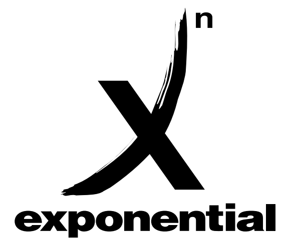
Advance Information

Exponential X704™ RISC Microprocessor

An Implementation of the PowerPC™ Architecture

technical summary



© Exponential Technology 1996

All rights reserved. No part of the contents of this document may be reproduced or transmitted in any form without the express written permission of Exponential Technology, Inc.

This document contains information on a product which has not been commercially released, and may contain technical inaccuracies or typographical errors. Information in this document is subject to change without notice. Exponential makes no express or implied warranty of any kind and assumes no liability or responsibility for any errors or omissions in this document.

PowerPC, PowerPC Architecture, POWER and POWER Architecture are trademarks of International Business Machines Corporation. Exponential and the Exponential logo are trademarks of Exponential Technology, Inc. All other trademarks and registered trademarks are the property of their respective owners.

Contents

Preface

1. Processor Overview 5

1.1 Processor Features 5

1.2 Processor Organization 7

1.2.1 Instruction Fetch Unit 8

1.2.2 Decode Unit 10

1.2.3 Branch Unit 10

1.2.4 Integer Execution Unit 10

1.2.5 Load/Store Unit 11

1.2.6 Floating-Point Execution Unit 13

1.2.7 Level 2 Cache 14

1.2.8 Bus Interface Unit 15

2. PowerPC Architecture Compliance 17

2.1 X^{704} User Instruction Set Architecture (UISA) 17

2.1.1 Reserved Fields 17

2.1.2 Classes of Instructions 17

2.1.2.1 Defined Instruction Class 17

2.1.2.2 Illegal Instruction Class 18

2.1.2.3 Reserved Instruction Class 18

2.1.3 Exceptions 19

2.1.4 Branch Processor 19

2.1.4.1 Instruction Fetching 19

2.1.4.2 Branch Prediction 19
2.1.4.3 Invalid Branch Instruction Forms 19
2.1.5 Fixed-Point Processor 20
2.1.5.1 Load/Store Unit 20
2.1.5.2 Invalid Load/Store Instruction Forms 21
2.1.5.3 Reservation Granularity 21
2.1.5.4 Synchronization Instruction 21
2.1.5.5 Data Breakpoints 21
2.1.6 Fixed-Point Unit 22
2.1.6.1 Invalid Fixed-Point Instruction Forms 22
2.1.7 Floating-Point Unit 23
2.1.7.1 Conformance with IEEE Standard 23
2.1.7.2 Floating-Point Load/Store Operations 23
2.1.7.3 Floating-Point Arithmetic Instructions 23
2.1.7.4 Floating-Point Status and Control Register Instructions 24
2.1.7.5 Optional Instructions 24
2.1.7.6 Denormalized Numbers 24
2.1.7.7 Floating-Point Exceptions 24
2.1.7.8 Invalid Floating-Point Instruction Forms 25

2.2 X^{704} Virtual Environment Architecture (VEA) 25

2.2.1 Storage Model 25
2.2.1.1 Caches 25
2.2.1.2 Storage Consistency 27
2.2.2 Effect of Operand Placement on Performance 27
2.2.3 Cache Management Instructions 27
2.2.3.1 Instruction Cache Block Invalidate (***icbi***) 27
2.2.3.2 Instruction Synchronize (***isync***) 28
2.2.3.3 Data Cache Block Touch (***dcbt***) 28
2.2.3.4 Data Cache Block Touch for Store (***dcbtst***) 28
2.2.3.5 Data Cache Block Zero (***dcbz***) 29
2.2.3.6 Data Cache Block Store (***dcbst***) 29
2.2.3.7 Data Cache Block Flush (***dcbf***) 30
2.2.4 Additional Diagnostic Instructions. 30
2.2.5 Storage Access Ordering 32

2.2.6 Executing Modified Code . . . 32
2.2.7 Atomic Update Primitives . . . 33
2.2.8 Timer Facilities . . . 33
2.3 X^{704} Operating Environment Architecture (OEA) . . . 33
2.3.1 Reserved Fields in Storage Tables . . . 33
2.3.2 Exceptions . . . 33
2.3.3 Branch Processor . . . 34
2.3.3.1 SRR0 and SRR1 . . . 34
2.3.3.2 MSR . . . 34
2.3.4 Fixed-Point Processor . . . 35
2.3.4.1 Software Use SPRs . . . 35
2.3.4.2 Processor Version Register . . . 35
2.3.4.3 Additional Special Purpose Registers . . . 35
2.3.4.4 TLB Miss Registers . . . 38
2.3.4.4.1 TLB Miss Address Register (MAR) . . . 38
2.3.4.4.2 TLB Miss Interrupt Status Register (MISR) . . . 39
2.3.4.4.3 TLB Miss PTE Compare Register (CMP) . . . 39
2.3.4.4.4 TLB Miss PTEG Address Hash Registers (HASH1 and HASH2) . . . 40
2.3.4.4.5 TLB Miss Update LRU Registers (TLBLRU0 and TLBLRU1) . . . 40
2.3.4.4.6 TLB Miss Update MRF Register (TLBMRF) . . . 41
2.3.4.5 Debugging Registers . . . 41
2.3.4.5.1 Breakpoint Control Register (BPTCTL) . . . 42
2.3.4.5.2 Instruction Address Breakpoint Register (IABR) . . . 44
2.3.4.5.3 Extended Data Address Breakpoint Register (XDABR) . . . 44
2.3.4.5.4 Data Address Breakpoint Register (DABR) . . . 45
2.3.4.5.5 Event Register (EVENT) . . . 45
2.3.4.6 Processor Control Registers . . . 47
2.3.4.6.1 Modes Register (MODES) . . . 47
2.3.4.6.2 Machine Check Register (CHECK) . . . 48
2.3.4.6.3 L2/Bus Control Register (L2CTL) . . . 50
2.3.4.6.4 L2 Column Disable Register (L2CDR) . . . 52
2.3.4.7 Processor Identification Register (PIR) . . . 52
2.3.5 Storage Control . . . 53
2.3.5.1 Translation Lookaside Buffer (TLB) . . . 53
2.3.5.2 Block Address Translation . . . 54
2.3.5.3 Storage Access Modes . . . 54
2.3.5.4 Reference and Change Recording . . . 54

2.3.5.5 Storage Control Instructions 54
2.3.5.5.1 Data Cache Block Invalidate (***dcbi***) 55
2.3.5.5.2 TLB Invalidate Entry (***tlbie***) 55
2.3.5.5.3 TLB Invalidate All (***tlbia***) 55
2.3.5.5.4 TLB Synchronize (***tlbsync***) 55

2.3.6 Interrupts 56
2.3.6.1 Interrupt Classes 56
2.3.6.2 Interrupt Definitions 56
2.3.6.2.1 System Reset Interrupt 57
2.3.6.2.2 Machine Check Interrupt 57
2.3.6.2.3 Data Storage Interrupt 58
2.3.6.2.4 Instruction Storage Interrupt 59
2.3.6.2.5 External Interrupt 59
2.3.6.2.6 Alignment Interrupt 60
2.3.6.2.7 Program Interrupt 60
2.3.6.2.8 Floating-Point Unavailable 60
2.3.6.2.9 Decrementer Interrupt 60
2.3.6.2.10 System Call Interrupt 60
2.3.6.2.11 Trace Interrupt 60
2.3.6.2.12 Floating-Point Assist Interrupts 61
2.3.6.2.13 TLB Miss interrupt 61
2.3.6.2.14 TLB Store Interrupt 62
2.3.6.3 Exception Ordering 64

2.3.7 Synchronization Requirements for Special Registers 64

3. Processor Operation 67

3.1 Execution Pipeline 67
3.1.1 Fetch Stage (F) 68
3.1.2 Decode Stage (D) 68
3.1.3 Address Generation Stage (A) 68
3.1.4 Cache Access Stage (C) 68
3.1.5 Tag Match Stage (M) 68
3.1.6 Writeback Stage (W) 68
3.1.7 ALU Operations 69
3.1.8 Floating-Point Operations 69

3.2 Instruction Cache 69

3.3 Data Cache 70

3.4 Level 2 Cache 72
3.4.1 Level 2 Cache Tags 72
3.4.2 Address Translation and the Level 2 Cache 73
3.4.3 Level 2 Cache Replacement Policy. 73
3.4.4 Disabling the Level 2 Cache. 75
3.4.5 Flushing the Level 2 Cache 75
3.4.6 Cache Coherency Protocol. 77
3.4.7 Cache Prefetching 78
3.5 Translation Lookaside Buffer (TLB) 79
3.6 Instruction TLB (ITLB) 81
3.7 Block Address Translation 81
3.8 Branch Prediction 81
3.8.1 Branch Direction Prediction 82
3.8.2 Branch Target Prediction 83
3.8.3 Finder Initialization 83
3.9 Diagnostic Accesses. 84
3.10 Power-On Reset and Hard Reset Initialization. 85
4. Instruction Execution. 87
4.1 Pipeline Diagrams. 87
4.2 Sliding ALU Stage. 88
4.3 Branch Resolution. 91
4.4 Instruction Grouping Rules 92
4.5 Fetch Stalls 94
4.6 Decode Stalls 95
4.7 Pipe Stalls 97
4.8 Penalties for Algebraic and Misaligned Loads and Stores 99
4.8.1 Pipeline Diagrams for Algebraic Loads. 99

