WIRELESS & µC

TSC691E
Integer Unit

User's Manual

for Embedded Real time 32–bit Computer

(ERC32)

for SPACE Applications

Table of contents

1. Introduction 1

2. TSC691E OVERVIEW 2
2.1. SPARC RISC STANDARD FUNCTIONS : 2
2.2. Fault Tolerant and Test Mechanism Improvements: 2
2.3. Presentation of the ERC32 computing core 2
2.3.1. Concept 2
2.3.2. Functional Description 3

3. Standard IU Function 4
3.1. Introduction 4
3.2. Description Of Parts 5
3.3. Programming Model 5
3.3.1. Register Windows 5
3.3.1.1. Windowing 6
3.3.1.1.1. Parameter Passing 7
3.3.1.1.2. Window Overflow and Underflow 8
3.3.1.1.3. Alternate Register Window Usage 9
3.3.1.2. Special Registers 9
3.3.2. Processor States 11
3.3.3. Supervisor/User Modes 11
3.3.4. Control/Status Registers 11
3.3.4.1. Program Counters (PC and nPC) 11
3.3.4.2. Processor State Register (PSR) 12
3.3.4.3. Window Invalid Mask Register (WIM) 14
3.3.4.4. Trap Base Register (TBR) 14
3.3.4.5. Y Register 15
3.3.5. Data Types 15
3.3.5.1. Data Organization In Registers 15
3.3.5.2. Data Organization In Memory 17
3.3.5.3. Extended Precision 17
3.4. Instruction Set 18
3.4.1. Instruction Formats 18
3.4.2. Addressing 20
3.4.2.1. Two–Register 20
3.4.2.2. Register Plus 13-Bit Immediate 20
3.4.2.3. 13-Bit Immediate 20
3.4.2.4. CALL 21
3.4.2.5. Branch 21
3.4.2.6. ASI 22
3.4.3. Instruction Types 22
3.4.3.1. Load/Store 22
3.4.3.1.1. ASI 22
3.4.3.1.2. Multiprocessing Instructions 23
3.4.3.2. Arithmetic/Logical/Shift 24
3.4.3.2.1. Register r[0] 24
3.4.3.2.2. SETHI 25
3.4.3.2.3. Tagged Arithmetic 25

3.4.3.3. Control Transfer 25
3.4.3.3.1. Branching and the Condition Codes 26
3.4.3.3.2. Trap Instructions 27
3.4.3.3.3. Calls and Returns 28
3.4.3.4. Delayed Control Transfer 29
3.4.3.4.1. PC and nPC 29
3.4.3.4.2. Delay Instruction 29
3.4.3.4.3. Annul Bit 29
3.4.3.4.4. Delayed Control Transfer Couples 30
3.4.3.5. Read/Write Control Registers 33
3.4.3.6. Floating-Point-Operate and Coprocessor-Operate 33
3.4.3.7. Miscellaneous 33
3.4.4. Op Codes 33
3.4.4.1. Load/Store Instructions 34
3.4.4.2. Arithmetic/Logical/Shift Instructions 36
3.4.4.3. Control Transfer Instructions 38
3.4.4.4. Read/Write Control Register Instructions 39
3.4.4.5. Floating-Point/Coprocessor Instructions 40
3.4.4.6. Miscellaneous Instructions 41
3.4.4.7. Opcodes In Ascending Numeric Order 42
3.5. Signal Description 48
3.5.1. Memory Subsystem Interface Signals 51
3.5.1.1. A[31:0]—Address Bus (output) 51
3.5.1.2. APAR—Address Bus Parity (output) 51
3.5.1.3. $\overline{\text{AOE}}$—Address Output Enable (input) 51
3.5.1.4. ASI[7:0]—Address Space Identifier (output) 51
3.5.1.5. ASPAR—ASI and SIZE Parity (output) 52
3.5.1.6. $\overline{\text{BHOLD}}$—Bus Hold (input) 52
3.5.1.7. $\overline{\text{COE}}$—Control Output Enable (input) 52
3.5.1.8. D[31:0]—Data Bus (bidirectional) 52
3.5.1.9.DPAR—Data Bus Parity (bidirectional) 53
3.5.1.10. $\overline{\text{DOE}}$—Data Output Enable (input) 53
3.5.1.11. DXFER—Data Transfer (output) 53
3.5.1.12. $\overline{\text{IFT}}$—Instruction Cache Flush Trap (input) 53
3.5.1.13. INULL—Integer Unit Nullify Cycle (output) 53
3.5.1.14. LDSTO—Atomic Load–Store (output) 53
3.5.1.15. LOCK—Bus Lock (output) 54
3.5.1.16. MAO—Memory Address Output (input) 54
3.5.1.17. $\overline{\text{MDS}}$—Memory Data Strobe (input) 54
3.5.1.18. $\overline{\text{MEXC}}$—Memory Exception (input) 54
3.5.1.19.$\overline{\text{MHOLD}}$(A/B)—Memory Holds (inputs) 54
3.5.1.20. RD—Read Access (output) 54
3.5.1.21. SIZE[1:0]—Bus Transaction Size (outputs) 55
3.5.1.22. $\overline{\text{WE}}$—Write Enable (output) 55
3.5.1.23. WRT—Advanced Write (output) 55
3.5.1.24. IMPAR—IU to MEC Control Parity (output) 55
3.5.2. Floating-Point/Coprocessor Interface Signals 55
3.5.2.1. CCC[1:0]—Coprocessor Condition Codes (input) 55
3.5.2.2. CCCV—Coprocessor Condition Codes Valid (input) 56
3.5.2.3. $\overline{\text{CEXC}}$—Coprocessor Exception (input) 56
3.5.2.4. $\overline{\text{CHOLD}}$—Coprocessor Hold (input) 56

3.5.2.5. CINS1—Coprocessor Instruction in Buffer 1 (output) 56
3.5.2.6. CINS2—Coprocessor Instruction in Buffer 2 (output) 56
3.5.2.7. $\overline{\text{CP}}$—Coprocessor Unit Present (input) 56
3.5.2.8. CXACK—Coprocessor Exception Acknowledge (output) 56
3.5.2.9. FCC[1:0]—Floating-Point Condition Codes (input) 56
3.5.2.10. FCCV—Floating-Point Condition Codes Valid (input) 56
3.5.2.11. $\overline{\text{FEXC}}$—Floating-Point Exception (input) 57
3.5.2.12. $\overline{\text{FHOLD}}$—Floating-Point Hold (input) 57
3.5.2.13. FIPAR—FPU to IU Control Parity (input) 57
3.5.2.14. FINS1—Floating-Point Instruction In Buffer 1 (output) 57
3.5.2.15. FINS2—Floating-Point Instruction In Buffer 2 (output) 57
3.5.2.16. FLUSH—Floating-Point/Coprocessor Instruction Flush (output) 57
3.5.2.17. $\overline{\text{FP}}$—Floating-Point Unit Present (input) 57
3.5.2.18. FXACK—Floating-Point Exception Acknowledge (output) 57
3.5.2.19. INST—Instruction Fetch (output) 58
3.5.2.20. IFPAR—IU to FPU Control Parity (output) 58
3.5.3. Interrupt and Control Signals 58
3.5.3.1. $\overline{\text{ERROR}}$—Error State (output) 58
3.5.3.2. $\overline{\text{HWERROR}}$—Hardware Error (output) 58
3.5.3.3. $\overline{\text{FLOW}}$ —Enable Flow Control (input) 58
3.5.3.4. $\overline{\text{MCERR}}$—Comparison Error (output) 58
3.5.3.5. $\overline{\text{601MODE}}$—Normal 601 Mode Operation (input) 58
3.5.3.6. $\overline{\text{CMODE}}$—Checker Mode (input) 58
3.5.3.7. FPSYN—Floating-Point Synonym Mode (input) 58
3.5.3.8. INTACK—Interrupt Acknowledge (output) 59
3.5.3.9. IRL[3:0]—Interrupt Request Level (input) 59
3.5.3.10. $\overline{\text{RESET}}$—Integer Unit reset (input) 59
3.5.3.11. $\overline{\text{TOE}}$—Test Mode Output Enable (input) 59
3.5.3.12. $\overline{\text{HALT}}$—Halt (input) 59
3.5.4. TAP signals 59
3.5.4.1. TCLK—Test Clock (input) 59
3.5.4.2. $\overline{\text{TRST}}$—TEST Reset (input) 59
3.5.4.3. TMS—Test Mode Select (input) 59
3.5.4.4. TDI—Test Data Input (input) 59
3.5.4.5. TDO—Test Data Output 60
3.5.5. Power and Clock Signals 60
3.5.5.1. CLK—Clock (input) 60
3.5.5.2. VCCO, VCCI, VCCT—Power (inputs) 60
3.5.5.3. VSSO, VSSI, VSST—Ground (inputs) 60
3.6. Pipeline and Instruction Execution Timing 60
3.6.1.Stages 61
3.6.1.1. Internal Opcodes 61
3.6.2. Multicycle Instructions 62
3.6.2.1. Register Interlocks 64
3.6.2.2. Branching 65
3.6.3. Pipeline Freezes 66
3.6.4. Traps 66
3.7. Bus Operation and Timing 67
3.7.1. Instruction Fetch 69
3.7.2. Load 70
3.7.3. Load with Interlock 70

3.7.4. Load Double 71
3.7.5. Store 72
3.7.6. Store Double 73
3.7.7. Atomic Load–Store 74
3.7.8. Floating-Point Operations 75
3.7.9. Bus Arbitration 76
3.7.10. Load with Cache Miss 77
3.7.11. Store with Cache Miss 78
3.7.12. Load/Store instruction with Trap 80
3.7.13. Memory Exceptions 81
3.7.14. Floating-Point Exceptions 91
3.7.15. Interrupts 91
3.7.16. Reset Condition 92
3.7.17. Error Condition 92
3.8. Exception Model 94
3.8.1. Reset 94
3.8.2. Synchronous Traps 94
3.8.2.1. External Signals 94
3.8.2.1.1. Hardware error 95
3.8.2.1.2. Instruction access exeption 95
3.8.2.1.3. Data access exception 95
3.8.2.2. Internal/Software 95
3.8.2.2.1. Illegal instruction 95
3.8.2.2.2. Privileged instruction 95
3.8.2.2.3. Fp disabled 95
3.8.2.2.4. Cp disabled 95
3.8.2.2.5. Window overflow 95
3.8.2.2.6. Window underflow 96
3.8.2.2.7. Memory address not aligned 96
3.8.2.2.8. Tag overflow 96
3.8.2.2.9. Trap instruction 96
3.8.3. Interrupts (Asynchronous Traps) 97
3.8.3.1. Priority 97
3.8.3.2. Response Time 97
3.8.3.3.Interrupt Acknowledge 99
3.8.4. Floating-Point/Coprocessor Traps 99
3.8.4.1. Floating-Point Exception 100
3.8.4.2. Coprocessor Exception 100
3.8.5. Trap Operation 100
3.8.5.1. Recognition 100
3.8.5.2. Trap Addressing 100
3.8.5.3. Trap Types and Priority 101
3.8.5.4. Return From Trap 101
3.9. Coprocessor Interface 102
3.9.1. Protocol 102
3.9.1.1. Coprocessor Interface Signals 102
3.9.2. Register Model 103
3.9.3. Exceptions 104

4. Fault Tolerant and Test Mechanism 105
4.1. Fault Tolerant and Test Support signals 106
4.1.1. Address Parity Generation: 106
4.1.2. Data Parity Generation/Checking: 106
4.1.3. MEC control signal Parity Generation: 106
4.1.4. FPU control signal Parity Generation/Checking: 106
4.1.5. Parity Checking Error Output: 106
4.1.6. Master/Checker Mode: 106
4.1.7. Test Access Port: 106
4.1.8. Miscellaneous: 106
4.2. Program Flow Control 107
4.2.1. Introduction 107
4.2.2. Example of Program Flow Control 107
4.3. Parity Checking 108
4.3.1. Introduction 108
4.3.2. Trap handling 108
4.3.3. Priority within hardware traps for IU 109
4.3.4. Parity Checking on Register File and Control/Status Registers 109
4.3.5. Parity Checking on Control Signal for the FPU 110
4.3.5.1. Output control signals 110
4.3.5.2. Input control signals 110
4.3.6. Parity Checking on Control Pads for the TSC693E (MEC) 110
4.3.6.1. Output control signals 110
4.3.6.2. Input control signals 110
4.3.7. Parity Checking on Control Pads for the Coprocessor 110
4.3.8. Parity Generation on ADDRESS Bus 110
4.3.9. Parity Checking on DATA Bus 111
4.3.10. Non CY7C601 Mode 111
4.3.11. Error Type for external signals parity errors 111
4.4. Master/checker operation 111
4.4.1. Basic function 111
4.4.1.1. Master/Checker Signal description 112
4.4.1.1.1. $\overline{\text{MCERR}}$—Comparison Error (output) 112
4.4.1.1.2.$\overline{\text{CMODE}}$—checker Mode (input) 112
4.5. IEEE Standard Test Access Port & Boundary-Scan Architecture 112
4.5.1. TAP 112
4.5.1.1. TCLK (input) 113
4.5.1.2. TMS (input) 113
4.5.1.3. TDI (input) 113
4.5.1.4. $\overline{\text{TRST}}$ (input) 113
4.5.1.5. TDO (output) 113
4.5.2. TAP Controller 113
4.5.3. The Instruction Register 113
4.5.3.1. Design and Construction of the instruction register 113
4.5.3.2. BYPASS Instruction 113
4.5.3.3. EXTEST Instruction 114
4.5.3.4. INTEST Instruction 114
4.5.3.5. SAMPLE/PRELOAD Instruction 114
4.5.4. The Device Identification Register 114
4.5.5. Internal Scan Path 114
4.5.6. Boundary scan test register 114

4.6. Interleaving register file bits 115

5. Electrical and Mechanical Specification 116

5.1. Maximum rating and DC Characteristics 116
5.1.1. Maximum Ratings 116
5.1.2. Operating Range 116
5.1.3. DC Characteristics Over the Operating Range 116
5.1.4. Capacitance Ratings [4, 5] 116
5.1.5. AC Test Loads and Waveforms 117
5.2. TSC691E AC Characteristics 117
5.2.1. AC Characteristics Over the Operation Range [1] 117
5.2.2. Waveforms 120
5.2.2.1. Clock and ERROR / RESET Timing 120
5.2.2.2. Clock and HWERROR Timing for Parity Error Type 120
5.2.2.3. TOE De–assertion /Assertion 121
5.2.2.4. Load Timing 121
5.2.2.5. Store Timing 122
5.2.2.6. Load with Cache Miss 123
5.2.2.7. Memory Exception Timing 124
5.2.2.8. Bus Arbitration Timing 124
5.2.2.9. Floating-Point and Coprocessor Timing 125
5.2.2.10. TAP Signals 125
5.2.2.11. PARITY Signals 126
5.2.2.12. MASTER/CHECKER Signals 126
5.2.2.13. IRL[3:0] Signals 127
5.2.2.14. HALT Signal timing 127
5.3. Package Description 128
5.3.1. 256-Pin MQFP_F Package 128
5.3.2. 256-Pin MQFP_F Pin Assignments 129

List of Tables

Table 1. Register Addressing 6
Table 2. Floating–Point Formats 15
Table 3. Extended–Precision Floating–Point Format 18
Table 4. op field Coding 20
Table 5. op2 Field Coding 20
Table 6. ASI Assignments 22
Table 7. Load/Store Instructions 23
Table 8. Arithmetic/Logical/Shift Instructions 24
Table 9. Control Transfer Instructions 26
Table 10. Control Transfer Instruction Characteristics 26
Table 11. Bicc and Ticc Condition Codes 27
Table 12. FBfcc Condition Codes 27
Table 13. CBccc Condition Codes 27
Table 14. Delayed Control Transfer Instruction Example 29
Table 15. Effect of Annul Bit Reset (a=0) 29
Table 16. Effect of Annul Bit Reset (a=1) 29
Table 17. Effect of Annul Bit on Delay Instruction 30
Table 18. Delayed Control Transfer Couple Instruction Sequence 31
Table 19. Execution of Delayed Control Transfer Couples 31
Table 20. Read/Write Control Register Instructions 32
Table 21. Floating–Point–Operate and Coprocessor–Operate Instructions 33
Table 22. Miscellaneous Instructions 33
Table 23. Load/Store Instruction Opcodes 34
Table 24. Arithmetic/Logical/Shift Instruction Opcodes 36
Table 25. Control Transfer Instruction Opcodes 38
Table 26. Bicc and Ticc Condition Codes 38
Table 27. FBfcc Condition Codes 39
Table 28. CBccc Condition Codes 39
Table 29. Read/Write Control Register Instruction Opcodes 39
Table 30. Floating–Point /Coprocessor Instruction Opcodes 40
Table 31. Miscellaneous Instruction Opcodes 41
Table 32. Instruction Opcode Numeric Listing 42
Table 33. TSC691E External Signal Summary 49
Table 34. ASI Assignments 52
Table 35. SIZE Bit Encoding 55
Table 36. Internally Generated Opcodes 62
Table 37. Externally Generated Synchronous Exception Traps 94
Table 38. Trap Type and Priority Assignments 101
Table 39. Error Type Assignments 109
Table 40. Hardware Priority 109
Table 41. Hardware error type for user registers 110
Table 42. Hardware error type for external signals 111

List of Figures

Figure 1. ERC32 Architecture . . . 3
Figure 2. Integer Unit Block Diagram . . . 4
Figure 3. SPARC Register Model . . . 5
Figure 4. Circular Stack of Overlapping Windows . . . 6
Figure 5. Overlapping Windows . . . 7
Figure 6. Registers as Seen by a Procedure . . . 8
Figure 7. Register Banks for Fast Context Switching . . . 10
Figure 8. Processor State Register . . . 12
Figure 9. Window Invalid Mask . . . 14
Figure 10. Trap Base Register . . . 14
Figure 11. Processor Data Types . . . 16
Figure 12. Byte Operand Load and Store . . . 17
Figure 13. Data Organization in Memory . . . 17
Figure 14. Extended–Precision Data Organization in Registers . . . 18
Figure 15. Extended–Precision Data Organization in Memory . . . 18
Figure 16. Instruction Format Summary . . . 19
Figure 17. Address Generation . . . 21
Figure 18. Tagged Data Example . . . 25
Figure 19. Ticc Trap Address Generation . . . 28
Figure 20. Delayed Control Transfer . . . 30
Figure 21. Delayed Control Transfer Couples . . . 32
Figure 22. TSC691E External Signals . . . 48
Figure 23. ASI timing with a WRPSR Instruction . . . 52
Figure 24. Processor Instruction Pipeline . . . 60
Figure 25. Pipeline with All Single–Cycle Instructions . . . 61
Figure 26. Pipeline with One Double–Cycle Instruction (Load) . . . 62
Figure 27. Pipeline with One Triple–Cycle Instruction (Store) . . . 63
Figure 28. Pipeline with Hardware Interlock (Load) . . . 64
Figure 29. Pipeline During Branch Instruction . . . 65
Figure 30. Branch with Annulled Delay Instruction . . . 65
Figure 31. Pipeline Frozen During Bus Arbitration . . . 66
Figure 32. Pipeline Operation for Taken Trap (Internal) . . . 67
Figure 33. Data Bus Contents During Data Transfers (1 of 2) . . . 68
Figure 34. Data Bus Contents During Data Transfers (2 of 2) . . . 69
Figure 35. Instruction Fetch . . . 69
Figure 36. Load Single Integer Timing . . . 70
Figure 37. Load Single with Interlock Timing . . . 70
Figure 38. Load Double Integer Timing . . . 71
Figure 39. Store Single Integer Timing . . . 72
Figure 40. Store Double Integer Timing . . . 73
Figure 41. Atomic Load–Store Timing . . . 74
Figure 42. Floating–Point Operation Timing . . . 75
Figure 43. Bus Arbitration Timing . . . 76
Figure 44. Load with Cache Miss Timing . . . 77
Figure 45. Store with Cache Miss Timing (1 of 2) . . . 78
Figure 46. Store with Cache Miss Timing (2 of 2) . . . 79
Figure 47. Ld, LdSt, St and Swap Inst with Trap Taken . . . 80
Figure 48. Load with Memory Exception Timing (1 of 2) . . . 81

Figure 49. Load with Memory Exception Timing (2 of 2) 82
Figure 50. Instruction Memory Access Exception Timing 83
Figure 51. Instruction Memory Access Exception Timing (LD in Execute stage) 84
Figure 52. Store with Memory Exception Timing (1 of 2) 85
Figure 53. Store with Memory Exception Timing (2 of 2) 86
Figure 54. Store double with Memory Exception on 1st data address (1 of 2) 87
Figure 55. Store double with Memory Exception on 1st data address (2 of 2) 88
Figure 56. Store double with Memory Exception on 2nd data address (1 of 2) 89
Figure 57. Store double with Memory Exception on 2nd data address (2 of 2) 90
Figure 58. Floating–Point Exception Handshake Timing 91
Figure 59. Asynchronous Interrupt Timing 91
Figure 60. Power–On Reset Timing 92
Figure 61. Error/Reset Timing 93
Figure 62. Best–Case Interrupt Response Timing (one cycle instruction) 97
Figure 63. Double Cycles Instruction Interrupt Response Timing (ex: Load) 98
Figure 64. Triple-Cycles Instruction Interrupt Response Timing (ex: Store) 98
Figure 65. Four-Cycles Instruction Interrupt Response Timing (Store Double) 98
Figure 66. Interrupt Response Timing on conditional branch instruction (B*A,a & B*cc,aNT) 99
Figure 67. Coprocessor Register Model 103
Figure 68. Fault Tolerant and Test Mechanism Block Diagram 105
Figure 69. Example of Program Flow Control 107
Figure 70. Master/Checker Configuration 112
Figure 71. Instruction Register (IR) Cell 113
Figure 72. Boundary Scan Cell 114

TSC691E RT Integer Unit

1. Introduction

This document presents the specification of the TSC691E Radiation Tolerant Integer Unit. It is organized in three main chapters:

- Standard IU (**TSC691E**) Functions (Chapter 3)
- Fault MECHANISM and Test MECHANISM (Chapter 4)
- Electrical and Mechanical Specification (Chapter 5)

Chapter 3 presents the SPARC RISC USER'S GUIDE from Cypress Semiconductor including some adaptations due to the introduction of fault tolerant MECHANISMs, without losing the full binary compatibility with the entire SPARC V7.0 application software base.

Chapter 4 and Chapter 5 deal with the new added functions introduced in the TSC691E to improve the reliability of space applications. These new functions also do not impact the SPARC V7.0 compatibility.

2. TSC691E Overview

2.1. SPARC RISC STANDARD FUNCTIONS :

- Full binary compatibility with entire SPARC V7.0 application software base
- Architecture 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

2.2. Fault Tolerant and Test Mechanism Improvements:

- Parity checking on 98.7% of the total number of latches with hardware error traps
- Parity checking of address, data pads and control pads
- Program flow control
- Master/Checker operation
- IEEE Standard Test Access Port & Boundary-Scan Architecture
- Possibility to disable the bus parity checking
- Manufactured using Space hardened 0.8 µm SCMOS RT TECHNOLOGY
- Part of the ERC32 high performance 32-bit computing core

To support applications requiring an extremely high level of reliability, the following improvements were introduced in the standard SPARC RISC processor TSC691E:

- Several independent fault detection MECHANISMs to support the design of fault tolerant systems
- Such as odd parity checking, Program Flow Control and Master/Checker operations.
- Support of sophisticated PC board level test using the IEEE Standard Test Access Port and
- Boundary Scan Architecture
- Hardening of the process by construction, applying restricted full static CMOS design rules for
- all critical blocks of the circuit such as register file, PLAs, ROMs etc...
- Hardened device processing using the 0.8 µm SCMOS-RT TECHNOLOGY.

Thanks to careful handling of the improvements, the introduced modifications have neither reduced the performance of the device nor changed the full binary compatibility with the entire SPARC V7.0 application software.

2.3. Presentation of the ERC32 computing core

The TSC691E Integer Unit is, with the TSC692E Floating Point Unit and the TSC693E Memory controller, a part of the ERC32 computing core.

2.3.1. Concept

The objective of the ERC32 is to provide a high performance 32-bit computing core, with which computers for on-board embedded real-time applications can be built. The core will be characterized by low circuit complexity and power consumption. Extensive concurrent error detection and support for fault tolerance and reconsideration will also be emphasized.

In addition to the main objective the ERC32 core should be possible to use for performance demanding research applications in deep space probes. The radiation tolerance and error masking are therefore important. For the real-time applications the system might be fail-operational rather than fail-safe. By including support for reconfiguration of the error-handling the different demands from the applications can be optimized for the best purpose in each case.

The ERC32 will be used as a building block only requiring memory and application specific peripherals to be added to form a complete on-board computer. All other system support functions are provided by the core.

2.3.2. Functional Description

The ERC32 incorporates the followings functions:

- Processor, which consists of one integer unit (IU–TSC691E) and one floating point unit (FPU–TSC692E). The processor includes concurrent error detection facilities.
- Memory controller (MEC–TSC693E), which is a unit consisting of all necessary support functions such as memory control and protection, EDAC, wait state generator, timers, interrupt handler, watch dog, UARTs and test and debug support. The unit also includes concurrent error detection facilities.
- Oscillator (optional).
- Buffers necessary to interface with memory and peripherals.

Figure 1 schematically shows the ERC32 architecture and external functions added to form a complete system.

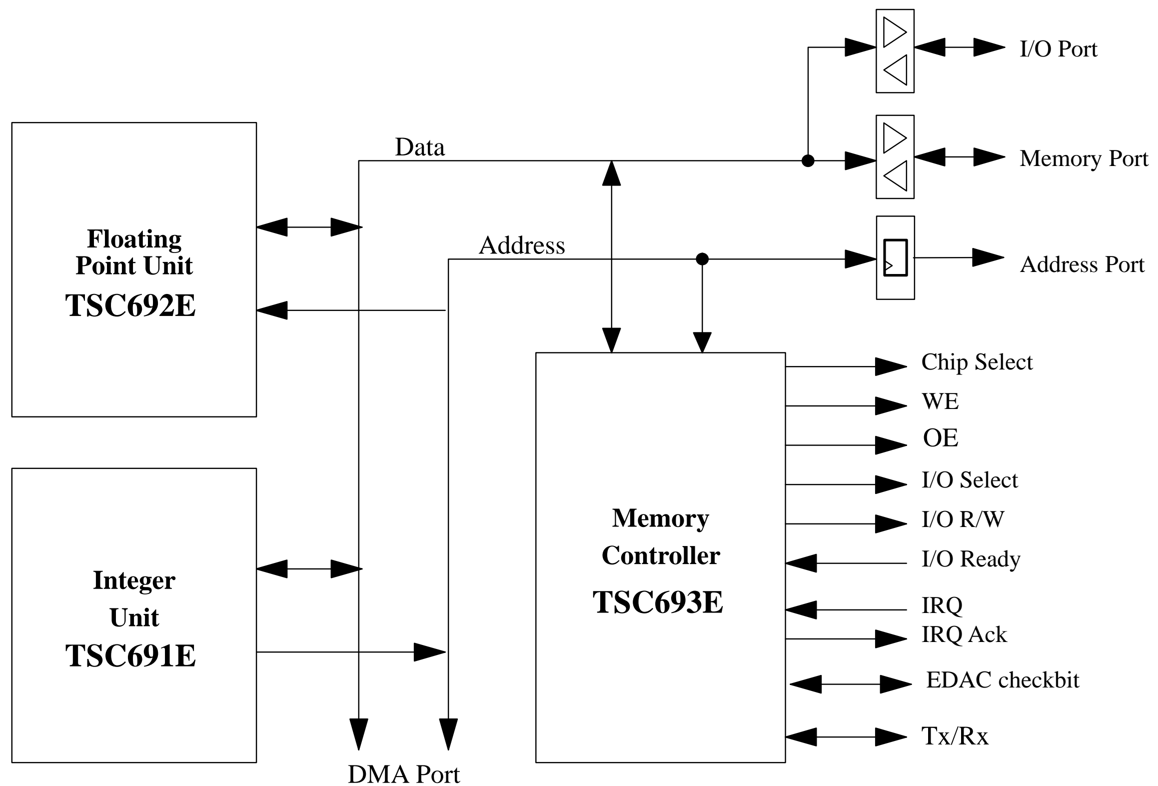


Figure 1. ERC32 Architecture

3. Standard IU Function

3.1. Introduction

This section describes the workings of the TSC691E RT Integer processing Unit (IU), the main computing engine in the SPARC architecture. The TSC691E is 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 TSC691E supports a tightly-coupled Floating-Point coprocessor Unit (FPU) and a second, system-specific coprocessor, all three of which may operate concurrently. The TSC691E executes all instructions except floating-point-operate and coprocessor-operate instructions.

A block diagram of the TSC691E is shown in Figure 2 . 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.

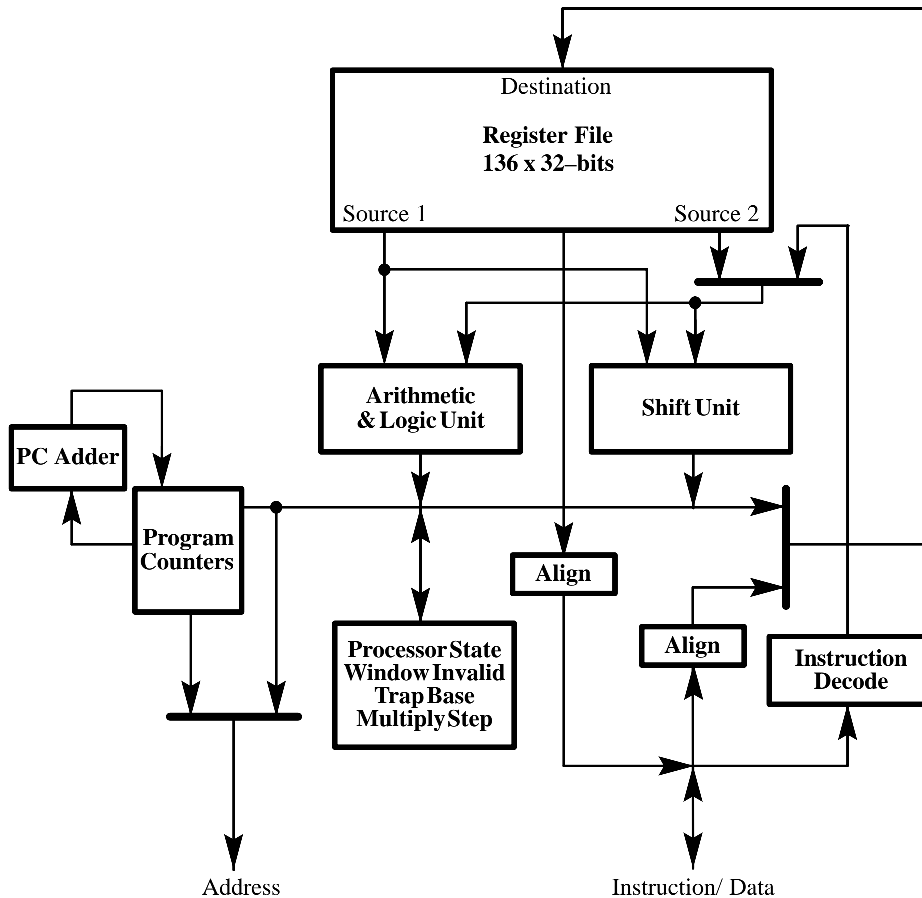


Figure 2. Integer Unit Block Diagram

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 busses 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.

3.2. Description Of Parts

The integer unit TSC691E contains a 136 x 32 register file divided into eight overlapping windows. It is supplied in 256-pins MQFP packages, which allows 32-bit address and data busses, an eight-bit ASI bus, a number of control lines, and floating-point–coprocessor, second coprocessor interfaces and 29 signals supporting fault tolerance and test MECHANISM.

3.3. Programming Model

This section describes the TSC691E'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 TSC691E's operation.

The register set shown in Figure 3 is a snapshot of the registers the TSC691E 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.

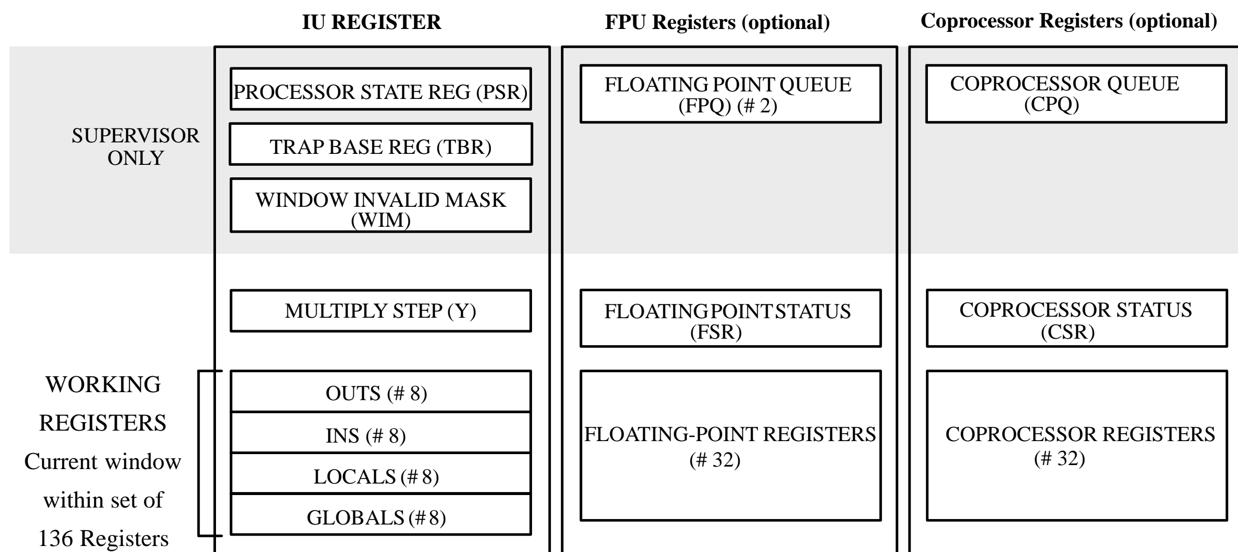


Figure 3. SPARC Register Model

Working registers are used for normal operations and are called ***r*** registers in the TSC691E, ***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.

3.3.1. Register Windows

The 136 *r* registers of the TSC691E 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 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 3.3.4.2).

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 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.

3.3.1.1. Windowing

The register file is implemented as a circular stack, with the highest numbered window joined to the lowest. In the TSC691E, window 7 adjoins window 0 (see Figure 4).

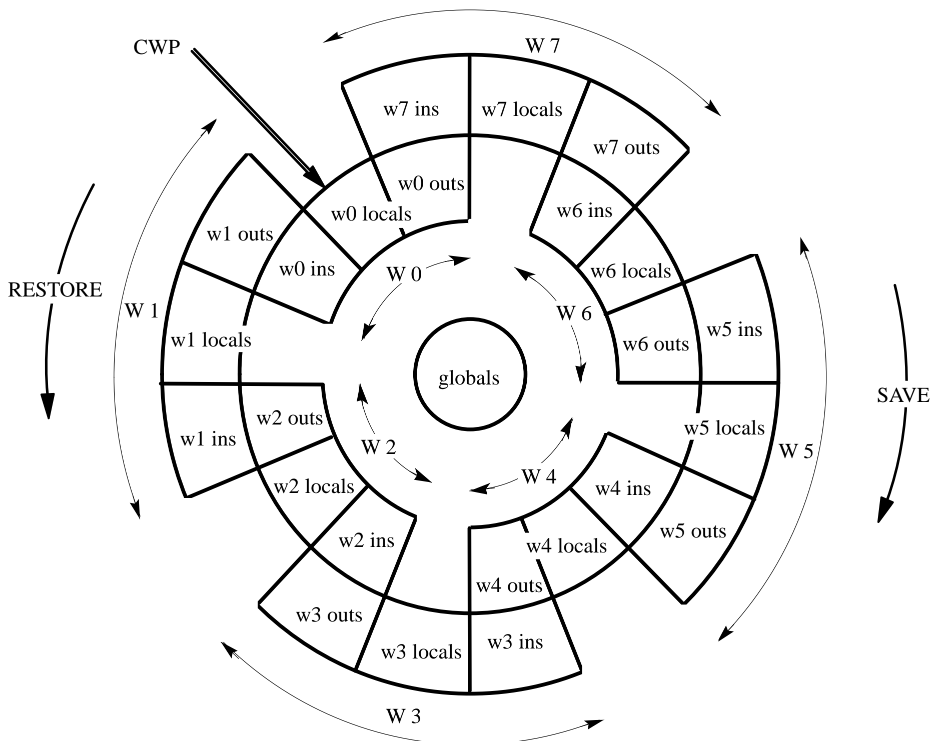


Figure 4. Circular Stack of Overlapping Windows

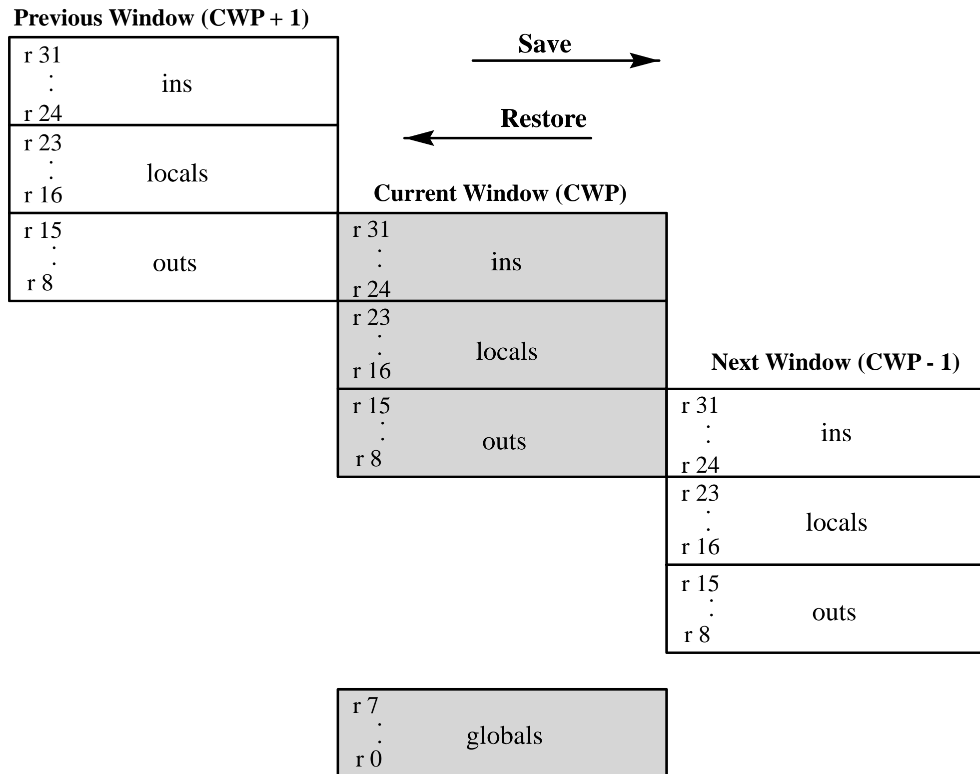


Figure 5. Overlapping Windows

Note that each window shares its *ins* and *outs* with adjacent windows (refer to Figure 5). *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 3.3.4.3) are undefined. The power–on reset trap routine must initialize the CWP and WIM register for correct operation.

3.3.1.1.1. 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 6). 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.

in	r31	(i7) return address
	r30	(FP) frame pointer
	r29	(i5) incoming param reg 5
	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
local	r23	(l7) local 7
	r22	(l6) local 6
	r21	(l5) local 5
	r20	(l4) local 4
	r19	(l3) local 3
	r18	(l2) local 2
	r17	(l1) local 1
	r16	(l0) local 0
out	r15	(o7) temp
	r14	(SP) stack pointer
	r13	(o5) outgoing param reg 5
	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
global	r7	(g7) global 7
	r6	(g6) global 6
	r5	(g5) global 5
	r4	(g4) global 4
	r3	(g3) global 3
	r2	(g2) global 2
	r1	(g1) global 1
	r0	(g0) 0
floating point	f31	floating–point value
	:	:
	f0	floating–point value

Figure 6. Registers as Seen by a Procedure

3.3.1.1.2. 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 TSC691E 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 3.3.4.3). If a SAVE instruction points the

CWP to a marked window, a window overflow trap is generated. This means that in the TSC691E, 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 TSC691E 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 under trap. SAVE, RESTORE, and RETT always check the WIM register before completing their actions.

As an example, in Figure 4 , 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 TSC691E windows, minus one for the reserved trap window.

3.3.1.1.3. 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 3.3.1.2), 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 3.3.4.3). Switching from one register bank to another is accomplished by writing to the CWP field of the processor state register. Figure 7 shows the TSC691E register file divided into four banks, each with its own trap handler window of eight local registers. *Globals* are accessible by all processes.

3.3.1.2. 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.

