2–10
processor state register. *See* PSR
processor states, 2–8
PSR, 2–9 to 2–10
PTE, 4–3, 4–4, 4–10 to 4–11
PTP, 4–9 to 4–10
PTPC, 4–11

R

r registers, 2–2 to 2–8
ins, 2–3 to 2–4, 2–25 to 2–26
locals, 2–3 to 2–4, 2–25 to 2–26
outs, 2–3 to 2–4, 2–25 to 2–26
r[0], 2–21
special *r* registers, 2–6 to 2–8
RDPSR, 2–10, 2–30, 6–86
RDTBR, 2–11, 2–30, 6–87
RDWIM, 2–11, 2–30, 6–88
RDY, 2–30, 6–89
register windows, 2–3 to 2–6
RESET
See also CY7C604/605 reset
signal, 2–8, 2–52, 3–24
state, 2–8, 2–78
timing, 2–76
RESTORE, 2–4, 2–6, 2–10, 2–11, 2–22, 2–25, 6–90
RETT, 2–6, 2–8, 2–10, 2–11, 2–22, 2–53, 2–84, 6–91
RPR (root pointer reg.), 4–38
RR (reset register), 4–37

S

SAVE, 2–4, 2–5, 2–6, 2–11, 2–22, 2–25, 6–93
SCR ('604 system control reg.), 4–35 to 4–36
SCR ('605 system control register), 4–36 to 4–37
SETHI, 2–15, 2–21, 6–94
SFAR (synch. fault addr. reg.), 4–40
SFSR (synch. fault status reg.), 4–39
SLL, 6–95
SNULL, 4–56
SRA, 6–96
SRL, 6–97
ST, 6–98
STA, 6–99
stack pointer, 2–4, G-3
STB, 6–100
STBA, 6–101
STC, 6–102
STCSR, 6–103

STD, 6–104
STDA, 6–105
STDC, 6–106
STDCQ, 6–107
STDF, 6–108
STDFQ, 6–109
STF, 6–110
STFSR, 2–22, 6–111
STH, 6–112
STHA, 6–113
SUB, 6–114
SUBcc, 6–115
SUBX, 6–116
SUBXcc, 6–117
supervisor
  bit, 2–10
  mode, 2–8
SWAP, 2–20, 6–118
SWAPA, 6–119

T

table walk, 4–8 to 4–9
TADDcc, 2–22, 6–120
TADDccTV, 2–22, 6–121
tagged arithmetic, 2–22
tagged data, 2–13, 2–22
TBR, 2–8, 2–11, 2–23, 2–83
Ticc, 2–11, 2–22, 2–23, 6–122
TLB, 4–4
  entries, 4–6, 4–43
  locking, 4–6
  look-up, 4–6
  table walk, 4–8 to 4–9
translation lookaside buffer. *See* TLB
trap, 2–78 to 2–84
  addressing, 2–83
  asynchronous, 2–78
trap (continued)
  floating-point, 3–17, 3–22
  FP/CP, 2–81 to 2–82
  IEEE exceptions, 3–22
  interrupts, 2–75, 2–80 to 2–81
  operation, 2–82
  pipeline timing, 2–58
  reset, 2–78
  synchronous, 2–78
  types, 2–78 to 2–80, 2–83
trap base register. *See* TBR
TRCR (TLB replacement control reg.), 4–39
TSUBcc, 2–22, 6–124
TSUBccTV, 2–22, 6–125

U

UNIMP, 2–30, 6–126
user mode, 2–8

W

WIM, 2–5, 2–6, 2–8, 2–11, 2–25
window overflow and underflow, 2–5, 2–11
  See also WIM
windows. *See* register windows
WRPSR, 2–10, 2–22, 2–30, 6–127
WRTBR, 2–11, 2–30, 6–128
WRWIM, 2–11, 2–30, 6–129
WRY, 2–30, 6–130

X

XNOR, 6–131
XNORcc, 6–132
XOR, 6–133
XORcc, 6–134

Y

Y register, 2–11

Cypress Semiconductor
3901 North First Street
San Jose, CA 95134
(408) 943-2600