4.8.2 Pipeline Diagrams for Misaligned Loads 100
4.8.3 Pipeline Diagram for Misaligned Stores 101
4.9 Floating-Point Execution 101
4.9.1 Floating-Point Computational Instructions 103
4.9.2 Floating-Point Compare Instructions 103
4.9.3 Floating-Point Load and Store Instructions 104
4.9.4 Floating Point Exceptions and Condition Register Updates 105
4.9.5 Floating-Point and Integer Pipeline Synchronization 105
4.9.6 Optimizing Floating-Point Performance 106
5. Signal Descriptions 107
5.1 Bus Interface Signals 107
5.2 Signal Descriptions 109
5.2.1 Clock and Phase-Locked Loop Signals 111
5.2.2 Test Signals 113
5.2.3 Thermal Monitoring and Control Signals 113
6. Processor Interface 115
6.1 Address Bus 116
6.2 Data Bus 116
6.3 Coherency Protocol 116
6.4 Features for Improved Bus Performance 117
7. Test Interface 119
7.1 JTAG Interface 119
7.2 Scan Chains 119
7.3 At-Speed Testing 120
8. Package Description 121
Appendix A. Sample TLB Interrupt Handlers 127
Index 133

Figures

Figure 1: Data Path Simplified Block Diagram . . . 8
Figure 2: Instruction Fetch Unit Simplified Block Diagram . . . 10
Figure 3: Integer Execution Unit Simplified Block Diagram. . . . 11
Figure 4: Load/Store Unit Simplified Block Diagram. . . . 13
Figure 5: Floating-Point Execution Unit Block Diagram . . . 14
Figure 6: TLB Miss PTE Compare Register . . . 39
Figure 7: TLB Miss PTEG Address Hash Registers . . . 40
Figure 8: TLBLRU Registers . . . 40
Figure 9: TLB Entry. . . . 41
Figure 10: Breakpoint Control Register. . . . 42
Figure 11: Instruction Address Breakpoint Register. . . . 44
Figure 12: Data Address Breakpoint Register . . . 45
Figure 13: Event Register . . . 46
Figure 14: Modes Register . . . 47
Figure 15: Machine Check Register. . . . 49
Figure 16: L2/Bus Control Register . . . 50
Figure 17: L2 Column Disable Register. . . . 52
Figure 18: Instruction Cache Tags . . . 70
Figure 19: Data Cache Tags. . . . 71
Figure 20: Level 2 Cache Tags . . . 72
Figure 21: L2 Cache Use Record . . . 73
Figure 22: TLB Entry. . . . 79
Figure 23: Predicted Branch Direction State Transitions . . . 83
Figure 24: X^{704} Bus Interface Signals . . . 108
Figure 25: Pinout Diagram for the X^{704} Package. . . . 121
Figure 26: X^{704} Package Structure. . . . 124
Figure 27: X^{704} Package Drawing. . . . 125

Tables

Table 1: Processor Revision Values 35

Table 2: Special Purpose Registers 36

Table 3: Event Counter Selections 46

Table 4: Interrupt Vector Offsets 56

Table 5: Synchronization Requirements for Implementation-Dependent SPRs 64

Table 6: Level 2 Cache Tag MESI State Values 73

Table 7: Diagnostic Address Space 84

Table 8: Hard Reset State Initialization 86

Table 9: Floating-Point Instruction Bandwidth and Latency 102

Table 10: Floating-Point and Integer Pipeline Alignments 105

Table 11: Processor-Dependent Signal Descriptions 109

Table 12: Typical PLL_CFG Settings 112

Table 13: CLK_CTL Settings 112

Table 14: Pinout Listing for the X^{704} Package 122

Preface

Scope of this Document

This document presents information supporting hardware design and systems programming with Exponential Techology's X^{704}. Note that this document is in progress and is subject to change at any time. Presently, it consists of eight chapters and an appendix, as follows:

- Chapter 1, "Processor Overview," describes basic features of the X^{704} and provides an overview of the processor's organization and architecture.
- Chapter 2, "PowerPC Architecture Compliance," contrasts the implementation of the X^{704} to the Apple-IBM-Motorola specification for PowerPC architecture. It provides details of features specific to the X^{704} implementation.
- Chapter 3, "Processor Operation," provides a detailed look at the X^{704} hardware features that are of particular interest to systems programmers.
- Chapter 4, "Instruction Execution," provides details on the operation of the instruction pipelines. It will benefit software engineers working to predict processor performance or to optimize software.
- Chapter 5, "Signal Descriptions," presents details on the X^{704} hardware interface.
- Chapter 6, "Processor Interface," summarizes aspects of the hardware interface that are unique to the X^{704}.
- Chapter 7, "Test Interface," presents details on X^{704} testability.
- Chapter 8, "Package Description," provides a physical and mechanical description of the X^{704}.
- Appendix A, "Sample TLB Interrupt Handlers," provides code examples.

Documentation Conventions and Definitions

This document follows the notation conventions used in the *PowerPC Architecture Specification* referenced on page 3. In addition, the following notation is used:

- 0b*nnnn* indicates a number expressed in binary format; 0x*nnnn* indicates a number expressed in hexadecimal format (for example, 0x4F00).
- Instruction mnemonics appear in lowercase, bold italic typefaces (for example, ***sync***, ***tlbsync***).
- Bits are numbered from left to right, starting with the lower numbered bit.
- Ranges of bits are specified in parentheses with starting and ending numbers separated by a colon. For example, (5:7) denotes bits five through seven.
- Register names, fields of instructions, fields of special purpose registers, and macro names appear in uppercase (for example, MSR, BO, RA, and VSID).
- REG[FIELD] indicates a specific field within a register (for example, FPSCR[NI]).
- REG(p) or REG(p:q) indicates a specific bit or range of bits, respectively, within a register (for example, BO(2)).
- (x) indicates the contents of register x, when x is an instruction field name. For example, (RA) means the contents of register RA, and (FRA) means the contents of register FRA, where RA and FRA are instruction fields.
- (RA|0) indicates the contents of register RA where RA has the value of 1-31, or the value 0 when RA contains 0.
- ACTIVE_HIGH signals appear in uppercase text (for example, SCAN_EN and SCAN_SER).
- ACTIVE_LOW signals appear in uppercase text with an overbar (for example, $\overline{\text{ABB}}$ and $\overline{\text{DBB}}$).
- SIG0–SIG7 indicates a group of signals from SIG0 to SIG7.
- The term *power-endian* is used to refer to the pseudo-little-endian mode defined for the PowerPC.

Applicable Documents

The X^{704} is compatible with the PowerPC architecture as specified in the following documents:

- IBM, *PowerPC User Instruction Set Architecture (Book I)*, Morgan Kaufmann, San Francisco, CA, second edition, December 13, 1994.
- IBM, *PowerPC Virtual Environment Architecture (Book II–AIM)*, Morgan Kaufmann, San Francisco, CA, second edition, December 13, 1994.
- IBM, *PowerPC Operating Environment Architecture (Book III–AIM)*, Morgan Kaufmann, San Francisco, CA, second edition, December 13, 1994.

These documents are collectively referred to as the *PowerPC Architecture Specification, PowerPC architecture,* or simply as the architecture specification and are individually referred to as Book I, Book II, and Book III. Readers of this document should be familiar with these books.

The X^{704} bus interface is compatible with the interface described in:

- *Motorola: PowerPC 604 Microprocessor Interface Specification*, March 28, 1994.

This document is referred to as the bus specification.

The X^{704} supports a test interface compatible with the IEEE 1149.1 standard described in:

- *IEEE, New York, NY: IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1, May, 1990

1. Processor Overview

This document describes the X^{704} implementation of the PowerPC architecture.

1.1 Processor Features

The Exponential Technology X^{704} is a single-chip implementation of the 32-bit PowerPC architecture that conforms fully to the *PowerPC Architecture Specification*. The X^{704} processor features:

- separate integer, load/store, branch, and floating-point units
- up to three instructions issued each cycle
- separate level 1 data and instruction caches
- unified data and instruction level 2 cache
- on-chip translation lookaside buffer (TLB)

Integer Unit

- executes all arithmetic, logical, compare, rotate, and shift instructions except multiply and divide in a single cycle
- executes multiply instructions in 3 to 6 cycles
- bypasses results to following instructions with no delay

Load/Store Unit

- supports issue of a load or store each cycle
- handles all big-endian mode misaligned loads and stores in hardware
- supports power-endian mode, including some misaligned accesses
- forwards load data to the integer unit with no load-use penalty

Branch Unit

- supports issue of a branch or condition register logical instruction each cycle
- maintains 2-bit dynamic branch prediction in hardware
- supports prediction through both PC-relative and indirect branches
- no penalty for following correctly predicted branches
- recovers quickly from mispredicted branches

Floating-Point Unit

- complies with IEEE-754 single-precision and double-precision arithmetic standard
- implements optional ***fsel*** and ***stfiwx*** instructions
- supports denormalized numbers in hardware

Caches

- 2-level cache hierarchy
- 2KB direct-mapped instruction cache with 32-byte blocks
- 2KB direct-mapped write through data cache with 32-byte blocks
- 32KB 8-way set-associative unified level 2 cache with 32-byte blocks
- supports write through and copy back protocols (level 2 cache)
- supports all PowerPC cache operations
- physically indexed and physically tagged caches
- features 4-doubleword store queue between load/store unit and data/level 2 caches
- features software disables
- maps out damaged blocks and columns (level 2 cache)

Memory Management Unit

- contains 128-entry, 4-way set-associative TLB with hardware-assisted software refill
- contains four-entry, fully associative instruction TLB with hardware refill from main TLB
- supports block address translation for four instruction blocks and four data blocks

MultiProcessing Support

- supports MESI cache coherency protocol
- supports ***lwarx*** and ***stwcx.*** memory synchronization instructions for atomic updates
- broadcast synchronization of cache operations and serialization
- broadcast TLB invalidates

Bus Interface

- supports standard 64-bit data, 32-bit address 60x bus
- supports data streaming with optional *fast L2* mode
- supports pipelined and split transactions
- supports processor clock that is an integral multiple of bus clock

Test Interface

- features JTAG TAP controller with boundary scan
- proprietary scan access to all internal flip flops
- supports scan access to all internal RAM structures
- supports instruction-level access to all internal RAM structures
- performs at-speed fault testing

1.2 Processor Organization

This section presents a high-level view of the X^{704} processor. See Chapter 3 for detailed descriptions of the X^{704} micro-architecture and implementation.