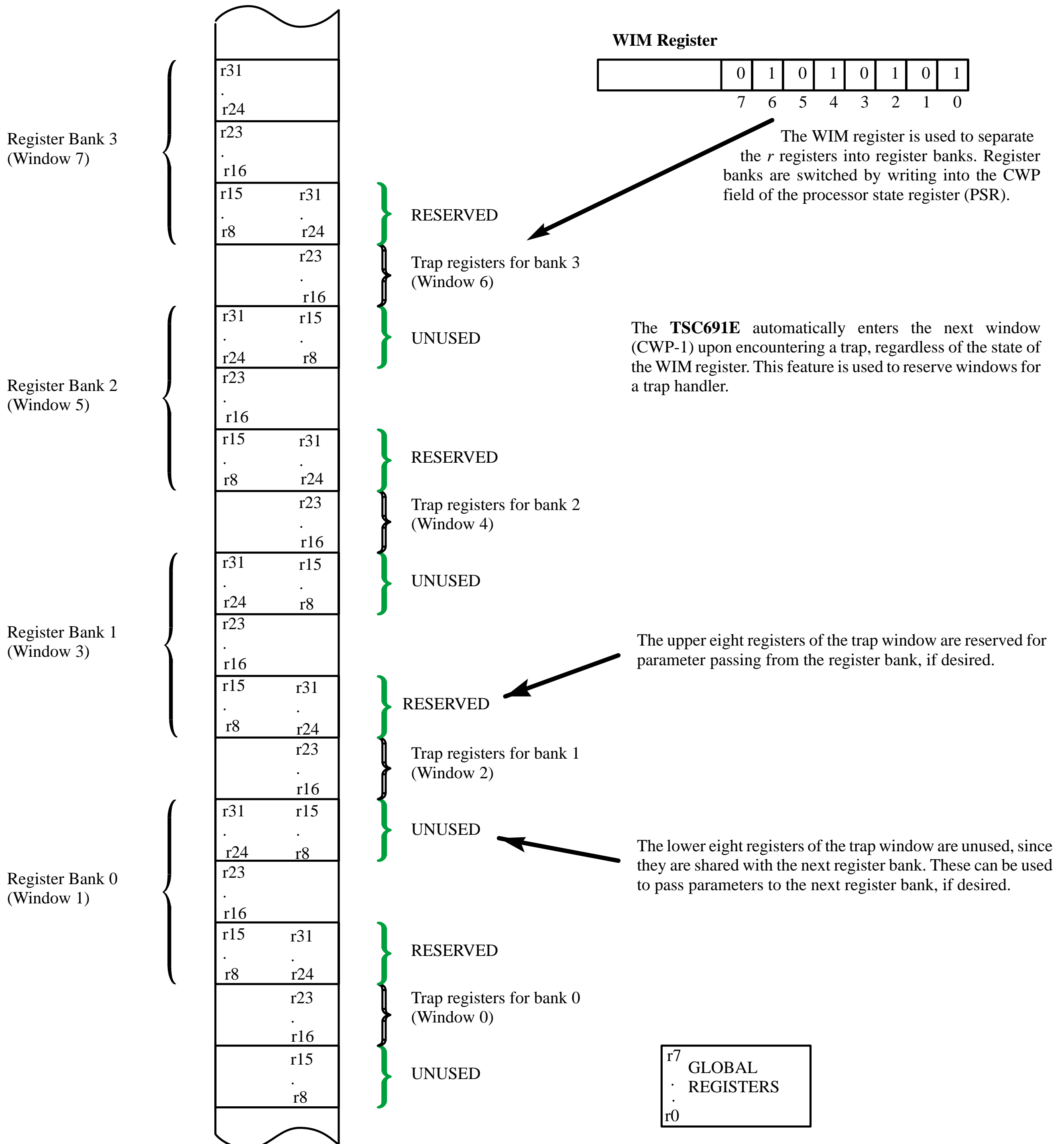


Figure 7. Register Banks for Fast Context Switching

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.

3.3.2. Processor States

The TSC691E 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 3.8). The TSC691E 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 3.5). The processor remains in that mode until $\overline{\text{RESET}}$ is deasserted. RESET signal must be asserted nine clocks at least. 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 3.3.3 and 3.4.2.6).

The TSC691E 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.

3.3.3. Supervisor/User Modes

In support of multitasking, the TSC691E 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 3.3.4.2). The state of this bit determines which address space is accessed with the ASI bits (see Section 3.4.2.6) 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 3.4.3.4.4). This restores both the PC and nPC and the previous state of the S bit.

3.3.4. Control/Status Registers

TSC691E 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 8 and Figure 10 , 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).

3.3.4.1. Program Counters (PC and nPC)

The Program Counter (PC) contains the address of the instruction currently being executed by the TSC691E, 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 3.4.3.4). 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.

3.3.4.2. Processor State Register (PSR)

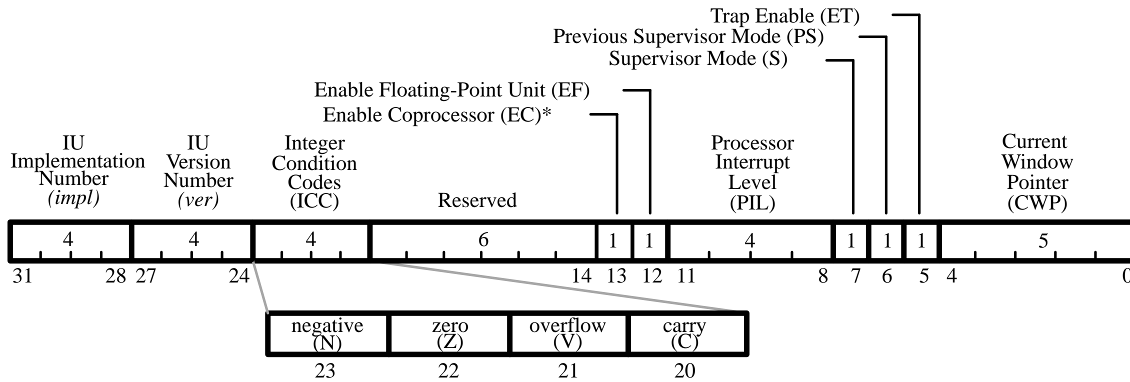


Figure 8. Processor State Register

This is the TSC691E'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 **TSC691E** is 0001_b. Writting PSR (WRPSR) does not modify this field.

ver—Version

Bits 24 through 27 contain the **TSC691E**'s version number. Writting PSR (WRPSR) does not modify this field. The current version number for the **TSC691E** is 0001_b.

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 Writting PSR 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 3.4.3.2.3). 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.

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 **TSC691E** halts and enters the error mode (see Section 3.8).

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 signal, $\overline{\text{FP}}$, 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.

3.3.4.3. Window Invalid Mask Register (WIM)

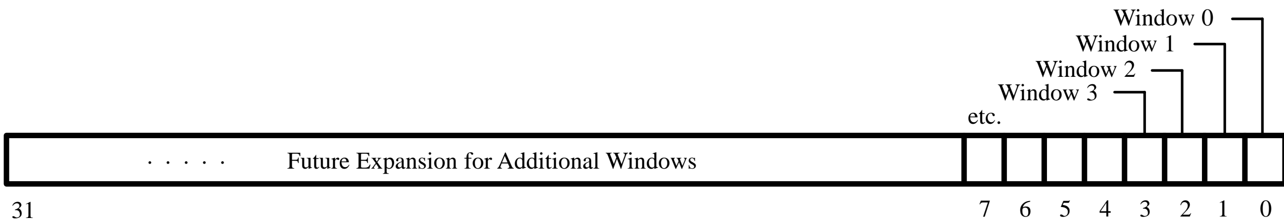


Figure 9. Window Invalid Mask

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 9) 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 3.3.1.1.3).

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.

3.3.4.4. Trap Base Register (TBR)

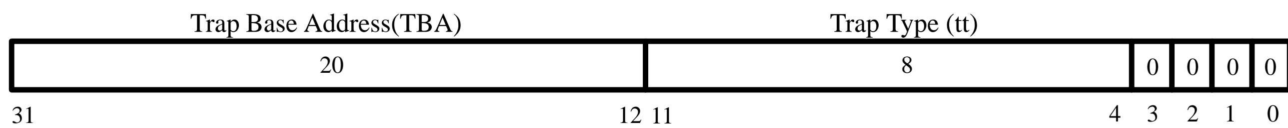


Figure 10. Trap Base Register

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 10). 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 3.8.

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 3.8.5.3). This field retains its value until the next trap is taken.

3.3.4.5. 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.

3.3.5. Data Types

The TSC691E supports ten data types (eleven with extended-precision floating-point, see Section 3.3.5.3). 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 shows the formats for single-precision and double-precision floating–point numbers.

Table 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):	$f \neq 0$	$(-1)^s * 2^{-126} * 0.f$
zero (e = 0):	$f \neq 0$	$(-1)^s * 0$
signaling NaN:	$f \neq 0$	s = u; e = 255 (max); f = .0uu···uu (at least one bit must be nonzero)
quiet NaN:	$f \neq 0$	s = u; e = 255 (max); f = .1uu···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):	$f \neq 0$	$(-1)^s * 2^{-1022} * 0.f$
zero (e = 0):	$f \neq 0$	$(-1)^s * 0$
signaling NaN:	$f \neq 0$	s = u; e = 2047 (max); f = .0uu···uu (at least one bit must be nonzero)
quiet NaN:	$f \neq 0$	s = u; e = 2047 (max); f = .1uu···uu
infinity:		s = 0 or 1, depending upon sign; e = 2047 (max); f = .00···00 (all zeros)

3.3.5.1. Data Organization In Registers

The organization of the ten data types when loaded into registers is shown in Figure 11 .

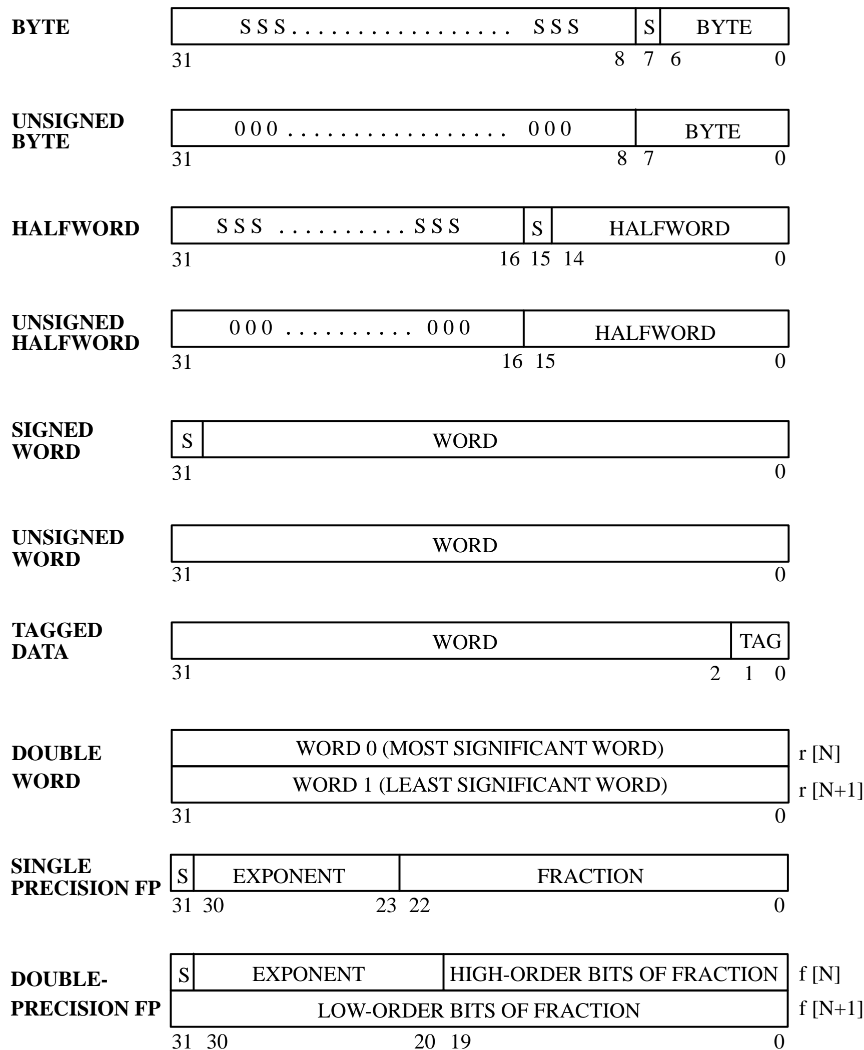
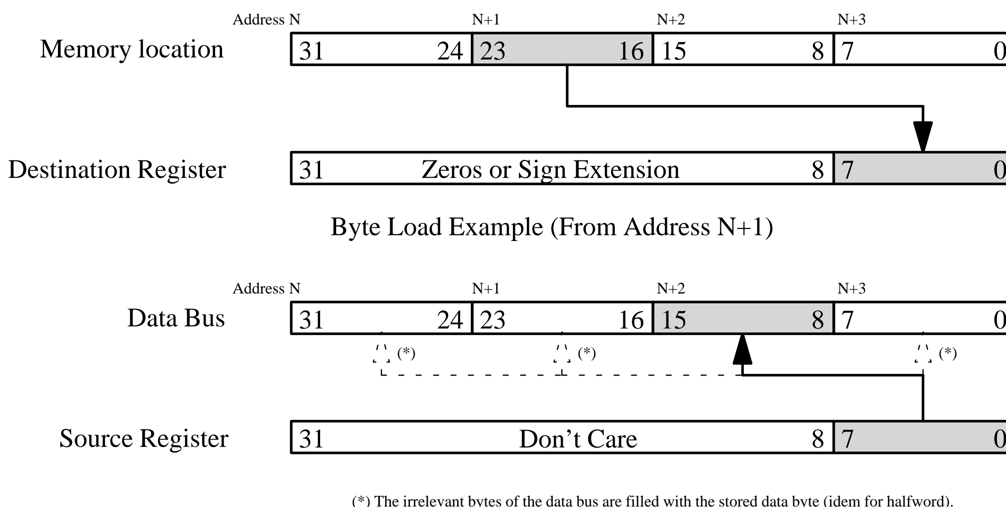


Figure 11. Processor Data Types

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 12 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.



Byte Store Example (To Address N+2)

Figure 12. Byte Operand Load and Store

3.3.5.2. 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 13).
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.

63 Doubleword 0							
31 Word 0				31 Word 0			
15 Halfword 0		15 Halfword 0		15 Halfword 0		15 Halfword 0	
7 Byte 0	7 Byte 0	7 Byte 0	7 Byte 0	7 Byte 0	7 Byte 0	7 Byte	7 Byte 0
Address N	N+1	N+2	N+3	N+4	N+5	N+6	N+7

Figure 13. Data Organization in Memory

3.3.5.3. 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 3). When loaded to the working registers, extended-precision operands require a register quadruple (see Figure 14). 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 15). An extended-precision datum must be located on an extended-precision boundary (address bits [3:0] = 0), which is evenly divisible by 16.

Table 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) ($f \neq 0$): zero (s = 0; e = 0) ($f \neq 0$) ($j \neq 0$):	$(-1)^s * 2^{e-16383} * j.f$ $(-1)^s * 2^{-16382} * j.f$ $(-1)^s * 0$

signaling NaN: $f \neq 0$
 	
quiet NaN: $f \neq 0$
 	
infinity:
	s = u; e = 32767 (max); j = u;
	f = .0 uu···uu (at least one bit must be must be nonzero)
	s = u; e = 32767 (max); j = u;
	f = .1 uu···uu
	s = 0 or 1, depending upon sign;
	e = 32767 (max); j = u;
	f = .00··· 00 (all zeros)

EXTENDED PRECISION FP

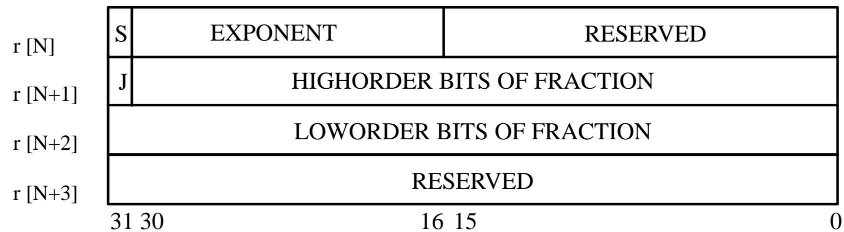


Figure 14. Extended–Precision Data Organization in Registers

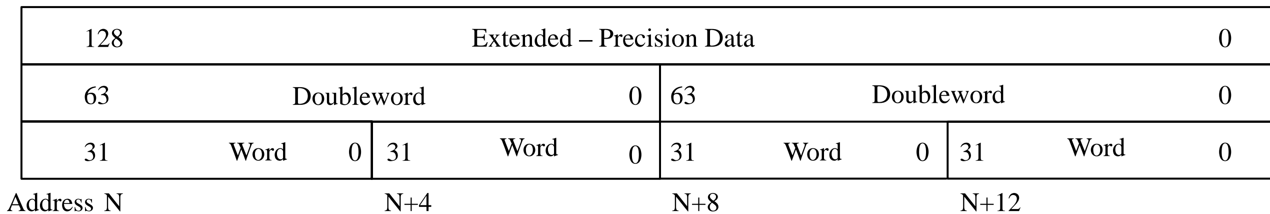


Figure 15. Extended–Precision Data Organization in Memory

3.4. Instruction Set

This section describes the TSC691E instruction set as defined by the SPARC architecture. Included are subsections on instruction formats, addressing, instruction types, and an op code summary. A specific document, SPARC V7.0 Instruction Set contains a description of the assembly language syntax and a complete set of instruction definitions.

3.4.1. 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 [1] and Branch instructions, and format 3 for the remaining integer and floating-point/coprocessor instructions. Figure 16 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.

Note:

See chapter 4.2 for application of this instruction in Program Flow Control.

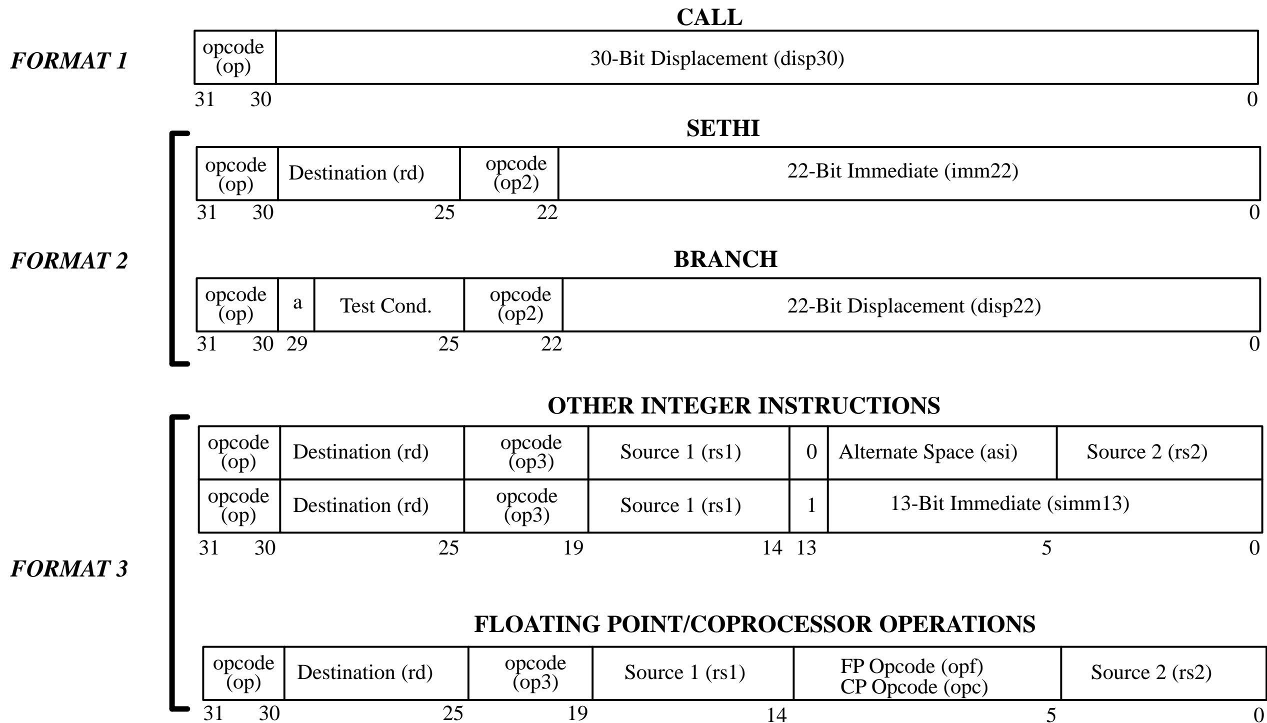


Figure 16. Instruction Format Summary

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 3.4.3.4.3).

asi The address space identifier is an eight-bit field used in load/store alternate instructions. See Section 3.4.2.6.

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. (See Chapter 4.2 for Program Flow Control)

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

op2 The *op2* field (Table 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 3.4.3.6.

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 3.4.3.6.

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 4. op field Coding

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

Table 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.

3.4.2. Addressing

Because it uses a load/store architecture, the TSC691E 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 17 .

3.4.2.1. 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.

3.4.2.2. 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.

3.4.2.3. 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.

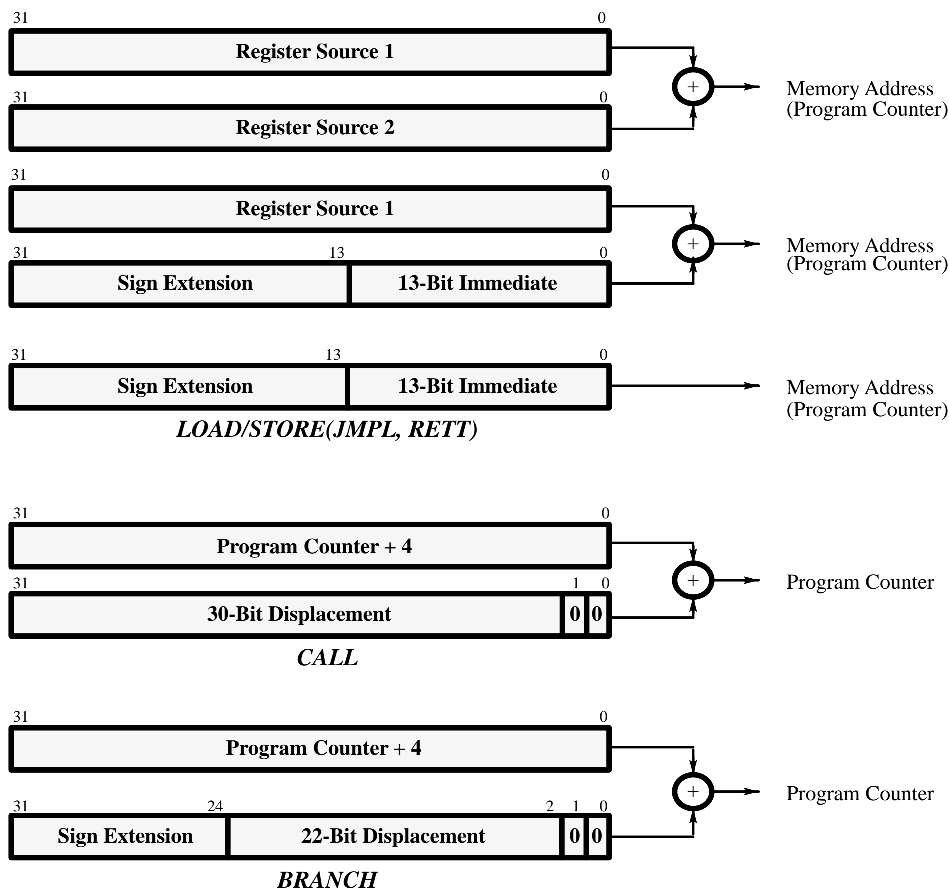


Figure 17. Address Generation

3.4.2.4. CALL

Address generation for the CALL instruction is program counter-relative, that is, the target address is based on the program counter. Because the TSC691E is a delayed-control-transfer machine (see Section 3.4.3.4), before the address is calculated, the PC is replaced by the nPC, so the calculation is actually done with PC + 4 (see Figure 17).

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.

3.4.2.5. 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 6. ASI Assignments

TSC691E Address Space Identifier (ASI)	Address Space
00001000 (08 H)	User Instruction
00001010 (0A H)	User Data
00001001 (09 H)	Supervisor Instruction
00001011 (0B H)	Supervisor Data

3.4.2.6. 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 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 TSC693E control registers, TLB entries, cache tag entries, etc...

The ASI value is supplied by the TSC691E for each instruction fetch and each data access encountered. The TSC690 family assigns a number of these ASI values to the TSC693E and a number are reserved for future assignment. Nevertheless, nearly 80 are left unassigned for use by the system.

3.4.3. Instruction Types

TSC691E 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, refer to its definition in SPARC V7.0 Instruction Set.

3.4.3.1. Load/Store

Load and store instructions (see Table 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 TSC691E generates the memory address and the FPU or CP receives or supplies the data.

The TSC691E 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 TSC691E also generates an eight-bit address space identifier.

3.4.3.1.1. 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 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

3.4.3.1.2. Multiprocessing Instructions

In addition to alternate address spaces, the TSC691E 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.

3.4.3.2. 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 [1] is a special case because it is a single–operand instruction.

Table 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 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 [1]		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 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 SPARC V7.0 Instruction Set.

Note [1]: See section 4.2 for application of this instruction in Program Flow Control.

3.4.3.2.1. 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 [1]. 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 pseudoinstructions, it translates the pseudoinstruction into the equivalent instruction in the native assembly language. Refer to your assembly language manual for details.

Note [1]: Refer to Program Flow Control for more information. (see section 4.2)

3.4.3.2.2. 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.

SETHI can also be used in Program Flow Control to compare the precomputed checksum given as a special SETHI instruction (SETHI 0,%SUM) with the checksum. This special SETHI instruction can be inserted after every branch, call, and before every branch-in point.

3.4.3.2.3. 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 18). 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.

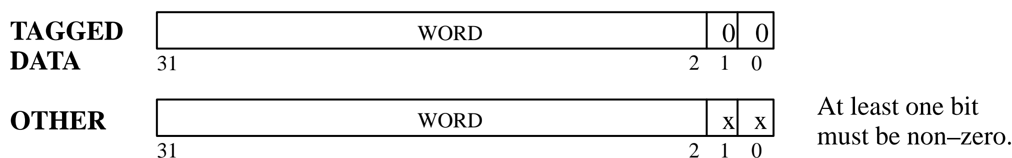


Figure 18. Tagged Data Example

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.

3.4.3.3. 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 9).

Table 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

In the TSC691E, 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 3.4.3.4.3). Table 10 shows the characteristics of each control transfer type.

Table 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

Program Counter Relative

PC-relative addressing computes the target address by adding a displacement to the program counter. See Section 3.4.2.

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 3.4.2.

Delayed

A control-transfer instruction is delayed if it transfers control to the target address after a one-instruction delay. See Section 3.4.3.4.

Annul Bit

In an instruction with an annul bit, the delay instruction that follows may be annulled. See Section 3.4.3.4.3.

3.4.3.3.1. 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.

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 3.4.3.4.

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 3.4.3.4.3 for details.

As illustrated in Table 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 floating–point unit (see Table 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 13). Both FBN and CBN behave in the same way as BN.

Table 11. Bicc and Ticc Condition Codes

Condition	Test	Condition	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 12. FBfcc Condition Codes

Condition	Test	Condition	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 13. CBccc Condition Codes

Opcode	Condition	CCC[1:0] Test	Opcode	Condition	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

3.4.3.3.2. 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 19 . 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.

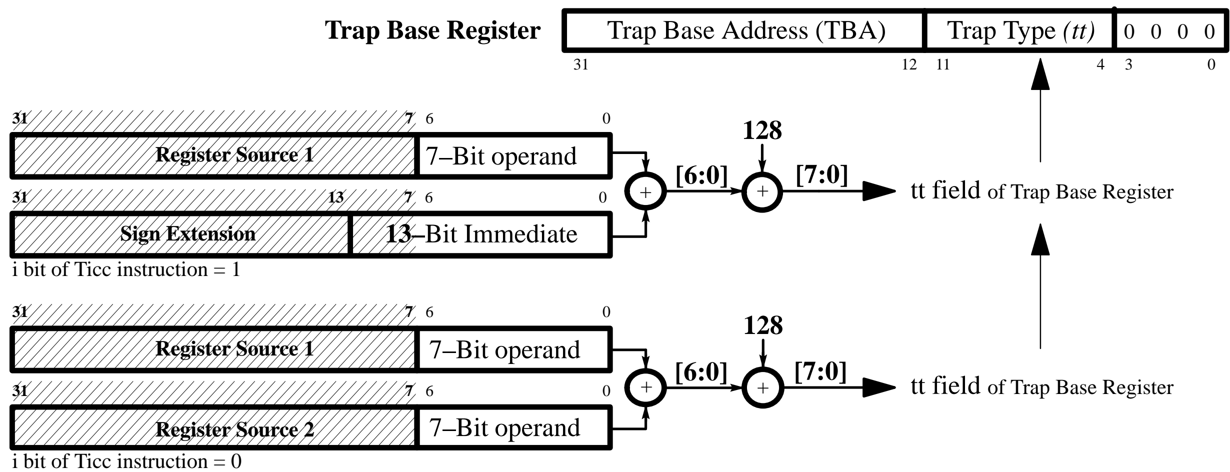


Figure 19. Ticc Trap Address Generation

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.

3.4.3.3.3. 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 5). 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 3.3.1.1.1).

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.

3.4.3.4. 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 TSC691E 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 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 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 16. Effect of Annul Bit Reset (*a*=1)

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

3.4.3.4.1. PC and nPC

The Program Counter (PC) contains the address of the instruction currently being executed by the TSC691E, 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.

3.4.3.4.2. 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 3.4.3.4.4.

Table 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.

3.4.3.4.3. 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 TSC691E 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 15 and Table 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 20 displays the effect the *a* bit has on any branch for either the set or reset state. Table 17 summarizes the effect the annul bit has on the execution of delay instructions.

Table 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**

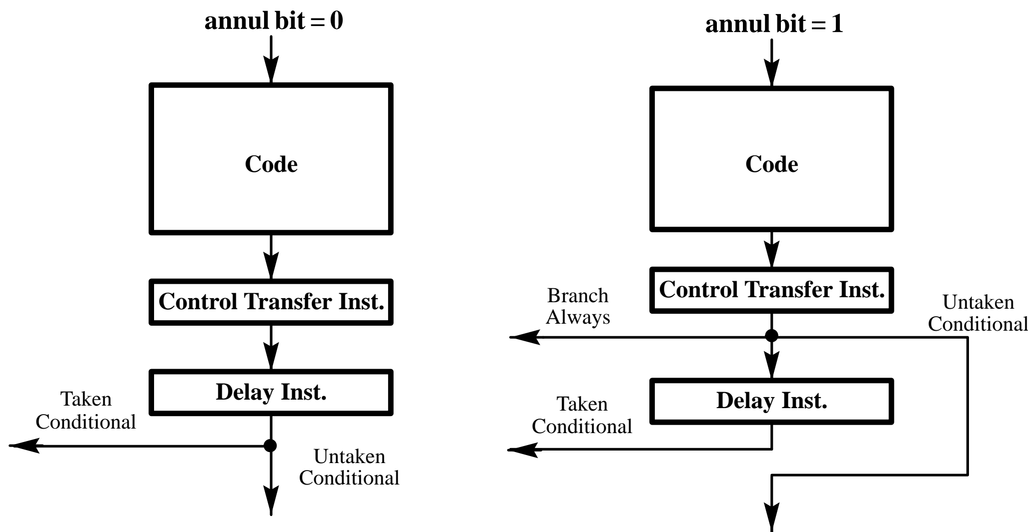


Figure 20. Delayed Control Transfer

3.4.3.4.4. 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 18 , and the order of execution for the six different cases of back-to-back, delayed control transfer instructions is shown in Table 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 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 3.4.3.4.2). 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 SPARC V7.0 Instruction Set for further information.

Table 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 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 annuled)
4	DCTI Unconditional	B*A (a = 1)	12, 16, 60, 64, ... (40 annuled)
5	B*A (a = 1)	any CTI	12, 40, 44, ... (40 annuled)
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 19 are illustrated in Figure 21 . In case 1, the first DCTI is fetched at address 0x12 and the target address is calculated while the delay slot instruction is fetched. The delay slot instruction for the first DCTI (located at address 0x16) 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 0x40. 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 0x60) 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 0x40 (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 0x44.

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 0x40 to be annulled. Since the second DCTI is an untaken conditional branch, instruction fetch continues after the annulled target instruction (address 0x44).

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 0x40) to be annulled, and program control is transferred to the target of the second DCTI at address 0x60.

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.

Case 6 When the first instruction of a delayed control transfer couple is a conditional branch, control transfer is undefined. 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.

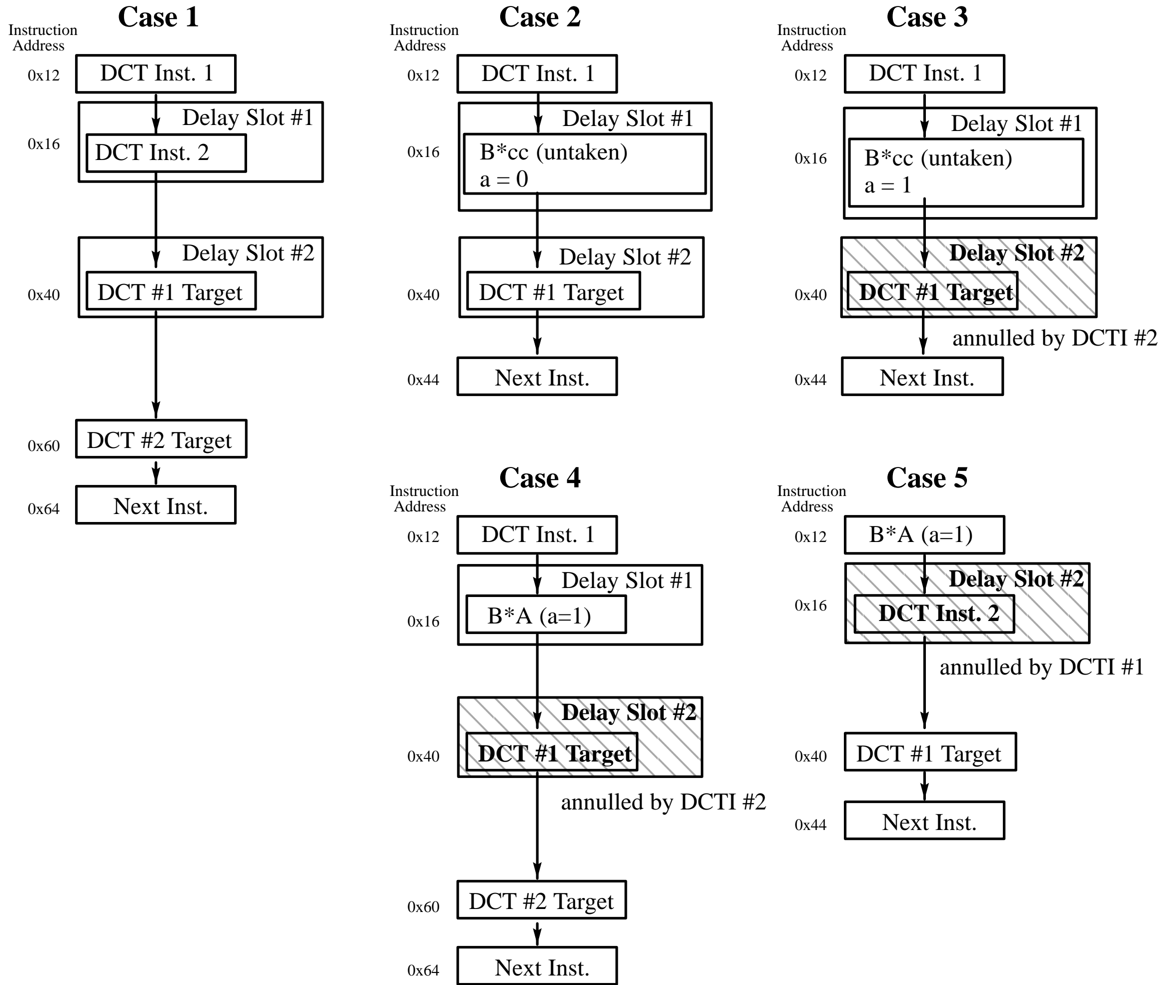


Figure 21. Delayed Control Transfer Couples

3.4.3.5. Read/Write Control Registers

This class of instruction reads or writes the contents of the various control registers (see Table 20).

Table 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

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.

3.4.3.6. 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 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 FPU's register file. If no FPU is present, or if the EF bit of the PSR is not set, executing a floating–point instruction will generate a $\overline{\text{FP}}$ disabled trap.

Coprocessor-operate instructions (FPops) 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 $\overline{\text{FP}}$ disabled trap.

Floating-point and coprocessor load/store instructions are not operate instructions; they fall under the **TSC691E**'s load /store instruction category (see Section 3.4.3.1).

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 FPU or coprocessor. Floating-point-operate instructions execute concurrently with **TSC691E** instructions. FPops can also execute concurrently with both **TSC691E** and FPop instructions if they are designed to do so.

Because the **TSC691E** and FPU 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 **TSC691E**, the architecture will support a coprocessor queue that functions in the same fashion as the floating-point queue.

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

Name	Operation	Cycles
FPop	Floating–Point Operations	1 to launch
FPop	Coprocessor Operations	1 to launch

3.4.3.7. Miscellaneous

Instructions in this category handle special circumstances within the integer unit (see Table 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 SPARC V7.0 Instruction Set.

The IFLUSH instruction is used to flush a word from an internal (to the **TSC691E**) instruction cache. Current integer unit implementations (**TSC691E**) 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 3.5)

Table 22. Miscellaneous Instructions

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

3.4.4. Op Codes

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

3.4.4.1. Load/Store Instructions

Table 23. Load/Store Instruction Opcodes

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	ignored	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	ignored	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	ignored	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	ignored	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	ignored	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	ignored	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	ignored	rs2
					i =1	simm13	

Mnemonic	Opcodes with Format 31 30	29 25	24 19	18 14	13	12 5	4 0
LDUHA	1 1	rd	0 1 0 0 1 0	rs1	i =0	asi	rs2
ST	1 1	rd	0 0 0 1 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
STA	1 1	rd	0 1 0 1 0 0	rs1	i =0	asi	rs2
STB	1 1	rd	0 0 0 1 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
STBA	1 1	rd	0 1 0 1 0 1	rs1	i =0	asi	rs2
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	
STD	1 1	rd	0 0 0 1 1 1	rs1	i =0	ignored	rs2
					i =1	simm13	
STDA	1 1	rd	0 1 0 1 1 1	rs1	i =0	asi	rs2
STDC	1 1	rd	1 1 0 1 1 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	
STDF	1 1	rd	1 0 0 1 1 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	
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	
STH	1 1	rd	0 0 0 1 1 0	rs1	i =0	ignored	rs2
					i =1	simm13	
STHA	1 1	rd	0 1 0 1 1 0	rs1	i =0	asi	rs2
SWAP	1 1	rd	0 0 1 1 1 1	rs1	i =0	ignored	rs2
					i =1	simm13	
SWAPA	1 1	rd	0 1 1 1 1 1	rs1	i =0	asi	rs2

3.4.4.2. Arithmetic/Logical/Shift Instructions

Table 24. Arithmetic/Logical/Shift Instruction Opcodes

Mnemonic	Opcodes with Format 31 30	29 25	24 19	18 14	13	12 5	4 0
ADD	1 0	rd	0 0 0 0 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	
ADDX	1 0	rd	0 0 1 0 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
ADDXcc	1 0	rd	0 1 1 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	
ANDcc	1 0	rd	0 1 0 0 0 1	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	
ANDNcc	1 0	rd	0 1 0 1 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
MULScc	1 0	rd	1 0 0 1 0 0	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	
ORcc	1 0	rd	0 1 0 0 1 0	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	
ORNcc	1 0	rd	0 1 0 1 1 0	rs1	i =0	ignored	rs2
					i =1	simm13	
SLL	1 0	rd	1 0 0 1 0 1	rs1	i =0	ignored	rs2
					i =1	shcnt	
SRA	1 0	rd	1 0 0 1 1 1	rs1	i =0	ignored	rs2
					i =1	shcnt	

Mnemonic	Opcodes with Format 31 30	29 25	24 19	18 14	13	12 5	4 0
SRL	1 0	rd	1 0 0 1 1 0	rs1	i =0	ignored	rs2
					i =1	shcnt	
SUB	1 0	rd	0 0 0 1 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
SUBcc	1 0	rd	0 1 0 1 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	
SUBXcc	1 0	rd	0 1 1 1 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
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
XOR	1 0	rd	0 0 0 0 1 1	rs1	i =1	simm13	
XORcc	1 0	rd	0 1 0 0 1 1	rs1	i =0	ignored	rs2
					i =1	simm13	

	31 30	29 25	24 22	21 0
SETHI	0 0	rd	1 0 0	imm22

3.4.4.3. Control Transfer Instructions

Table 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 26. Bicc and Ticc Condition Codes

Condition	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 27. FBfcc Condition Codes

Condition	Test
0000	Never
0001	Not equal
0010	Less than or greater to
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 28. CBccc Condition Codes

Opcode	Condition	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

3.4.4.4. Read/Write Control Register Instructions

Table 29. Read/Write Control Register Instruction Opcodes

<table>
<tr><th rowspan="2">Mnemonic</th><th colspan="7">Opcodes with Format</th></tr>
<tr><th>31 30</th><th>29 25</th><th>24 19</th><th>18 14</th><th>13</th><th>12 5</th><th>4 0</th></tr>
<tr><td>RDPSR</td><td>1 0</td><td>rd</td><td>1 0 1 0 0 1</td><td>ignored</td><td>I*</td><td colspan="2">ignored</td></tr>
<tr><td>RDTBR</td><td>1 0</td><td>rd</td><td>1 0 1 0 1 1</td><td>ignored</td><td>I*</td><td colspan="2">ignored</td></tr>
<tr><td>RDWIM</td><td>1 0</td><td>rd</td><td>1 0 1 0 1 0</td><td>ignored</td><td>I*</td><td colspan="2">ignored</td></tr>
<tr><td>RDY</td><td>1 0</td><td>rd</td><td>1 0 1 0 0 0</td><td>ignored</td><td>I*</td><td colspan="2">ignored</td></tr>
<tr><td rowspan="2">WRPSR</td><td rowspan="2">1 0</td><td rowspan="2">ignored</td><td rowspan="2">1 1 0 0 0 1</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">WRTBR</td><td rowspan="2">1 0</td><td rowspan="2">ignored</td><td rowspan="2">1 1 0 0 1 1</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
</table>

Mnemonic	Opcodes with Format						
	31 30	29 25	24 19	18 14	13	12 5	4 0
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.