The major functional blocks of the X^{704} include the following:

- instruction fetch unit, including the instruction cache
- decode unit
- integer execution unit
- load/store unit, including the data cache and TLB
- floating-point execution unit
- level 2 cache
- bus interface unit

The block diagram in Figure 1 depicts an overview of the X^{704} data paths.

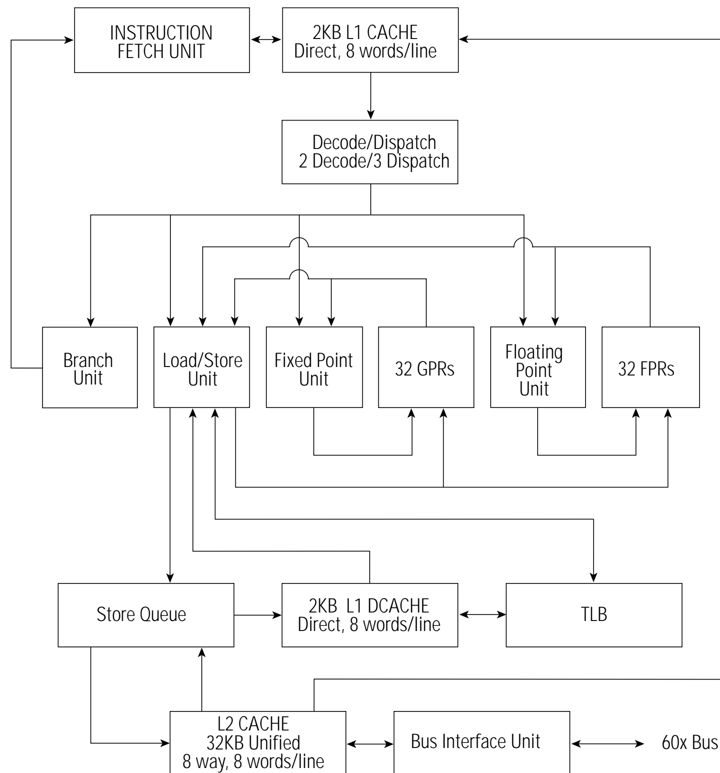


Figure 1: Data Path Simplified Block Diagram

1.2.1 Instruction Fetch Unit

The instruction fetch unit contains the instruction cache, the instruction TLB and the IBAT registers, a branch prediction RAM known as the *finder*, and a 6-word instruction buffer. The instruction buffer consists of a four-entry decode buffer and a two-entry fetch buffer. Figure 2 shows a simplified block diagram of the instruction fetch unit.

As the decode unit empties the decode buffer, the fetch unit continually reads instructions from the instruction cache and places them in the instruction buffer. Instructions not consumed by the decode unit are moved to the front of the decode buffer. An aligned doubleword can be read from the instruction cache on each cycle. The instruction cache is not read during an

instruction TLB miss, during an instruction cache miss, or when the fetch buffer is not empty. Only one instruction can be placed in the decode buffer after the instruction stream branches to an instruction on an odd word address. Instructions are placed directly in the decode buffer portion of the instruction buffer if it is not full; the fetch buffer holds any overflow.

The fetch unit maintains its own copy of the program counter, called the fetch PC, that is updated in one of three ways:

- If the finder indicates that the instruction being read is not a branch or is a branch that is predicted to be not taken, the fetch unit increments the counter.
- If the finder predicts that a branch will be taken, the fetch unit sets the counter to the branch target address.
- If the decode unit indicates that a previous branch was predicted incorrectly, the fetch unit sets the counter to the correct branch target address provided by the decode unit.

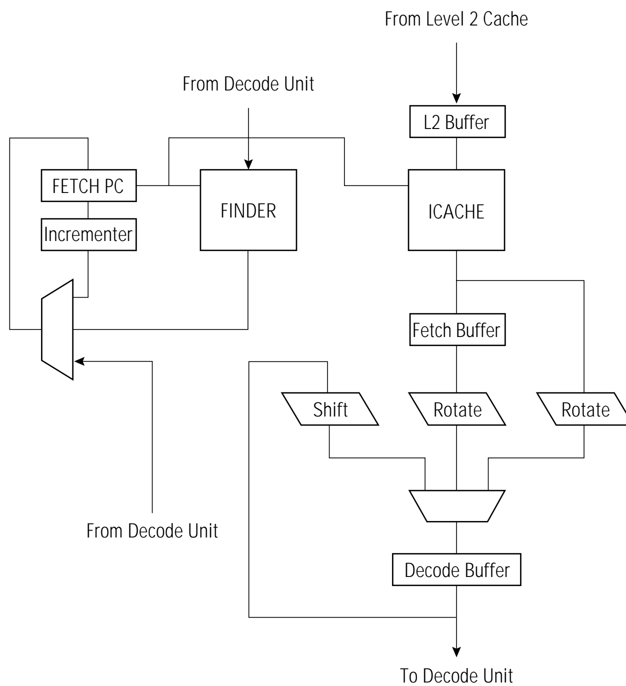


Figure 2: Instruction Fetch Unit Simplified Block Diagram

The fetch unit predicts indirect branches, including return-from-interrupt, and some interrupts. However, not all branches can be predicted. See Section 3.8 on page 81 for more information on branch prediction.

1.2.2 Decode Unit

The decode unit examines the contents of the first three entries in the decode buffer and determines whether the first, the first and the second, or all three of those instructions can be *issued*—sent to the appropriate execution unit—on each cycle. Integer register operands are read from the integer register file, which is part of the decode unit. The decode unit tracks inter-instruction interlocks and ensures that all results are correctly bypassed to any instructions that need them. This unit also processes exceptions.

1.2.3 Branch Unit

The branch unit determines whether branches are taken and computes branch target addresses. The branch unit tracks all branch predictions made by the fetch unit and handles mispredicted branches by flushing the pipeline and sending the correct branch target address back to the fetch unit.

The condition register resides in the branch unit, so all condition register logical instructions are executed here. The branch unit also contains the link register, count register, XER, MSR, SRR0, SRR1, DEC, TBU, and TBL special purpose registers.

1.2.4 Integer Execution Unit

The integer execution unit consists of a single pipe stage that executes instructions in one of five subunits: an adder unit, a logical operation unit, a shifter/rotator, a leading-zero counter, and a multiplier. The multiplier takes multiple cycles and includes two internal registers, MQ1 and MQ2, that hold intermediate results. Divides use a combination of the adder and the shifter/rotator. Only one subunit can execute an instruction in any one cycle. Figure 3 shows a simplified block diagram of the integer execution unit.

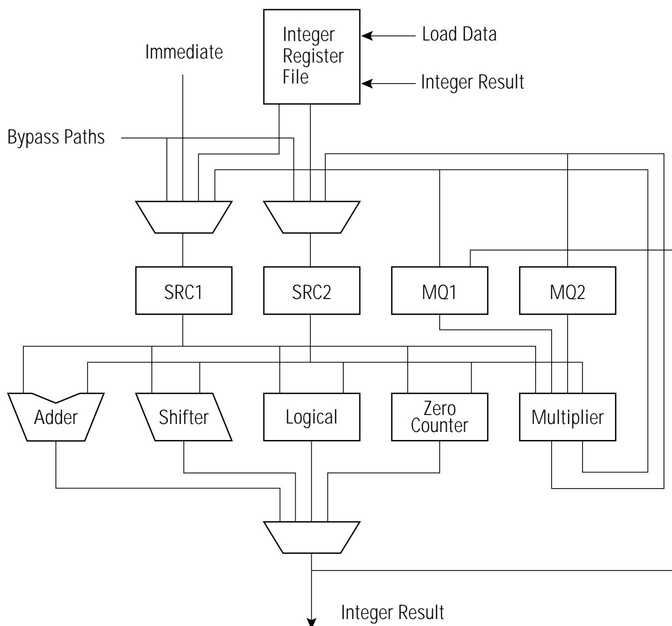


Figure 3: Integer Execution Unit Simplified Block Diagram

1.2.5 Load/Store Unit

In conjunction with the level 2 cache and bus interface unit, the load/store unit executes load, store, and cache operation instructions. This unit consists of an adder that produces the effective address from the address operands, the TLB, the data cache, a rotator that handles misaligned data and performs byte-reversal, and a store queue. Figure 4 on page 13 shows a simplified block diagram of the load/store unit. A number of SPRs, including the DAR, DSISR, SDR1, SPRG, and DBATs, and the segment registers, reside in the load/store unit.

The load/store unit reads the results of load instructions from the data cache. It then immediately bypasses the results to any execution unit that may need them as operands for other instructions, even to fixed-point instructions that are issued in the same cycle as the load. Because of this intra-cycle bypassing, the load-use penalty for ALU operations on the X^{704} is effectively zero cycles. Store instructions do not usually delay the execution pipeline. Store data is placed in the store queue where it waits for a free cycle when it can be written to the level 2 cache and possibly to the data cache.

Data in the store queue can be written to both the data cache and the level 2 cache. Store queue entries also hold data sent from the level 2 cache in response to a data cache miss. The store queue allows subsequent load instructions to proceed and potentially complete before it writes the store data to either cache. Stores that miss in the data cache do not cause a data cache miss; instead, the data is sent directly to the level 2 cache, where a miss occurs if necessary. This process is known as *store-around*.

The store queue contains four doubleword entries. Cacheable stores that hit existing entries in the store queue combine with the existing entry rather than allocate a new entry. This significantly improves the performance of consecutive stores, particularly those using the store multiple instruction. In most cases, data held in the store queue can be bypassed back into the pipeline when a load instruction hits it; the load need not wait until the store queue data is written into the data cache.

Cache operations are placed in the store queue and sent to the level 2 cache, which actually performs the operations, without holding up the execution pipeline. The ***sync***, ***tlbsync***, and ***eieio*** synchronization instructions also execute in the load/store unit and do not complete until all of the appropriate entries have been removed from the store queue.

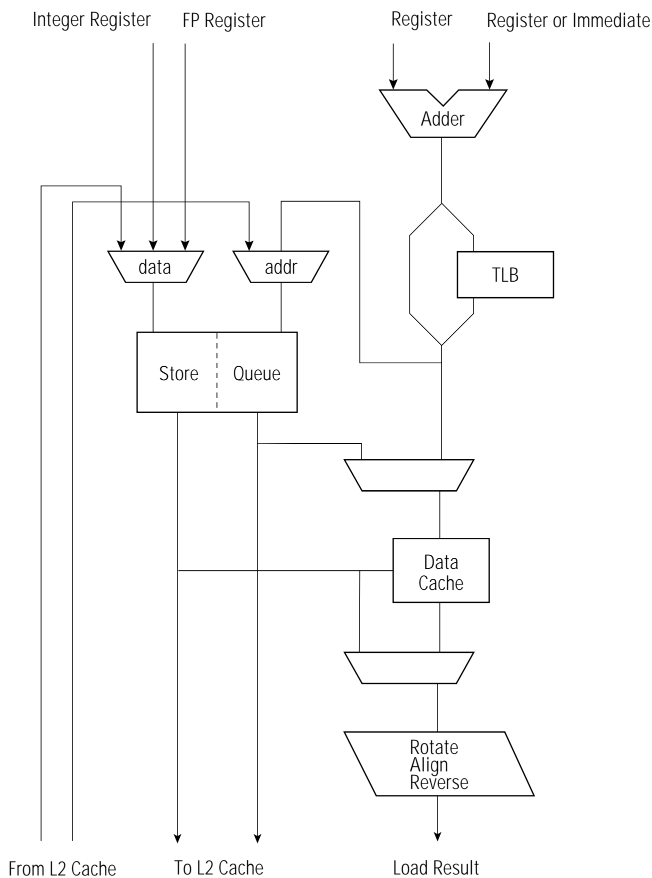


Figure 4: Load/Store Unit Simplified Block Diagram

1.2.6 Floating-Point Execution Unit

The floating-point execution unit contains the floating-point register file, a pipelined adder, a pipelined multiplier, and a divider that all support the IEEE-754 standard for floating-point arithmetic. Figure 5 shows a simplified block diagram of the floating-point execution unit.

The X^{704} processor supports IEEE NaNs and denormalized numbers. See Section 2.1.7.6 on page 24 for more information on denormalized numbers.

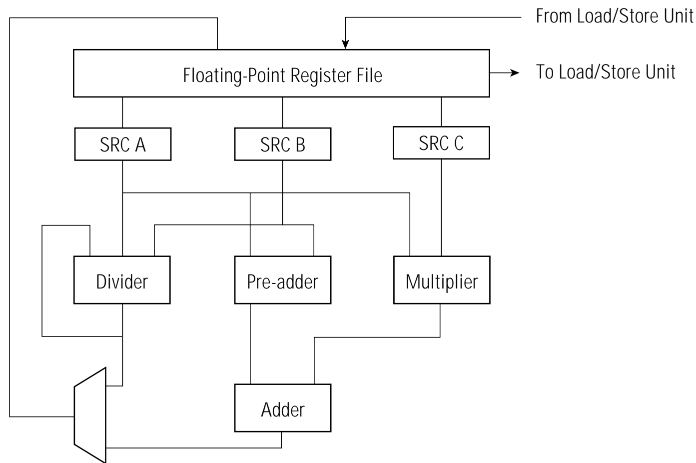


Figure 5: Floating-Point Execution Unit Block Diagram

1.2.7 Level 2 Cache

The level 2 cache moves data between external caches or memory and the faster instruction and data caches. The level 2 cache controller handles multiple misses and processes hits from either level 1 cache while satisfying a miss from one or both of them.

The level 2 cache also executes cache operations such as block touch, block store, and block zero, maintaining both the storage reservation used by the ***stwcx***. instruction and cache coherency in a multiprocessor system using the MESI protocol.

The level 1 caches must be subsets of the level 2 cache; the level 2 cache tags record which lines, if any, are present in either level 1 cache. This allows the level 2 cache to execute most cache operations and process most snoop requests without interfering with the operation of the level 1 caches.

1.2.8 Bus Interface Unit

The bus interface unit passes information between the level 2 cache and the system bus using the basic transfer protocol described in the bus specification. It contains:

- a 32-byte writeback buffer that stages data being evicted from the level 2 cache
- a 16-byte write buffer that holds write data being sent to the system in response to a snoop
- address buffers that hold information for up to three read, write, coherency, or synchronization requests and a single incoming snoop request

In a multiprocessor system, the bus interface unit maintains information on the state of broadcast coherency and synchronization operations.

2. PowerPC Architecture Compliance

This chapter describes implementation-dependent features of the X^{704} and elaborates on general architectural details where necessary.

2.1 X^{704} User Instruction Set Architecture (UISA)

This section follows the structure of Book I of the *PowerPC Architecture Specification*. The reader should be familiar with that book.

2.1.1 Reserved Fields

The PowerPC architecture does not require the implementation of reserved bits in special purpose registers. For reserved bits, do not assume that a read returns the last value written when writing software. When the X^{704} writes zero to a reserved bit, subsequent reads return zero; when the X^{704} writes one to a reserved bit, subsequent reads return an undefined value.

2.1.2 Classes of Instructions

The PowerPC architecture defines three instruction classes:

- Defined
- Illegal
- Reserved

The following sections describe the X^{704}'s implementation.

2.1.2.1 Defined Instruction Class

The X^{704} supports all required instructions defined for 32-bit implementations of the PowerPC architecture. It also supports the optional ***fsel***, ***stfiwx***, and ***tlbie*** instructions.

The X^{704} does not support the optional ***fres***, ***frsqrte***, ***fsqrt***, ***fsqrts***, ***tlbia***, ***eciwx***, and ***ecowx*** instructions. Attempts to execute these instructions cause the system illegal instruction error handler to be invoked.

Book I of the architecture defines certain instructions to have preferred forms. The X^{704} does not distinguish between preferred forms and other forms of valid defined instructions.

The architecture allows invalid forms of defined instructions to cause boundedly undefined results. The operation of the X^{704} on invalid forms of instructions is described for each of the functional units in Section 2.1.4.3 on page 19, Section 2.1.5.2 on page 21, Section 2.1.6.1 on page 22, and Section 2.1.7.8 on page 25, respectively.

2.1.2.2 Illegal Instruction Class

Attempts to execute instructions in this class cause the system illegal instruction error handler to be invoked. PowerPC instructions defined only for 64-bit implementations are treated as illegal instructions.

2.1.2.3 Reserved Instruction Class

The reserved instruction class comprise the following four subclasses:

- the instruction having primary opcode zero except for the instruction consisting entirely of zeros
- POWER instructions that were not included in the PowerPC architecture
- implementation-dependent instructions required to conform to the architecture specification
- other implementation-dependent instructions

The X^{704} invokes the system illegal instruction error handler on attempts to execute instructions with primary opcode zero or POWER instructions that are not included in the PowerPC architecture. See Section G.27 of Book I for POWER instructions not implemented in the PowerPC architecture.

There are no implementation-dependent instructions required to conform to the *PowerPC Architecture Specification*.

The X^{704} supports two implementation-dependent instructions, ***lwdx*** and ***stwdx***, that provide diagnostic access to the on-chip caches and TLB. See Section 2.2.4 on page 30.

The X^{704} also supports several privileged special purpose registers (SPRs) that are not defined in the architecture specification. The ***mfspr*** and ***mtspr*** instructions provide access to these registers. These implementation-dependent SPRs are described in Section 2.3.4.3 on page 35.

2.1.3 Exceptions

The X^{704} supports the standard PowerPC exceptions, including single-step and branch tracing; it also supports two additional exceptions not described in the architecture specification: TLB miss and TLB store. These exceptions handle software reloading and updating of the translation lookaside buffer (TLB). See Section 2.3.6.2 on page 56 for additional information on these interrupts.

2.1.4 Branch Processor

The following sections describe the X^{704} UISA for the branch processor.

2.1.4.1 Instruction Fetching

The X^{704} prefetches instructions before it determines whether they will actually execute. Instructions are never fetched from guarded storage unless they are either in the cache, known to be on the branch path, or are on the same page as an instruction known to be on the branch path. The X^{704} requires software cache operations when a program modifies an instruction it intends to execute. Software must execute the instruction sequence described in Section 2.2.6 on page 32 to ensure that the modified instruction is visible to the fetch unit.