3.4.4.5. Floating-Point/Coprocessor Instructions

Table 30. Floating–Point /Coprocessor Instruction Opcodes

Mnemonic	Opcodes with Format					
	31 30	29 25	24 19	18 14	13 5	4 0
FPOP1	1 0	rd	1 1 0 1 1 0	rs1	OPC	rs2
FPOP2	1 0	rd	1 1 0 1 1 1	rs1	OPC	rs2
FABSs	1 0	rd	1 1 0 1 0 0	ignored	0 0 0 0 0 1 0 0 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
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
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
FMOVs	1 0	rd	1 1 0 1 0 0	ignored	0 0 0 0 0 0 0 0 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
FNEGs	1 0	rd	1 1 0 1 0 0	ignored	0 0 0 0 0 0 1 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

Mnemonic	Opcodes with Format					
	31 30	29 25	24 19	18 14	13 5	4 0
FSQRTx	1 0	rd	1 1 0 1 0 0	ignored	0 0 0 1 0 1 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
FdTOi	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 1 0	rs2
FdTOs	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 0 1 1 0	rs2
FdTOx	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 1 0	rs2
FiTOd	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 0 0	rs2
FiTOs	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 0 1 0 0	rs2
FiTOx	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 0 0	rs2
FsTOd	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 0 1	rs2
FsTOi	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 0 1	rs2
FsTOx	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 0 1	rs2
FxTOi	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 1 1	rs2
FxTOs	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 0 1 1 1	rs2
FxTOd	1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 1 1	rs2

3.4.4.6. Miscellaneous Instructions

Table 31. Miscellaneous Instruction Opcodes

Mnemonic	Opcodes with Format							
	31 30	29 25	24 22	21 19	18 14	13	12 5	4 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				

3.4.4.7. Opcodes In Ascending Numeric Order

Table 32. Instruction Opcode Numeric Listing

<table>
<tr><th rowspan="2">Mnemonic</th><th colspan="9">Opcodes with Format</th></tr>
<tr><th>31 30</th><th>29</th><th>25</th><th>24 22</th><th>21 19</th><th>18 14</th><th>13</th><th>12 5</th><th>4 0</th></tr>
<tr><td>UNIMP</td><td>0 0</td><td colspan="2">ignored</td><td>0 0 0</td><td colspan="5">const22</td></tr>
<tr><td>Bicc</td><td>0 0</td><td>a</td><td>cond</td><td>0 1 0</td><td colspan="5">disp22</td></tr>
<tr><td>SETHI</td><td>0 0</td><td colspan="2">rd</td><td>1 0 0</td><td colspan="5">imm22</td></tr>
<tr><td>FBfcc</td><td>0 0</td><td>a</td><td>cond</td><td>1 1 0</td><td colspan="5">disp22</td></tr>
<tr><td>CBccc</td><td>0 0</td><td>a</td><td>cond</td><td>1 1 1</td><td colspan="5">disp22</td></tr>
<tr><td>CALL</td><td>0 1</td><td colspan="8">disp30</td></tr>
<tr><td rowspan="2">ADD</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 0 0 0</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">AND</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 0 0 1</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">OR</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 0 1 0</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">XOR</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 0 1 1</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">SUB</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 1 0 0</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">ANDN</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 1 0 1</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">ORN</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 1 1 0</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">XNOR</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 0 1 1 1</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">ADDX</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 1 0 0 0</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">SUBX</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 0 1 1 0 0</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">ADDcc</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 1 0 0 0 0</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
<tr><td rowspan="2">ANDcc</td><td rowspan="2">1 0</td><td rowspan="2" colspan="2">rd</td><td rowspan="2" colspan="2">0 1 0 0 0 1</td><td rowspan="2">rs1</td><td>i =0</td><td>ignored</td><td>rs2</td></tr>
<tr><td>i =1</td><td colspan="2">simm13</td></tr>
</table>

Mnemonic	Opcodes with Format						
	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
ORcc	1 0	rd	0 1 0 0 1 0	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	
SUBcc	1 0	rd	0 1 0 1 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
ANDNcc	1 0	rd	0 1 0 1 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
ORNcc	1 0	rd	0 1 0 1 1 0	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	
ADDXcc	1 0	rd	0 1 1 0 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
SUBXcc	1 0	rd	0 1 1 1 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
TADDcc	1 0	rd	1 0 0 0 0 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	
TADDccTV	1 0	rd	1 0 0 0 1 0	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	
MULScc	1 0	rd	1 0 0 1 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
SLL	1 0	rd	1 0 0 1 0 1	rs1	i =0	ignored	rs2
					i =1	shcnt	
SRL	1 0	rd	1 0 0 1 1 0	rs1	i =0	ignored	rs2
					i =1	shcnt	
SRA	1 0	rd	1 0 0 1 1 1	rs1	i =0	ignored	rs2
					i =1	shcnt	
RDY	1 0	rd	1 0 1 0 0 0	ignored	I*	ignored	

Mnemonic	Opcodes with Format						
	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
RDPSR	1 0	rd	1 0 1 0 0 1	ignored	I*	ignored	
RDWIM	1 0	rd	1 0 1 0 1 0	ignored	I*	ignored	
RDTBR	1 0	rd	1 0 1 0 1 1	ignored	I*	ignored	
WRY	1 0	ignored	1 1 0 0 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
WRPSR	1 0	ignored	1 1 0 0 0 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	
WRTBR	1 0	ignored	1 1 0 0 1 1	rs1	i =0	ignored	rs2
					i =1	simm13	
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

Mnemonic	Opcodes with Format						
	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
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
FPOP1	1 0	rd	1 1 0 1 1 0	rs1	OPC		rs2
FPOP2	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	
RETT	1 0	ignored	1 1 1 0 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
Ticc	1 0	I* cond	1 1 1 0 1 0	rs1	i =0	ignored	rs2
					i =1	simm13	
IFLUSH	1 0	ignored	1 1 1 0 1 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	
RESTORE	1 0	rd	1 1 1 1 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
LD	1 1	rd	0 0 0 0 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	

Mnemonic	Opcodes with Format 31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
LDUB	1 1	rd	0 0 0 0 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
LDUH	1 1	rd	0 0 0 0 1 0	rs1	i =0	ignored	rs2
					i =1	simm13	
LDD	1 1	rd	0 0 0 0 1 1	rs1	i =0	ignored	rs2
					i =1	simm13	
ST	1 1	rd	0 0 0 1 0 0	rs1	i =0	ignored	rs2
					i =1	simm13	
STB	1 1	rd	0 0 0 1 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
STH	1 1	rd	0 0 0 1 1 0	rs1	i =0	ignored	rs2
					i =1	simm13	
STD	1 1	rd	0 0 0 1 1 1	rs1	i =0	ignored	rs2
					i =1	simm13	
LDSB	1 1	rd	0 0 1 0 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
LDSH	1 1		0 0 1 0 1 0	rs1	i =0	ignored	rs2
					i =1	simm13	
LDSTUB	1 1	rd	0 0 1 1 0 1	rs1	i =0	ignored	rs2
					i =1	simm13	
SWAP	1 1	rd	0 0 1 1 1 1	rs1	i =0	ignored	rs2
					i =1	simm13	
LDA	1 1	rd	0 1 0 0 0 0	rs1	i =0	asi	rs2
LDUBA	1 1	rd	0 1 0 0 0 1	rs1	i =0	asi	rs2
LDUHA	1 1	rd	0 1 0 0 1 0	rs1	i =0	asi	rs2
LDDA	1 1	rd	0 1 0 0 1 1	rs1	i =0	asi	rs2
STA	1 1	rd	0 1 0 1 0 0	rs1	i =0	asi	rs2
STBA	1 1	rd	0 1 0 1 0 1	rs1	i =0	asi	rs2
STHA	1 1	rd	0 1 0 1 1 0	rs1	i =0	asi	rs2
STDA	1 1	rd	0 1 0 1 1 1	rs1	i =0	asi	rs2
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

Mnemonic	Opcodes with Format						
	31 30	29 25	24 22 21 19	18 14	13	12 5	4 0
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		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	

3.5. Signal Description

This section provides a description of the **TSC691E**'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.

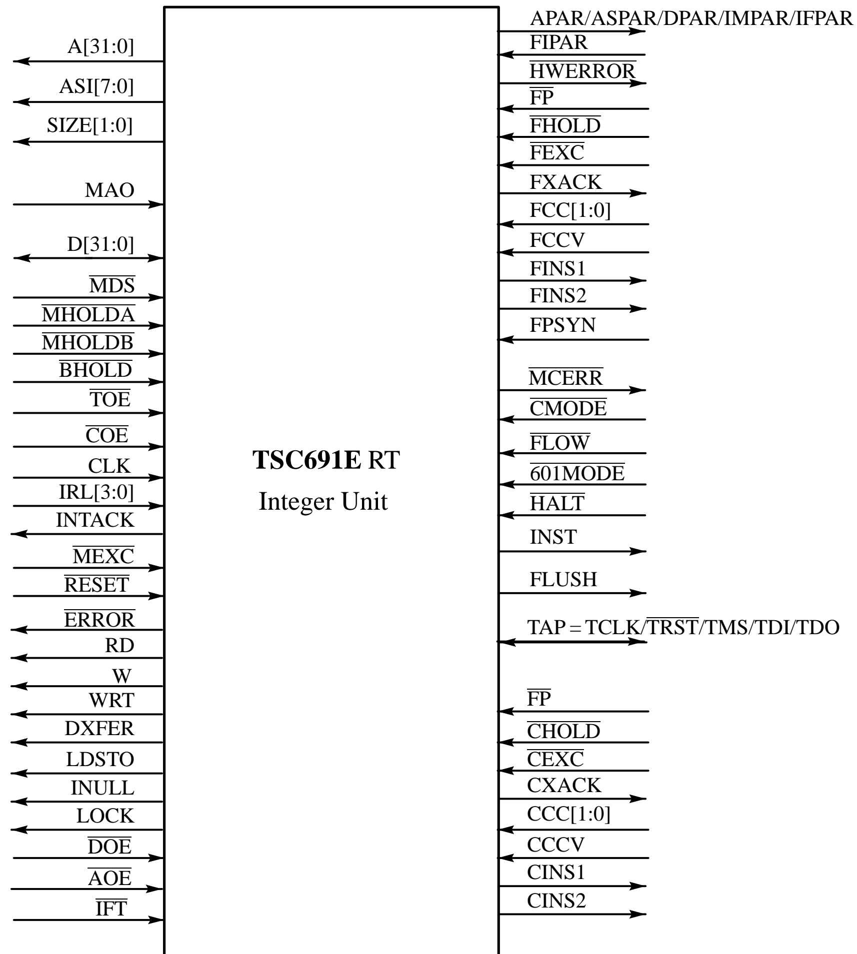


Figure 22. TSC691E External Signals

Signals that are active LOW are marked with an overscore; all others are active HIGH. Figure 22 summarizes the signals described in this section. Table 33 provides a summary of the external signals for the **TSC691E**.

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 33. TSC691E External Signal Summary

Memory Subsystem Interface Signals:

Signal Name	Description	Signal Type (Impedance of Three–State Output=20kΩ)	Active
A[31:0]	Address Bus	Three–State Output	
APAR	Address Bus Parity	Three–State Output	HIGH
$\overline{AOE}$	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.	
DPAR	Data Bus Parity	Three-State BiDir.	HIGH
$\overline{DOE}$	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	
ASPAR	ASI and SIZE Parity	Three–State Output	HIGH
$\overline{WE}$	Write Enable	Three–State Output	LOW
WRT	Advanced Write	Three–State Output	HIGH
IMPAR	IU to MEC [1] Control Parity	Three–State Output	HIGH

Note 1 **:**
TSC693E = Memory controller system support circuit which contains fault detection and peripheral control function.

Floating–Point / Coprocessor Interface Signals:

Signal Name	Description	Signal Type (Impedance of Three–State Output=20kΩ)	Active
CCC[1:0]	Coprocessor Condition Codes	Input	
CCCV	Coprocessor Condition Codes Valid	Input	HIGH
$\overline{CEXC}$	Coprocessor Exception	Input	LOW

Floating–Point / Coprocessor Interface Signals:

Signal Name	Description	Signal Type (Impedance of Three–State Output=20kΩ)	Active
$\overline{CHOLD}$	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
FIPAR	FPU to IU Control Parity	Input	HIGH
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
IFPAR	IU to FPU Control Parity	Three-State Output	HIGH

Interrupt and Control Signals:

Signal Name	Description	Signal Type (Impedance of Three–State Output=20kΩ)	Active
IRL[3:0]	Interrupt Request Level	Input	
INTACK	Interrupt Acknowledge	Three–State Output	HIGH
$\overline{RESET}$	Reset	Input	LOW
$\overline{ERROR}$	Error State	Three–State Output	LOW
$\overline{HWERROR}$	Hardware error Detected	Three–State Output	LOW
$\overline{MCERR}$	Comparison error	Three–State Output	LOW
$\overline{FLOW}$	Enable Program Flow Control	Input	LOW
$\overline{CMODE}$	Checker Mode	Input	LOW
$\overline{601MODE}$	Normal **601** Mode	Input	LOW
FPSYN	Floating–Point Synonym Mode	Input	HIGH
$\overline{TOE}$	Test Mode Output Enable	Input	LOW
$\overline{HALT}$	Halt Mode	Input	LOW

Test Access Port Signals:

Signal Name	Description	Signal Type (Impedance of Three–State Output=20kΩ)	Active
TCLK	Test Clock	Input	
$\overline{\text{TRST}}$	Test reset	Input	LOW
TMS	Test Mode Select	Input	HIGH
TDI	Test Data Input	Input	
TDO	Test Data Output	Three-State Output	

Power and Clock Signals:

Signal 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

3.5.1. 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.

3.5.1.1. A[31:0]—Address Bus (output)

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 **TSC691E**. 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 (on chip pull_up resistor=20kΩ) when the $\overline{\text{AOE}}$ or $\overline{\text{TOE}}$ signal is deasserted (HIGH).

3.5.1.2. APAR—Address Bus Parity (output)

This signal contains the odd parity over the 32-bit address bus and is asserted simultaneously with the memory address. It is high-Z (on chip pull–up resistor=20kΩ) when the $\overline{\text{AOE}}$ or $\overline{\text{TOE}}$ signal is deasserted.

3.5.1.3. $\overline{\text{AOE}}$—Address Output Enable (input)

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 (on chip pull_up resistor=20kΩ) 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).

3.5.1.4. ASI[7:0]—Address Space Identifier (output)

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 (on chip pull_up resistor=20kΩ) when the $\overline{\text{AOE}}$ or $\overline{\text{TOE}}$ signal is deasserted (HIGH). Encoding of the ASI bits is shown in Table 34.

Table 34. ASI Assignments

TSC691E Address Space Identifier (ASI)	Address Space
00001000 (0x08)	User Instruction
00001010 (0x0a)	User Data
00001001 (0x09)	Supervisor Instruction
00001011 (0x0b)	Supervisor Data

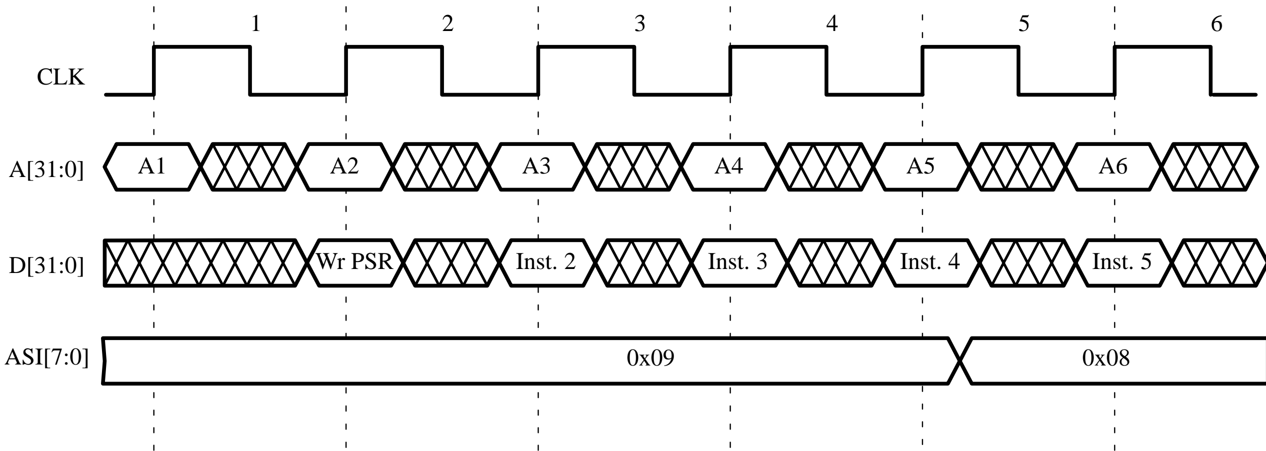


Figure 23. ASI timing with a WRPSR Instruction

3.5.1.5. ASPAR—ASI and SIZE Parity (output)

This signal contains the odd parity over the 8-bit address space identifier and 2 bit Bus Transaction Size. It is asserted simultaneously with the ASI and SIZE and will be high-Z (on chip pull_up resistor=20kΩ) when the $\overline{\text{AOE}}$, $\overline{\text{COE}}$ or $\overline{\text{TOE}}$ signal is deasserted.

3.5.1.6. $\overline{\text{BHOLD}}$—Bus Hold (input)

$\overline{\text{BHOLD}}$ 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 $\overline{\text{BHOLD}}$, external logic must guarantee that the data at all inputs to the **TSC691E** is the same as it was before $\overline{\text{BHOLD}}$ 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 **TSC691E** (transparent latch with clock high) before they are used. Because $\overline{\text{MDS}}$ and $\overline{\text{MEXC}}$ signals are recognized while this input is active, $\overline{\text{BHOLD}}$ should only be used for bus access requests by an external device. $\overline{\text{BHOLD}}$ should not be asserted when LOCK is asserted.

3.5.1.7. $\overline{\text{COE}}$—Control Output Enable (input)

Assertion of this signal enables the output drivers for SIZE[1:0], RD, $\overline{\text{WE}}$, WRT, LOCK, LDSTO, and DXFER outputs, and is the normal condition. Deassertion of $\overline{\text{COE}}$ three-states (on chip pull_up resistor=20kΩ) these 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).

3.5.1.8. D[31:0]—Data Bus (bidirectional)

These signals 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 **TSC691E** during the execution of integer store instructions and the store cycle of atomic-load-store instructions. Similarly, the 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.

The Data bus is three-stated (on chip pull_up resistor=20kΩ) when the $\overline{\text{DOE}}$ or $\overline{\text{TOE}}$ signal is deasserted (HIGH)

3.5.1.9.DPAR—Data Bus Parity (bidirectional)

This signal contains the odd parity over the 32-bit bidirectional data bus.

In case of store data operations the parity bit is generated and launched in parallel by the IU. In case of load data operations the parity is checked by the IU.

This signal will be high-Z (on chip pull_up resistor=20kΩ) when the $\overline{\text{DOE}}$ or $\overline{\text{TOE}}$ signal is deasserted.

3.5.1.10. $\overline{\text{DOE}}$—Data Output Enable (input)

Assertion of this signal enables the output drivers for the data bus, D[31:0], and is the normal condition. Deassertion of $\overline{\text{DOE}}$ three-states (on chip pull_up resistor=20kΩ) the data bus 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).

3.5.1.11. DXFER—Data Transfer (output)

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.

3.5.1.12. $\overline{\text{IFT}}$—Instruction Cache Flush Trap (input)

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

3.5.1.13. 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 $\overline{\text{MEXC}}$ 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.

3.5.1.14. 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.

3.5.1.15. 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 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.

3.5.1.16. MAO—Memory Address Output (input)

This signal is asserted during an MHOLD condition to force the previous (missed) memory access parameters back on their various busses 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 MHOLD was asserted. A logic HIGH value at this signal during a cache miss causes the integer unit to put A[31:0], ASI[7:0], SIZE[1:0], RD, 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 WE.

MAO must be driven LOW while RESET is LOW.

3.5.1.17. MDS—Memory Data Strobe (input)

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 MHOLDA/B. In a system with cache, 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, MDS tells the processor when the read data is available on the bus. During a cache line replacement, MDS may be asserted anywhere within the MHOLD cycle and deasserted before MHOLD is released. For example, if a cache miss occurs on word 2 of a 4-word cache line, MDS should only be driven active while word 2 is being replaced in the cache.

MDS is also used to strobe in the MEXC memory exception signal. MDS may only be asserted when the pipeline is frozen with MHOLDA/B. The **TSC691E** samples MDS with an on-chip transparent latch before it is used.

3.5.1.18. 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 **TSC691E** that the memory system was unable to supply a valid instruction or data. If MEXC is asserted 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.

MEXC is used as a qualifier for the MDS signal, and must be asserted when both MHOLDA/B and MDS are already asserted. If MDS is applied without MEXC, the **TSC691E** accepts the contents of the data bus as valid. If MEXC accompanies MDS, an exception is generated and the data bus content is ignored.

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

3.5.1.19.MHOLD(A/B)—Memory Holds (inputs)

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 MHOLDA is asserted and the **TSC691E** outputs revert to and maintain the value they had at the rising edge of the clock in the cycle in which 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.

MHOLDB behaves in the same fashion as 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 (MHOLDA, MHOLDB, and BHOLD). All HOLD signals are latched in the **TSC691E** (transparent latch with clock high) before they are used.

Note that MHOLD must be driven HIGH while RESET is LOW.

3.5.1.20. 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.

3.5.1.21. 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 **TSC691E** 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. The SIZE[1:0] pins are three-state (on chip pull_up resistor=20kΩ) when the $\overline{\text{COE}}$ or $\overline{\text{TOE}}$ signal is deaserted. Encoding of the size bits is shown in Table 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 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)

3.5.1.22. $\overline{\text{WE}}$—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, $\overline{\text{WE}}$ must be externally qualified by the $\overline{\text{MHOLDA/B}}$ signals.

3.5.1.23. 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.

3.5.1.24. IMPAR—IU to MEC Control Parity (output)

This signal contains the odd parity over the DXFER, LDSTO, LOCK, RD, $\overline{\text{WE}}$ and WRT bits. The parity bit is generated by the IU and will be checked by the MEC.

It will be high-Z (on chip pull_up resistor=20kΩ) when the $\overline{\text{COE}}$ or $\overline{\text{TOE}}$ signal is deasserted.

3.5.2. 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:

3.5.2.1. CCC[1:0]—Coprocessor Condition Codes (input)

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.

3.5.2.2. CCCV—Coprocessor Condition Codes Valid (input)

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 3.9). 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 **TSC691E** before it is used.

3.5.2.3. $\overline{\mathrm{CEXC}}$—Coprocessor Exception (input)

$\overline{\mathrm{CEXC}}$ is used to signal the integer unit that a coprocessor exception has occurred. $\overline{\mathrm{CEXC}}$ must remain asserted until the **TSC691E** takes the trap and acknowledges the coprocessor exception via the CXACK signal. Although coprocessor exceptions can occur at any time, they are taken by the **TSC691E** only during the execution of a subsequent FPop, a CBfcc instruction, or a coprocessor load or store instruction. A coprocessor implementation should deassert $\overline{\mathrm{CHOLD}}$ if it detects an exception while $\overline{\mathrm{CHOLD}}$ is asserted. In such a case, $\overline{\mathrm{CEXC}}$ should be asserted one cycle before $\overline{\mathrm{CHOLD}}$ is deasserted. $\overline{\mathrm{CEXC}}$ is latched in the **TSC691E** before it is used.

3.5.2.4. $\overline{\mathrm{CHOLD}}$—Coprocessor Hold (input)

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 $\overline{\mathrm{CHOLD}}$ (if necessary) in the next cycle. If the integer unit receives a $\overline{\mathrm{CHOLD}}$, it freezes the instruction pipeline in the same cycle. Once the conditions causing the $\overline{\mathrm{CHOLD}}$ are resolved, the coprocessor deasserts $\overline{\mathrm{CHOLD}}$, releasing the instruction pipeline. $\overline{\mathrm{CHOLD}}$ is latched in the **TSC691E** before it is used.

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

3.5.2.5. CINS1—Coprocessor Instruction in Buffer 1 (output)

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.

3.5.2.6. CINS2—Coprocessor Instruction in Buffer 2 (output)

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.

3.5.2.7. $\overline{\mathrm{CP}}$—Coprocessor Unit Present (input)

When pulled low, $\overline{\mathrm{CP}}$ 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 $\overline{\mathrm{CP}}$=1 during the execution of an CPop, CBfcc, or coprocessor load or store instruction.

3.5.2.8. CXACK—Coprocessor Exception Acknowledge (output)

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

3.5.2.9. 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.

3.5.2.10. 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 FPU deasserts FCCV (FCCV=0) as soon as a floating-point compare instruction enters the floating-point queue, unless an exception is detected. 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 **TSC691E** before it is used.

3.5.2.11. $\overline{\text{FEXC}}$—Floating-Point Exception (input)

$\overline{\text{FEXC}}$ is used to signal the integer unit that a floating-point exception has occurred. $\overline{\text{FEXC}}$ must remain asserted until the **TSC691E** 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 **TSC691E** only during the execution of a subsequent FPop, an FBfcc instruction, or a floating-point load or store instruction. The FPU deasserts $\overline{\text{FHOLD}}$ if it detects an exception while $\overline{\text{FHOLD}}$ is asserted. In such a case, $\overline{\text{FEXC}}$ is asserted one cycle before $\overline{\text{FHOLD}}$ is deasserted. $\overline{\text{FEXC}}$ is latched in the **TSC691E** before it is used.

3.5.2.12. $\overline{\text{FHOLD}}$—Floating-Point Hold (input)

This signal is asserted by the FPU 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 $\overline{\text{FHOLD}}$ (if necessary) in the next cycle. If the integer unit receives an $\overline{\text{FHOLD}}$, it freezes the instruction pipeline in the same cycle. Once the conditions causing the $\overline{\text{FHOLD}}$ are resolved, the FPU deasserts $\overline{\text{FHOLD}}$, releasing the instruction pipeline. $\overline{\text{FHOLD}}$ is latched in the **TSC691E** before it is used.

An $\overline{\text{FHOLD}}$ 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.

3.5.2.13. FIPAR—FPU to IU Control Parity (input)

This signal contains the odd parity over the FCC[1:0], FCCV, $\overline{\text{FEXC}}$ and $\overline{\text{FHOLD}}$ bits. The parity bit is generated by the FPU and will be checked by the IU.

3.5.2.14. 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. The FPU uses this signal to begin decoding and execution of the D1 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.

3.5.2.15. 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. 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.

3.5.2.16. 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 **TSC691E**'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 $\overline{\text{FP}}$ (or $\overline{\text{FP}}$) queue must be emptied before the FPU (coprocessor) can resume execution.

3.5.2.17. $\overline{\text{FP}}$—Floating-Point Unit Present (input)

When pulled low, $\overline{\text{FP}}$ 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 $\overline{\text{FP}}$=1 during the execution of an FPop, FBfcc, or floating-point load or store instruction.

3.5.2.18. 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 $\overline{\text{FEXC}}$ signal. Receipt of the asserted FXACK causes the FPU to deassert $\overline{\text{FEXC}}$. FXACK is a latched output and should not be latched externally.

3.5.2.19. 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. 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.

3.5.2.20. IFPAR—IU to FPU Control Parity (output)

This signal contains the odd parity over the FINS1, FINS2, FLUSH, FXACK and INST bits. The parity bit is generated by the IU and will be checked by the FPU. It will be high-Z (on chip pull_up resistor=20kΩ) when the TOE signal is deasserted.

3.5.3. Interrupt and Control Signals

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

3.5.3.1. 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 **TSC691E** 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 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 RESET signal.

3.5.3.2. HWERROR—Hardware Error (output)

The HWERROR outputs indicate a parity error occurs, except Master/Checker errors. When asserted low, the IU trap with Trap Type value depending of the internal parity error (see Table 39, page 109). It is deasserted when the parity error is removed (i.e. by resuming this instruction), or by a reset cycle.

3.5.3.3. FLOW —Enable Flow Control (input)

Forcing this input low will enable the program flow control. It is a static signal and shall not change when running.

3.5.3.4. MCERR—Comparison Error (output)

This signal is asserted low in checker mode when a comparison error occurs on the internal output signals vis-à-vis the output signal (excepted TAP, MCERR, HWERROR and ERROR signals) of the master IU. It is deasserted when the error disappears. See chapter 4.4 for more information.

This signal is also asserted in master mode when the output doesn't match the value of the pin.

This output is high-Z (on chip pull_up resistor=20kΩ) when the TOE signal is deasserted.

3.5.3.5. 601MODE—Normal 601 Mode Operation (input)

Forcing this input low will disable the parity checking of all input signals. This means the IU will operate with the standard input signals. Nevertheless generation and checking of internal parity bit is still active. Parity on the data bus is generated internally and parity checking on the control bus is disabled.

3.5.3.6. CMODE—Checker Mode (input)

Assertion of this signal will set the IU to act as a checker to support master/checker operation. All output signals except ERROR, HWERROR, MCERR and TAP signals will be high-Z (on chip pull_up resistor=20kΩ). It is a static signal and shall not change when running. CMODE signal can change when RESET signal is asserted or when the IU is in halt mode.

3.5.3.7. FPSYN—Floating-Point Synonym Mode (input)

This is a mode signal which will be used to allow execution of additional instructions in future designs. For the **TSC691E**, it should be kept grounded.

3.5.3.8. 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.

3.5.3.9. 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 **TSC691E**. Logic must also keep an interrupt valid until it is taken and acknowledged. External interrupts can be acknowledged by system software or by the **TSC691E**'s INTerrupt ACKnowledge (INTACK) signal.

3.5.3.10. $\overline{\text{RESET}}$—Integer Unit reset (input)

Assertion of this signal will reset the integer unit. $\overline{\text{RESET}}$ must be asserted for a minimum of nine processor clock cycles. After $\overline{\text{RESET}}$ is deasserted, the integer unit starts fetching from address 0. $\overline{\text{RESET}}$ is latched by the **TSC691E** before it is used.

The $\overline{\text{RESET}}$ signal input is protected by a glitch removal filter and pulses which are so short that they are detected only during one clock period are not influencing the IU. $\overline{\text{RESET}}$ signal is also protected with two-rail coding and an error detected will lead to error mode.

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

When *deasserted*, this signal will three-state all integer unit output drivers (on chip pull_up resistor=20kΩ). Thus, in normal operation, this signal should always be asserted (tied to ground). Deassertion of $\overline{\text{TOE}}$ isolates the **TSC691E** from the system for debugging purposes.

3.5.3.12. $\overline{\text{HALT}}$—Halt (input)

When asserted this input will freeze the IU pipeline and the clock. All information placed in the registers of the IU remains unchanged. By deasserting $\overline{\text{HALT}}$, execution of the IU will resume. (see timing section 5.2.2.14, page 127)

When the IU is in halt mode, the TAP is still operating.

3.5.4. TAP signals

The following Test Access Port interface (IEEE standard 1149.1-1990) is used to perform boundary scan for test and debugging purposes.

3.5.4.1. TCLK—Test Clock (input)

This clock signal permits test data to be shifted into or out of the instruction or test data register cells without interfering with the on chip system logic.The IEEE standards requires that TCLK can be stopped at 0 indefinitely without causing any change to the state of the test logic.

3.5.4.2. $\overline{\text{TRST}}$—TEST Reset (input)

The TAP's test logic is reset when a logical 0 is applied to this port.

3.5.4.3. TMS—Test Mode Select (input)

The TMS input signal is interpreted by the TAP controller to control the test operations.

The received signal is sampled at the rising edge of the TCLK pulses.

3.5.4.4. TDI—Test Data Input (input)

Serial input data applied to this port is fed either into the instruction register or into a test data register, depending on the sequence previously applied to the TMS input.

The received input data is sampled at the rising edge of the TCLK pulse.

3.5.4.5. TDO—Test Data Output

Depending on the sequence previously applied to the TMS input, the contents of either the instruction register or the data register are serially shifted out toward the TDO.

The data out of the TDO is clocked at the falling edge of the TCLK pulses. TDO should be in the inactive state except when scanning is in progress. (Use of 3 state driver)

3.5.5. Power and Clock Signals

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

3.5.5.1. 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.

3.5.5.2. VCCO, VCCI, VCCT—Power (inputs)

These pins provide +5V power to various sections of the processor. Power is supplied on three different busses 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.

3.5.5.3. VSSO, VSSI, VSST—Ground (inputs)

These pins provide ground return for the power signals. Ground is supplied on three different busses 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.

3.6. Pipeline and Instruction Execution Timing

One of the major contributing factors to the **TSC691E**'s very high performance is an instruction execution rate approaching one instruction per clock cycle. To achieve that rate of execution, the **TSC691E** employs a four-stage instruction pipeline that permits parallel execution of multiple instructions.

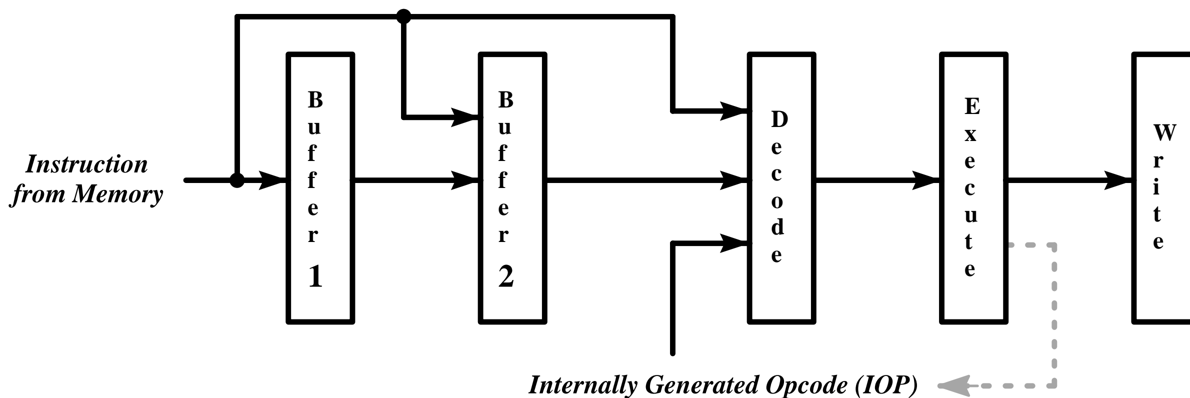


Figure 24. Processor Instruction Pipeline

3.6.1.Stages

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

2. **Fetch**—The processor outputs the instruction address to fetch the instruction.
3. **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.
4. **Execute**—The processor executes the instruction and saves the results in temporary registers. Pending traps are prioritized and internal traps taken during this stage.
5. **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 25).

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.

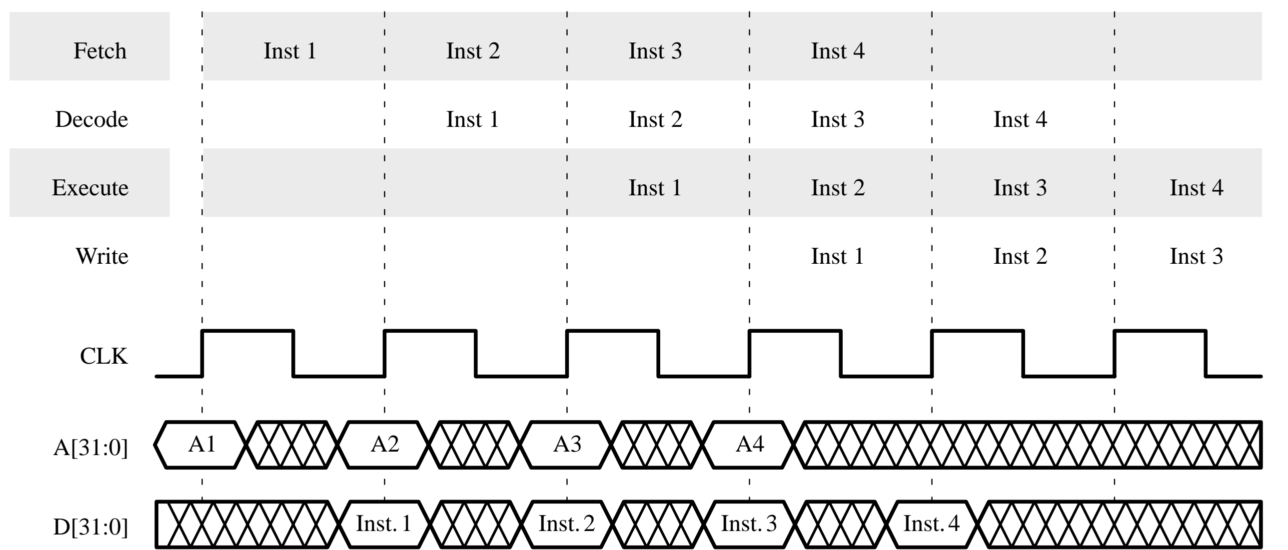


Figure 25. Pipeline with All Single–Cycle Instructions

3.6.1.1. 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 26 , 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 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 24). 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 36. Internally Generated Opcodes

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

3.6.2. 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 26, the fetch of instruction Inst 3 is delayed by one cycle for the data load, and in Figure 27, the store sequence delays the Inst 3 fetch by two cycles.

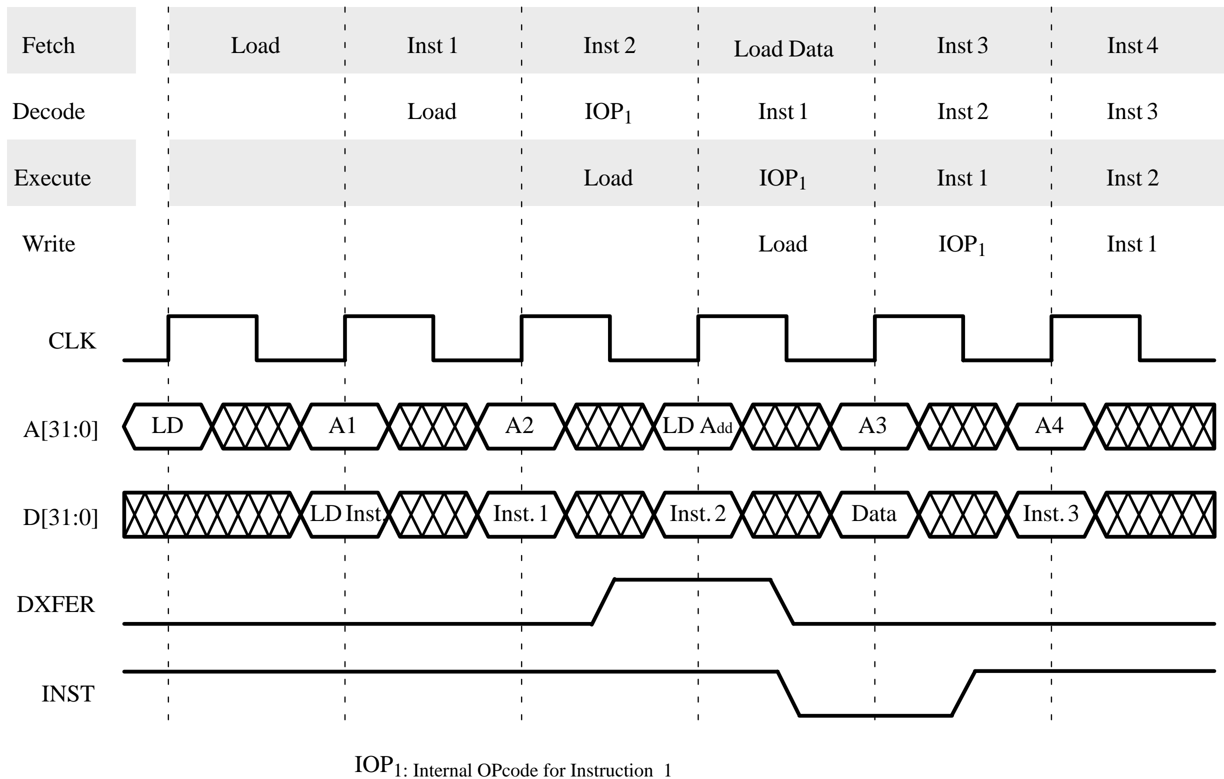


Figure 26. Pipeline with One Double–Cycle Instruction (Load)

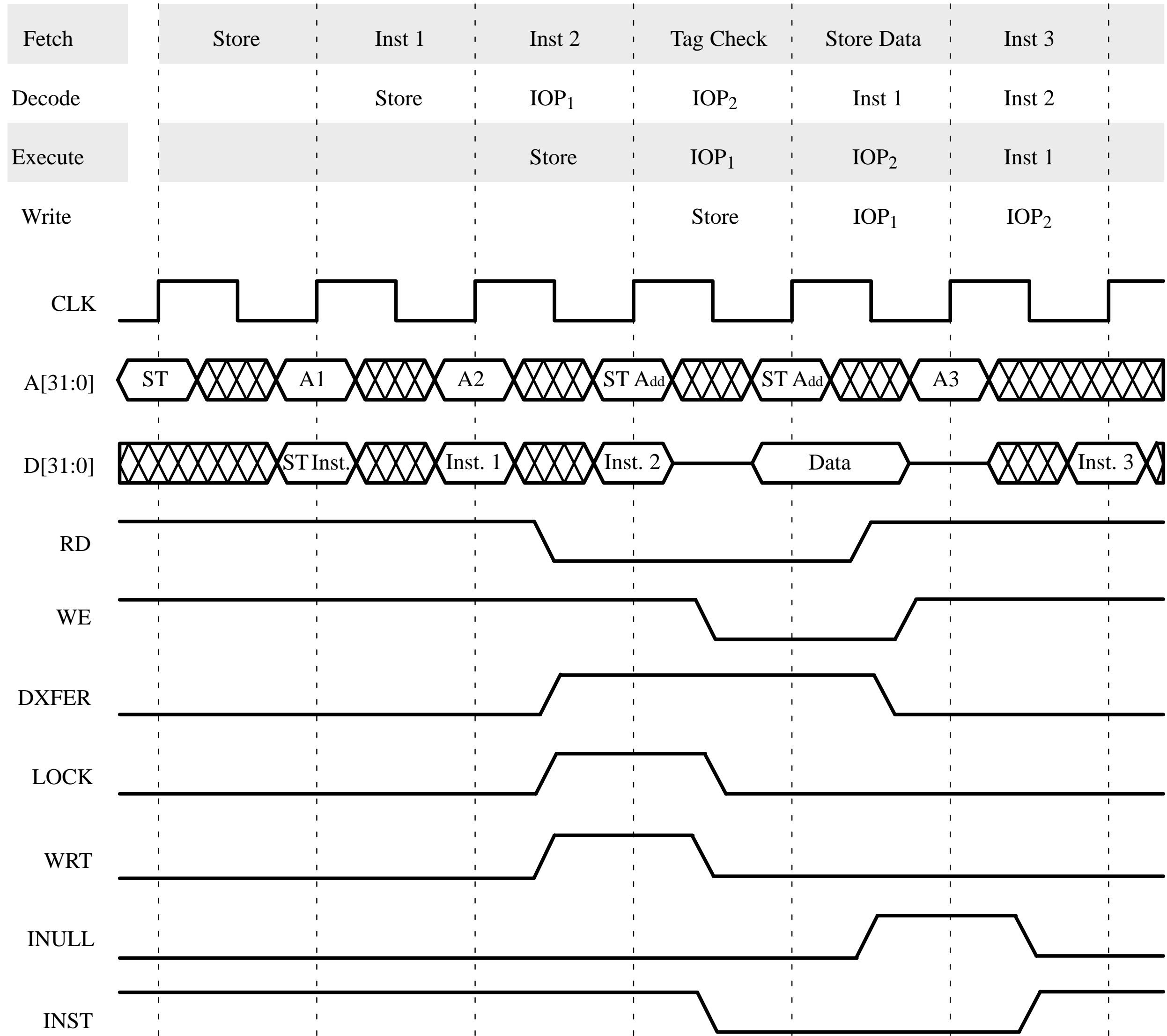


Figure 27. Pipeline with One Triple–Cycle Instruction (Store)

3.6.2.1. 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 **TSC691E** 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 28). This also applies to 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.