The instruction fetch unit includes an Instruction Address Breakpoint Register (IABR) that triggers a trace interrupt when an instruction is fetched from a specified address. See Section 2.3.4.5 on page 41 for more information on IABR and its associated interrupt.

2.1.4.2 Branch Prediction

The X^{704} improves its performance by predicting branch directions and target addresses using the algorithms described in Section 3.8 on page 81. The X^{704} ignores the *y* bit in the BO field of branch conditional instructions.

2.1.4.3 Invalid Branch Instruction Forms

Attempts to execute invalid forms of the ***bcctr*** instruction where BO(2) is clear will cause the system illegal instruction error handler to be invoked.

Execution of invalid forms of the ***mcrf*** and condition-register logical instructions with the Rc bit set may cause CR0 to be set to an undefined value.

2.1.5 Fixed-Point Processor

The following sections describe the X^{704} implementation of the fixed-point processor.

2.1.5.1 Load/Store Unit

Book I of the architecture specification notes that the load algebraic, load with byte reversal, and load with update instructions may have greater latency than other load instructions. On the X^{704}, load algebraic instructions (***lha***, ***lhax***, ***lhau***, and ***lhaux***) require an additional cycle before the result can be used by another instruction, but can still issue at the rate of one per cycle. Load with byte reversal and load with update instructions, however, incur no additional latency penalty, although update instructions do prevent the simultaneous issue of an integer instruction. See Chapter 4 for more information on instruction latency and performance.

When operating in big-endian mode, the load/store unit supports arbitrary alignment of halfword, word, floating-point single, and floating-point double scalar values. In power-endian mode, the load/store unit supports misaligned word and halfword loads and stores that do not cross a doubleword boundary. Some alignments may incur additional cycles of execution time as described in Section 4.8 on page 99. Power-endian mode elementary loads and stores that cross a doubleword boundary, and any ***lwarx*** or ***stwcx.*** instruction with a misaligned target address causes the system alignment error handler to be invoked.

An unaligned access that does not cause the system alignment error handler to be invoked may cross a page boundary. If this happens, the TLB miss, TLB store fault, or system data storage error handlers can be invoked with the instruction partially completed, but the RT register will not have been altered for elementary fixed-point load instructions. Aligned move assist (***lswi***, ***lswx***, ***stswi***, and ***stswx***), ***lmw***, and ***stmw*** instructions that cross page boundaries can also cause these handlers to be invoked with the instruction partially completed.

The X^{704} does not support direct-store segments or accesses to direct-store segments. All attempts to reference data in a direct-store segment cause either the system data storage error handler or the system alignment error handler to be invoked.

2.1.5.2 Invalid Load/Store Instruction Forms

Attempts to execute load with update instructions with RA = RT cause the system illegal instruction error handler to be invoked.

The execution of load with update or store with update instructions with RA = 0 performs the storage access with an effective address of the contents of RB (for X-form instructions) or of the displacement (for D-form instructions). If the access is successful, r0 is set to the effective address.

Execution of the ***lmw***, ***lswi***, and ***lswx*** instructions with RA or RB in the range of target registers, including the RA = 0 case, functions correctly, but the instructions cannot be restarted reliably if interrupted because the address value in RA or RB may have been overwritten.

Execution of an ***lswx*** instruction specifying a zero-byte transfer causes the contents of RT to become undefined.

Execution of invalid forms of load/store instructions with the Rc bit set does not alter CR0.

Execution of the ***stwcx.*** instruction with the Rc bit clear sets CR0 as if the Rc bit were set.

2.1.5.3 Reservation Granularity

The storage reservation granularity established by the ***lwarx*** instruction is 32 bytes—the same size as a cache block.

2.1.5.4 Synchronization Instruction

This instruction causes significant performance penalties and should not be used indiscriminately. See Section 2.2.5 on page 32 for a detailed description of the ***sync*** instruction.

2.1.5.5 Data Breakpoints

The load store unit includes a Data Address Breakpoint Register (DABR) and a Breakpoint Control register (BPTCTL) that cause a data storage interrupt to occur when a specified address or address range is referenced. See Section 2.3.4.5 on page 41 for more information on BPTCTL, DABR, and breakpoint interrupts.

2.1.6 Fixed-Point Unit

Book I of the architecture specification notes that instructions with the OE bit set or those that are defined to set CA can execute slowly or prevent the execution of subsequent instructions until the operation is complete. With the X^{704} processor, instructions that set CA never cause performance penalties. The performance of multiply instructions, however, is affected by setting the OE bit.

On the X^{704}, the ***mullwo*** and ***mullwo.*** instructions always require six cycles, as opposed to the three to five cycles required by other fixed-point multiply instructions. The only other performance penalty that may occur for fixed-point instructions occurs when recovering from mispredicted branches based on the value of the SO bit in CR0. In this case, a penalty is incurred only when the branch issues while the instruction with OE set is still in the pipeline.

The performance of the ***mtcrf*** instruction does not depend on the value of the FXM field in the instruction.

Execution of ***mtspr*** and ***mfspr*** instructions with undefined values in the SPR field triggers either the system privileged instruction handler (if SPR(0) is set) or the system illegal instruction interrupt handler (if SPR(0) is clear). The X^{704} defines additional SPR values beyond those defined in the *PowerPC Architecture Specification*.

The X^{704} does not support the optional EAR special purpose register. Attempts to reference this register cause the system illegal instruction error handler to be invoked.

2.1.6.1 Invalid Fixed-Point Instruction Forms

The X^{704} processor ignores the Rc bit value in compare, trap, ***mtspr***, ***mfspr***, ***mcrxr***, and ***mfcr*** instructions. Execution of those instructions with the Rc bit set will not cause CR0 to be set to an undefined value.

Execution of the ***mtcrf*** instruction with the Rc bit set may cause CR0 to be set to an undefined value.

Execution of compare instructions with Rc set and with BF not equal to zero will set CR field BF correctly.

Execution of compare instructions is unaffected by the value of either the L bit or bit 9 of the instruction.

Execution of instructions such as ***neg*** that do not use the RB field is unaffected by the contents of that field.

2.1.7 Floating-Point Unit

The following sections describe the X^{704} implementation of the floating-point unit.

2.1.7.1 Conformance with IEEE Standard

The X^{704} floating-point unit complies with the IEEE-754 floating-point standard while the NI bit in the FPSCR is clear. When FPSCR[NI] is set, the X^{704} deviates from the standard by replacing denormalized results of floating-point computational instructions with zeros. The check for a denormalized result is made before rounding, so a result that would have been rounded from the largest denormalized number to the smallest normalized number is still forced to zero.

Setting the NI bit does not alter either the definitions of other FPSCR fields or the behavior of floating-point exceptions, including underflow and inexact traps resulting from denormalized results that are forced to zero. Applications that want to suppress all floating-point exceptions should clear all five exception enable bits in the FPSCR.

2.1.7.2 Floating-Point Load/Store Operations

The floating-point register file stores all operands in double-precision format. Single-precision loads and stores perform the appropriate conversions to and from the single-precision memory format. Conversions of single-precision denormalized values on load instructions cause a performance penalty. See Section 4.9 on page 101 for more information on floating-point execution.

2.1.7.3 Floating-Point Arithmetic Instructions

The architecture specification requires operands to single-precision floating-point arithmetic instructions to be representable in single-precision format. If they are not, the results of single-precision arithmetic instructions are undefined. On the X^{704}, the results are undefined only when one or more operands are not representable in single-precision format and the result is also not representable in single-precision format. The undefined result may not be representable in single-precision format and therefore may not be a valid input for subsequent single-precision computational or store instructions.

2.1.7.4 Floating-Point Status and Control Register Instructions

The performance of the ***mtfsf*** instruction does not depend on the value of the FLM field in the instruction.

2.1.7.5 Optional Instructions

The X^{704} implements the optional ***fsel*** and ***stfiwx*** instructions, but not the optional ***fres***, ***frsqrte***, ***fsqrt***, and ***fsqrts*** instructions. The hardware never sets FPSCR[VXSQRT] except when one of the floating-point status and control instructions sets that bit explicitly.

2.1.7.6 Denormalized Numbers

The X^{704} provides complete support for denormalized values in both single- and double-precision formats. When the processor is in non-IEEE mode (FPSCR[NI] is set), denormalized results of floating-point computational instructions, but not floating-point load instructions, are forced to zero. Conversion of single-precision denormalized values on load instructions causes a performance penalty. See Section 4.9 on page 101 for more on floating-point execution.

2.1.7.7 Floating-Point Exceptions

All floating-point exceptions on the X^{704} are reported as precise exceptions. The X^{704} does not use the imprecise recoverable and imprecise non-recoverable exception modes. When a floating-point exception occurs while the processor is not in floating-point interrupts disabled mode, SRR0 always points to the instruction that caused the exception, all instructions prior to that instruction have completed, and no instructions following that instruction have caused any architecturally visible effects.

Enabling inexact, overflow, and underflow exceptions degrades performance more than enabling zero divide and invalid operation exceptions. Enabling inexact, overflow, and underflow exceptions does not cause the floating-point operations to take longer, but it does prevent the fixed-point and branch processors from completing instructions and issuing additional instructions for an extended period of time. See Section 4.9 on page 101 for more information.

The X^{704} processor does not use the floating-point assist interrupt.

2.1.7.8 Invalid Floating-Point Instruction Forms

The execution of floating-point load with update or floating-point store with update instructions with RA = 0 performs the storage access with an effective address of the contents of RB (for X-form instructions) or of the displacement (for D-form instructions). If the access is successful, R0 is set to the effective address.

Execution of floating-point load and store instructions with the Rc bit set does not alter CR1.

The X^{704} processor ignores the value of the Rc bit in ***fcmpo*** and ***fcmpu***. Execution of those instructions with the Rc bit set does not cause CR1 to be set to an undefined value.

Execution of floating-point compare instructions with Rc set and with BF not equal to one will set CR field BF correctly.

2.2 X^{704} Virtual Environment Architecture (VEA)

This section follows the structure of Book II of the *PowerPC Architecture Specification*. The reader should be familiar with that book.

2.2.1 Storage Model

The following sections describe the implementation of the storage model for the X^{704} processor.

2.2.1.1 Caches

The X^{704} contains three on-chip caches: level 1 data and instruction caches, and a unified level 2 cache. In this document, level 1 is used only when referring to the instruction and data caches as a group; otherwise, those caches are known simply as the instruction cache and the data cache.

All three caches are made up of 32-byte blocks, are physically addressed, and have physical tags. Cache validity is maintained on a doubleword basis in the level 1 caches. A level 2 cache miss requests all 32 bytes from off chip. This doubleword validity scheme allows partially satisfied level 1 cache misses to be abandoned when a higher-priority miss occurs. For example, if the instruction stream executes a branch from the middle of a cache block, there is no need to supply the remainder of that block to the instruction cache. Instead, the level 2 cache may immediately begin supplying data from the target of the branch if that data is not already present in the instruction cache.

The instruction and data caches are 2KB direct-mapped caches. The level 2 cache is a 32KB, eight-way set-associative cache. The level 2 cache always includes any block that is present in either level 1 cache; this is known as the *inclusion* property. The level 2 cache uses a modified pseudo-LRU algorithm to manage the blocks in an associativity set: a block marked as present in either the data cache or the instruction cache is never considered to be the least-recently-used block and is not replaced in the level 2 cache.

The X^{704} processor disables the caches when it is reset and must be individually enabled by setting the appropriate bits in the L2CTL register. See Section 2.3.4.6.3 on page 50 and Section 3.10 on page 85.

Level 1 cache blocks can be either valid or invalid. Level 2 cache blocks are each in one of the four MESI cache line states: invalid (I), exclusive clean (E), shared clean (S), and exclusive modified (M).

The data cache is a write through cache. The level 2 cache uses the write through required (W) storage control attribute to determine whether each individual block is write through or copy back. Blocks are treated as copy back unless the W bit is set.

Most memory references are either instruction fetches, data loads, or data stores. When all caches are enabled, those operations have the following effects:

- Instruction fetches read the target storage block into the level 2 cache if it is not already present there, and into the instruction cache if it is not already in that cache.
- Data loads read the target storage block into the level 2 cache if it is not already present there, and into the data cache if it is not already in that cache.
- Data stores read the target storage block into the level 2 cache if it is not already present there. If the block is present in the data cache, the modified data is written to the data cache; if the block is not present, it is not brought into the data cache. The X^{704} always writes modified data into the level 2 cache, but never to the instruction cache, even if the target block is present there.

All other memory references are performed either with cache management instructions (described in Section 2.2.3 on page 27) or by other processors referencing coherent storage (described in Section 3.4.6 on page 77).

2.2.1.2 Storage Consistency

In order to maintain sequential consistency for memory operations executed within a single processor, the X^{704} load/store unit wraps data from the store queue when a load hits a recent store. Store data is not placed in the cache or sent off chip until any possible exceptions caused by the store instruction or any instruction issued before the store instruction have occurred.

2.2.2 Effect of Operand Placement on Performance

The alignment of operands in memory affects the performance of load and store instructions. In big-endian mode, the X^{704} handles misaligned accesses with minimal performance degradation, and then only when the accesses cross a doubleword boundary. In power-endian mode, misaligned accesses that cross a doubleword boundary always invoke the system alignment error handler, resulting in poor performance.

2.2.3 Cache Management Instructions

The X^{704} implements all cache management instructions described in Book II.

Execution of all of these instructions except ***isync*** can update the LRU state of the TLB and level 2 cache. Management of the PTE Reference and Change bits is left to the software interrupt handlers, but the handlers should not be expected to distinguish accesses on behalf of cache management instructions from other storage accesses.

2.2.3.1 Instruction Cache Block Invalidate (***icbi***)

Execution of the ***icbi*** instruction invokes the TLB miss handler if data address translation is enabled and no translation for the effective address is found in the TLB or DBAT. If a translation is found in the TLB, but read permission is not allowed, the system data storage error handler is invoked. If data address translation is disabled, or a translation is found and read access is allowed, the addressed block is removed from the instruction cache if it is present there. If the addressed storage is in coherence required mode, the operation is then broadcast on the bus to allow the line to be invalidated in the instruction caches of other processors.

The ***icbi*** instruction never invalidates a block in the level 2 cache.

The effect of this instruction is the same if the instruction cache is disabled.

2.2.3.2 Instruction Synchronize (***isync***)

Execution of the ***isync*** instruction flushes any subsequent instructions from the pipeline and causes the fetch unit to invalidate the contents of the instruction buffer and to re-fetch the instruction following the ***isync*** in the current context. All previously issued instructions complete.

This instruction must be used when changing the processor's endian mode (see Section 2.3.3.2 on page 34).

2.2.3.3 Data Cache Block Touch (***dcbt***)

If data address translation is disabled, or if a translation for the effective address is found, read access is allowed and the addressed storage is not in caching inhibited mode, the X^{704} may read the addressed block into the level 2 cache. If any of these conditions are not met, if the level 2 cache is disabled, or if processor resources are busy on higher-priority memory operations, the ***dcbt*** instruction is treated as a ***nop***.

Because of resource limitations, the X^{704} generally performs a read for only the last in a sequence of touch operations. Data references caused by the ***dcbt*** instruction are treated as prefetches. See Section 3.4.7 on page 78 for more information on prefetching.

2.2.3.4 Data Cache Block Touch for Store (***dcbtst***)

The ***dcbtst*** instruction is treated as a ***nop*** when:

- the level 2 cache is disabled
- processor resources are busy on higher-priority memory operations
- data address translation is enabled and no translation for the effective address is found
- read access is not allowed
- the addressed storage is in caching inhibited mode

If none of these conditions are true, and the addressed storage is marked as memory coherence required, the addressed block is read into the level 2 cache with a read with intent to modify bus operation and marked as modified in the cache. If the addressed storage is marked as memory coherence not required, the block is read into the cache with a simple read operation and placed in the exclusive state.

This instruction should be used only when there is a high probability that the target cache block will be modified before it is evicted from the cache. If the line is very likely to be read, but less likely to be modified, the ***dcbt*** instruc-

tion should be used instead. Because of resource limitations, the X^{704} generally performs a read for only the last in a sequence of touch operations. Data references caused by the ***dcbtst*** instruction are treated as prefetches. See Section 3.4.7 on page 78 for more information on prefetching.

2.2.3.5 Data Cache Block Zero (*dcbz*)

Execution of the ***dcbz*** instruction invokes the TLB miss handler if data address translation is enabled and no translation for the effective address is found in the TLB or DBAT. It also invokes the TLB store handler if a matching TLB entry is found with the C bit clear, and invokes the system data error handler if a translation is found that does not allow write permission.

If the addressed storage is marked as memory coherence required and not caching inhibited, ***dcbz*** broadcasts an invalidate request that removes the line from the caches of any other processors.

If the addressed storage is caching allowed, ***dcbz*** zeroes the line in the level 2 cache, allocating a cache block if the line is not already present. No read request will be issued on the bus. If the addressed storage is present in the data cache, ***dcbz*** invalidates the cache block containing that storage.

If the level 2 cache is disabled, or if the storage is marked as either caching inhibited or write through required, the ***dcbz*** instruction sets each byte of the addressed block in off-chip memory to zero. The *PowerPC Architecture Specification* invokes the system alignment error handler in these cases, but the X^{704} implementation does not.

2.2.3.6 Data Cache Block Store (*dcbst*)

Execution of the ***dcbst*** instruction invokes the TLB miss handler if data address translation is enabled and no translation for the effective address is found in the TLB or DBAT. If a translation is found, but read access is not allowed, the system data storage error handler is invoked.

If data address translation is disabled, or a translation is found and the addressed block is marked as modified in the level 2 cache, the contents of the block are written back to off-chip memory, and the state of the block is changed to exclusive clean. If the addressed storage is marked as memory coherence required, the clean operation is broadcast on the bus.

The operation of this instruction is independent of the state of the cache enables. If the block is not present and modified in any processor's level 2 cache, the instruction is treated as a ***nop***.

2.2.3.7 Data Cache Block Flush (***dcbf***)

Execution of the ***dcbf*** instruction invokes the TLB miss handler if data address translation is enabled and no translation for the effective address is found in the TLB or DBAT. If a translation is found, but read access is not allowed, the system data error handler is invoked.

If data address translation is disabled or a translation is found, and the addressed block is marked as valid in the level 2 cache, the block is invalidated in the data cache, the instruction cache, and the level 2 cache. The data cache block addressed by bits (21:26) of the effective address is invalidated regardless of whether the addressed storage is present in the cache. If the addressed block is marked as modified in the level 2 cache, the contents of the block is written back to main memory. If the addressed storage is marked as coherence required, the flush operation is broadcast on the bus.

The operation of this instruction is independent of the state of the cache enables. If the block is not present in any processor's level 2 cache, the instruction is treated as a ***nop***.