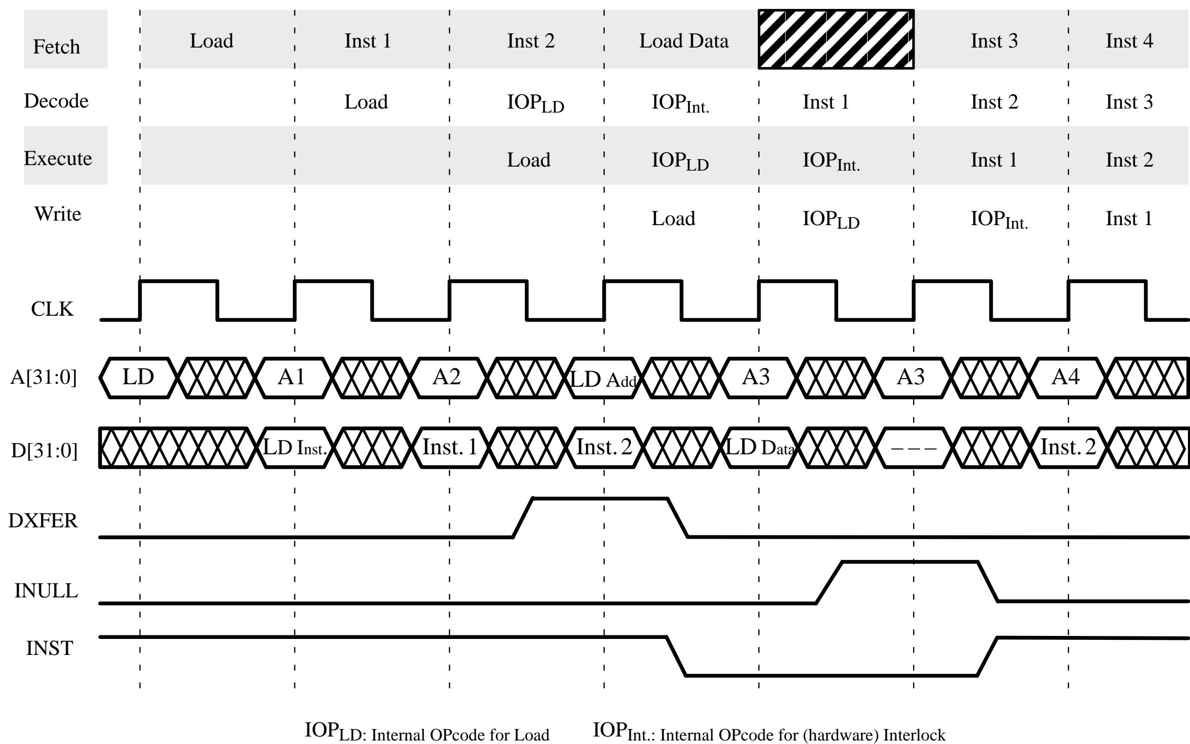


Figure 28. Pipeline with Hardware Interlock (Load)

3.6.2.2. Branching

The **TSC691E**'s delayed-control-transfer mechanism allows branches (taken or untaken) to occur without creating a bubble in the pipeline (see Figure 29). 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 3.4.3.3.1 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 30 .

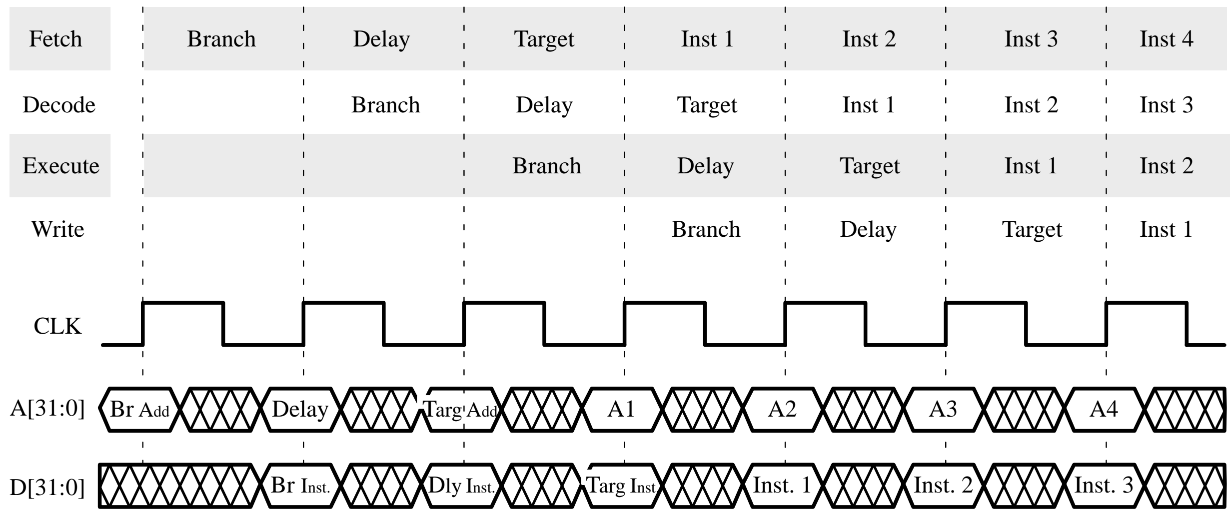


Figure 29. Pipeline During Branch Instruction

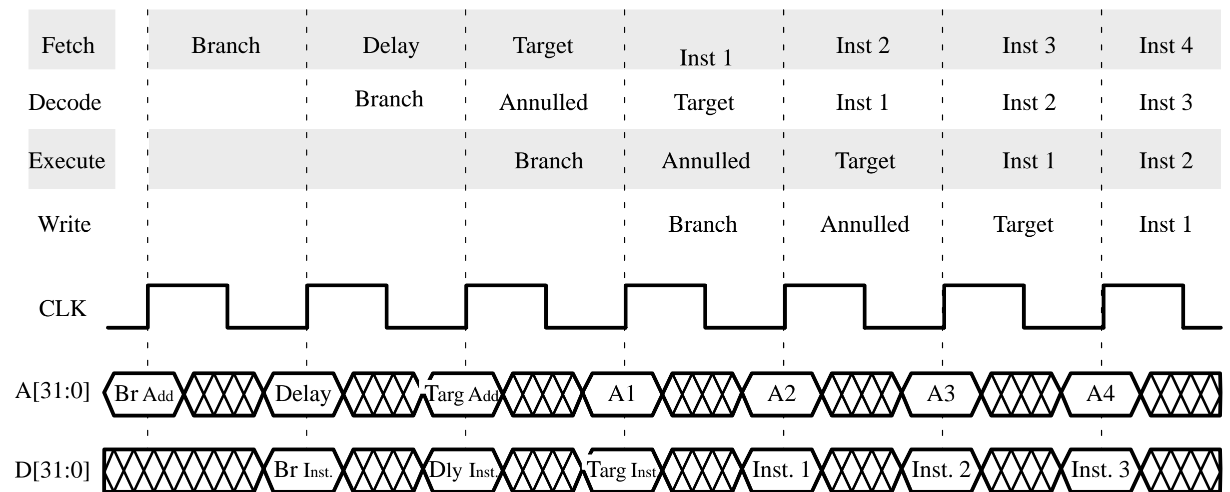


Figure 30. Branch with Annulled Delay Instruction

3.6.3. 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 31 shows the pipeline frozen by a $\overline{\text{BHOLD}}$ as the result of bus arbitration initiated by another bus master in the system.

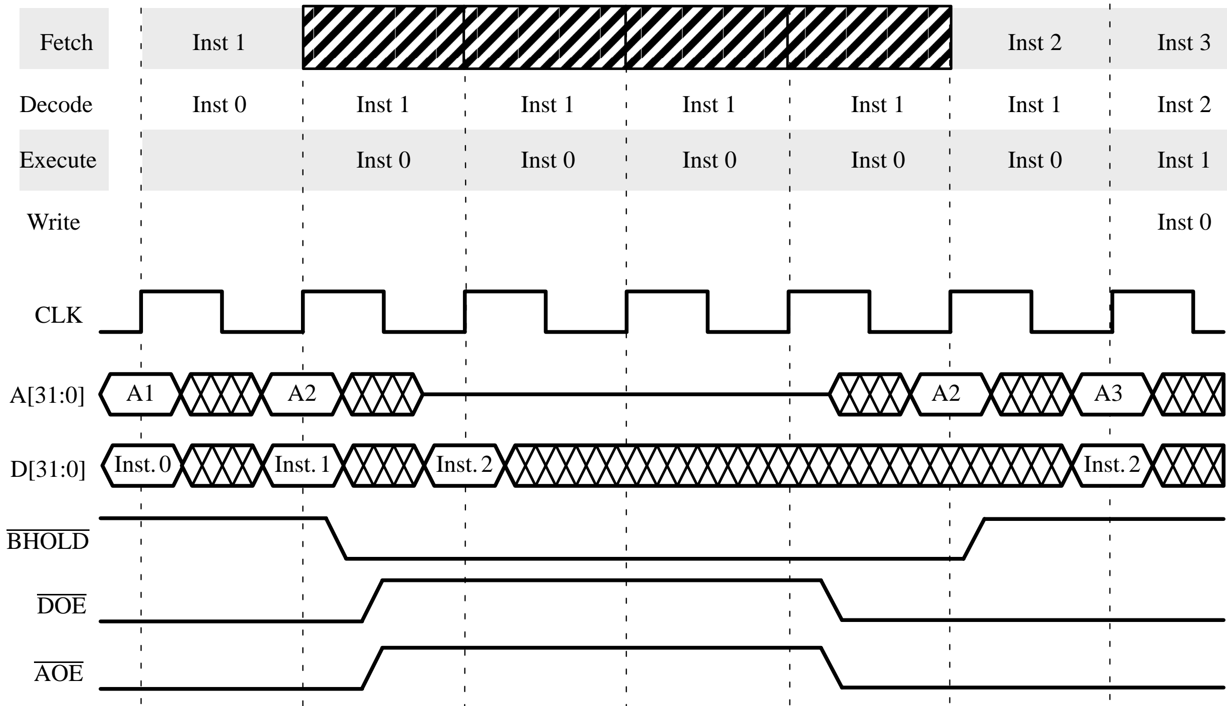


Figure 31. Pipeline Frozen During Bus Arbitration

3.6.4. Traps

Figure 32 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.

3.6.5. Traps

Figure 32 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.

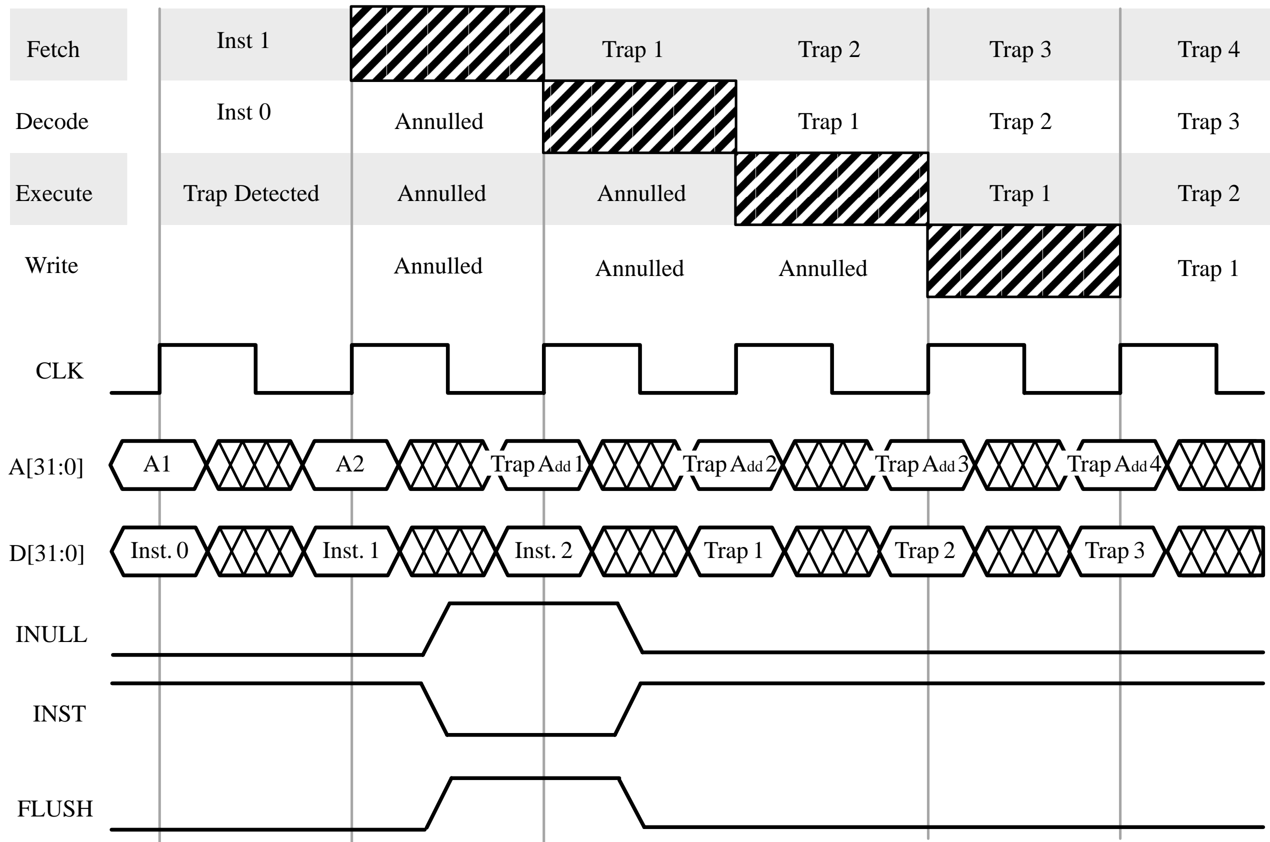


Figure 32. Pipeline Operation for Taken Trap (Internal)

3.7. Bus Operation and Timing

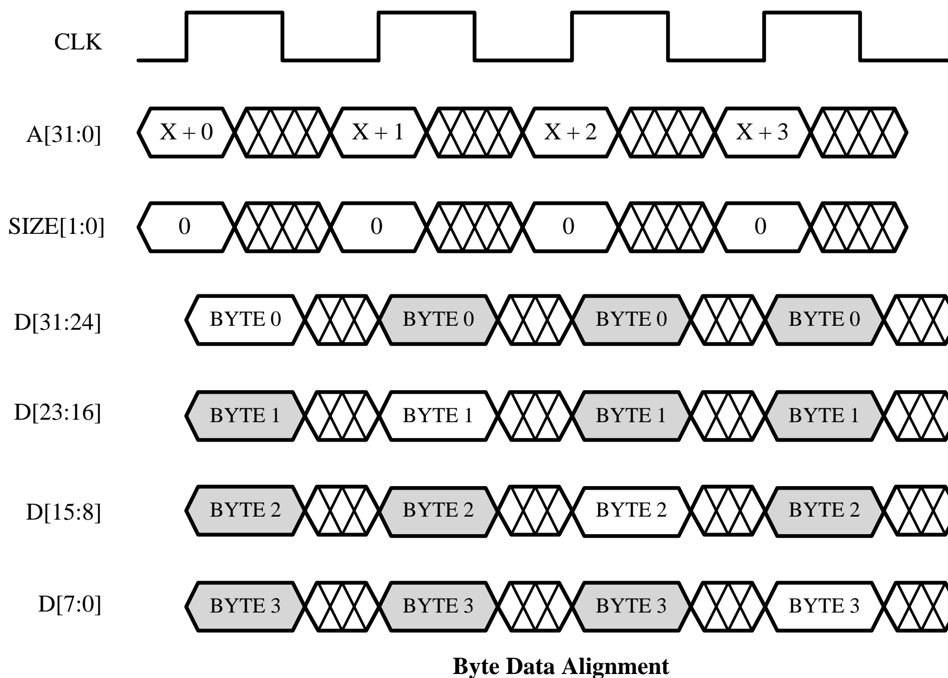
This section covers standard and non-standard bus operations. Standard operations include instruction fetch, load integer, load double integer, load floating-point, load double floating-point, store integer, store double integer, store floating-point, store double floating-point, atomic load-store unsigned 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.

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 **TSC691E**'s high-speed operation, many signals are sent out unlatched. Refer to Section 3.5 for further details on **TSC691E** signals.

The processor automatically aligns byte (and halfword) transfers as previously shown in Figure 12. Firures 33 & 34 show 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 **TSC691E** 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 33 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.



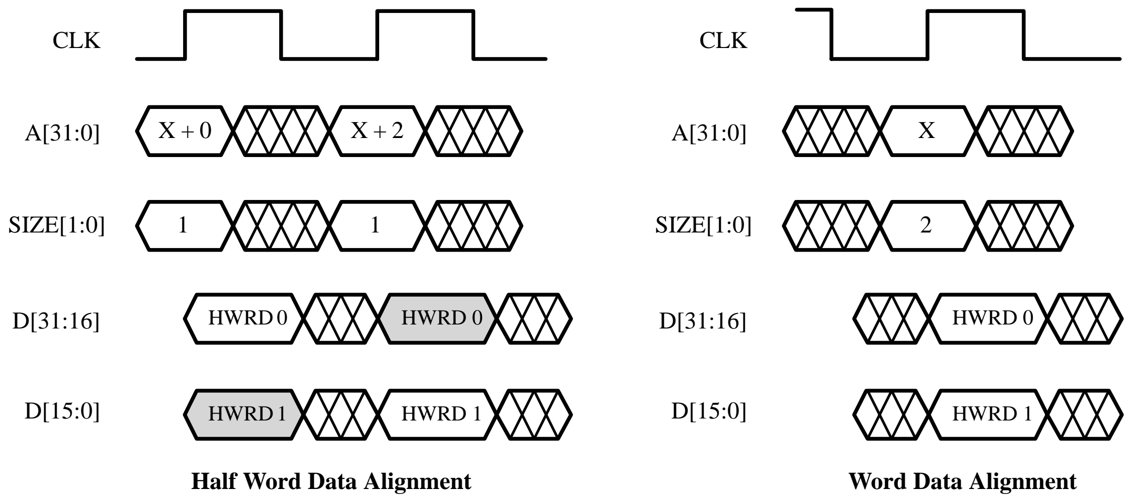
Byte Data Alignment

X = word boundary address

Note 1: The parity bit of undef data in/out must match with the data

Note 2 :This illustration depicts data alignment and is not intended to illustrate a timing case.

Figure 33. Data Bus Contents During Data Transfers (1 of 2)



X = word boundary address

Note 1: The parity bit of undef data in/out must match with the data

Note 2 :This illustration depicts data alignment and is not intended to illustrate a timing case.

Figure 34. Data Bus Contents During Data Transfers (2 of 2)

3.7.1. 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 busses (see Figure 35). 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.

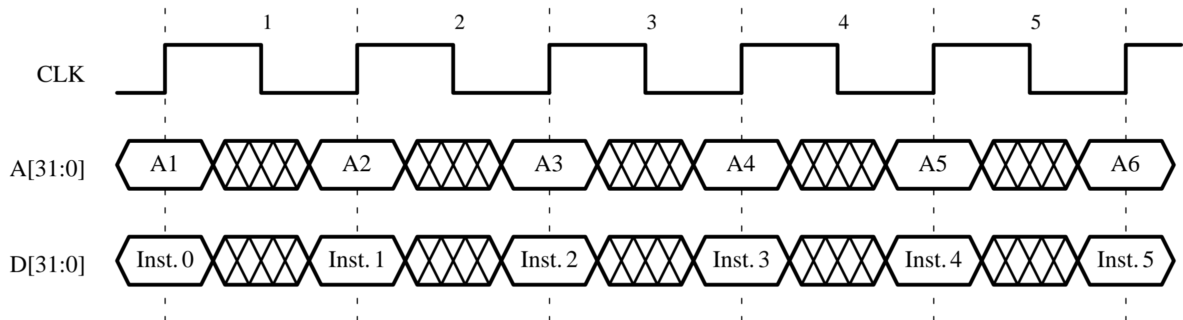


Figure 35. Instruction Fetch

3.7.2. Load

Figure 36 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.

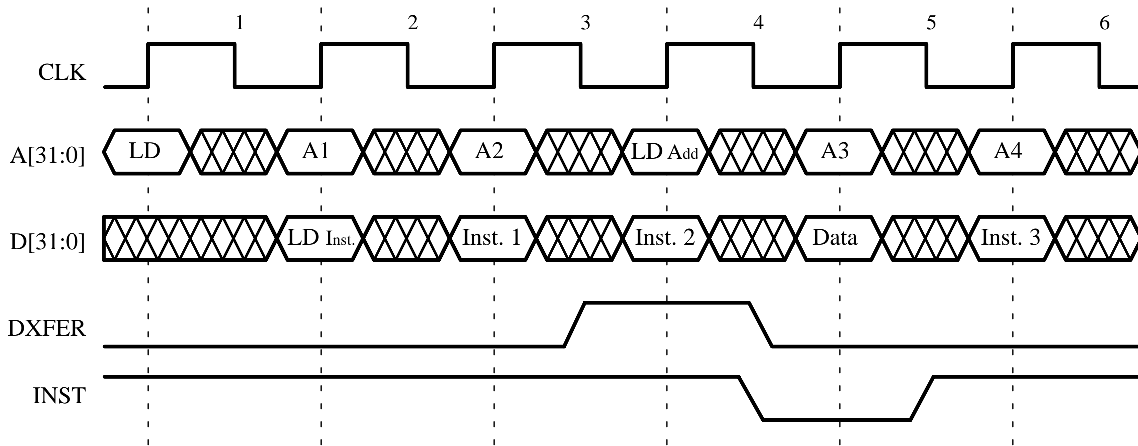


Figure 36. Load Single Integer Timing

3.7.3. 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 3.6.5.1) in the fourth cycle, which must be matched by a null bus cycle in the fetch stage to balance the pipeline (see Figure 37).

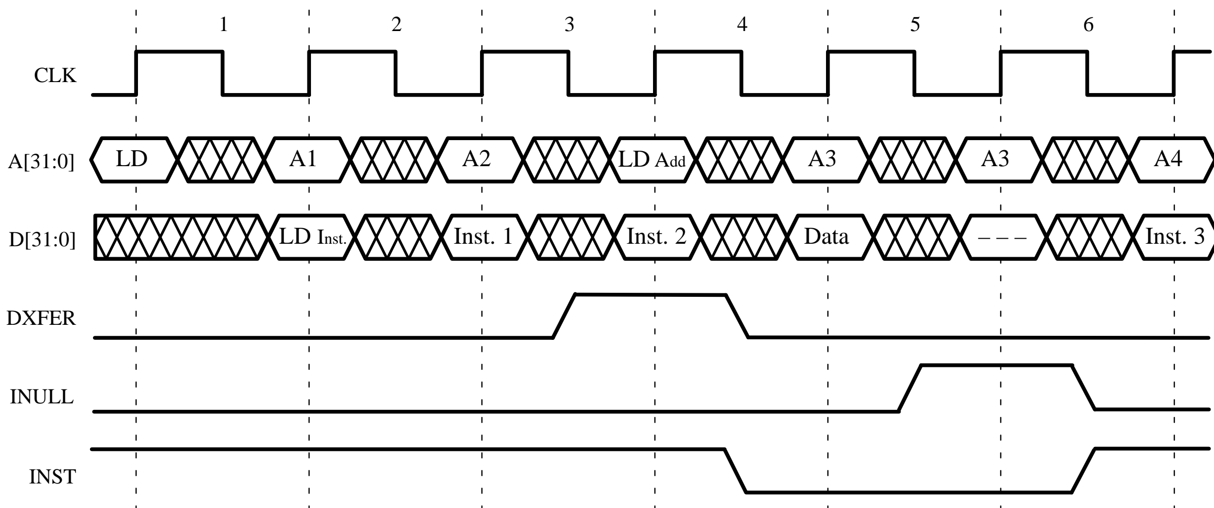


Figure 37. Load Single with Interlock Timing

3.7.4. Load Double

The timing for a load double integer is shown in Figure 38 . 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.

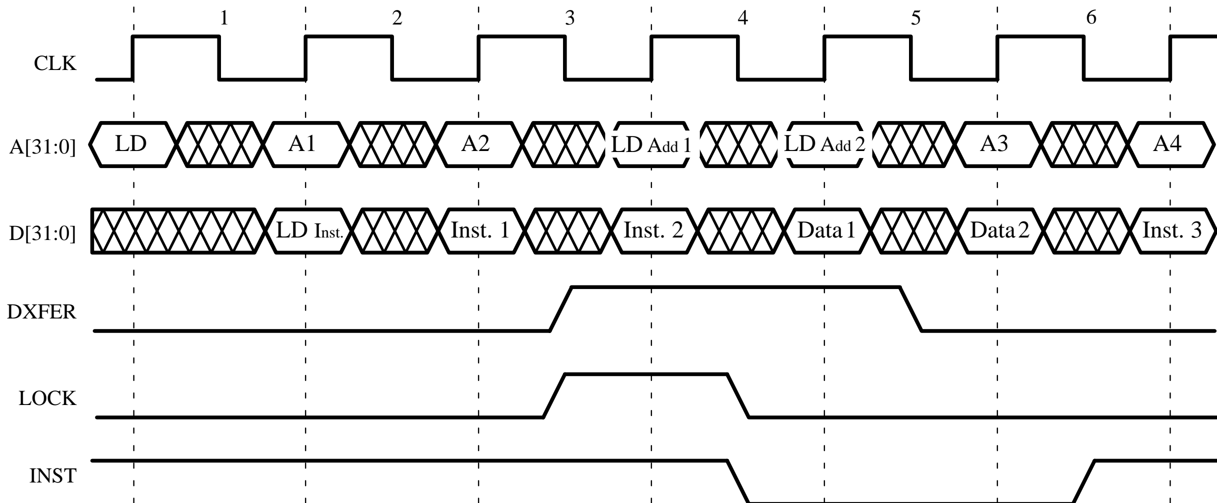


Figure 38. Load Double Integer Timing

3.7.5. Store

Store transactions involve more bus activity than loads, as shown in the store single integer timing in Figure 39 . 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 (on chip pull_up resistor=20kΩ)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.

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 busses.

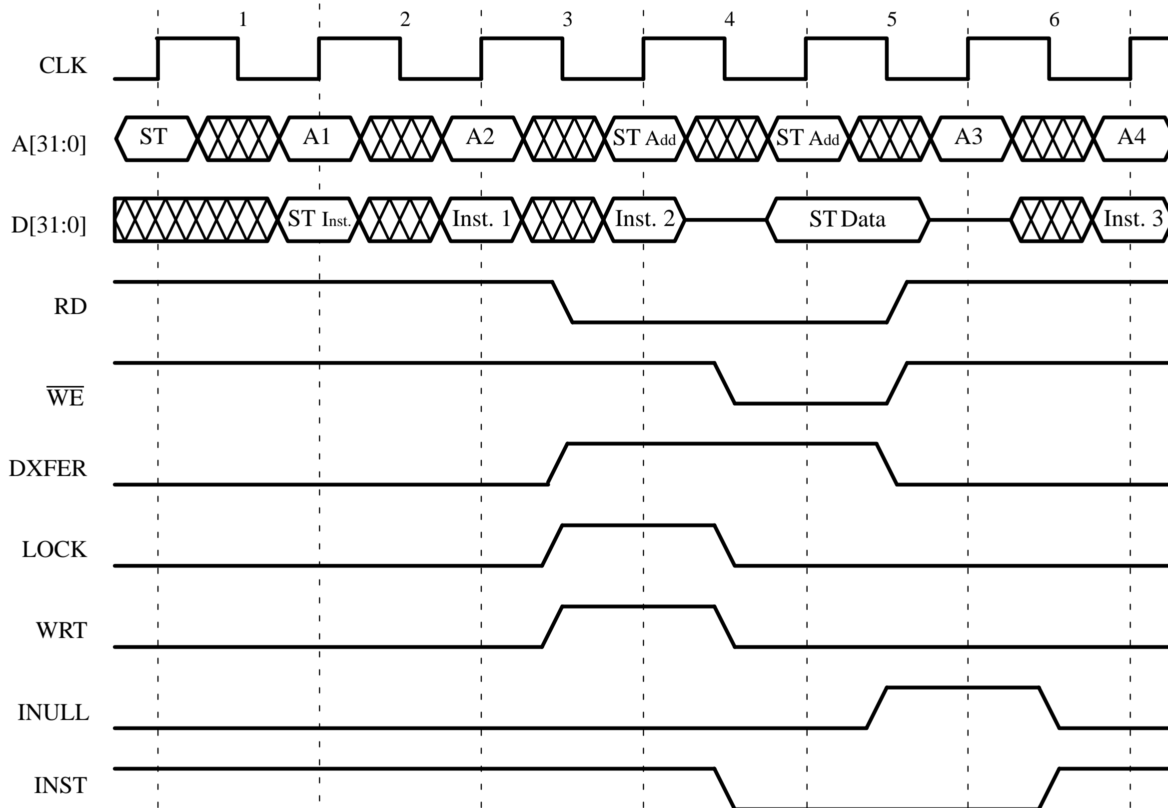


Figure 39. Store Single Integer Timing

3.7.6. Store Double

The timing for a store double integer is shown in Figure 40 . 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.

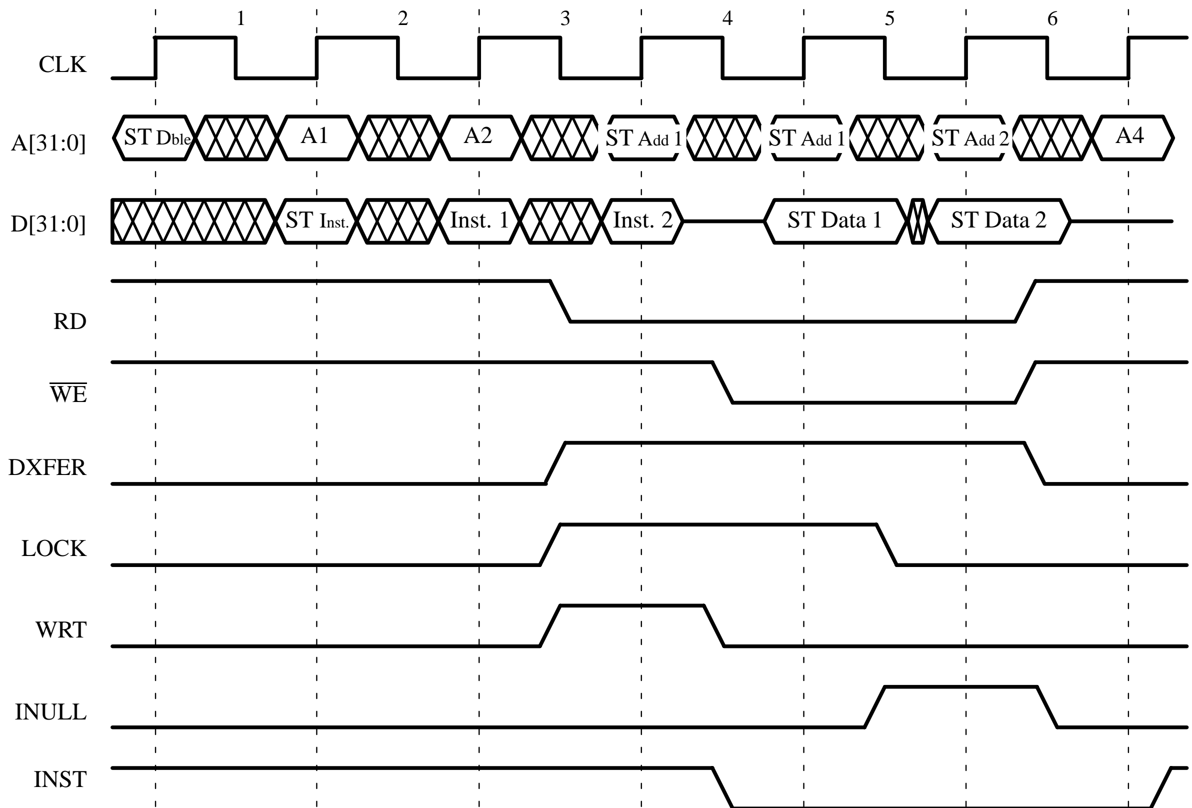


Figure 40. Store Double Integer Timing

3.7.7. 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 **TSC691E** asserts LOCK for as long as necessary to make sure that no interruption occurs on the bus. Figure 41 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.

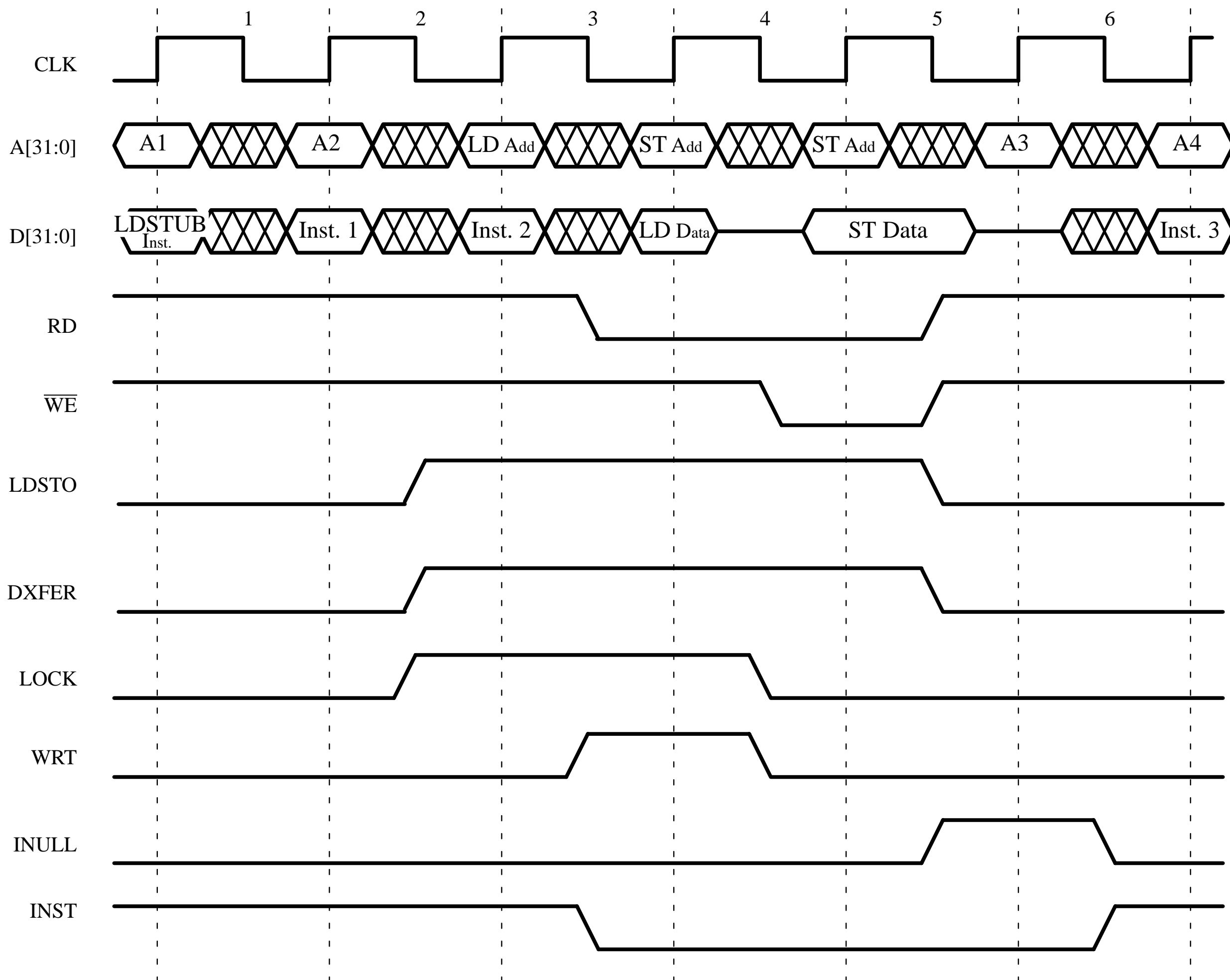


Figure 41. Atomic Load–Store Timing

3.7.8. 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 42). 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.

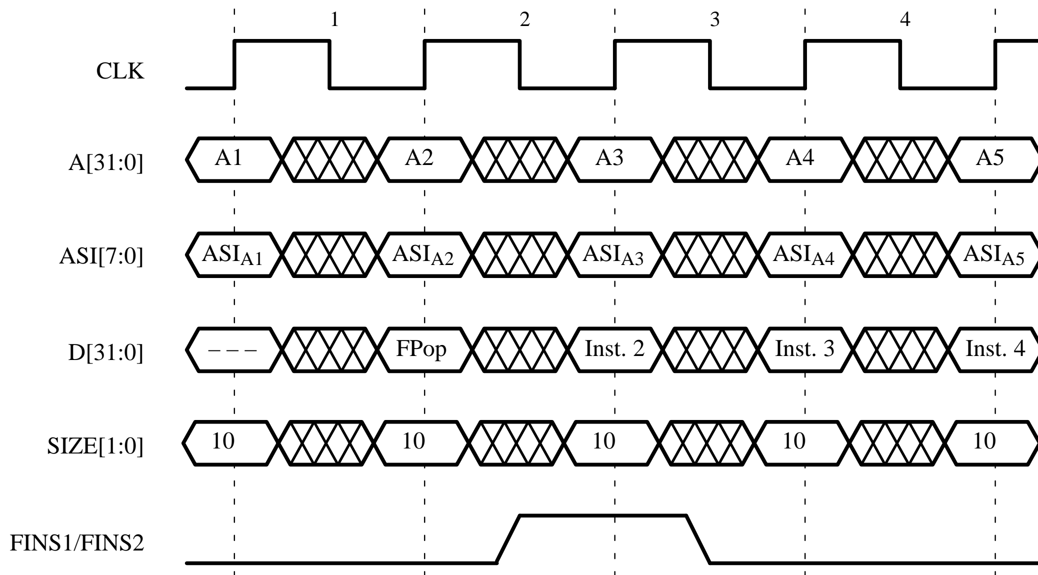


Figure 42. Floating–Point Operation Timing

3.7.9. Bus Arbitration

The **TSC691E** 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 **TSC691E**. When the **TSC691E** needs to retain the busses it asserts the LOCK signal. The arbitration circuitry should assert $\overline{BHOLD}$ when it needs to keep the **TSC691E** off the busses. When $\overline{BHOLD}$ is asserted, the processor's instruction pipeline is frozen until it is deasserted. The arbitration circuitry should also deassert the $\overline{DOE}$, $\overline{AOE}$, and $\overline{COE}$ signals to three-state the **TSC691E**'s address bus (on chip pull_up resistor=20kΩ), data bus and control signal output drivers so they may be driven by an external source (see Figure 43).