2.2.4 Additional Diagnostic Instructions

The X^{704} implements two diagnostic instructions allowing direct access to the TLB, cache data, cache tags, and other internal processor structures. These instructions use an alternate address space to reference the structures. See Section 3.9 on page 84 for a description of the diagnostic address space.

In the following instruction descriptions, *DIAG(X, Y)* refers to the contents of *Y* bytes of diagnostic memory at address *X* in the diagnostic address space.

Load Word Diagnostic Indexed **X-Form**

```
lwdx                          RT, RA, RB
```

31	RT	RA	RB	0x21E	/
0 5	6 10	11 15	16 20	21 30	31

```
if RA = 0 then                b ← 0
else                          b ← (RA)
a ← b + (RB)
RT ← DIAG (a, 4)
```

Let the diagnostic address (*a*) be the sum (RA | 0) + (RB). The word in diagnostic memory addressed by *a* is loaded into RT.

This instruction is privileged, and defined only when the DE bit in the MODES register is set. An attempt to execute this instruction with MODES[DE] clear or MSR[PR] set will cause the system illegal instruction error handler to be invoked.

If the instruction references an undefined diagnostic address, the system data storage interrupt handler may be invoked (see Section 3.9 on page 84). The low two bits of the effective address must be zero or the results of executing the instruction are boundedly undefined; this instruction never causes an alignment interrupt.

Special Registers Altered: None

Store Word Diagnostic Indexed **X-Form**

```
stwdx                          RS, RA, RB
```

31	RS	RA	RB	0x29E	/
0 5	6 10	11 15	16 20	21 30	31

```
if RA = 0 then                 b ← 0
else                           b ← (RA)
a ← b + (RB)
DIAG (a, 4) ← (RS)
```

Let the diagnostic address (*a*) be the sum (RA | 0) + (RB). (RS) is stored into the word in diagnostic memory addressed by *a*.

This instruction is privileged, and defined only when the DE bit in the MODES register is set. An attempt to execute this instruction with MODES[DE] clear or MSR[PR] set will cause the system illegal instruction error handler to be invoked.

If the instruction references an undefined diagnostic address, the system data storage interrupt handler may be invoked (see Section 3.9 on page 84). The low two bits of the effective address must be zero or the results of executing the instruction are boundedly undefined; this instruction never causes an alignment interrupt.

Special Registers Altered: None

2.2.5 Storage Access Ordering

The X^{704} implements a weakly consistent storage model. The order in which stores are visible outside the processor may not be the same order in which the stores are performed. For example, multiple stores to the same doubleword can be merged and made visible as a single store operation. If a particular ordering is required, the ***eieio*** and ***sync*** instructions can be used to place barriers in the storage access stream.

The ***eieio*** instruction does not complete until all stores have been removed from the store queue and the level 2 cache has reported that all previous ***tlbie*** and ***tlbsync*** operations have been broadcast on the bus. Subsequent load and store instructions are delayed until after the ***eieio*** completes. The architecture specification defines two sets of operations that are ordered separately by ***eieio***, but the X^{704} orders all applicable operations as a single set. If synchronization of storage and TLB accesses is all that is required, the ***eieio*** instruction is preferable to the ***sync*** instruction.

The ***sync*** instruction does not complete until the store queue is empty and the level 2 cache reports that no operations are still in progress.

2.2.6 Executing Modified Code

When a program modifies an instruction stream that it wants to execute, cache management instructions must be used to ensure that all updates are visible to the instruction fetch unit. Without the use of these instructions, the X^{704} does not guarantee coherency between the instruction cache and either the data cache, level 2 cache, or off-chip memory.

After modifying instructions in the block of data addressed by general register RX, the program should execute the following instruction sequence:

```
   dcbst    RX                  ! update cache block in main memory
   sync                         ! wait for update to complete
   icbi     RX                  ! invalidate block in icache
   sync                         ! wait for invalidate to complete
   isync                        ! make sure instructions are re-
fetched
```

Because it appears before the ***icbi*** instruction, the first ***sync*** instruction ensures that the fetch unit reads instructions from the modified block after the updates are visible. The second ***sync*** instruction is necessary only on multiprocessor systems where the block must be flushed from the instruction caches of all processors before execution continues.

2.2.7 Atomic Update Primitives

The ***lwarx*** and ***stwcx.*** instructions function correctly regardless of the write through required attribute for the addressed storage.

The ***stwcx.*** instruction performs a store even if its storage address is not identical to the storage address used by the most recent ***lwarx*** instruction.

There are no causes of reservation loss other than those listed in Book II. On the X^{704}, the reservation is lost when another processor executes a ***dcbtst*** or ***dcbi*** instruction to the reservation granule, but is not lost when another processor executes a ***dcbf*** or ***dcbst***.

2.2.8 Timer Facilities

The X^{704} maintains the 64-bit time base register in two parts: the lower 32 bits in the TBL register and the upper 32 bits in the TBU register. These registers can be read separately with the ***mftb*** and ***mftbu*** instructions. The X^{704} is a 32-bit implementation of the PowerPC architecture and thus does not provide a way to read all 64 bits of the time base in one instruction.

The X^{704} increments the time base register every four system bus clock cycles as long as MODES[TBD] is clear. The X^{704} also implements the 32-bit decrementer (DEC) register. This register counts down every four system bus clock cycles.

2.3 X^{704} Operating Environment Architecture (OEA)

This section follows the structure of Book III of the *PowerPC Architecture Specification*. The reader should be familiar with that book.

2.3.1 Reserved Fields in Storage Tables

The X^{704} hardware does not automatically access the hashed page table and thus does not alter any reserved fields.

2.3.2 Exceptions

The X^{704} defines two additional interrupts: TLB miss and TLB store. These interrupts manage the software refill and update of the TLB. They are described in detail in Section 2.3.6.2 on page 56.

2.3.3 Branch Processor

The following sections describe the X^{704} implementation of the OEA specification for the branch processor.

2.3.3.1 SRR0 and SRR1

Any instruction fetch when MSR[IR] is set can set SRR0 to the address of the instruction being fetched, and can set SRR1 as described in Section 2.3.6.2.13 on page 61.

The execution of any instruction requiring an address translation when MSR[DR] is set can set SRR0 to the address of the instruction being executed, and can set SRR1 as described in Section 2.3.6.2.13 on page 61 or Section 2.3.6.2.14 on page 62.

2.3.3.2 MSR

The X^{704} implements the MSR as described in the *PowerPC Architecture Specification*, including the tracing functions supported by the Branch Trace Enable (BE) and Single-Step Trace Enable (SE) bits. When either of these trace enable bits is set, the processor disables superscalar instruction issue.

Caution: Use of the tracing facilities causes significant performance loss.

The X^{704} uses MSR(14), formerly called the Implementation-Dependent Function bit and known as MSR[TW] on the X^{704}, to prevent external TLB invalidates from interfering with a software page table walk. TLB miss and TLB store interrupts set MSR[TW], and the ***rfi*** instruction clears it. When the bit is set, the X^{704} defers processing of TLB invalidates received from other processors or devices.

The X^{704} does not use the Power Management Enable (POW) bit. This bit is treated as a reserved full-function bit. Future implementations that support power management features may make use of this bit.

In order to ensure that instructions are fetched using the correct address for the current endian format, care must be taken when altering the MSR[LE] bit with an ***mtmsr*** or ***rfi*** instruction. When using an ***mtmsr*** instruction, the change in endian-ness is not guaranteed to take effect until after a subsequent ***isync*** instruction. The following code sequence should be used to change endian modes with an ***mtmsr*** instruction:

```
.align 8
mtmsr    RX
isync
```

When altering MSR[LE] with an ***rfi*** instruction using an effective address in SRR0 that refers to the doubleword containing the ***rfi*** instruction, the results are boundedly undefined.

2.3.4 Fixed-Point Processor

The following sections describe the X^{704} implementation of the OEA specification for the fixed-point processor.

2.3.4.1 Software Use SPRs

The X^{704} provides eight software use SPRs rather than the four specified in Book III. SPRG0–SPRG7 are addressed by SPR values 272 through 279. The additional SPRG registers can be used by implementation-dependent interrupt handlers.

2.3.4.2 Processor Version Register

The **Version** field of the PVR contains 0x60 for the X^{704} processor. The **Revision** field is divided into two bytes: bits (16:23) contain a major version number and bits (24:31) contain a minor version number. The **Revision** field is incremented each time the processor is revised. Table 1 shows the values of the **Revision** field for all versions of the X^{704}.

Table 1: Processor Revision Values

Processor Release	PVR.Revision
prototype	0x0100
initial production	0x0101

Important Note: The **Version** field for the prototype version of the X^{704} processor was 0x54. This value will not be used for any other X^{704} versions.

2.3.4.3 Additional Special Purpose Registers

The X^{704} contains several implementation-dependent special purpose registers that can be accessed with the ***mfspr*** and ***mtspr*** instructions. Accesses to all of these registers are privileged. Table 2 shows all of the SPRs implemented on the X^{704}, with the implementation-dependent entries shown in shaded rows.