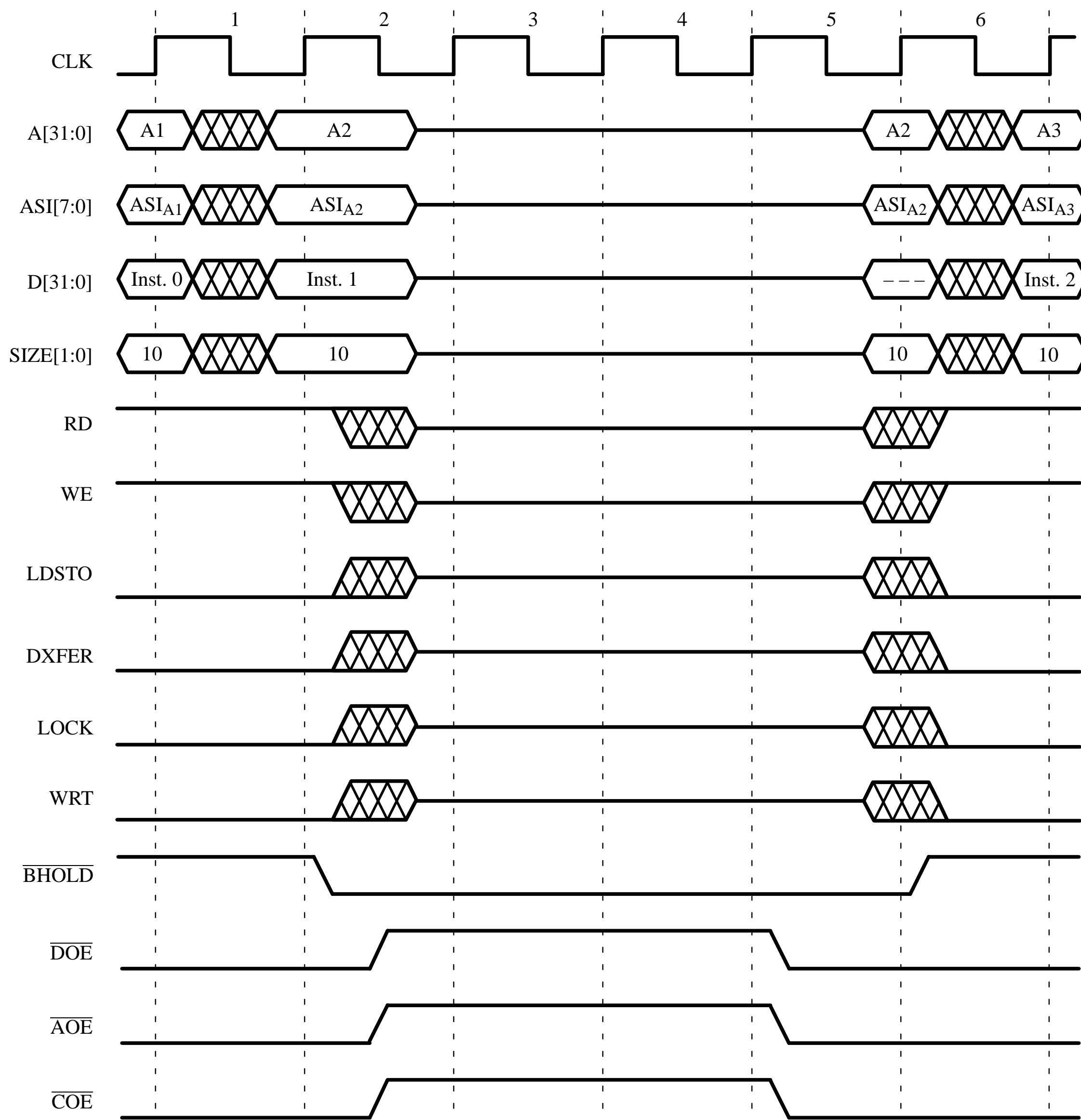


Figure 43. Bus Arbitration Timing

3.7.10. Load with Cache Miss

Figure 44 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 $\overline{\text{MHOLD}}$).

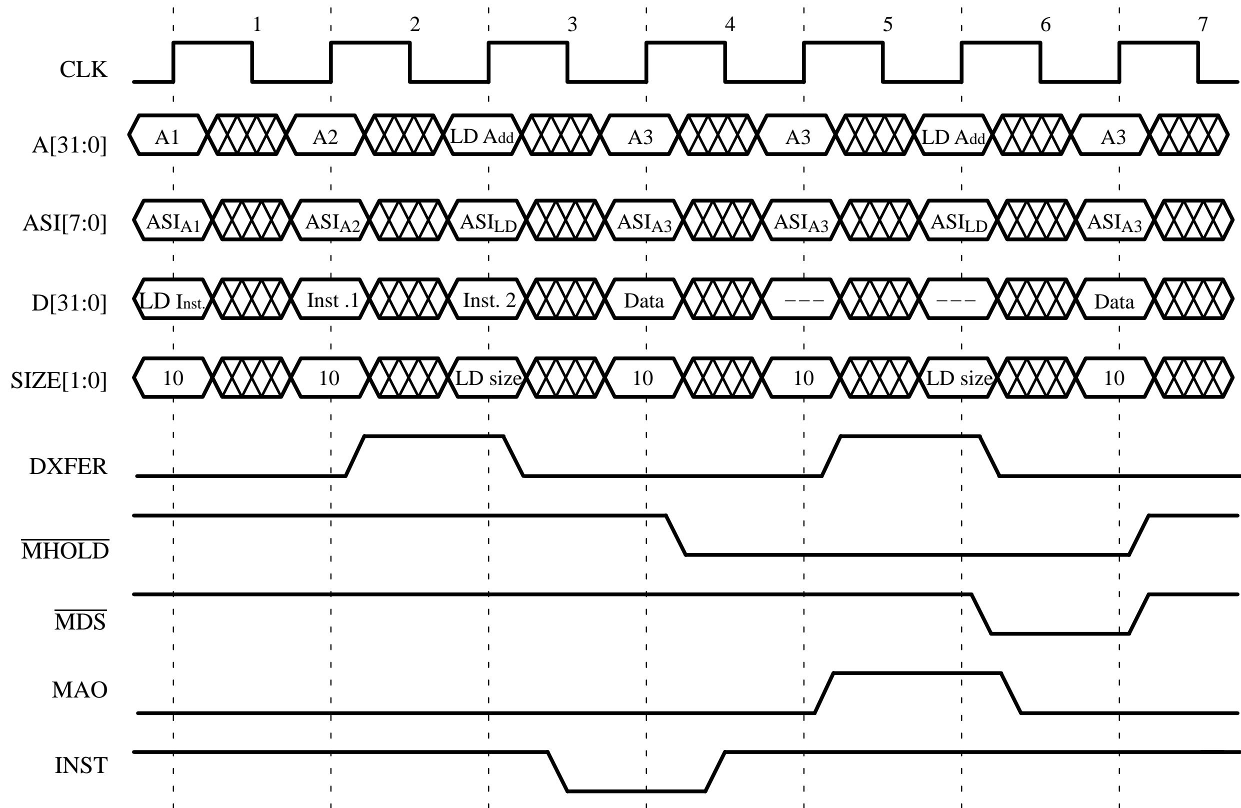


Figure 44. Load with Cache Miss Timing

3.7.11. 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 Figures 45 & 46). 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.

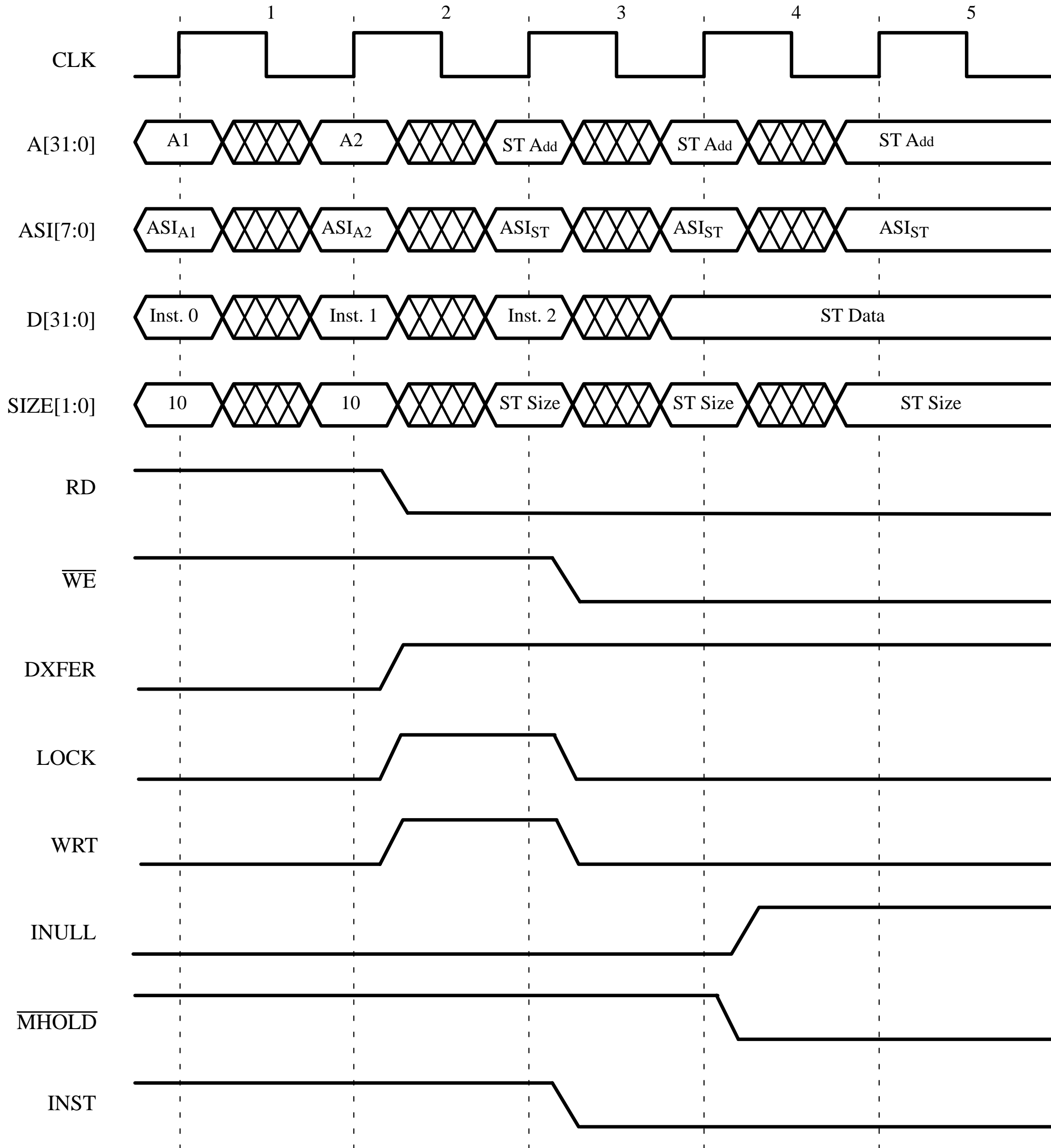


Figure 45. Store with Cache Miss Timing (1 of 2)

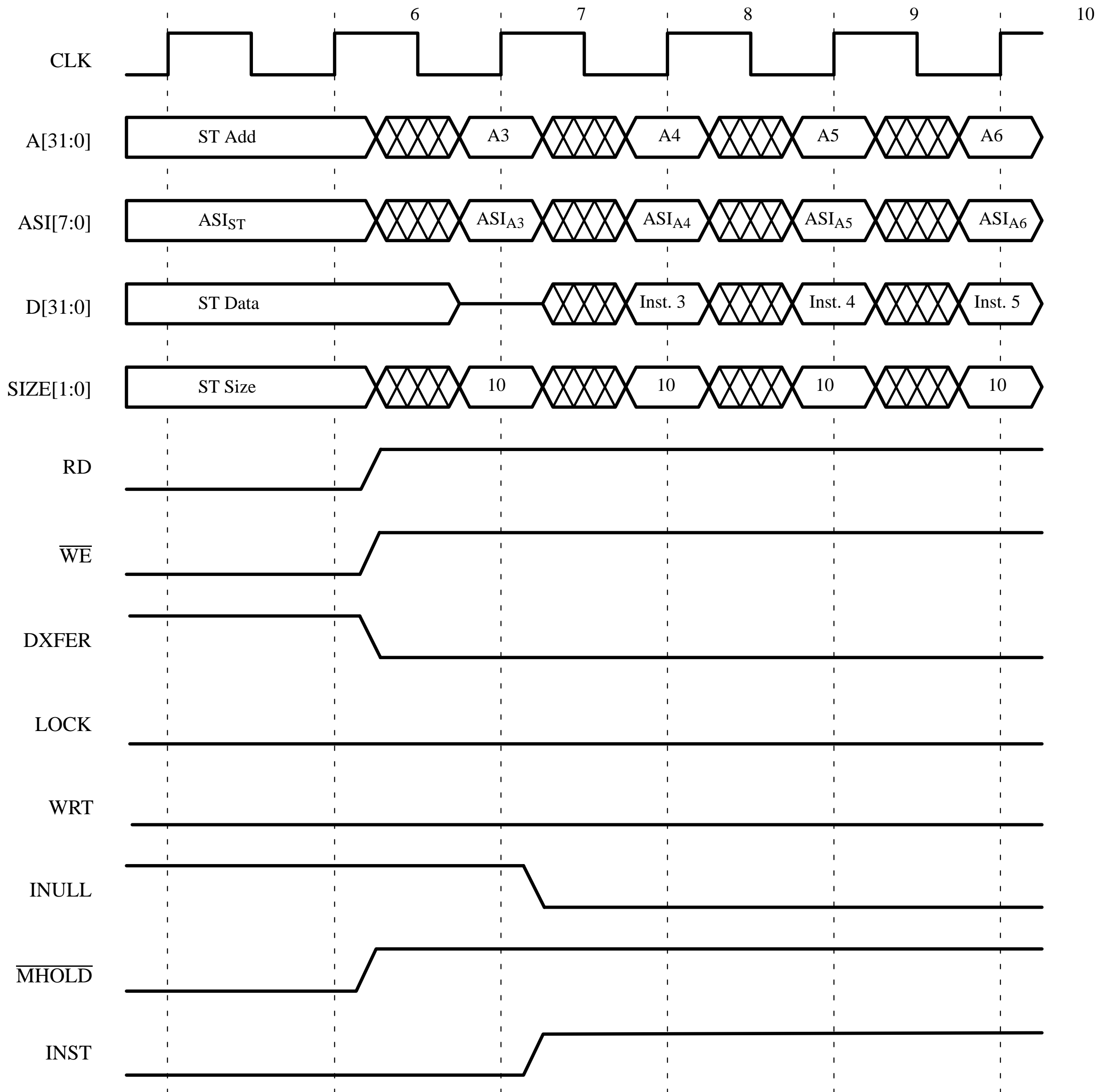


Figure 46. Store with Cache Miss Timing (2 of 2)

3.7.12. Load/Store instruction with Trap

Figure 47 gives the timing for a load instruction with a trap taken. This timing is similar for the load double, for the load-store, for the store and for the swap instructions.

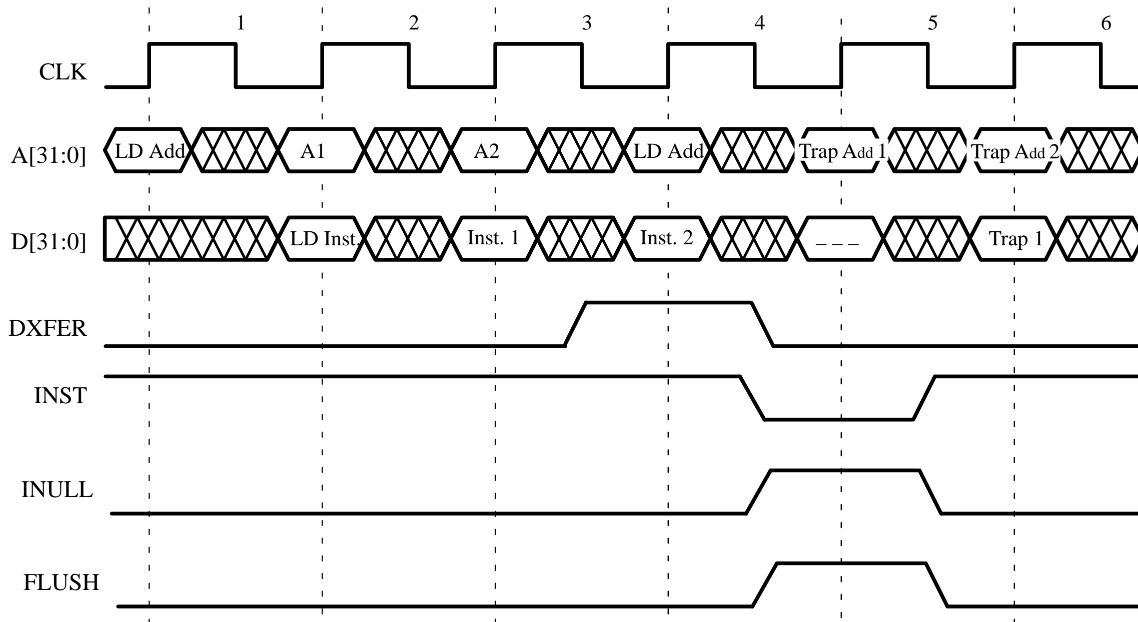


Figure 47. Ld, LdSt, St and Swap Inst with Trap Taken

3.7.13. Memory Exceptions

Load with memory exception timing is shown in Figures 48 & 49. As with a cache miss, memory logic must stop the processor by asserting $\overline{MHOLDA}$ or $\overline{MHOLDB}$ in the next cycle. The $\overline{MHOLD}$ signal must be maintained while the memory exception ($\overline{MEXC}$) signal is strobed into the processor with the $\overline{MDS}$ signal (it must be strobed in externally because the internal processor clock is frozen by the $\overline{MHOLD}$). $\overline{MEXC}$ must be deasserted in the same clock cycle in which $\overline{MHOLD}$ 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 32 for an internal trap. Store with memory exception has the same timing (see Figures 52 & 53) except INULL is asserted from the second store address through to the annulled cycle 8 instruction fetch.

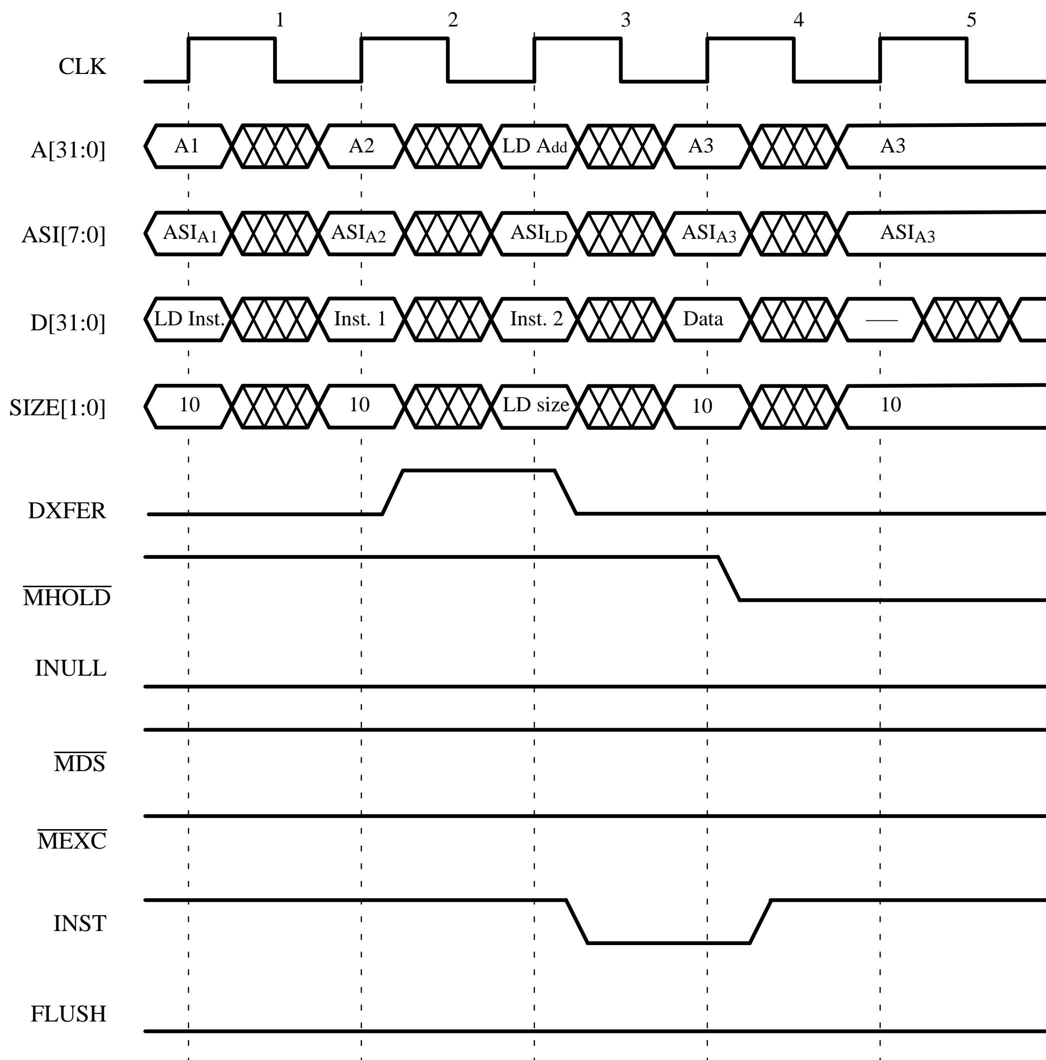


Figure 48. Load with Memory Exception Timing (1 of 2)

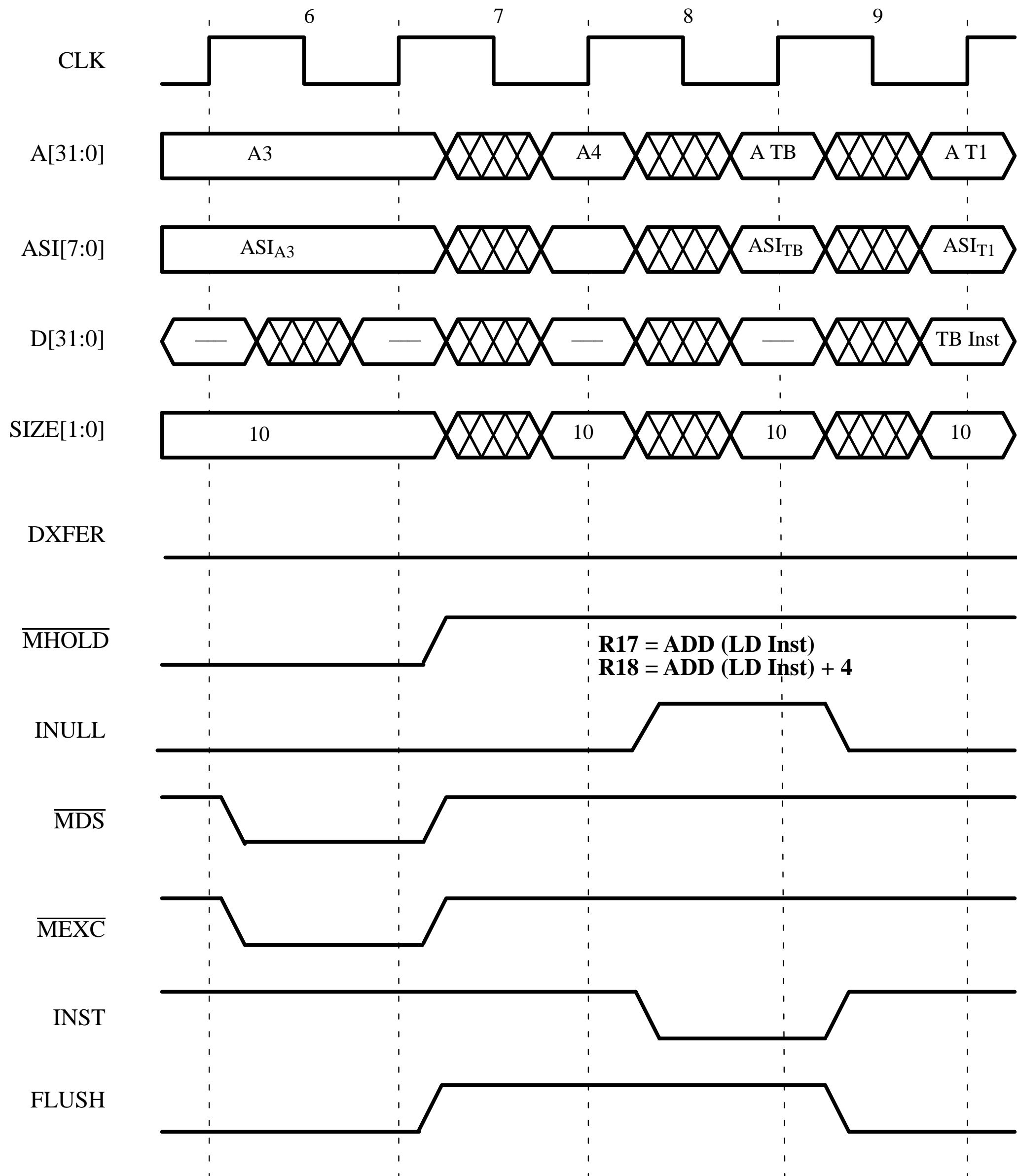


Figure 49. Load with Memory Exception Timing (2 of 2)

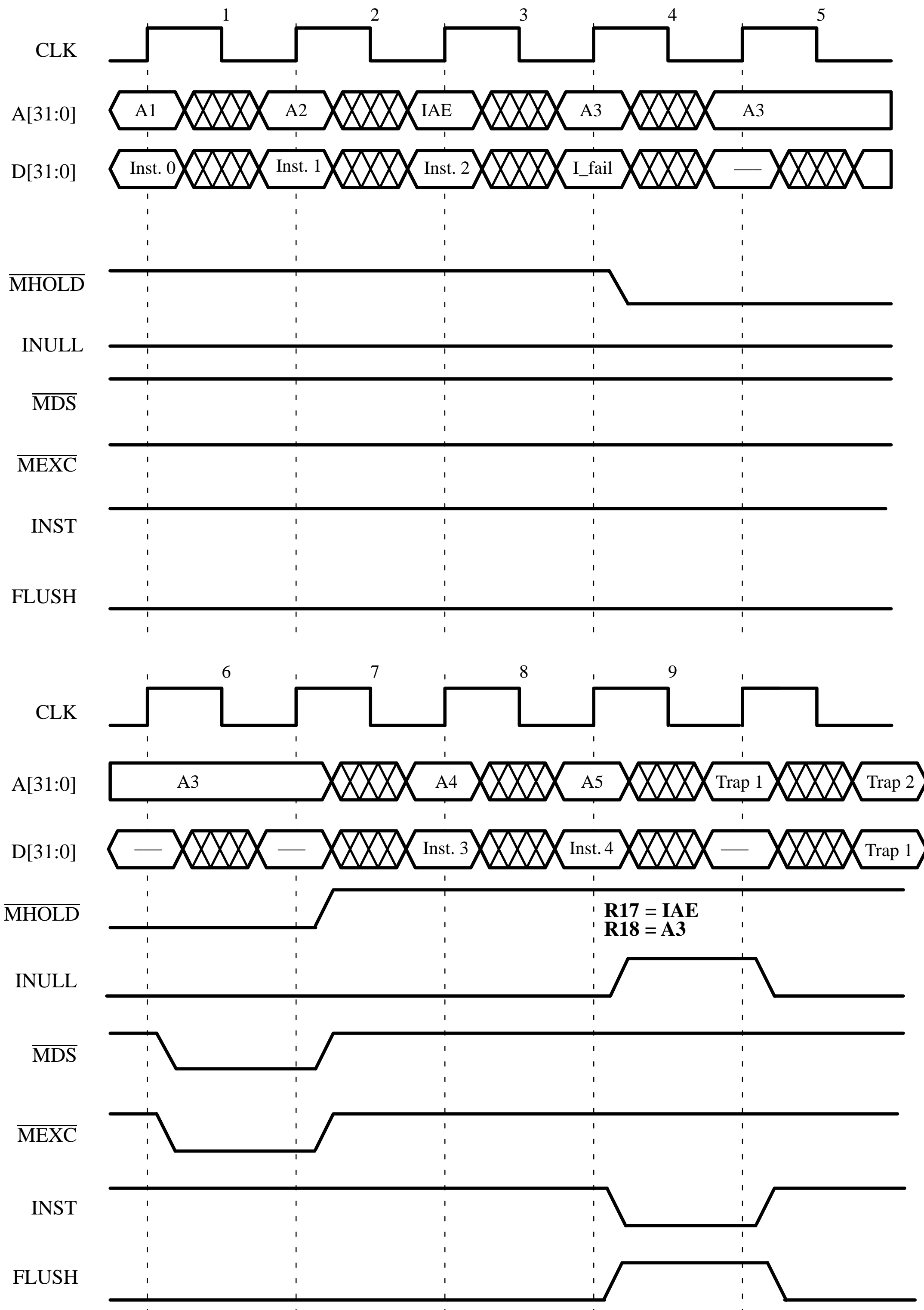


Figure 50. Instruction Memory Access Exception Timing

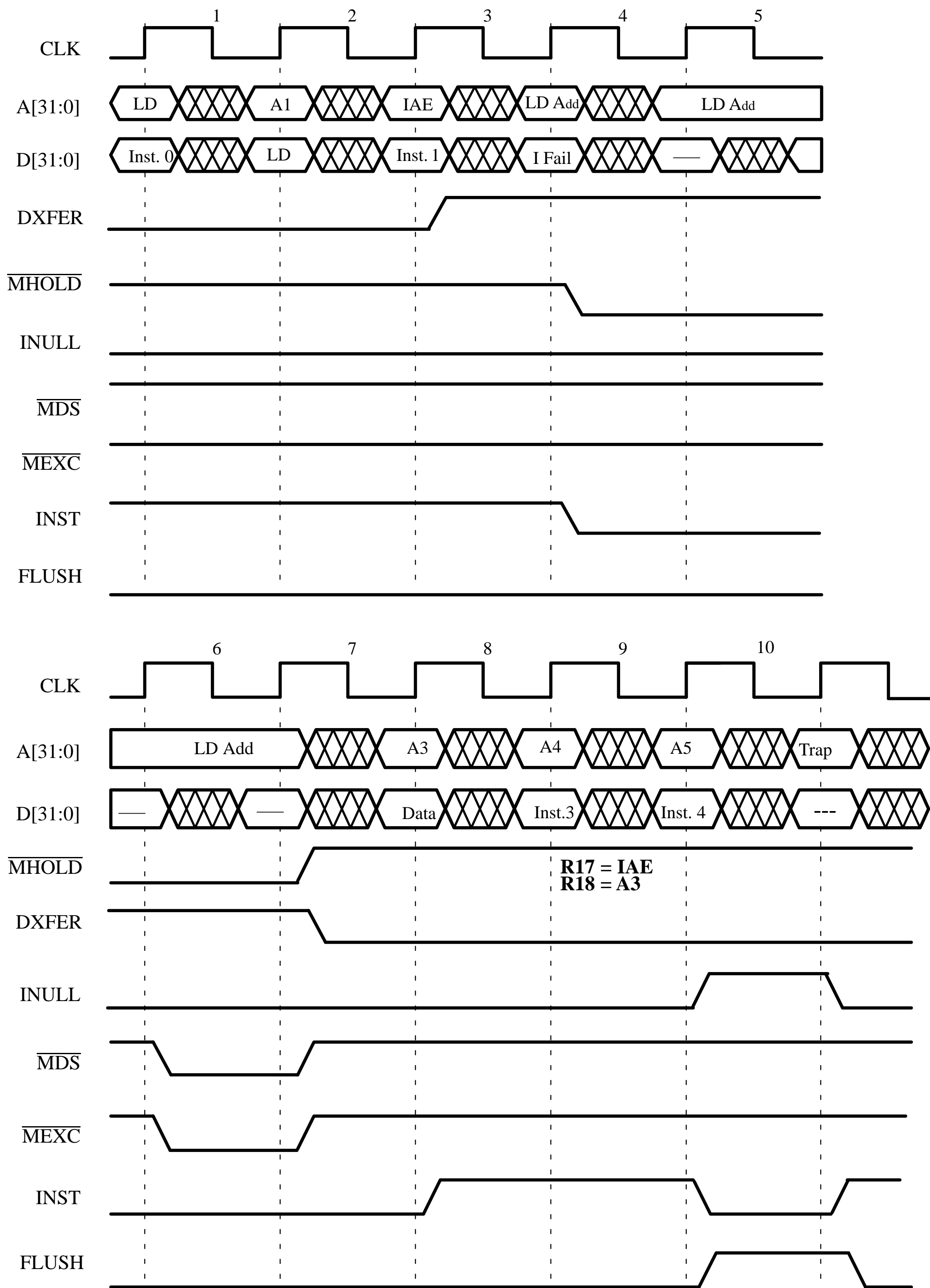


Figure 51. Instruction Memory Access Exception Timing (LD in Execute stage)

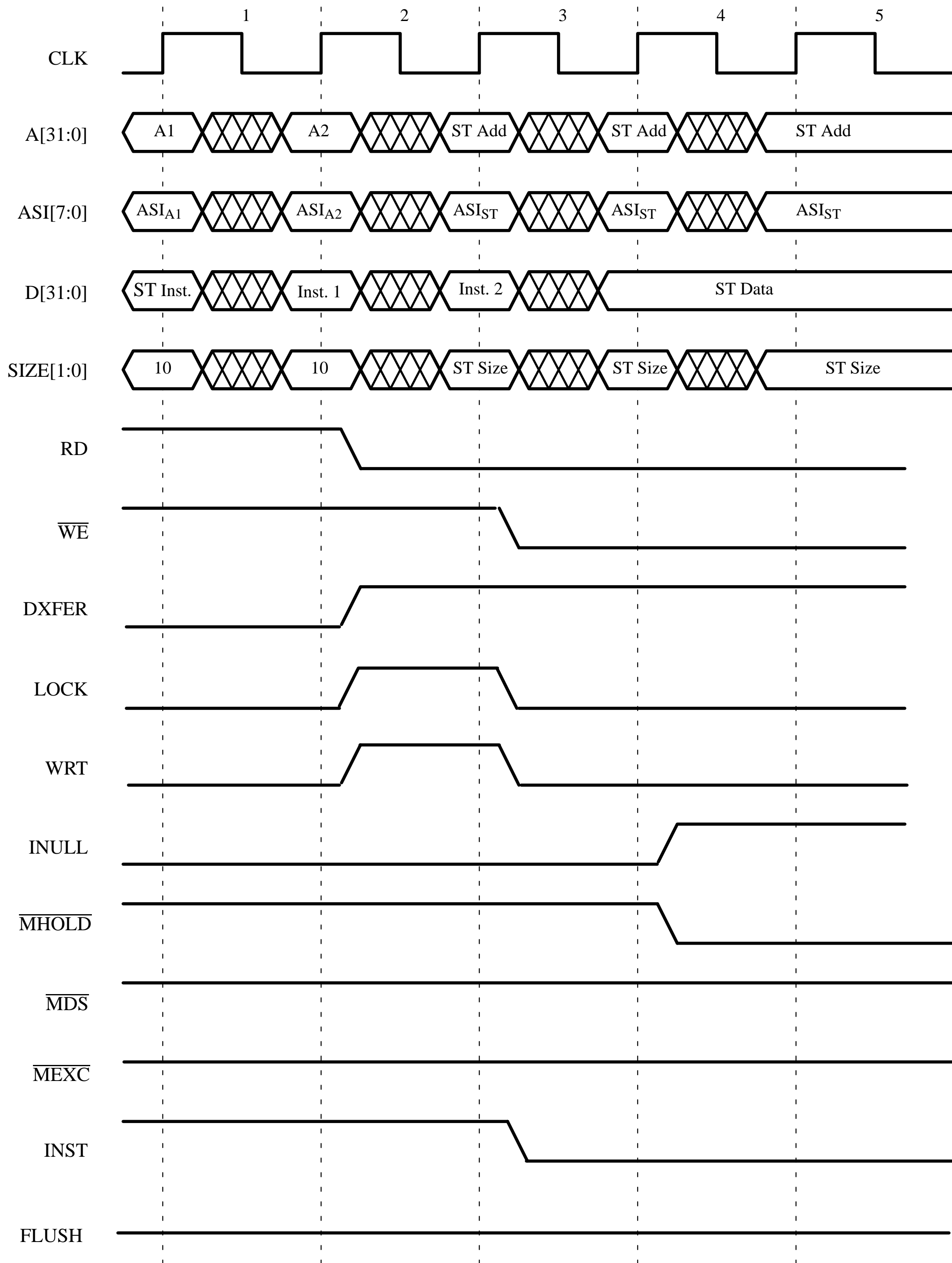


Figure 52. Store with Memory Exception Timing (1 of 2)

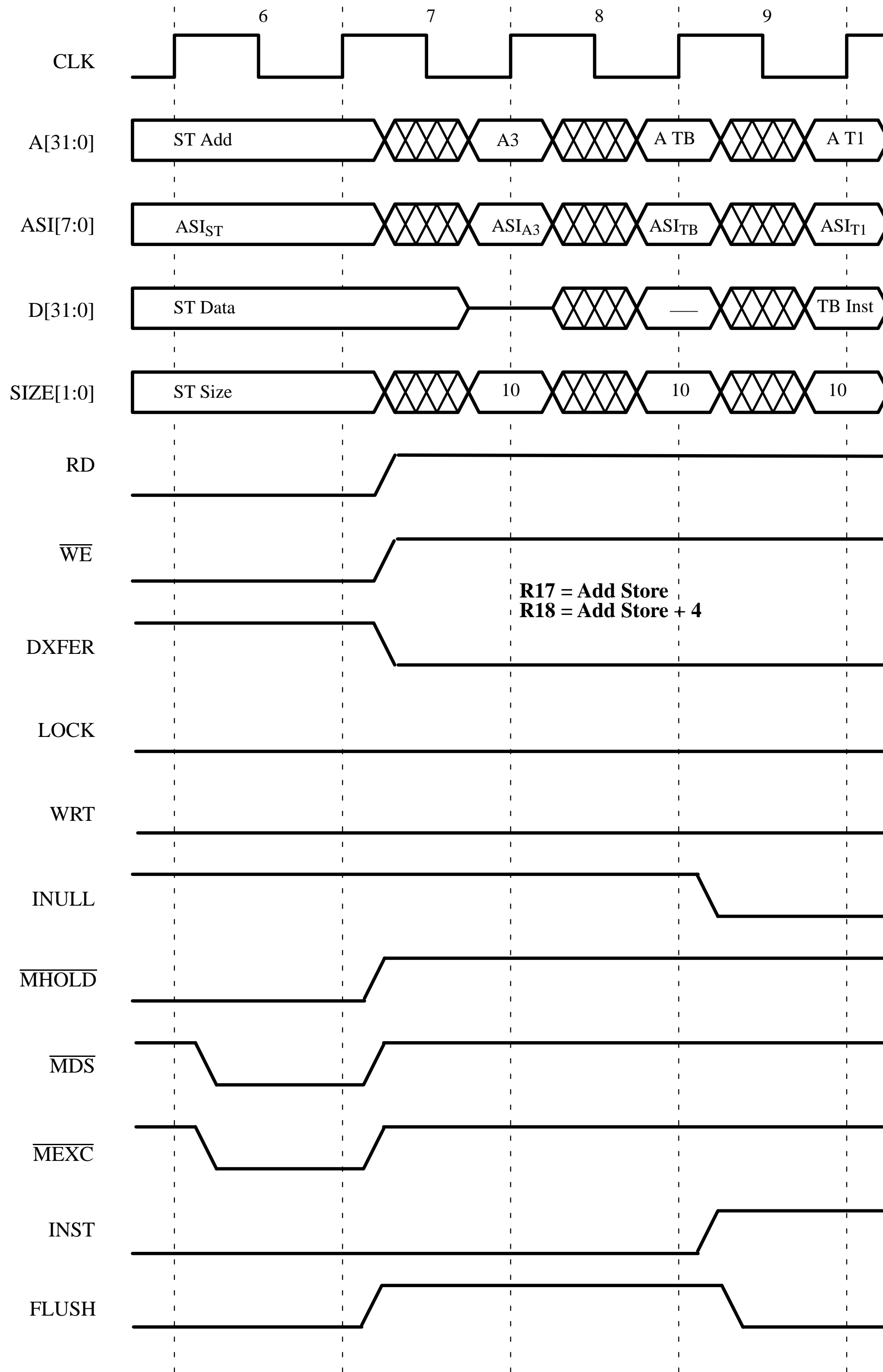


Figure 53. Store with Memory Exception Timing (2 of 2)

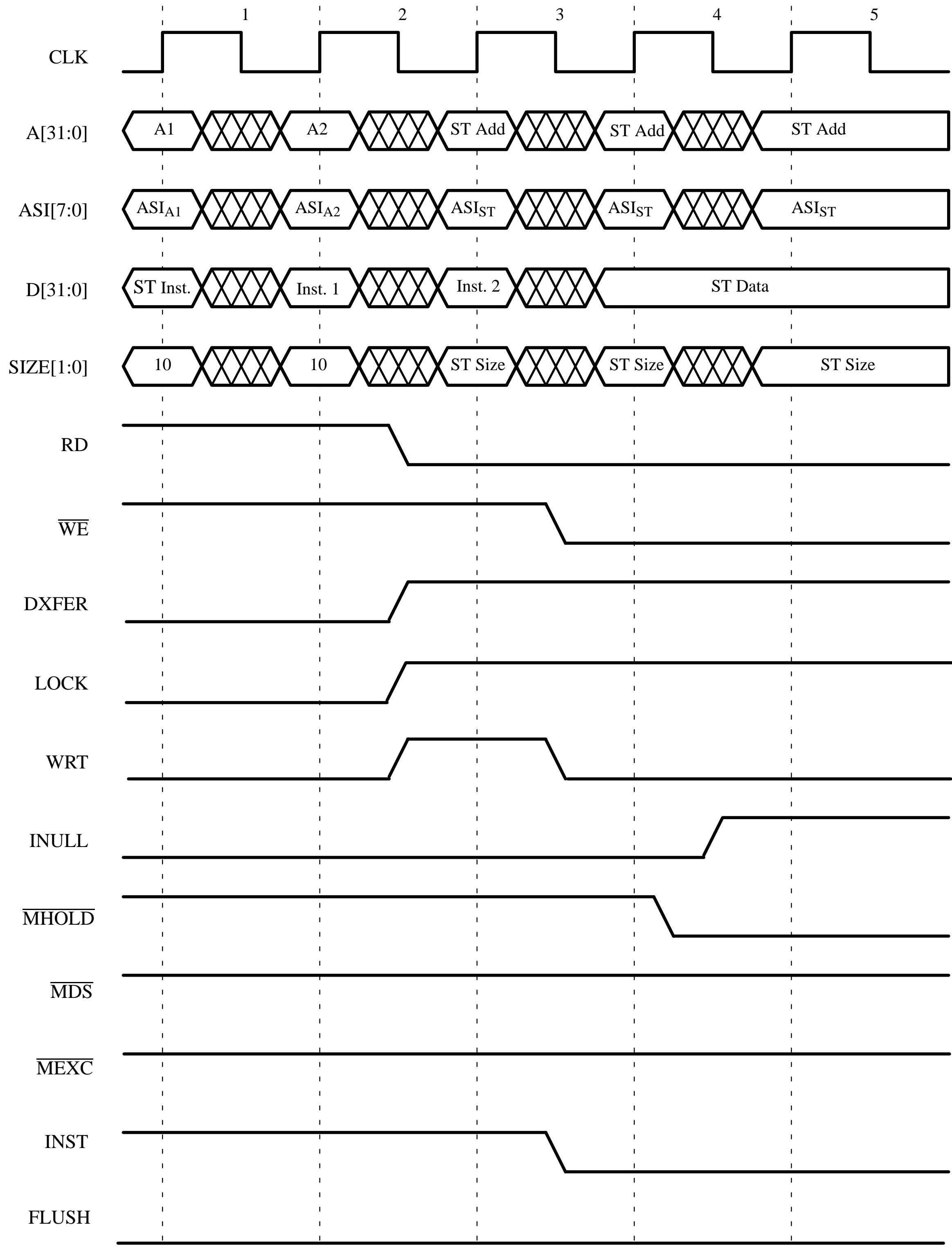


Figure 54. Store double with Memory Exception on 1st data address (1 of 2)

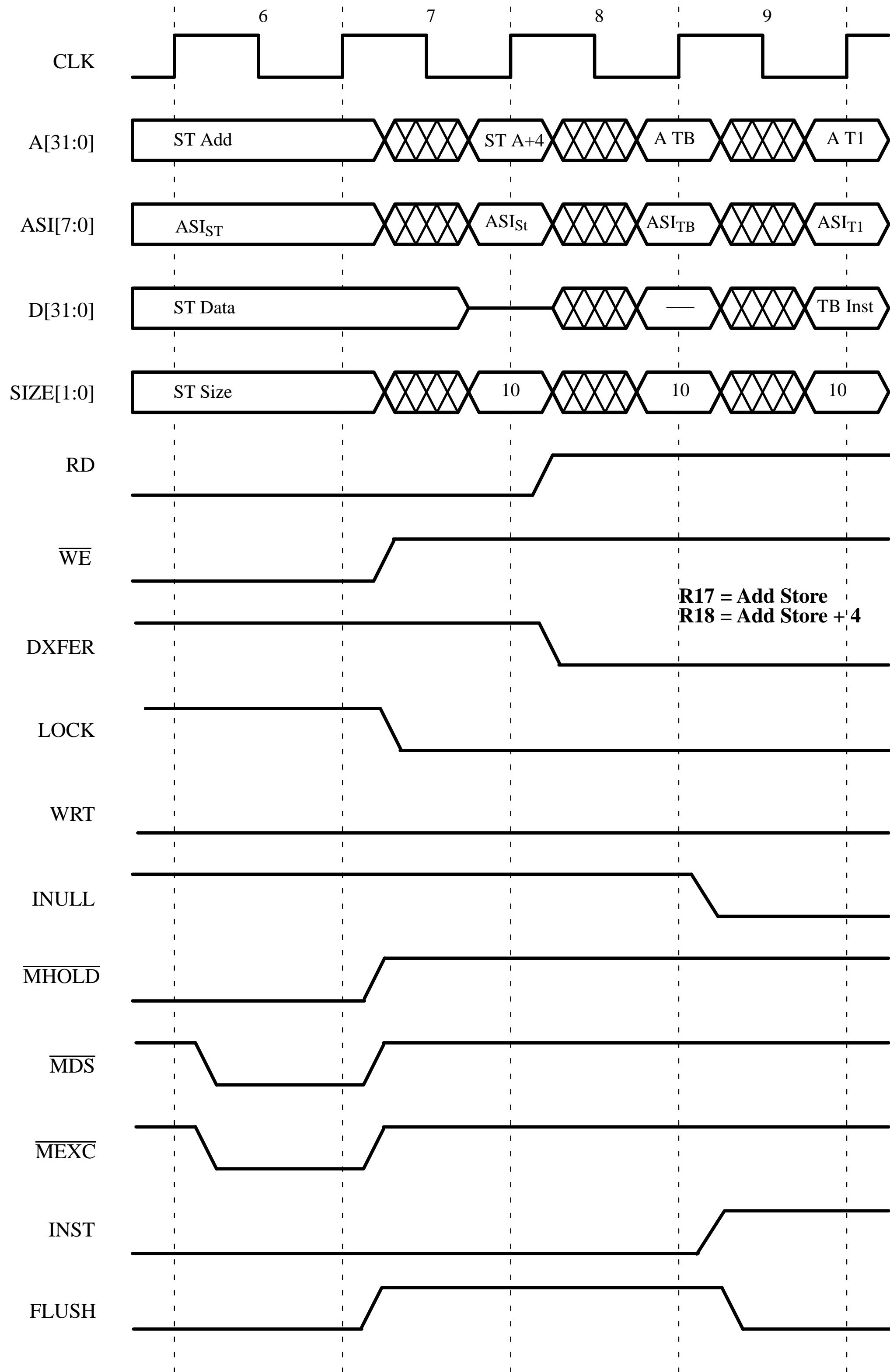


Figure 55. Store double with Memory Exception on 1st data address (2 of 2)

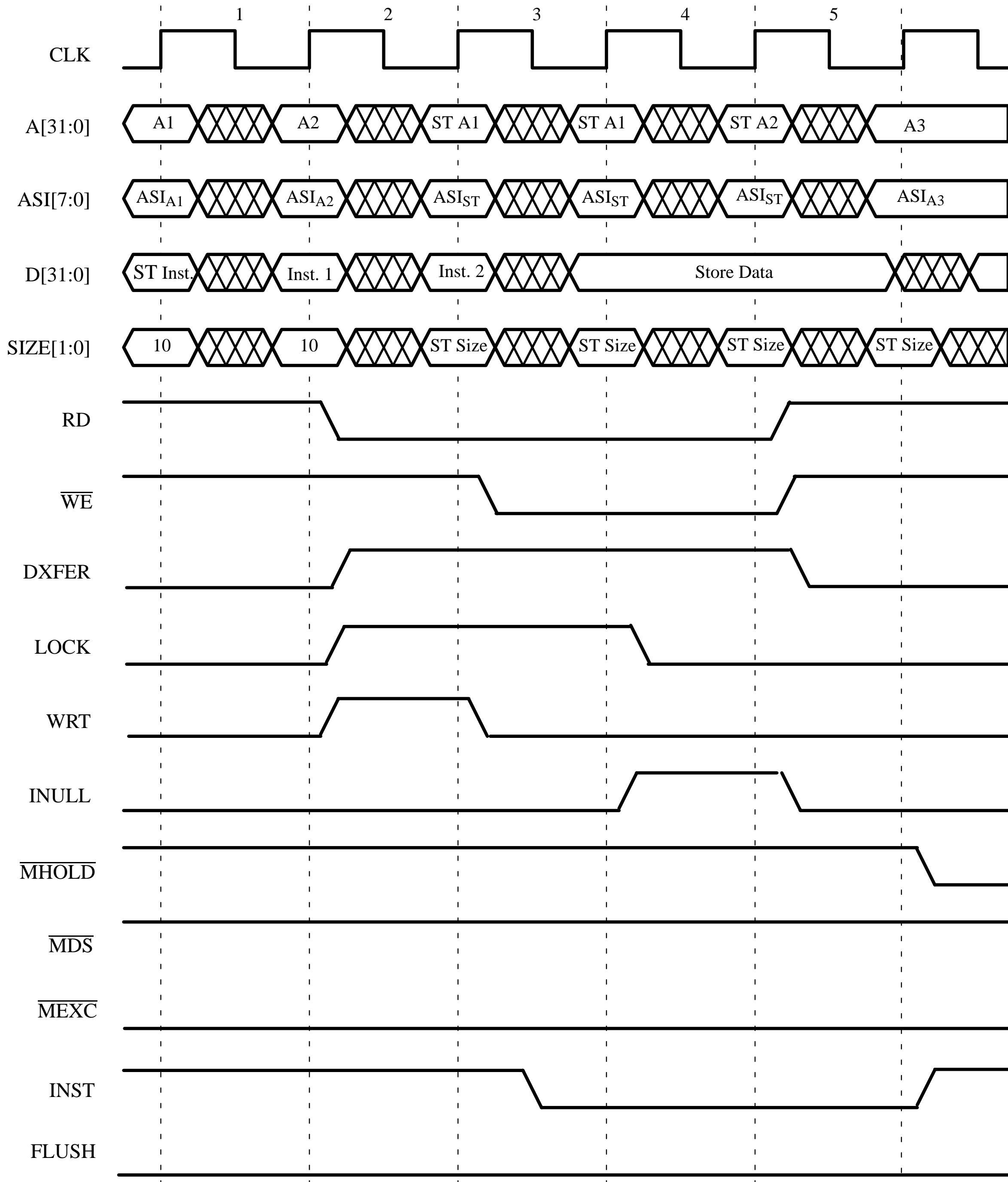


Figure 56. Store double with Memory Exception on 2nd data address (1 of 2)

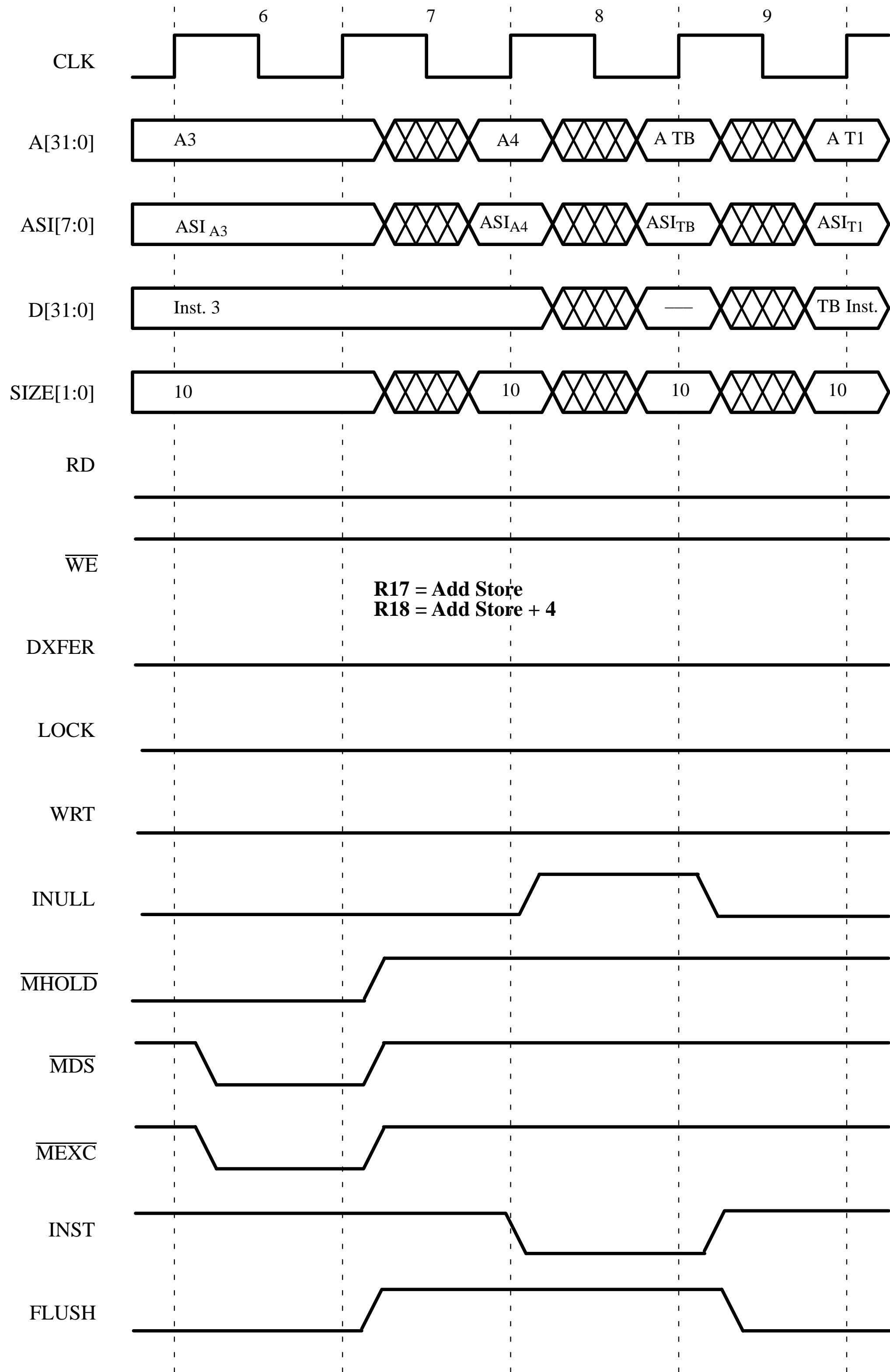


Figure 57. Store double with Memory Exception on 2nd data address (2 of 2)

3.7.14. Floating-Point Exceptions

The floating–point unit asserts $\overline{FEXC}$ to notify the **TSC691E** 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 58). The **TSC691E** 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.

$\overline{FEXC}$ is deasserted by the FPU after FXACK is asserted. FXACK is deasserted by the **TSC691E** after $\overline{FEXC}$ is deasserted.

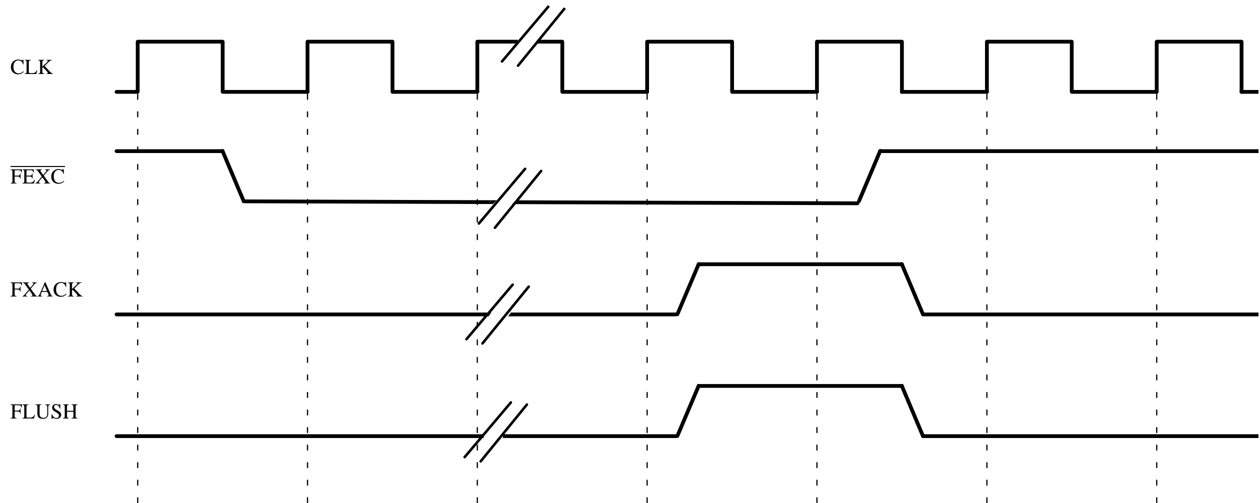


Figure 58. Floating–Point Exception Handshake Timing

3.7.15. 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 **TSC691E** will take the interrupt. The IRL input level should be held stable until the processor asserts INTACK.When the trap is taken, IRL line are ignored until ET=0 (until RETT instruction is executed). Figure 59 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 3.8.3 for more information on interrupts.

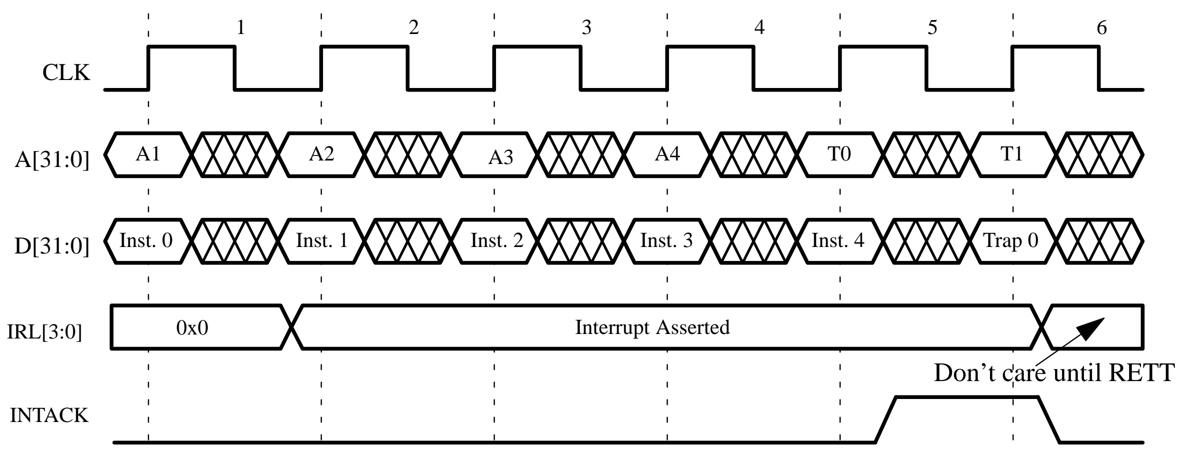


Figure 59. Asynchronous Interrupt Timing

3.7.16. Reset Condition

Figure 60 shows the timing for a power-on reset. $\overline{RESET}$ must be asserted for at least nine cycles so that the processor can synchronize the reset input and initialize its internal state. For $\overline{RESET}$ 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).

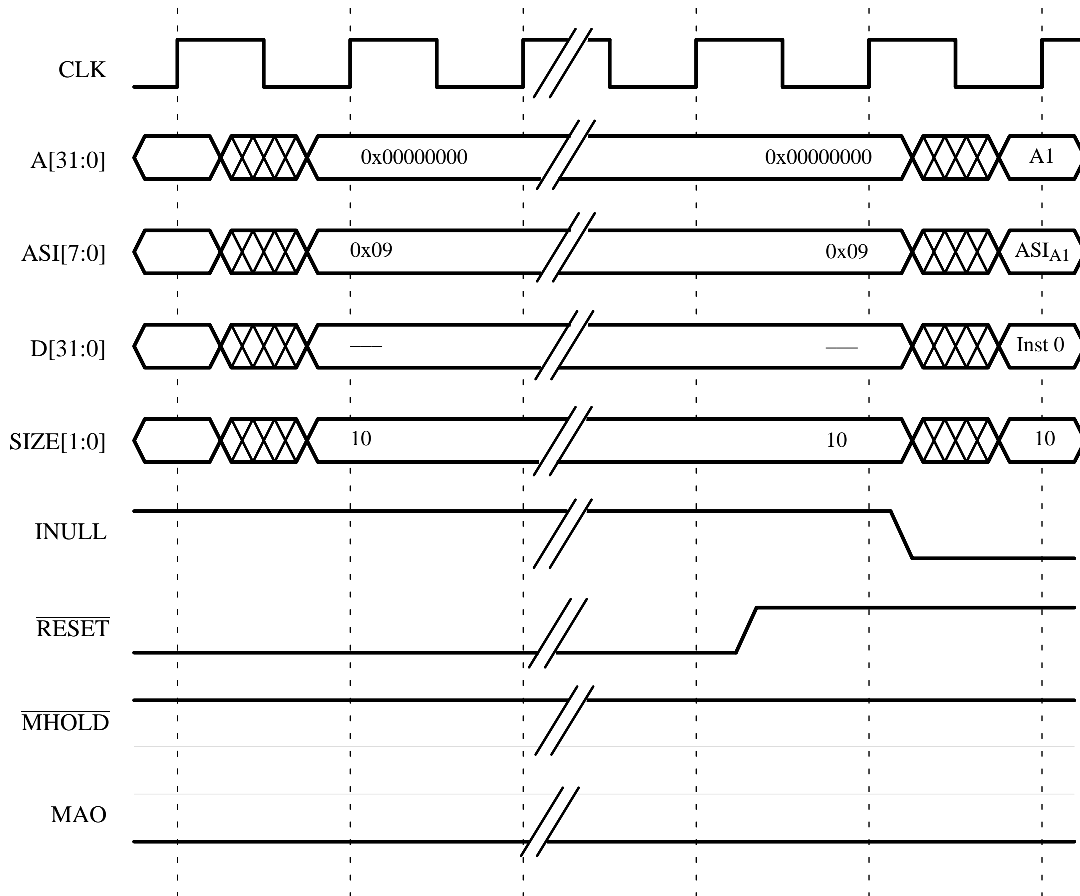
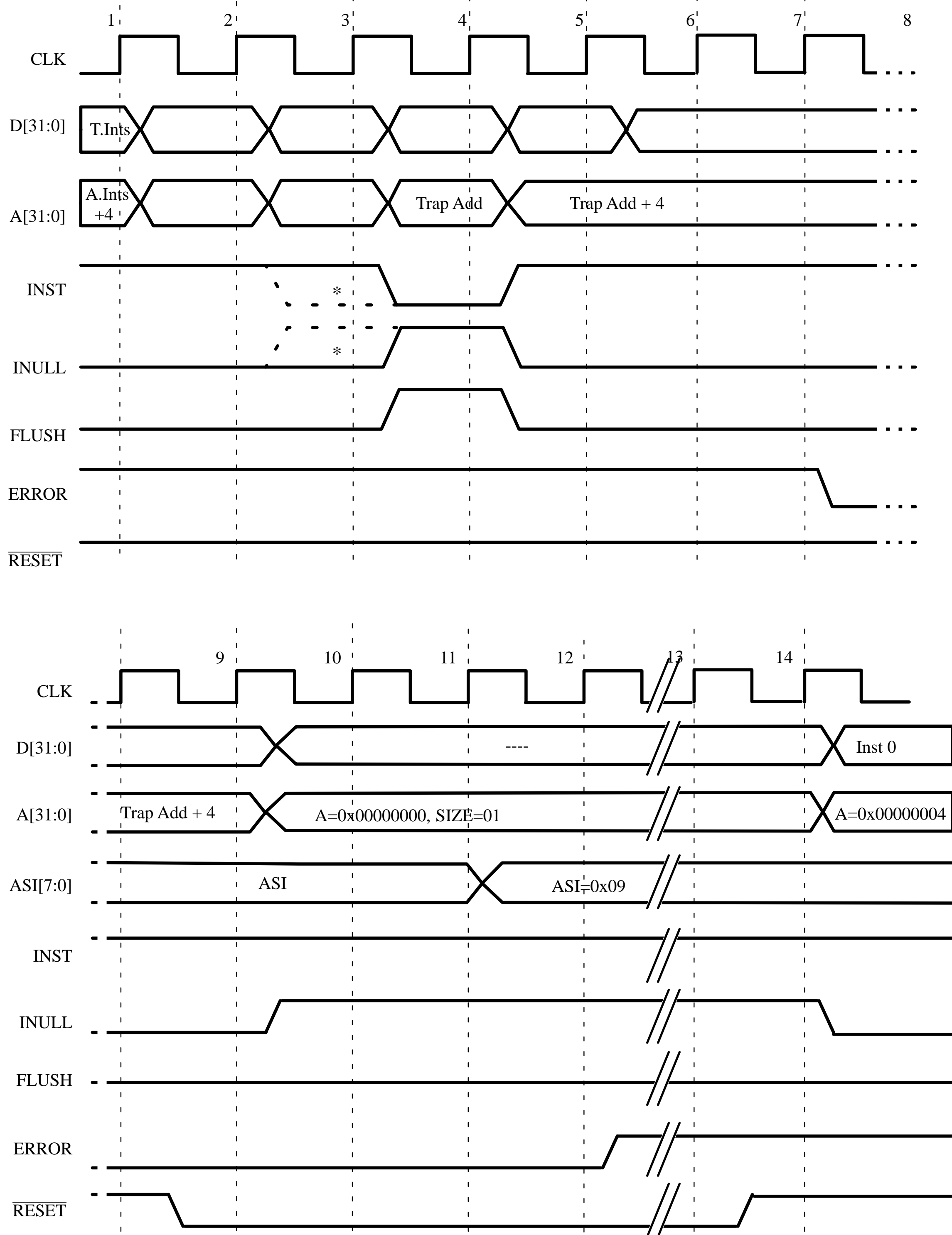


Figure 60. Power–On Reset Timing

3.7.17. Error Condition

Error mode is one of the three states in which the **TSC691E** can exist. To get into this 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 3.8), including setting the *tt* bits to identify the trap type. It then enters error mode, halts, and asserts the $\overline{ERROR}$ signal (see Figure 61).

The only way to leave error mode is to receive an external $\overline{RESET}$ signal, which forces the processor into reset mode. All information placed in the **TSC691E**'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.



T.Inst = Traping Instruction

* If T.Inst is a control transfert instruction

RESET must be asserted for a minimum of 9 clocks

Figure 61. Error/Reset Timing

3.8. Exception Model

The **TSC691E** 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.

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 3.8.3.

3.8.1. 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 3.8.3).

As soon as the **TSC691E** 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 3.8.5) 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.

3.8.2. Synchronous Traps

Synchronous traps are caused by the actions of an instruction, with the trap stimulus occurring either internally to the **TSC691E** 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.

A new type of trap has been added: Hardware traps. This trap occurs when a hardware error (i.e. SEU[1] on register) is detected by the IU. The trap type depends of the internal parity error (see Table 40). In case of hardware traps the $\overline{\text{HWERROR}}$ signal is asserted low.

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

Table 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

Note [1]:
SEU = Single Event Upset, a flip of register or memory cells, forced by heavy ions.

3.8.2.1. 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.

3.8.2.1.1. Hardware error

When a hardware error is detected, the trap handling routine saves the error information which the MEC has sampled.

The trap routine then resumes the instruction by returning from the trap routine. If the cause of the error was a transient fault, it may be removed by just resuming the instruction. If the error was caused by a fault that is not removable by resuming the instruction, another hardware error trap is generated and the trap handling routine propagates the error to a higher level of the application.

If the fault is in a critical register or latch which the trap handling routine uses, another hardware error trap is generated. A synchronous trap during the time when traps are disabled is a critical error and the IU enters the error mode and halts. This means that the error detection mechanism has to detect the error when the faulty instruction is in the execute stage in order to handle the trap normally, i.e. correct PC for the faulty instruction.

When an error trap occurs, the $\overline{\text{HWERROR}}$ signal is asserted (see Table 40).

3.8.2.1.2. Instruction access exeption

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

3.8.2.1.3. 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.

3.8.2.2. 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.

3.8.2.2.1. 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.
Floating-point instructions are coded with : op=10 & op3=11010x and coprocessor instructions : op=10 & op3=11011x. The IU decodes the fields op and op3 and generates FINS's or CINS's even if the instruction is unimplemented.

3.8.2.2.2. Privileged instruction

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

3.8.2.2.3. 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).

3.8.2.2.4. 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).

3.8.2.2.5. 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.

3.8.2.2.6. 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 3.8.1 on reset traps and SPARC V7.0 Instruction Set for the instruction definition for RETT.

3.8.2.2.7. 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 nonzero).

3.8.2.2.8. 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 3.4.3.2.3 for details.

3.8.2.2.9. 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 SPARC V7.0 Instruction Set, Ticc instruction).

3.8.3. 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 62). 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.

3.8.3.1. 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 38 .

3.8.3.2. Response Time

The **TSC691E** 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 62). In this case, the first instruction of the interrupt service routine is fetched during the fifth 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 65). 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.

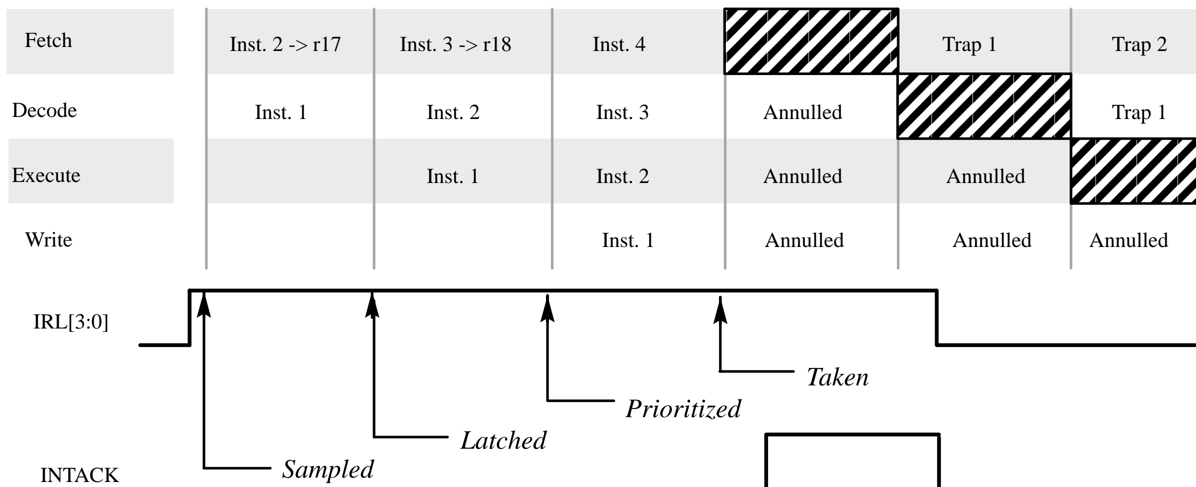


Figure 62. Best–Case Interrupt Response Timing (one cycle instruction)

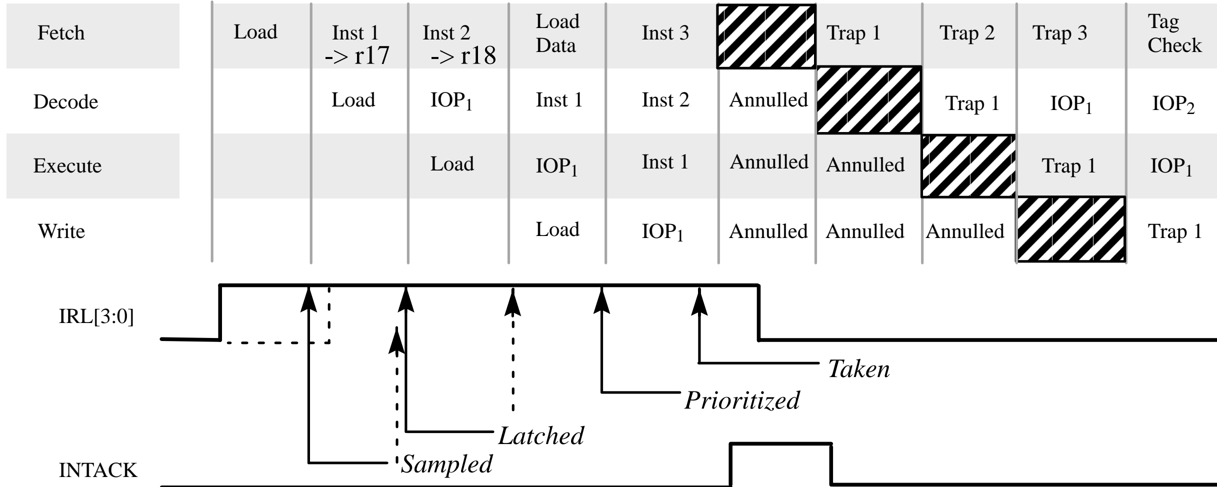


Figure 63. Double Cycles Instruction Interrupt Response Timing (ex: Load)

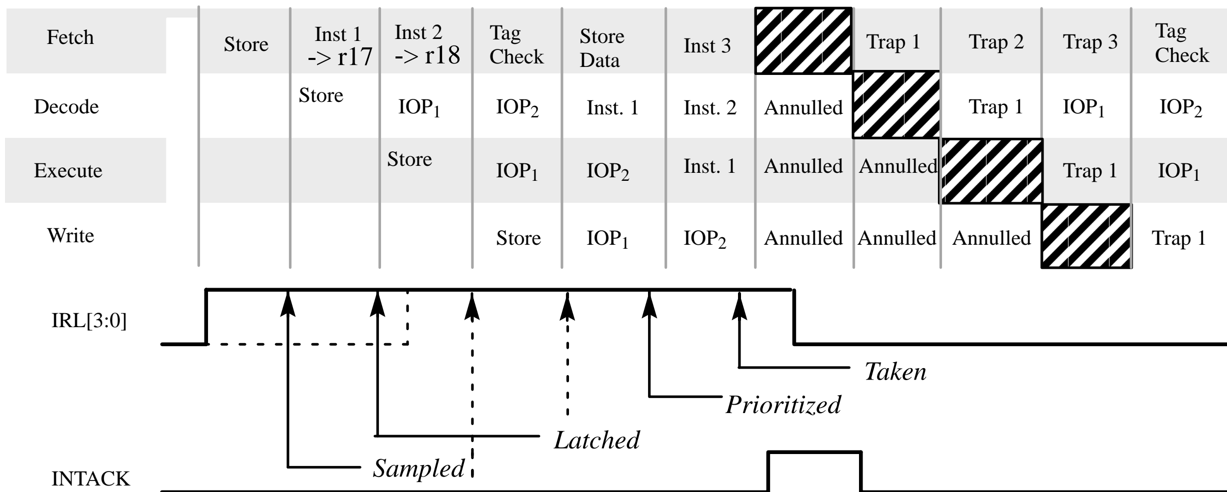


Figure 64. Triple-Cycles Instruction Interrupt Response Timing (ex: Store)

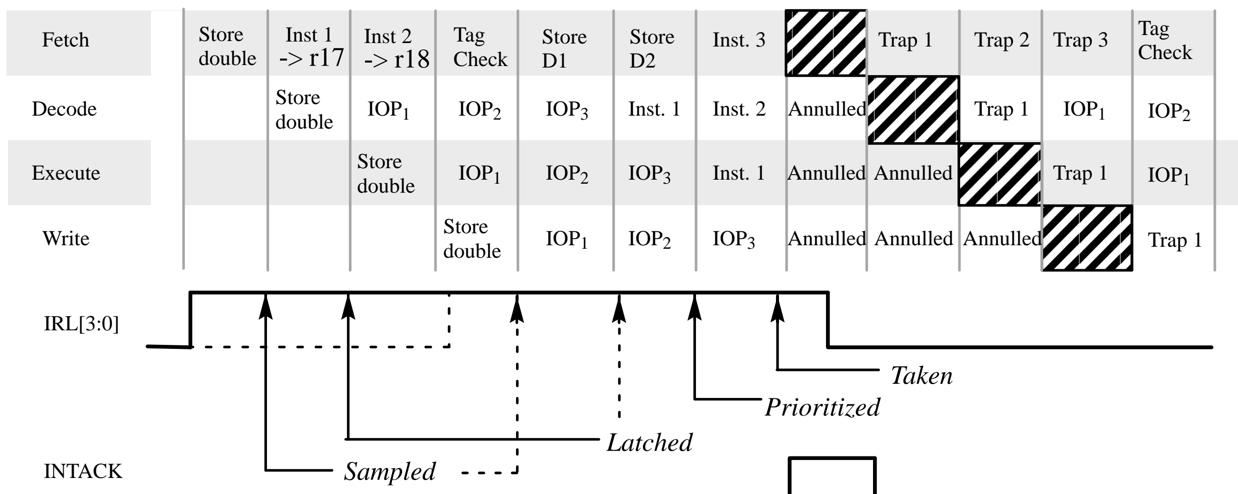


Figure 65. Four-Cycles Instruction Interrupt Response Timing (Store Double)

When the instruction present in the decode stage during sampling of IRL[3:0] is a CBI, the response time is the same than described in Figure 62 except when the delay instruction is annulled:

- BA, FBA, and CBA with annul bit = 1 (B*A,a)
- Bicc, FBicc, and CBicc not taken with annul bit = 1 (B*cc,aNT)

For those two cases, the INTACK signal and the first instruction of the interrupt service routine will be valid one cycle later (see Figure 66).

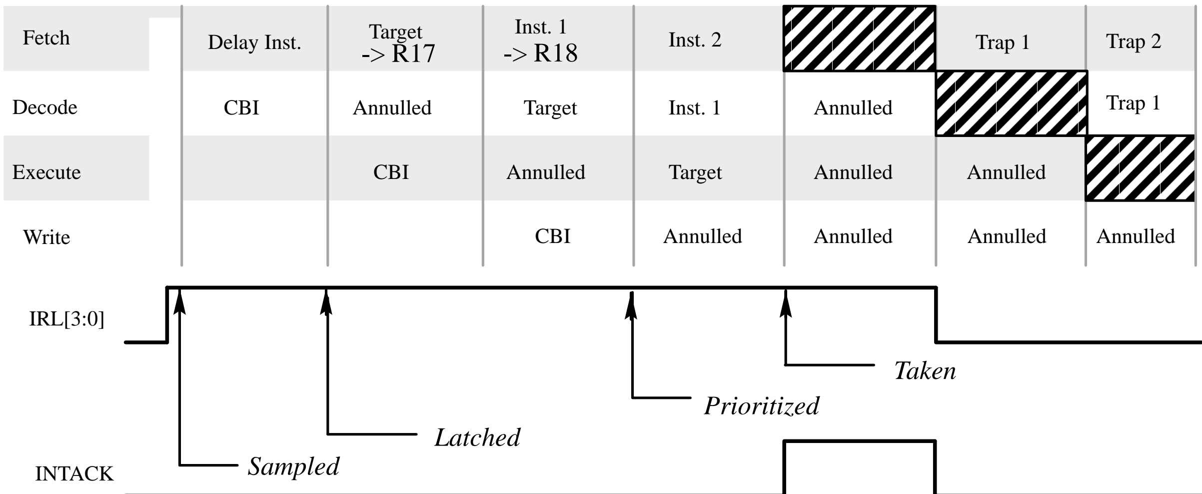


Figure 66. Interrupt Response Timing on conditional branch instruction (B*A,a & B*cc,aNT)

3.8.3.3.Interrupt Acknowledge

As shown in Figures 62 to 66, 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.

When trap is taken the PC and nPC are saved into r[17] and r[18] respectively (see Figures 63 to 66). Care must be taken in case of Response Timing on conditional branch instruction (B*A,a & B*cc,aNT), the PC value of inst1 instead of the Delay Instruction is saved in r[17]. If Branch is taken, r[17] and r[18] contain the 2 first addresses of the branch routine.

For the Best–Case Interrupt Response Timing (see Figure 66), r[18] contains the value of the first address of the branch routine if inst1 if a Branch instruction (different than B*A,a & B*cc,aNT).

3.8.4. 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 **TSC691E** 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 **TSC691E** 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.

3.8.4.1. 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).

3.8.4.2. 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.

3.8.5. 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.

3.8.5.1. 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 (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 **TSC691E** 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 $\overline{\text{FEXC}}$ or $\overline{\text{CEXC}}$ 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.

3.8.5.2. 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.

3.8.5.3. Trap Types and Priority

Each type of trap is assigned a priority (see Table 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 38. Trap Type and Priority Assignments

Trap	Priority	Trap Type (tt)	Synchronous or Asynchronous
Reset	1	-	Async.
Hardware error	2[1]	97-102	Sync.
Instruction Access	3[1]	1	Sync.
Illegal Instruction	4[1]	2	Sync.
Privileged Instruction	5[1]	3	Sync.
Floating–Point Disabled	6[1]	4	Sync.
Coprocessor Disabled	6[1]	36	Sync.
Window Overflow	7[1]	5	Sync.
Window Underflow	7	6	Sync.
Memory Address not Aligned	8	7	Sync.
Floating–Point Exception	9	8	Sync.
Coprocessor Exception	9	40	Sync.
Data Access Exception	10	9	Sync.
Tag Overflow	11	10	Sync.
Trap Instructions (Ticc)	12	128 - 255	Sync.
Interrupt Level 15	13	31	Async.
Interrupt Level 14	14	30	Async.
– – – – – –	– – – – – –	– – – – – –	– – – – – –
Interrupt Level 2	26	18	Async.
Interrupt Level 1	27	17	Async.

__Note__ 1:
The priority of those traps have changed in relation to the CY7C601.

3.8.5.4. 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.

3.9. 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 **TSC691E**, 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 MHS and/or 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.

In order for the **TSC691E** 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 as an example of a coprocessor interface application.

3.9.1. 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 busses are shared. The initial **TSC691E** instruction decode determines which unit should execute the instruction. The **TSC691E** 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 **TSC691E** 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 **TSC691E** by freezing the instruction pipeline with the $\overline{\text{CHOLD}}$ signal.

The signal interface between the **TSC691E** 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 **TSC691E**.

3.9.1.1. Coprocessor Interface Signals

The SPARC architecture defines two sets of signals intended for interfacing with two coprocessors. The **TSC691E** 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 **TSC691E** instruction fetch. The two groups of signals are symmetric, have identical timing requirements, and are listed in Table 33 .

Instruction fetch is signaled by the **TSC691E** 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 **TSC691E**, regardless of instruction type. The **TSC691E** 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 **TSC691E** 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 **TSC691E** 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 **TSC691E**. This signal is asserted in cases where the **TSC691E** 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 **TSC691E** will continue normal execution until the execution stage of the next coprocessor instruction. At that time, the **TSC691E** will acknowledge the interrupt with CXACK, and begin coprocessor trap execution.

Coprocessor branch on condition code (CBcc) instructions are executed by the **TSC691E** 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 **TSC691E** to halt execution. This interlock prevents the **TSC691E** 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.

3.9.2. Register Model

The coprocessor register model specified by the SPARC architecture is shown in Figure 67 . 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 **TSC691E**, using coprocessor load/store instructions.

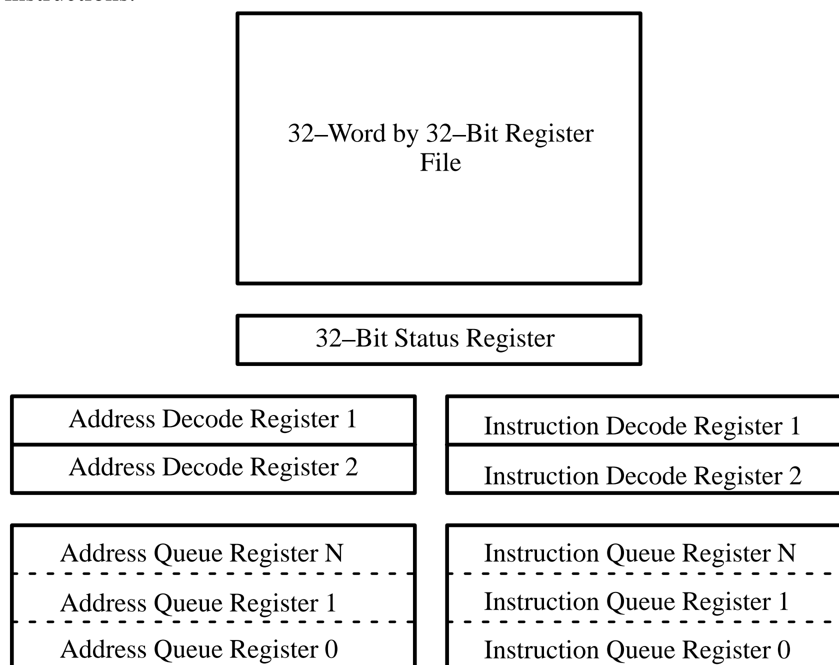


Figure 67. Coprocessor Register Model

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 **TSC691E** determines if they belong to the coprocessor. If one of the held instructions belongs to the coprocessor, the **TSC691E** 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 **TSC691E** 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.

3.9.3. 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.

4. Fault Tolerant and Test Mechanism

Fault Tolerant Mechanism:

- Parity checking on 98.7% of the total number of latches with hardware error traps
- Parity checking of address, data pads and control pads
- Program Flow Control
- Master/Checker operation
- Interleaving of the register file bits to reduce the risk for one impact to provoke dual (multiple) bit upsets
- Manufactured using Space Hardened 0.8 µm SCMOS RT TECHNOLOGY

Test Mechanism:

- IEEE Standard Test Access Port & Boundary-Scan Architecture
- Internal Scan Path to test the internal parity error detection during off-line test
- Possibility to halt the IU by an external signal

Figure 68. Fault Tolerant and Test Mechanism Block Diagram

4.1. Fault Tolerant and Test Support signals

Some signals have been added for fault tolerant and test mechanism improvement. These new signals can be classified as follows:

4.1.1. Address Parity Generation:

- APAR—Address Bus Parity (output)
- ASPAR—ASI and SIZE Parity (output)

4.1.2. Data Parity Generation/Checking:

- DPAR—Data Bus Parity (bidirectional)

4.1.3. MEC control signal Parity Generation:

- IMPAR—IU to MEC Control Parity (output)

4.1.4. FPU control signal Parity Generation/Checking:

- IFPAR—IU to FPU Control Parity (output)
- FIPAR—FPU to IU Control Parity (input)

4.1.5. Parity Checking Error Output:

- $\overline{\text{HWERROR}}$—Hardware Error Occurs (outputs)

  Odd parity definition: The number of one in a word, including the parity bit, is alway odd.
  (e.g. 00000000 --> P=1, 00000001 --> P=0)

4.1.6. Master/Checker Mode:

- $\overline{\text{CMODE}}$—checker Mode (input)
- $\overline{\text{MCERR}}$—Comparison Error (output)

4.1.7. Test Access Port:

- TCLK—Test Clock (input)
- $\overline{\text{TRST}}$—TEST Reset (input)
- TMS—Test Mode Select (input)
- TDI—Test Data Input (input)
- TDO—Test Data Output

4.1.8. Miscellaneous:

- $\overline{\text{601MODE}}$—Normal 601Mode Operation (input)
- $\overline{\text{HALT}}$—Halt (input)
- $\overline{\text{FLOW}}$—enable or disable Program Flow Control

A more detailed description of these signals is provided in Chapter 3.5

4.2. Program Flow Control

4.2.1. Introduction

A very high proportion of transient faults can cause errors in the program flow (75% in a traditional microprocessor). This type of error is detected by the TSC691E using Embedded Signature-Monitoring (ESM) techniques.

A program using ESM is partitioned in branch free basic blocks and branch instructions. For each executed instruction, the IU calculates a checksum of 32 bits of the operation code during the execution. The checksum result consist of the logic XOR of the instruction words with the previous checksum. The 16 MSB's are XORed with the 16 LSB's to provide a signature word.

This 16–bit signature is compared with the correct value, precomputed by the compiler, whenever a SETHI instruction (SETHI g0,%PRE_CHECKSUM) is executed. After the comparison, the checksum is reseted to zero.
The 6 MSB's in the immediate value of the SETHI instruction must be set to "011111".
In case of a comparison error, a hardware trap is taken with Trap Type=0x66 and $\overline{\text{HWERROR}}$ asserted.
There are three cases when the subsequent check is disabled:

1. When a trap is taken.
2. When executing a RETT instruction.
3. When executing a SETHI instruction to R[0] with the immediate value set to zero. This SETHI instruction does not perform a comparison but zero the checksum. It is reserved as a NOP instruction.

For these cases the subsequent check is disabled, and will not signal an error, but will enable the checking again with checksum equal to zero.

The Program Flow Control is enabled by the $\overline{\text{FLOW}}$ signal input. After reset the Program Flow Control is enabled (if $\overline{\text{FLOW}}$ signal is low), and the checksum is reseted to zero.

4.2.2. Example of Program Flow Control

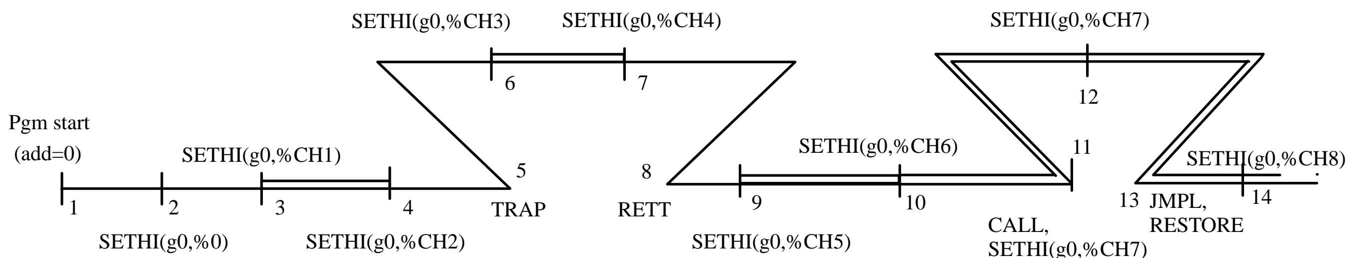


Figure 69. Example of Program Flow Control

1- Program starts at address=0x0 with Program Flow Control enabled and Checksum=0.

2- No comparison performed, next checking disabled and Checksum is reseted to zero. (NOP)

3- No comparison performed, next checking enabled and Checksum is reseted to zero.

4- Comparison performed, next checking enabled and Checksum is reseted to zeroomparison.

5- TRAP instruction disables the next checking.

6- No comparison performed, next checking enabled and Checksum is reseted to zero.

7- Comparison performed, next checking enabled and Checksum is reseted to zero.

8- RETT instruction disables the next checking.

9- No comparison performed, next checking enabled and Checksum is reseted to zero.

10- Comparison performed, next checking enabled and Checksum is reseted to zero.

11- When a CALL instruction is encountered, the delay instruction must be a SETHI instruction to perform a comparison, enable the next checking and reset the Checksum.

12- Comparison performed, next checking enabled and Checksum is reseted to zero.

13- When the JMPL instruction is encountered the Program Flow Control is not disabled and next checking is enable since the delay instruction is a RESTORE.

14- Comparison performed (checksum is calculated from the last SETHI encountered in the subroutine).

Conclusion: In this example, the sequences 3 to 4, 6 to 7 and 9 to 14 are checked.

Programming Note:

1- When returning from a CALL routine (13), the delay instruction is a RESTORE so, when encountering the next SETHI(g0,%CHS) (14), the comparison is performed with a checksum calculated from the last SETHI(g0,%CHS) of the subroutine (12).

2- All the delay instruction (instruction after a control transfer instruction: Bicc, FBfcc, CBccc, CALL, JMPL, Ticc or RETT) are added in the checksum even this instruction is annuled.

4.3. Parity Checking

4.3.1. Introduction

In the **TSC691E**, 98.7% of all registers are protected by a odd parity bit (100% of the register file is protected). The checking of registers and busses is be performed only if the registers or the busses are used by the current instruction. With this approach, unused registers/busses will not cause an error and decreasing the uptime of the system will be limited.

Address bus, Size and ASI busses, Data bus, Control signals of the MEC and of the FPU are also protected with parity bits.Control signals for coprocessor are not protected by parity bit. The parity checking is disabled during reset. Care has to be taken not to read a register before it has been written and its parity bits initialized.

When an error occurs, the $\overline{\text{HWERROR}}$ signal is asserted low and a trap is taken depending of the parity error type (see Table 39).

Definition of odd parity bit: The number of one in a word, including the parity bit, is alway odd.
(e.g. 00000000 --> P=1, 00000001 --> P=0)

4.3.2. Trap handling

When a hardware error is detected the $\overline{\text{HWERROR}}$ signal is asserted then a trap routine is taken depending of the error type (see Table 39). The $\overline{\text{HWERROR}}$ signal is asserted until the error trap is taken.This software routine reviews the failing instruction. If the cause of the error was a transient fault, it may be removed by just resuming this instruction. In this case, $\overline{\text{HWERROR}}$ is deasserted (see 5.2.2.2).

If the error was caused by a non removable error, another hardware error trap is generated. Because a synchronous trap is taken during a time when traps are disabled, the IU enters the error mode, asserts $\overline{\text{ERROR}}$ signal and halts ($\overline{\text{HWERROR}}$ will stay asserted until removed by reset).

This means that the error detection mechanism will detect the error when the failing instruction is in the execute stage in order to handle the trap normally, i.e. correct PC for the failing instruction.

The trap are grouped into the following Error-Type:

1- Restartable, precise error: Errors that can be removed by retrying the instruction and with correctly saved PC and nPC. These errors can be removed by simply returning from the trap routine.

2- Non-restartable, precise error: Errors that will remain even after an instruction retry, but with correctly saved PC and nPC. These errors are not removable and the trap routine should not attempt a retry. Since the address of failing instruction is know, the kernel can attempt a local clean-up, i.e. not having to restart the application.

3- Restartable, late error: Errors that can be removed by retrying the instruction but with PC and nPC pointing to the following instruction (was data load error). The trap routine can emulate the failing instruction or retry after the PC and nPC have been adjusted.

4- Non-restartable, imprecise error: Error that can not be associated with a particular instruction and cannot be removed by instruction retry. These errors are typically quite severe and will require a re-boot.

5- Register file error: Error that occurred in the register file (special case of Non-restartable, precise error)

6- Program flow error: Error detected by the program flow control.

Table 39. Error Type Assignments

Trap Type	Error Type	Error Signal
0x61	Restartable, precise error	HWERROR
0x62	Non-restartable, precise error	HWERROR
0x63	Restartable, late error	HWERROR
0x64	Non-restartable, imprecise error	HWERROR
0x65	Register file error	HWERROR
0x66	Program flow error	Trap only
-	Master / Checker error	MCERR
-	Error mode	ERROR

4.3.3. Priority within hardware traps for IU

When multiple hardware traps occur, the highest priority trap is taken, and lower priority traps are ignored. The priority applied on the harware traps of the IU are define as follow:

Table 40. Hardware Priority

Trap Type	Error Type	Error Signal
0x61	Restartable, precise error	5
0x62	Non-restartable, precise error	2
0x63	Restartable, late error	4
0x64	Non-restartable, imprecise error	1
0x65	Register file error	3
0x66	Program flow error	6
-	IU synchronous traps	7

Remark: Priority 1 is for highest priority and 5 for the lowest priority. All other synchronous traps (caused by the actions of an instruction) has a lower priority.

4.3.4. Parity Checking on Register File and Control/Status Registers

The register file and the control/status registers of the TSC691E are protected by a parity bit. Hardware error on those registers shall lead to hardware error trap as defined in .

Table 41. Hardware error type for user registers

Location or Register	Error type	Trap type
Register File	Register file error	65H
“Fetch” Parity Check	Restartable, precise error	61H
“Decode” Parity Check	Non-restartable, precise error	62H
“Execute” Parity Check / “Write” Parity Check	Non-restartable, imprecise error	64H
PSR	Non-restartable, imprecise error	64H
WIM	Non-restartable, precise error	62H
TBR	Non-restartable, precise error	62H
Y	Restartable, precise error	61H

4.3.5. Parity Checking on Control Signal for the FPU

The control signals between the IU and the FPU are protected by a parity bit.

4.3.5.1. Output control signals

The control bus contains five bits: FINS1, FINS2, FLUSH, FXACK and INST. The parity output for these five signals is IFPAR (IU to FPU PARity). This parity bit is generated by the IU.

Note:
IFPAR is a three-state (on chip pull–up resistor=20kΩ) output controlled by $\overline{TOE}$ signal.

4.3.5.2. Input control signals

The input control signals are: FCC[1:0], FCCV, $\overline{FEXC}$, $\overline{FHOLD}$ and $\overline{FP}$. The parity input for these signals is FIPAR (FPU to IU PARity). This parity bit is generated by the FPU and checked by the IU when a FBfcc instruction is executed. FCCV and FHOLD are re-synchronized on the rising edge of the clock to check the parity.

4.3.6. Parity Checking on Control Pads for the TSC693E (MEC)

The 13 control signals between the IU and the MEC are protected by a parity bit.

4.3.6.1. Output control signals

The output control bus contains six bits: DXFER, LDSTO, LOCK, RD, $\overline{WE}$ and WRT. The parity output for these five signals is IMPAR (IU to MEC PARity). This parity bit is generated by the IU.

Note:
IMPAR is a three-state (on chip pull–up resistor=20kΩ) output controlled by $\overline{COE}$ or $\overline{TOE}$ signals.

4.3.6.2. Input control signals

No parity is performed on the input control signals: MAO, $\overline{MDS}$, $\overline{MEXC}$, $\overline{MHOLDA/B}$ and $\overline{BHOLD}$.

4.3.7. Parity Checking on Control Pads for the Coprocessor

No parity is performed on the input and output control signals.

4.3.8. Parity Generation on ADDRESS Bus

The 32–bit address bus contains a parity bit calculated by the IU and sent out on the APAR pad. The ASI[7:0] and SIZE[1:0] busses contain also a parity bit called ASPAR which is calculated by the IU.

Note:
APAR and ASPAR are a three-state (on chip pull–up resistor=20kΩ) output controlled by $\overline{AOE}$ or $\overline{TOE}$ signals.

4.3.9. Parity Checking on DATA Bus

The DPAR bidirectional signal contains the odd parity over the 32-bit data bus. When the IU receives a data (LOAD) or an instruction, the parity bit is checked by the IU. In case of a STORE data instruction, the parity bit is generated and launched in parallel by the IU.

To be able to use a standard FPU (i.e. TSC692E), parity on the data bus has to be generated internally and parity checking on the control bus must be turned off.

4.3.10. Non CY7C601 Mode

This feature is controlled by asserting the $\overline{\text{601MODE}}$ input signal. This signal is static and shall not change when running.

4.3.11. Error Type for external signals parity errors

Data inputs (Inst. and Load) and FPU to IU control signals receive a parity bit which is checked by the IU. If an error is detected, the IU takes a trap depending of the error type Table 50 .

Table 42. Hardware error type for external signals

Register	Error type	Trap type
Data (inst.)	Restartable, precise error	0x61
Data (load)	Restartable, late error	0x63
FIPAR	Floating–Point Disabled [1]	0x04

Note [1]: The parity is only checked when a FBfcc instruction is executed.

4.4. Master/checker operation

The MHS **TSC691E** includes comparator circuits at the outputs to support fault detection. Applications requiring a high level of reliability can use this Master/Checker operation to introduce fault detection on a system level. By duplication of units without the use of external comparators, 100% of the internal errors can be detected, especially those errors which are not detected by the internal unit concurrent error detection mechanism.

4.4.1. Basic function

By asserting the signal $\overline{\text{CMODE}}$ the **TSC691E** can be configured either as master or checker. This signal is static and shall not change when running. Assertion of this signal will set the IU to act as a checker to support master/checker operation. All output signals except $\overline{\text{ERROR}}$,$\overline{\text{HWERROR}}$, $\overline{\text{MCERR}}$ and TAP signals will be high-Z (on chip pull_up resistor=20kΩ). The master and at least one checker circuit are working in parallel and execute the same program. When the master is forcing the address and data bus, the checker is in a read and compare mode. This means the output buffers are disabled and the external busses are compared by the checker with its internal results. If a mismatch occurs on any output, then the $\overline{\text{MCERR}}$ signals are asserted until the mismatch disappears. In this case, the system hardware and/or software can take appropriate action.

If the master IU signals an internal error before a comparison error is indicated, it is possible to stop execution of the two IUs by asserting the $\overline{\text{HALT}}$ signal, disable the master IU, change the checker IU to master IU and continue execution. $\overline{\text{CMODE}}$ signal can be changed when $\overline{\text{RESET}}$ signal is asserted or when the IU is in halt mode.

On a master processor, the three-state control signals (e.g : $\overline{\text{AOE}}$, $\overline{\text{COE}}$, $\overline{\text{DOE}}$, $\overline{\text{TOE}}$) disable the checker mode of the three-stated buffers.

An external/internal mismatch can occur for two reasons:

1- In a system with only one master processor, a short or other electrical failure can force the output signal to a fixed voltage. For example, a bus signal can be shorted to ground. When the circuit drives a high voltage on the bus, the external signal will be pulled low and a mismatch will occur and he IU asserts the $\overline{\text{CMPERR}}$ signals.

2- An external/internal mismatch can occur in the master/checker mode. Figure 70 shows a basic master/checker configuration using two **TSC691E** devices.

Using the master/checker solution there is a possibility that the system can continue with only the correct remaining unit, or with both after the restoration of state of the faulty unit. If an internal error is indicated in the checker, it could be ignored. The MEC requires error signals from both the master and the checker. In case of corruption, the system behavior is defined by the MEC.

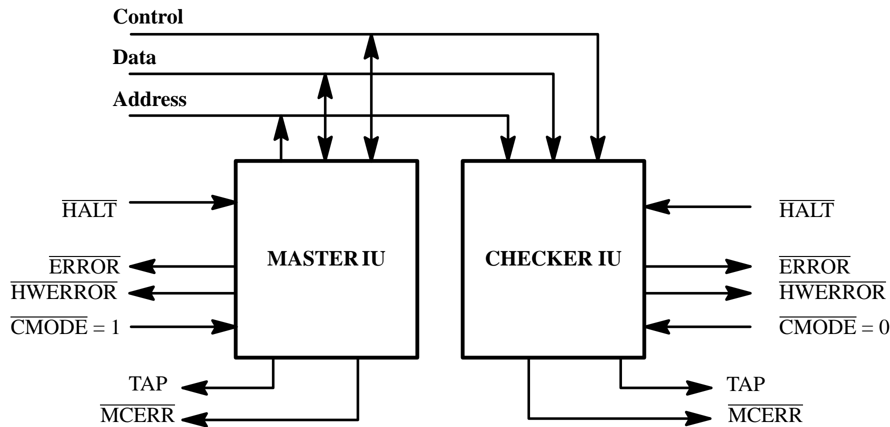


Figure 70. Master/Checker Configuration

4.4.1.1. Master/Checker Signal description

4.4.1.1.1. $\overline{\text{MCERR}}$—Comparison Error (output)

This signal is asserted in the checker mode when a comparison error occurs on the internal output signals (except $\overline{\text{ERROR}}$, $\overline{\text{HWERROR}}$, $\overline{\text{MCERR}}$ and TAP signals) vis-à-vis the output signal of the master IU. It is deasserted when the error disappears.

Note:
$\overline{\text{MCERR}}$ is a three-state (on chip pull–up resistor=20kΩ) output controlled by $\overline{\text{TOE}}$ signal.

4.4.1.1.2. $\overline{\text{CMODE}}$—checker Mode (input)

Assertion of this signal will set the IU to act as a checker to support master/checker operation. All output signals except $\overline{\text{ERROR}}$, $\overline{\text{HWERROR}}$, and TAP signals will be high-Z (on chip pull–up resistor=20kΩ). It is a static signal and shall not change when running. It can change only during reset cycle or halt mode.

4.5. IEEE Standard Test Access Port & Boundary-Scan Architecture

The IU includes a Boundary Scan using a Test Access Port (TAP) interface [**IEEE standard 1149.1**]. This interface is used for debugging and test purposes.

This interface provides standardized approaches to :

1– Testing the interconnections between integrated circuits once they have been assembled on a printed circuit board or other substrate.

2– Support of testing the integrated circuit itself.

3– Observing or modifying activity during the component's normal operation.

4.5.1. TAP

The Test Access Port includes the following connections : TCLK, TMS, $\overline{\text{TRST}}$, TDI and TDO. Dedicated TAP connections are required to allow access to the full range of mandatory features of this standard.

4.5.1.1. TCLK (input)

The Test Clock Input provides the clock for the test logic defined by this standard. TCK is active high. The IEEE standards requires that TCLK can be stopped at 0 indefinitely without causing any change to the state of the test logic. When TCLK is active, CKL must be held to one.

4.5.1.2. TMS (input)

The signal received by TMS is decoded by the TAP controller to control test operation. TMS is sampled on the rising edge of TCLK and has to change on the falling edge of TCLK.

4.5.1.3. TDI (input)

Serial test instructions and data are received by the test logic by TDI. TDI is sampled on the rising edge of TCLK and has to change on the falling edge of TCLK.

4.5.1.4. $\overline{\text{TRST}}$ (input)

The $\overline{\text{TRST}}$ input provides for asynchronous initialization of the TAP controller.

4.5.1.5. TDO (output)

TDO is the serial output for test instructions and data from the test logic defined in the standard.

4.5.2. TAP Controller

The TAP controller is a synchronous finite state machine that responds to changes at the TMS and TCLK signal of the TAP and controls the sequence of operations of the circuit defined by the IEEE standard.

4.5.3. The Instruction Register

The Instruction Register allows an instruction to be shifted into the design. The instruction is used to select the test to be performed or the test data register to be accessed or both. A number of mandatory and optional instructions are defined by the standard. The instructions SAMPLE/PRELOAD, INTEST, EXTEST and BYPASS are implemented on this chip.

The private instruction TESTPAR will be implemented to access the internal scan path registers. These registers are not publicly accessible and will be used to test the internal parity logic.

4.5.3.1. Design and Construction of the instruction register

The instruction register is a shift-based design having an optional parallel input. These parallel inputs permit capture of design-specific information in the Capture-IR state. Figure 71 illustrates an example implementation of an Instruction Register Cell.

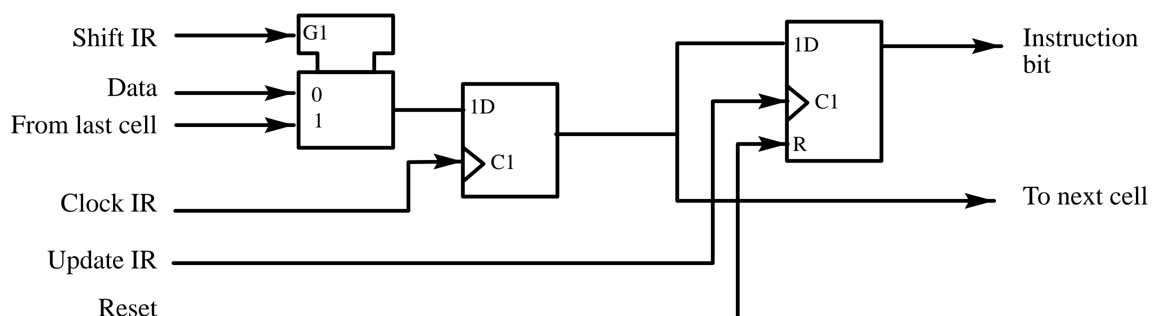


Figure 71. Instruction Register (IR) Cell

4.5.3.2. BYPASS Instruction

The BYPASS register contains a single shift register stage, used to speed-up shifting at the board level, through components which are not activated.

4.5.3.3. EXTEST Instruction

The EXTEST instruction shall connect the BOUNDARY SCAN register between TDI and TDO. It is used to test connections between components on the board level. All output signals can be disabled by using the EXTEST instruction (except TAP).

4.5.3.4. INTEST Instruction

INTEST instruction allows testing of the on-chip system logic while the component is assembled on the board, with each test pattern and response being shifted through the boundary-scan register.

4.5.3.5. SAMPLE/PRELOAD Instruction

The SAMPLE instruction allows normal operation if the system logic with the ability to sample signals entering and leaving the component without affecting circuit operation.

PRELOAD allows a value to be preloaded on the latched outputs of the boundary scan register. This instruction does not modify the system behavior.

4.5.4. The Device Identification Register

The Device Identification Register is implemented on the chip. It contains the TSC691E's assigned component identifier: 0x0b6400b1. It is selected by the IDCODE instruction.

4.5.5. Internal Scan Path

An Internal Scan Path will be implemented to provide the off-line test of the internal parity error detection. This Internal Scan Path will be controlled by the TAP and will force some nodes in the generation circuit of the parity bits. This would then result in a value with the wrong parity. When this value is read again an error will be detected if the error detection works correctly. This chain would have one bit for each parity generator.

4.5.6. Boundary scan test register

The Boundary-scan technique involves the inclusion of a shift register stage (contained in a Boundary-scan cell) adjacent to each component pin so that signals at component boundaries can be controlled and observed using scan testing principles.

Figure 72 illustrates an example implementation for a Boundary-scan cell that could be used for an input or output connection to an integrated circuit. Dependent on the control signals applied to the multiplexers, data can either be loaded into the scan register from the Signal-in port (e.g. the input pin), or driven from the register through the Signal-out port of the cell (e.g. into the core of the component design). The second flip-flop (controlled by clock B) is provided to ensure that the signals driven out of the cell in the latter case are held while new data is shifted into the cell using clock A.

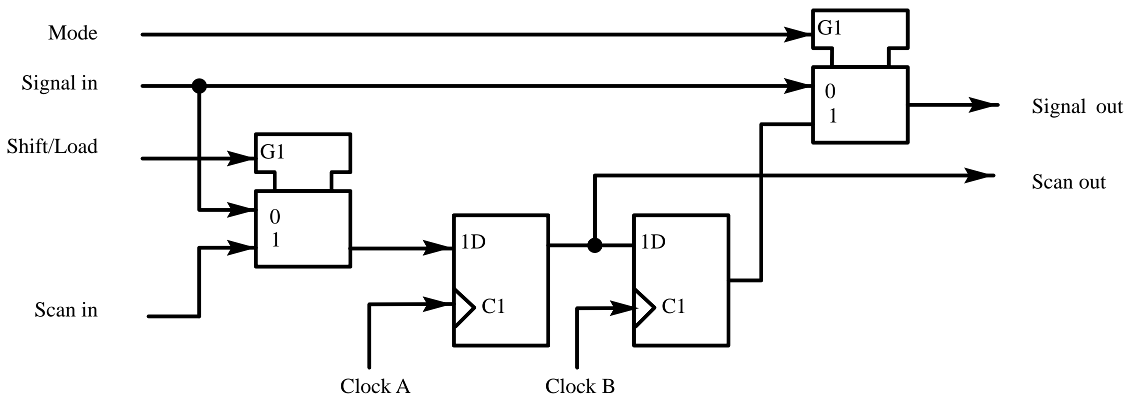


Figure 72. Boundary Scan Cell

4.6. Interleaving register file bits

It is known that the impact from an SEU will flip adjacent bits in a register file. These multiple bit errors might be impossible to detect with one parity bit error. Though these cases with multiple bit errors due to SEU are probably more rare than one bit errors, they cannot be neglected, especially in the register file, which corresponds to 70% of the entire amount of registers in the IU.

One solution to this problem is to interleave the bits of one word with the bits of another word. This is done in the register file and will remove all multiple bit errors due to SEU and full error detection is possible with a single parity bit checker.

5. Electrical and Mechanical Specification

5.1. Maximum rating and DC Characteristics

5.1.1. Maximum Ratings

Storage Temperature . . . . -65 ° C to +150 ° C
Ambient Temperature with Power Applied . . . . -55 ° C to +125 ° C
Supply Voltage[1] . . . . -0.5 V to +7.0 V
Input Voltage . . . . -0.5 V to +7.0 V

5.1.2. Operating Range

Range	Ambient Temperature [2]	Vcc
Military	-55° C to 125° C	5V +/- 10%

5.1.3. DC Characteristics Over the Operating Range

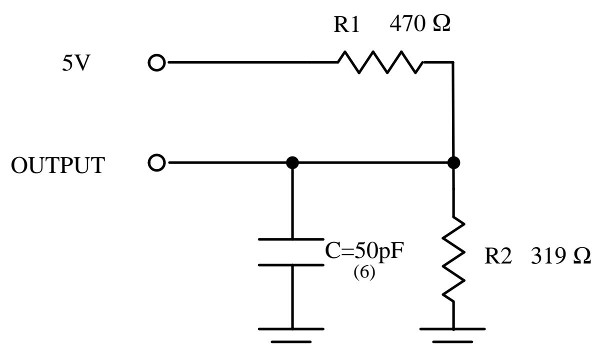
Parameters	Description	Test Conditions	Min.	Max.	Units
V_{OH}	Output HIGH Voltage	V_{CC} = min., I_{OH} = -2.0 mA	2.4		V
V_{OL}	Output LOW Voltage	V_{CC} = min., I_{OL} = +4.0 mA		0.5	V
V_{IH}	Input HIGH Voltage		2.1	V_{CC}	V
V_{IL}	Input LOW Voltage		-0.5	0.8	V
I_{IZ}	Input Leakage Current	V_{CC} = Max., $V_{SS} \leq V_{in} \leq V_{CC}$	-10	10	µA
I_{OZH} I_{OZL}	Output Leakage Current	V_{CC} = Max., $V_{out} = V_{CC}$ V_{CC} = Max., $V_{out} = V_{SS}$	-5 –50[3]	–5 –240[3]	µA
I_{SC}	Output Short Circuit Current	V_{CC} = Max., V_{out} = 0 V	-30	-350	mA
I_{CCop}	Supply Current	V_{CC} = Max., f = 14 MHz	-	200	mA
I_{CCsb}	Stand By Current	Vcc = Max, f = 0 Mhz	-	1	mA

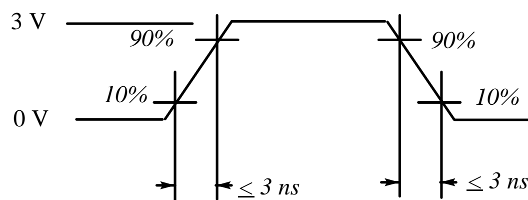
Notes :
1– All power and ground pins must be connected before power is applied.
2– Ambient temperature is defined as the 'instant on' case temperature.
3– On chip pull–up resistor=20kΩ

5.1.4. Capacitance Ratings [4, 5]

Parameters	Description	Max.	Units
C_{IN}	Input Capacitance	10	pF
C_{OUT}	Output Capacitance	12	pF
C_{IO}	Input/Output Bus Capacitance	16	pF

5.1.5. AC Test Loads and Waveforms





TEST LOAD *WAVEFORM*

Notes :
4. Tested initially and after any design or process changes that may affect these parameters.
5. Test conditions are: V_{CC}=5.0V, T_A=25° C, f=1MHz
6. C = 30 pF (for FINS [1:0] signal)

5.2. TSC691E AC Characteristics

5.2.1. AC Characteristics Over the Operation Range [1]

Parameter		Description	Reference Edge	TSC691E – 14 MHz Min	Max	Unit
1	t_{CY}	Clock cycle		71		ns
2	t_{CHL}	Clock high and low		33		ns
3	t_{CRF}	Clock rise and fall		1		V/ns
4	t_{AD}	Address/Control [2] output delay	CLK+		58	ns
5	t_{AH}	Address/Control [2] output valid	CLK+	7		ns
6	t_{DOD}	D[31:0] output delay	CLK-		35	ns
7	t_{DOH}	D[31:0] output valid	CLK-	4		ns
8	t_{DIS}	D[31:0] input setup	CLK+	7		ns
9	t_{DIH}	D[31:0] input hold	CLK+	9		ns
10	t_{MES}	$\overline{\text{MEXC}}$ input setup	CLK+	12		ns
11	t_{MEH}	$\overline{\text{MEXC}}$ input hold	CLK+	4		ns
12–1	t_{HS}	$\overline{\text{MHOLDA, B}}$ input setup	CLK-	7		ns
13–1	t_{HH}	$\overline{\text{MHOLDA, B}}$ input hold	CLK-	9		ns
12–2	t_{YS}	$\overline{\text{yHOLD}}$ [3] input setup	CLK-	10		ns
13–2	t_{YH}	$\overline{\text{yHOLD}}$ [3] input hold	CLK-	7		ns
14	t_{HOD}	$\overline{\text{xHOLD}}$ [7] to Address/Control output delay	XHOLD-		40	ns
15	t_{HOH}	$\overline{\text{xHOLD}}$ [7] to Address/Control output valid	XHOLD+	0		ns
16	t_{OE}	$\overline{\text{AOE}}$, $\overline{\text{COE}}$, $\overline{\text{DOE}}$ to output enable delay	$\overline{\text{xOE}}$- [8]		27	ns

Parameter		Description	Reference Edge	TSC691E – 14 MHz Min	Max	Unit
17	t_{OD}	$\overline{\text{AOE}}$, $\overline{\text{COE}}$, $\overline{\text{DOE}}$ to output disable delay	$\overline{\text{xOE}}$+ [8]		27	ns
18	t_{TOE}	$\overline{\text{TOE}}$ asserted to output enable delay	$\overline{\text{TOE}}$-		38	ns
19	t_{TOD}	$\overline{\text{TOE}}$ deasserted to output disable delay	$\overline{\text{TOE}}$+		38	ns
20	t_{SSD}	INST, FXACK, CXACK, INTACK, $\overline{\text{ERROR}}$ output delay	CLK+		36	ns
21	t_{SSH}	INST, FXACK, CXACK, INTACK, $\overline{\text{ERROR}}$ output valid	CLK+	3		ns
22	t_{RS}	$\overline{\text{RESET}}$ input setup	CLK+	27		ns
23	t_{RH}	$\overline{\text{RESET}}$ input hold	CLK+	3		ns
24	t_{FD}	FINS[1:0] output delay	CLK+		29	ns
25	t_{FH}	FINS[1:0] output valid	CLK+	3.5		ns
24	t_{FD}	CINS[1:0] output delay	CLK+		40	ns
25	t_{FH}	CINS[1:0] output valid	CLK+	3.5		ns
26	t_{FIS}	FCC[1:0], CCC[1:0] input setup	CLK+	18		ns
27	t_{FIH}	FCC[1:0], CCC[1:0] input hold	CLK+	4		ns
28	t_{DXD}	DXFER output delay	CLK+		57	ns
29	t_{DXH}	DXFER output valid	CLK+	10		ns
30	t_{HDXD}	$\overline{\text{XHOLD}}$ [3] asserted to DXFER output delay	$\overline{\text{XHOLD}}$-		36	ns
31	t_{HDXH}	$\overline{\text{XHOLD}}$ [3] deasserted to DXFER output valid	$\overline{\text{XHOLD}}$+	0		ns
32	t_{NUD}	INULL output delay	CLK+		34	ns
33	t_{NUH}	INULL output valid	CLK+	7		ns
34	t_{MDS}	$\overline{\text{MDS}}$ input setup	CLK-	4		ns
35	t_{MDH}	$\overline{\text{MDS}}$ input hold	CLK-	5		ns
36	t_{FLS}	FLUSH output delay	CLK+		30	ns
37	t_{FLH}	FLUSH output valid	CLK+	3		ns
38	t_{CCVS}	FCCV, CCCV input setup	CLK-	13		ns
39	t_{CCVH}	FCCV, CCCV input hold	CLK-	5		ns
40	t_{XES}	$\overline{\text{FEXC}}$, $\overline{\text{CEXC}}$ input setup	CLK+	18		ns
41	t_{XEH}	$\overline{\text{FEXC}}$, $\overline{\text{CEXC}}$ input hold	CLK+	4		ns
42	t_{MAD}	MAO Asserted to Address/Control Output Delay	MAO+		36	ns
43	t_{MAH}	MAO Deasserted to Address/Control Output Valid	MAO-	2		ns
44	t_{ERD}	$\overline{\text{HWERROR}}$ output delay	CLK+		45	ns
45	t_{ERH}	$\overline{\text{HWERROR}}$ output valid	CLK+	5		ns
46	t_{TMS}	TMS input setup	TCLK+	20		ns
47	t_{TMH}	TMS input hold	TCLK+	25		ns
48	t_{TDIS}	TDI input setup	TCLK+	20		ns
49	t_{TDIH}	TDI input hold	TCLK+	25		ns

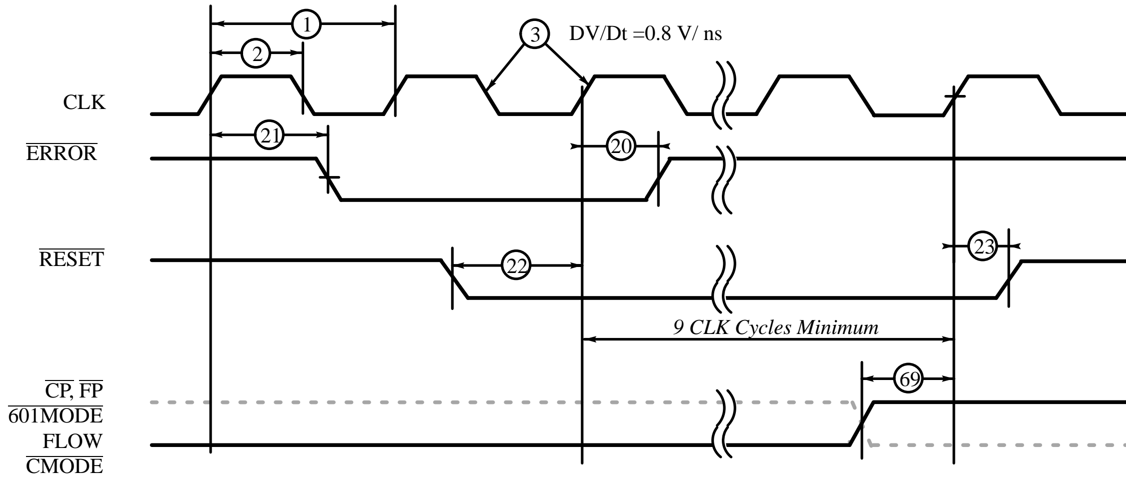
Parameter		Description	Reference Edge	TSC691E – 14 MHz Min	Max	Unit
50	t_{TRS}	$\overline{\text{TRST}}$ input setup	TCLK+	25		ns
51	t_{TRH}	$\overline{\text{TRST}}$ input hold	TCLK+	25		ns
52	t_{TDOD}	TDO output delay	TCLK-		45	ns
53	t_{TDOH}	TDO output valid	TCLK-	5		ns
54	t_{TCY}	TCLK clock cycle		100	1000	ns
55	t_{XAPD}	xAPAR[4] output delay	CLK+		61	ns
56	t_{XAPH}	xAPAR[4] output valid	CLK+	7		ns
57	t_{DPOD}	DPAR output delay	CLK-		45	ns
58	t_{DPOH}	DPAR output valid	CLK-	4		ns
59	t_{DPIS}	DPAR input setup	CLK+	6		ns
60	t_{DPIH}	DPAR input hold	CLK+	4		ns
61	t_{IFPD}	IFPAR output delay	CLK+		53	ns
62	t_{IFPH}	IFPAR output valid	CLK+	3		ns
63	t_{FIPS}	FIPAR input setup	CLK+	18		ns
64	t_{FIPH}	FIPAR input hold	CLK+	4		ns
65	t_{IMPD}	IMPAR output delay	CLK+		61	ns
66	t_{IMPH}	IMPAR output valid	CLK+	7		ns
67	t_{MCED}	$\overline{\text{MCERR}}$ output delay [5]	CLK+		45	ns
68	t_{MCEV}	$\overline{\text{MCERR}}$ output valid [5]	CLK+	5		ns
69	t_{STATS}	$\overline{\text{601MODE}}$/$\overline{\text{FLOW}}$/$\overline{\text{CMODE}}$/$\overline{\text{FP}}$ input setup [6]	CLK+	18		ns
70	t_{HAS}	$\overline{\text{HALT}}$ input setup	CLK-	13		ns
71	t_{HAH}	$\overline{\text{HALT}}$ input hold	CLK-	4		ns
72	t_{IRLS}	IRL[3:0] input setup	CLK+	2		ns
73	t_{IRLH}	IRL[3:0] input hold	CLK+	6		ns

Notes:

1– Test conditions assume signal transition times of 3 ns or less, a timing reference level of 1.5 V, input levels of 0 to 3.0 V and output loading of 50 pF.
2– Address/Control signals include: A[31:0], ASI[7:0], SIZE[1:0], RD, WRT, $\overline{\text{WE}}$, LOCK, and LDSTO.
3– $\overline{\text{yHOLD}}$ includes $\overline{\text{BHOLD}}$, $\overline{\text{FHOLD}}$, and $\overline{\text{CHOLD}}$.
4– xAPAR includes APAR and ASPAR.
5– When an error occurs on D[31:0] or DPAR, $\overline{\text{MCERR}}$ may be delayed for 1 cycles depending of frequency.
6– $\overline{\text{601MODE}}$/$\overline{\text{FLOW}}$/$\overline{\text{CMODE}}$/$\overline{\text{FP}}$ shall be change to be related to positive clock edge during reset active or $\overline{\text{HALT}}$ active.
7– $\overline{\text{xHOLD}}$ includes $\overline{\text{BHOLD}}$, $\overline{\text{MHOLDA}}$, $\overline{\text{MHOLDB}}$, $\overline{\text{FHOLD}}$ and $\overline{\text{CHOLD}}$.
8– $\overline{\text{xOE}}$ includes $\overline{\text{AOE}}$, $\overline{\text{COE}}$ and $\overline{\text{DOE}}$.

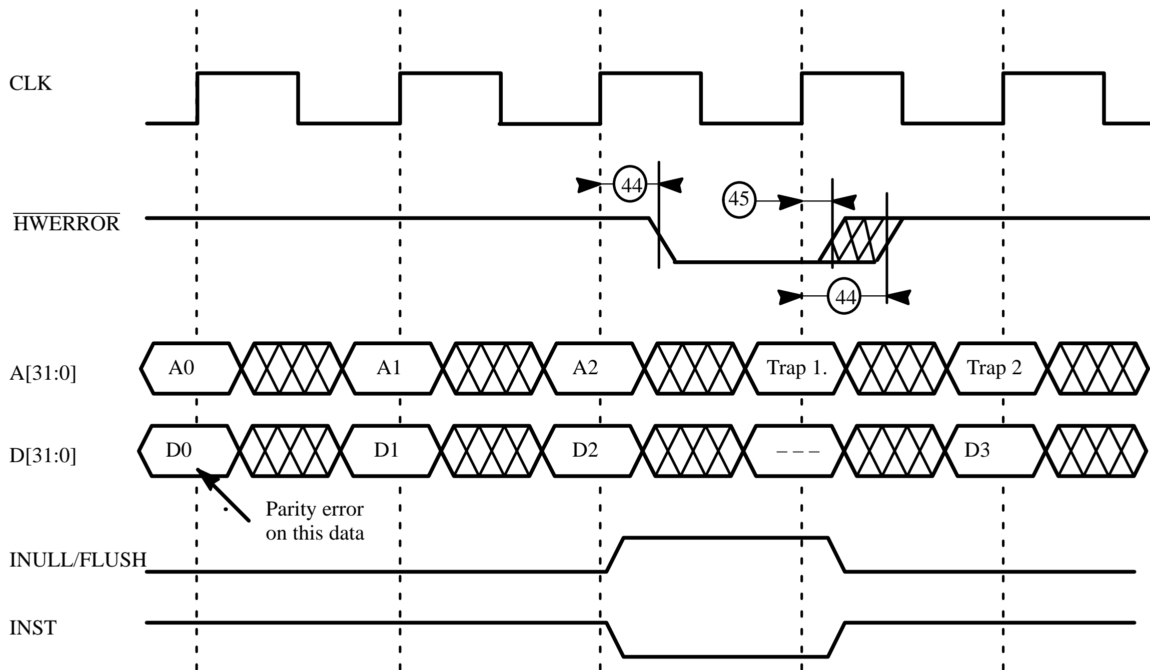
5.2.2. Waveforms

5.2.2.1. Clock and ERROR / RESET Timing



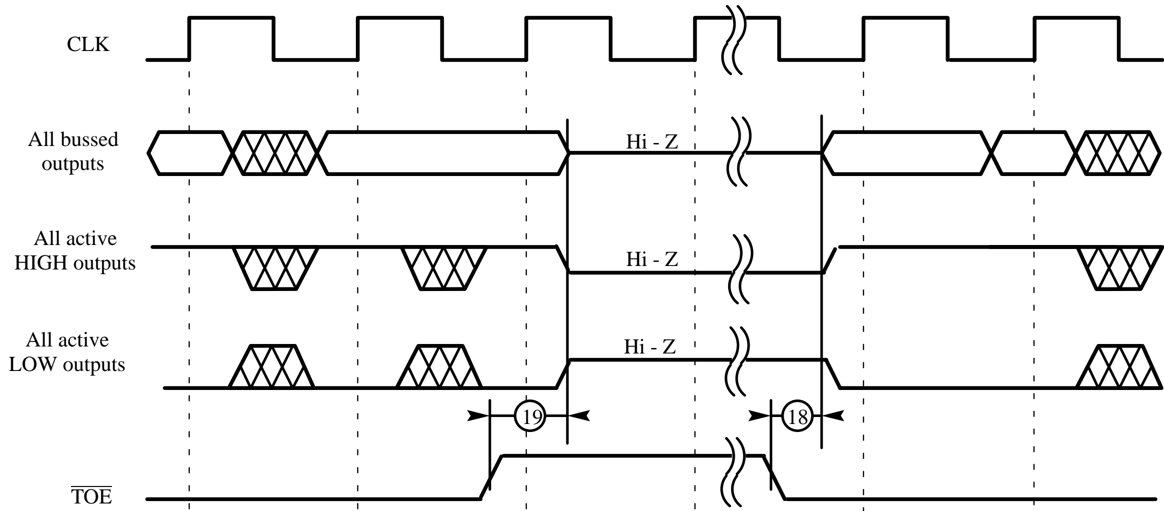
Reset needs to be synchronized with CLK only if the processor must be in step with other devices in the system.

5.2.2.2. Clock and HWERROR Timing for Parity Error Type

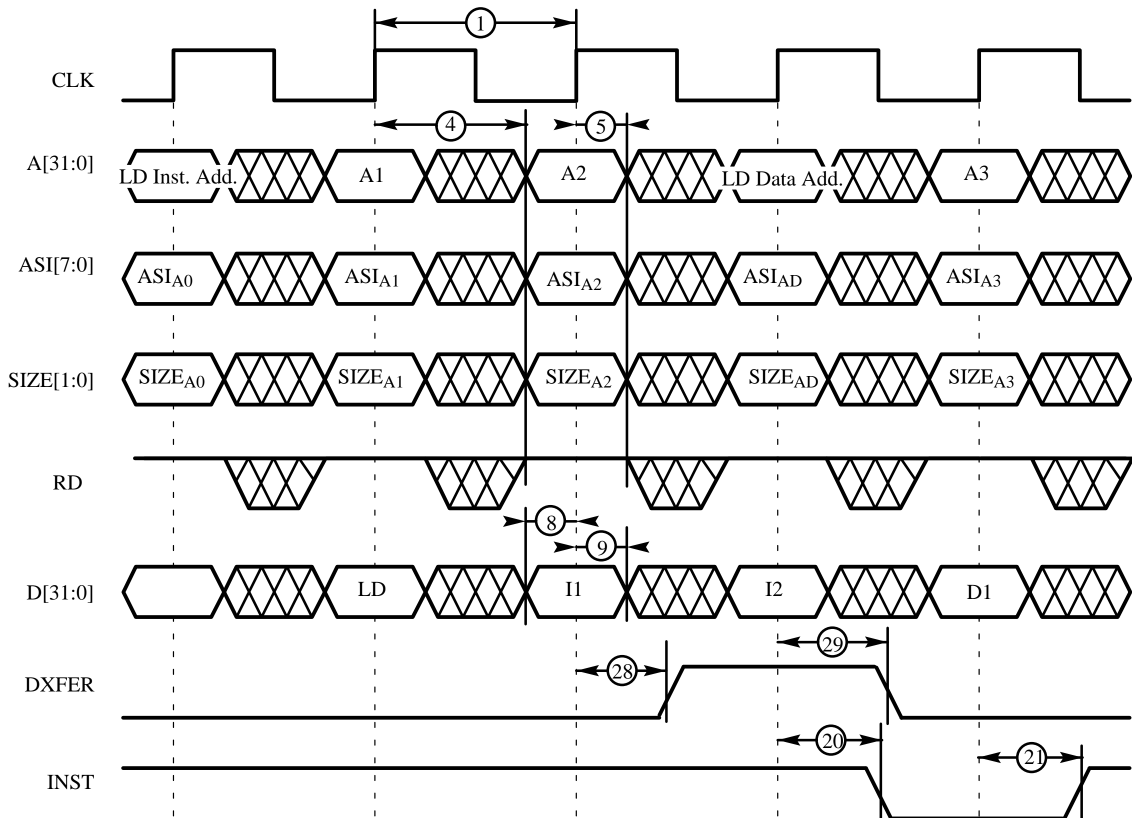


Note: The IU check the parity on internal register when the instruction is in the execute stag.

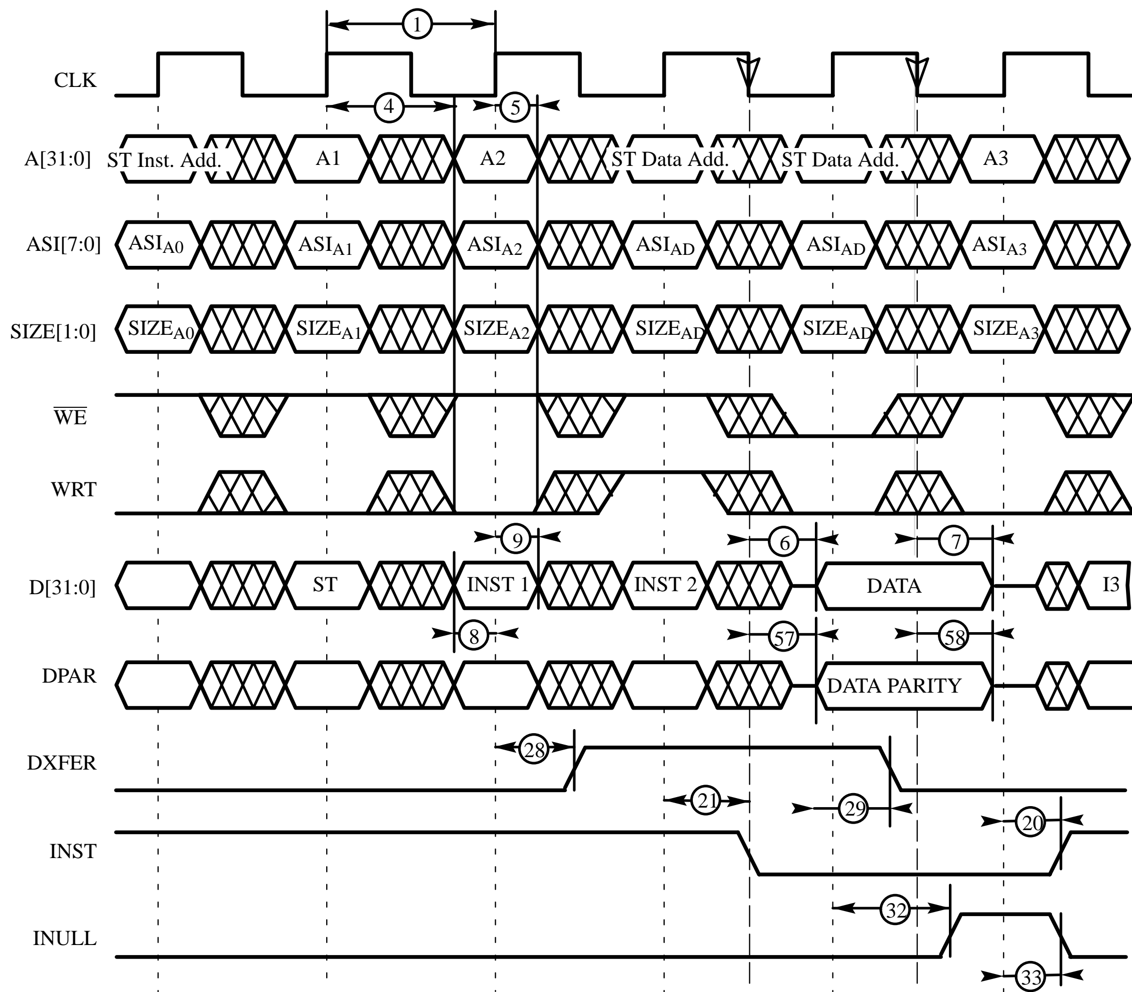
5.2.2.3. $\overline{\text{TOE}}$ De–assertion /Assertion



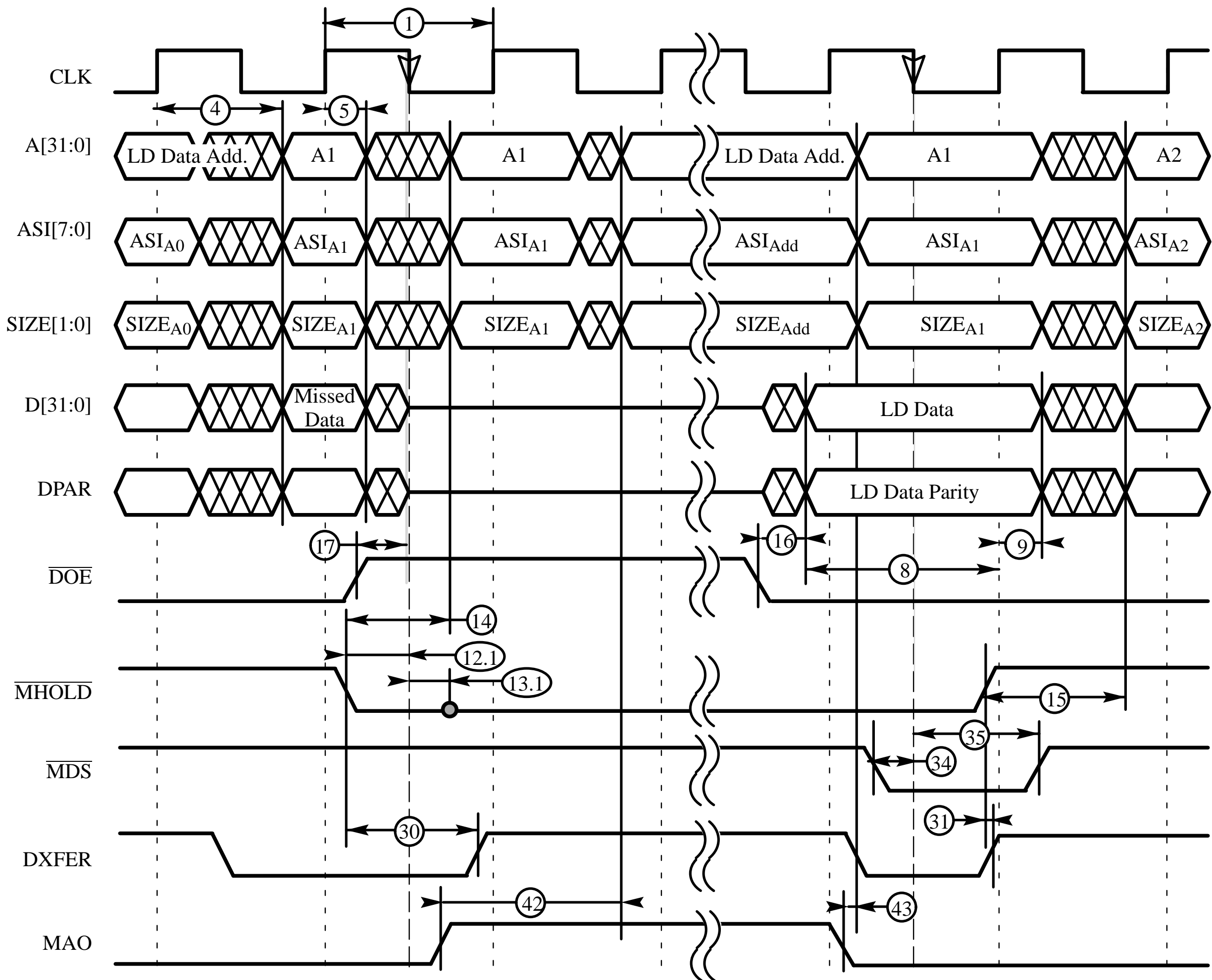
5.2.2.4. Load Timing



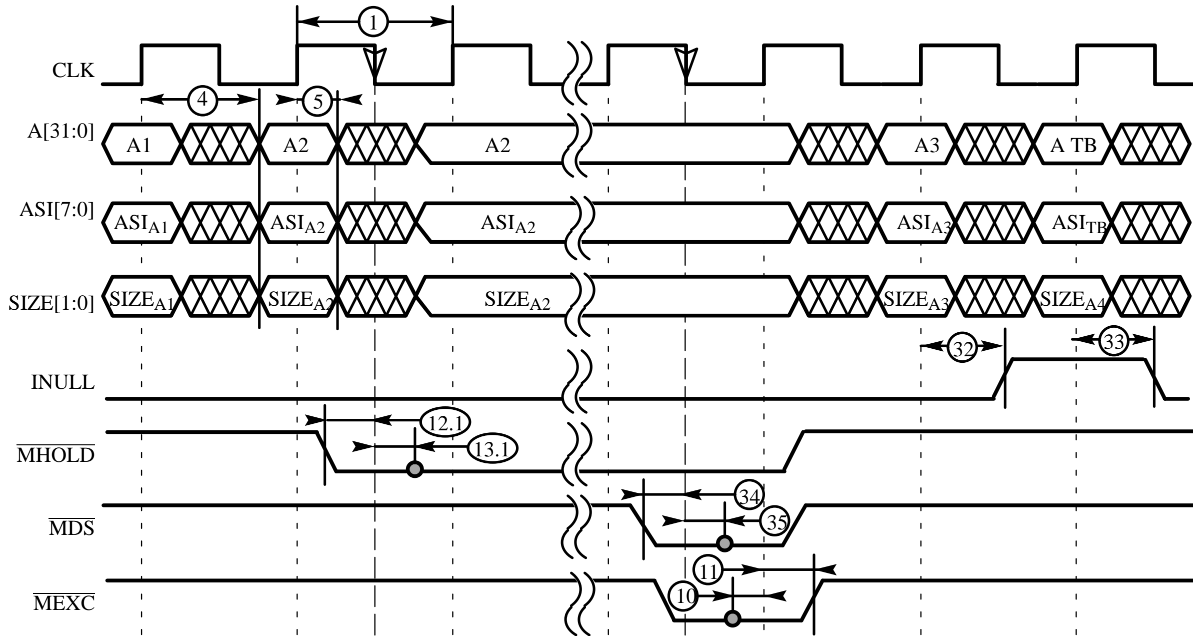
5.2.2.5. Store Timing



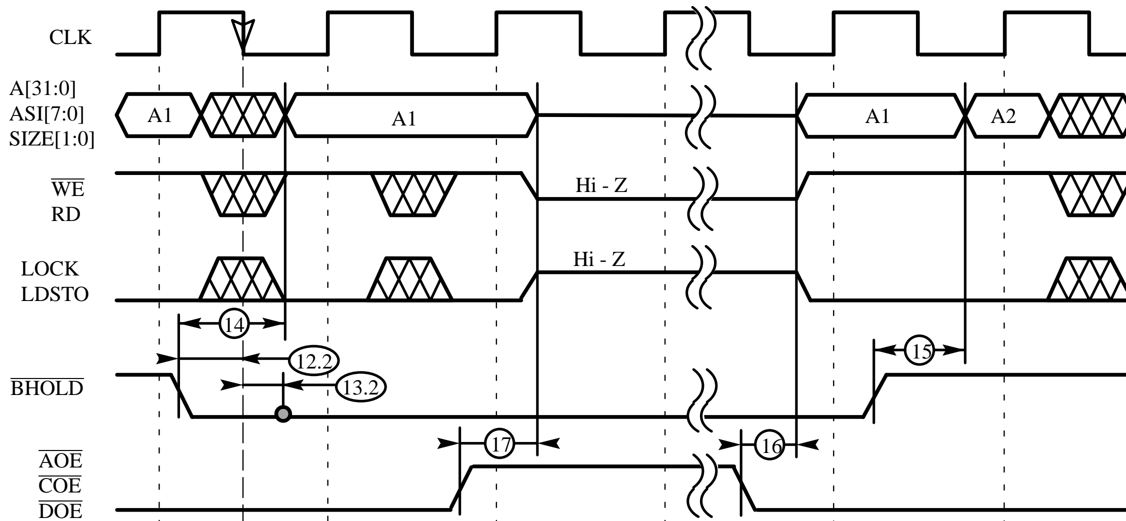
5.2.2.6. Load with Cache Miss



5.2.2.7. Memory Exception Timing

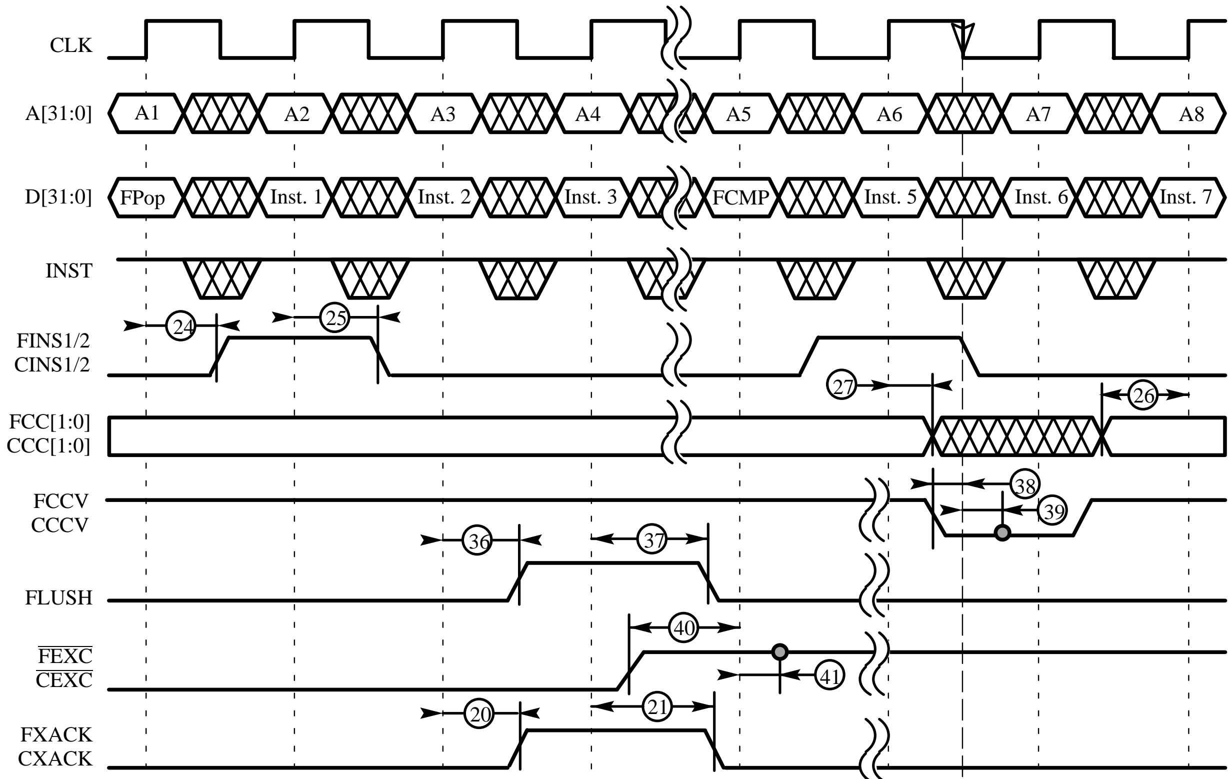


5.2.2.8. Bus Arbitration Timing

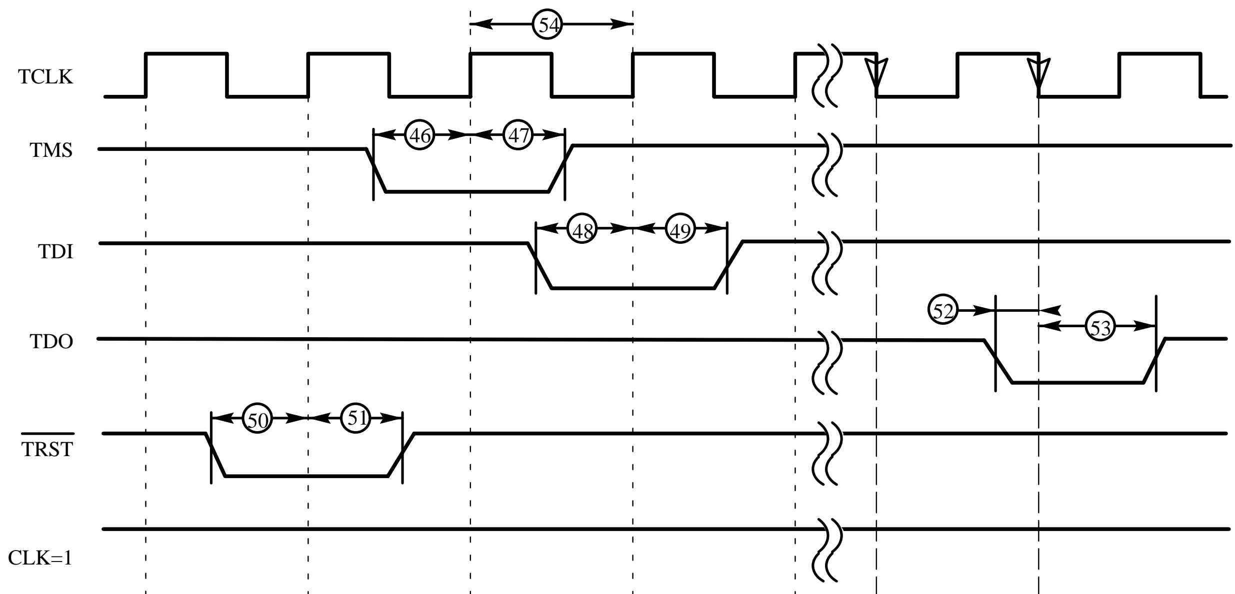


TOE can replace the combined function of AOE, COE, and DOE

5.2.2.9. Floating-Point and Coprocessor Timing



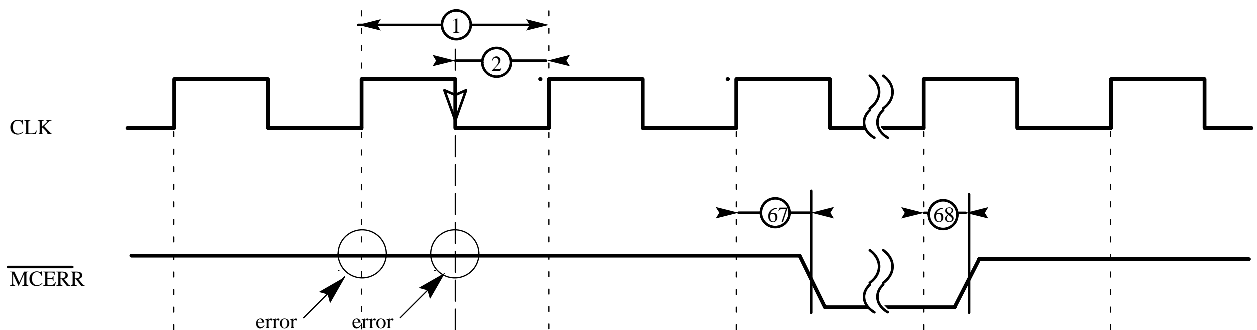
5.2.2.10. TAP Signals



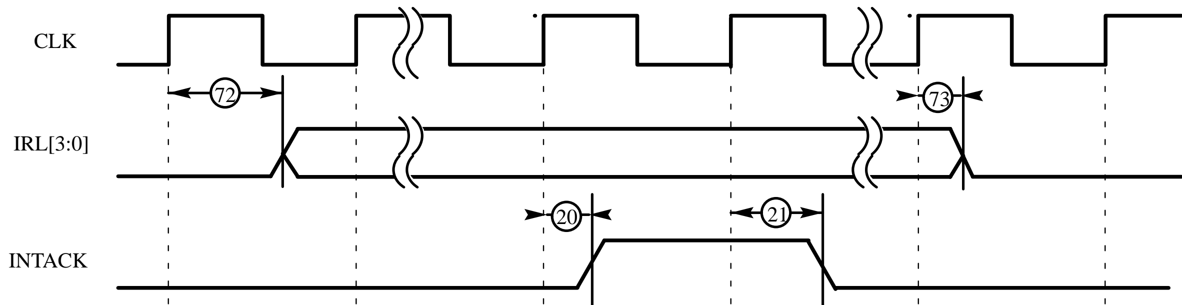
5.2.2.11. PARITY Signals

CLK
A[31:0]
APAR
D[31:0]
DPAR
FIPAR
IFPAR
IMPAR

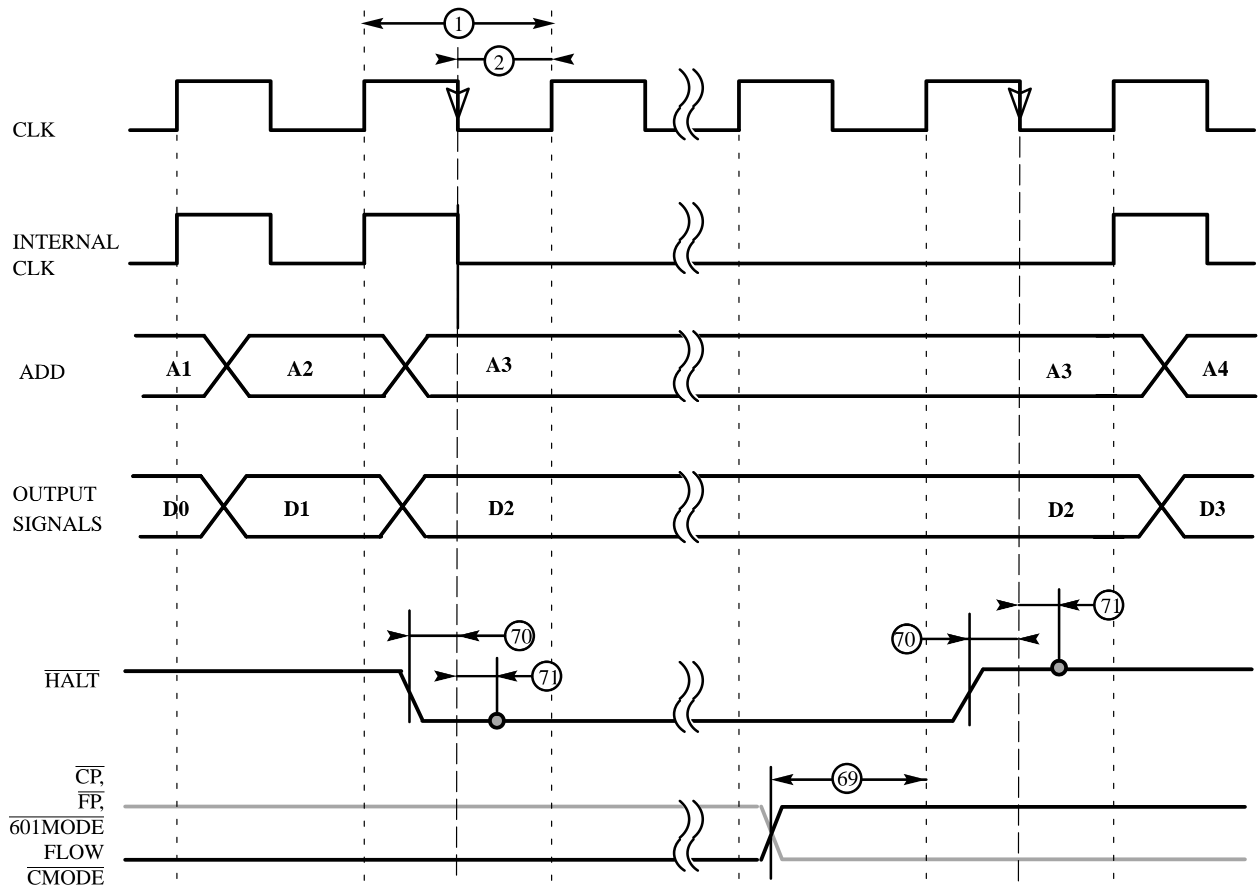
5.2.2.12. MASTER/CHECKER Signals



5.2.2.13. IRL[3:0] Signals

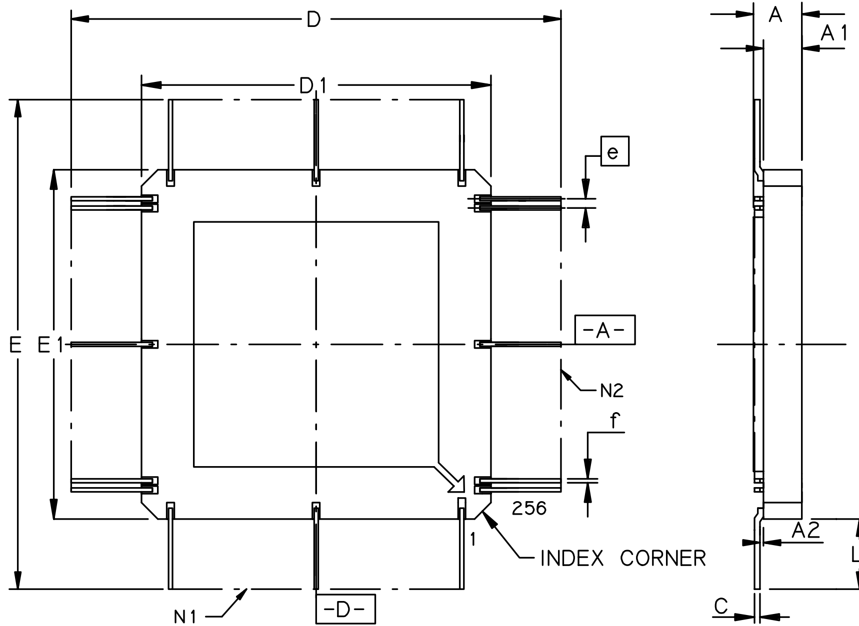


5.2.2.14. HALT Signal timing



5.3. Package Description

5.3.1. 256-Pin MQFP_F Package



	mm		mils	
	Min	Max	Min	Max
A	2.41	3.18	.095	.125
C	0.10	0.20	.004	.008
D	53.23	55.74	2.095	2.195
D1	36.83	37.34	1.450	1.470
E	53.23	55.74	2.095	2.195
E1	36.83	37.34	1.450	1.470
e	0.508 BSC		.020 BSC	
f	0.15	0.25	.006	.010
A1	2.06	2.56	.081	.101
A2	0.05	0.36	.002	.014
L	8.20	9.20	.323	.362
N1	64		64	
N2	64		64	

5.3.2. 256-Pin MQFP_F Pin Assignments

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	IMPAR	33	$\overline{\text{AOE}}$	65	VSSO	97	$\overline{\text{FLOW}}$
2	VCCO	34	APAR	66	VSSO	98	$\overline{\text{MCERR}}$
3	$\overline{\text{COE}}$	35	A0	67	VSSI	99	$\overline{\text{HALT}}$
4	VCCI	36	A1	68	VCCO	100	DPAR
5	DXFER	37	VCCI	69	A16	101	-NC-
6	LOCK	38	A2	70	A15	102	-NC-
7	VSSO	39	-NC-	71	A18	103	D0
8	WRT	40	-NC-	72	A17	104	VSSO
9	SIZE1	41	VSSO	73	A19	105	D1
10	MAO	42	A3	74	VSSO	106	D2
11	ASPAR	43	-NC-	75	A20	107	-NC-
12	SIZE0	44	-NC-	76	VCCI	108	VSSI
13	VCCO	45	A4	77	VSSI	109	D3
14	$\overline{\text{HWERROR}}$	46	A5	78	A21	110	VCCO
15	ASI1	47	-NC-	79	VCCO	111	D4
16	ASI0	48	-NC-	80	A22	112	D5
17	VSSI	49	A6	81	A24	113	-NC-
18	ASI2	50	VCCO	82	A23	114	D6
19	ASI3	51	A7	83	A25	115	VCCI
20	VSSO	52	A8	84	A26	116	D8
21	ASI4	53	A9	85	VSSO	117	D7
22	VCCI	54	A10	86	A27	118	-NC-
23	ASI5	55	VSSI	87	A28	119	-NC-
24	ASI6	56	VSSO	88	A29	120	D9
25	ASI7	57	-NC-	89	VSSI	121	VCCO
26	VCCO	58	A12	90	VSST	122	-NC-
27	VSST	59	A11	91	A30	123	-NC-
28	CLK	60	A14	92	VCCO	124	VSSI
29	-NC-	61	A13	93	A31	125	VCCT
30	VSSI	62	VCCI	94	VCCI	126	VSSO
31	-NC-	63	VCCI	95	$\overline{\text{601MODE}}$	127	VSSO
32	-NC-	64	VCCO	96	-NC-	128	-NC-

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
129	VCCI	161	-NC-	193	VSSO	225	VSST
130	VCCO	162	D24	194	VSSO	226	$\overline{\text{RESET}}$
131	D11	163	-NC-	195	$\overline{\text{IFT}}$	227	VSSI
132	VCCO	164	D25	196	VSSI	228	$\overline{\text{CHOLD}}$
133	D12	165	VCCO	197	FLUSH	229	$\overline{\text{FHOLD}}$
134	D10	166	VCCI	198	IFPAR	230	$\overline{\text{BHOLD}}$
135	VSSO	167	D26	199	VCCO	231	-NC-
136	D13	168	D27	200	$\overline{\text{ERROR}}$	232	$\overline{\text{MHOLDB}}$
137	D15	169	D28	201	CXACK	233	$\overline{\text{MHOLDA}}$
138	D14	170	VSSO	202	INTACK	234	$\overline{\text{MDS}}$
139	D16	171	D29	203	FXACK	235	-NC-
140	VSSI	172	D30	204	VSSO	236	$\overline{\text{FP}}$
141	D17	173	VSSI	205	CCC1	237	$\overline{\text{CEXC}}$
142	VCCO	174	VCCI	206	CCC0	238	$\overline{\text{MEXC}}$
143	D18	175	D31	207	FPSYN	239	-NC-
144	D19	176	$\overline{\text{DOE}}$	208	FCC1	240	-NC-
145	-NC-	177	VCCI	209	VSSI	241	$\overline{\text{FEXC}}$
146	-NC-	178	FINS2	210	FCC0	242	-NC-
147	-NC-	179	FINS1	211	IRL3	243	VSSI
148	-NC-	180	CINS1	212	IRL2	244	VSSO
149	D20	181	VCCO	213	-NC-	245	INST
150	D21	182	$\overline{\text{TOE}}$	214	-NC-	246	RD
151	VSSO	183	VSSI	215	IRL1	247	VCCI
152	-NC-	184	$\overline{\text{TRST}}$	216	-NC-	248	LDSTO
153	-NC-	185	CINS2	217	IRL0	249	VCCO
154	VCCI	186	TDI	218	-NC-	250	$\overline{\text{WE}}$
155	D22	187	TCLK	219	-NC-	251	$\overline{\text{CP}}$
156	-NC-	188	VSSI	220	CCCV	252	VCCT
157	D23	189	TMS	221	FIPAR	253	INULL
158	VSST	190	VCCI	222	VCCI	254	VSSO
159	-NC-	191	TDO	223	FCCV	255	VSSI
160	VSSI	192	VCCO	224	$\overline{\text{CMODE}}$	256	VSSO