Table 2: Special Purpose Registers

SPR Number			Register Name	Privileged	Unit	Defined in Book/Page
Dec.	$spr_{5:9}$	$spr_{0:4}$				
1	00000	00001	XER	no	Branch	Book I
8	00000	01000	LR	no	Branch	Book I
9	00000	01001	CTR	no	Branch	Book I
18	00000	10010	DSISR	yes	Load/Store	Book III
19	00000	10011	DAR	yes	Load/Store	Book III
22	00000	10110	DEC	yes	Branch	Book III
25	00000	11001	SDR1	yes	Load/Store	Book III
26	00000	11010	SRR0	yes	Branch	Book III
27	00000	11011	SRR1	yes	Branch	Book III
272	01000	10000	SPRG0	yes	Load/Store	Book III
273	01000	10001	SPRG1	yes	Load/Store	Book III
274	01000	10010	SPRG2	yes	Load/Store	Book III
275	01000	10011	SPRG3	yes	Load/Store	Book III
276	01000	10100	SPRG4	yes	Load/Store	Book III
277	01000	10101	SPRG5	yes	Load/Store	Book III
278	01000	10110	SPRG6	yes	Load/Store	Book III
279	01000	10111	SPRG7	yes	Load/Store	Book III
284	01000	11100	TBL	yes	Branch	Book II
285	01000	11101	TBU	yes	Branch	Book II
287	01000	11111	PVR[1]	yes	Load/Store	Book III
528	10000	10000	IBAT0U	yes	Fetch	Book III
529	10000	10001	IBAT0L	yes	Fetch	Book III
530	10000	10010	IBAT1U	yes	Fetch	Book III
531	10000	10011	IBAT1L	yes	Fetch	Book III
532	10000	10100	IBAT2U	yes	Fetch	Book III
533	10000	10101	IBAT2L	yes	Fetch	Book III
534	10000	10110	IBAT3U	yes	Fetch	Book III
535	10000	10111	IBAT3L	yes	Fetch	Book III
536	10000	11000	DBAT0U	yes	Load/Store	Book III
537	10000	11001	DBAT0L	yes	Load/Store	Book III
538	10000	11010	DBAT1U	yes	Load/Store	Book III
539	10000	11011	DBAT1L	yes	Load/Store	Book III
540	10000	11100	DBAT2U	yes	Load/Store	Book III
541	10000	11101	DBAT2L	yes	Load/Store	Book III
542	10000	11110	DBAT3U	yes	Load/Store	Book III

Table 2: Special Purpose Registers (Cont.)

SPR Number		Register Name	Privileged	Unit	Defined in Book/Page
543	10000 11111	DBAT3L	yes	Load/Store	Book III
953	11101 11001	EVENT	yes	Branch	page 45
954	11101 11010	MODES	yes	Branch	page 47
977	11110 10001	CMP[1]	yes	Load/Store	page 39
978	11110 10010	HASH1[1]	yes	Load/Store	page 40
979	11110 10011	HASH2[1]	yes	Load/Store	page 40
982	11110 10110	TLBLRU0[2]	yes	Load/Store	page 40
983	11110 10111	TLBMRF	yes	Load/Store	page 41
985	11110 11001	BPTCTL	yes	Load/Store	page 42
987	11110 11011	TLBLRU1[2]	yes	Load/Store	page 40
988	11110 11100	MISR	yes	Load/Store	page 39
989	11110 11101	MAR	yes	Load/Store	page 38
1008	11111 10000	CHECK	yes	Load/Store	page 48
1010	11111 10010	IABR	yes	Fetch	page 44
1012	11111 10100	L2CDR	yes	L2 Cache	page 52
1013	11111 10101	DABR	yes	Load/Store	Book III
1014	11111 10110	XDABR	yes	Load/Store	page 44
1019	11111 11011	L2CTL	yes	L2 Cache	page 50
1023	11111 11111	PIR	yes	Load/Store	page 52

1. read-only
2. write-only

These registers fall into five major categories:

- hardware aids for TLB miss handlers (MAR, MISR, CMP, HASH1, HASH2, TLBLRU0, TLBLRU1, and TLBMRF registers)
- scratch registers for TLB miss, TLB store, and instruction emulation handlers (SPRG4–SPRG7)
- debugging (BPTCTL, IABR, DABR, XDABR, and EVENT)
- various processor control functions (CHECK, MODES, L2CTL, and L2CDR)
- multiprocessor applications (PIR)

The following sections describe these registers in detail.

2.3.4.4 TLB Miss Registers

The MAR and MISR registers provide information about the address and type of reference that caused a TLB miss or TLB store interrupt. They are analogous to the DAR and DSISR registers used to hold information about the address and reference that caused a data storage or alignment interrupt.

The HASH1, HASH2, and CMP registers return page table access information designed to assist TLB miss, TLB store, instruction storage, and data storage interrupt handlers. The contents of these registers is based on the current contents of SDR1, MAR, and the segment register referenced by the effective address saved in MAR, all of which have well-defined values after TLB interrupts.

The TLB miss and TLB store handlers write the TLBLRU0, TLBLRU1, and TLBMRF registers to update the contents of the TLB in response to TLB interrupts. Writes to the TLBLRU0 and TLBLRU1 registers use the current contents of MAR to select the target TLB entry. The data written depends on the current contents of MAR and the segment register referenced by the effective address saved in MAR. The contents and location of the TLB entry written using the TLBLRU registers may be changed if MAR or the segment registers are modified. Read and write accesses to the TLBMRF register use the current contents of the MAR and MISR to select a TLB entry. Modifying either of those registers may change which TLB entry is accessed by a reference to the TLBMRF register.

In general, interrupt handlers should not modify SDR1, MAR, MISR, or the segment registers before using the HASH1, HASH2, CMP, TLBLRU0, TLBLRU1, and TLBMRF registers. Sample TLB miss and TLB store handlers making use of these registers are shown in Appendix A.

Outside of TLB-related interrupt handlers, software can alter the values in SDR1, MAR, or the segment registers and subsequently use HASH1, HASH2, and CMP to assist in other page table accesses. These registers should be altered only when both data and instruction relocation are disabled, and programs that update them must follow the synchronization requirements described in Section 2.3.7 on page 64.

2.3.4.4.1 TLB Miss Address Register (MAR)

The MAR register is a 32-bit register used for software TLB management. When a TLB miss or TLB store interrupt occurs, the MAR is loaded with the effective address of the faulting reference. For TLB miss interrupts, this address can be either an instruction address or the effective address of a data reference.

The X^{704} processor uses the contents of this register implicitly in references to the CMP, HASH1, HASH2, TLBLRU0, TLBLRU1, and TLBLMRF registers. MAR is both readable and writable.

2.3.4.4.2 TLB Miss Interrupt Status Register (MISR)

The MISR register contains information about the reference that caused a TLB miss or TLB store interrupt. When a TLB miss or TLB store interrupt occurs, MISR is loaded as described in Section 2.3.6.2.13 on page 61 and Section 2.3.6.2.14 on page 62. The MISR register is compatible with the DSISR register, so that its contents can be copied there when TLB miss interrupts must be passed to the operating system as page fault or page protection data storage interrupts. See the sample TLB interrupt handlers in Appendix A for examples of the use of this register.

2.3.4.4.3 TLB Miss PTE Compare Register (CMP)

The CMP register contains the high word of the PTE search target, made up of the V, VSID, H, and API fields as defined in the *PowerPC Architecture Specification*. TLB miss and TLB store interrupt handlers compare the value in the CMP register with the high word of PTEs in the system page table to locate the PTE that matches the reference that caused the interrupt. Figure 6 depicts the CMP register.

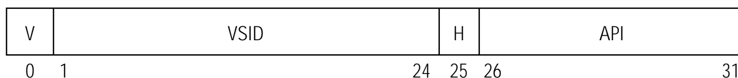


Figure 6: TLB Miss PTE Compare Register

The fields are defined as follows:

- V — is the valid bit. This bit is always returned as one because the miss handler is searching for a valid PTE.
- VSID — is the virtual segment ID copied from the VSID field of the segment register indexed by bits (0:3) of the MAR register.
- H — is the hash function identifier. This bit is always returned as zero because the miss handler begins the search using the primary hash function.
- API — is the abbreviated page index copied from bits (4:9) of the current contents of the MAR register.

An instruction that attempts to write CMP is invalid.

2.3.4.4.4 TLB Miss PTEG Address Hash Registers (HASH1 and HASH2)

The HASH1 register contains the physical address formed by the primary hash for the address currently in MAR. The HASH2 register contains the physical address formed by the secondary hash. TLB miss and TLB store handlers use the contents of these registers to address the two PTEGs that could contain the page table entry for the reference that caused the interrupt. Figure 7 shows the format of these registers.

Figure 7: TLB Miss PTEG Address Hash Registers

The fields are defined as follows:

HTABORG	is bits (0:6) of the HTABORG field of the sdr1 register.
HASH	is the output of the primary or secondary hash function as defined in the PowerPC Architecture Specification. The value in this field depends on the current contents of MAR, SDR1, and the segment register indexed by the address in MAR.

An instruction that attempts to write either of these SPRs is invalid.

2.3.4.4.5 TLB Miss Update LRU Registers (TLBLRU0 and TLBLRU1)

The TLBLRU registers are used by TLB miss handlers to create a TLB entry with a translation for the last address that missed in the TLB. The data written to this register contains the RPN, R, C, WIMG, and PP PTE fields and is formatted as the lower half of a PTE as shown in Figure 8.

Figure 8: TLBLRU Registers

The remaining information needed to build a translation is taken from the CMP register and the segment register indexed by bits (0:3) of MAR. The definitions of the fields in the TLBLRU registers are exactly the same as the definitions of the corresponding fields in the PTE.

In order to create a valid TLB entry (see Figure 22 on page 79), the same data must be written to both TLBLRU0 and TLBLRU1.

When either TLBLRU register is written, the least recently used entry in the TLB set corresponding to the address saved in MAR is updated with the new translation. Software that writes TLB entries should always set the R bit in the corresponding PTE.

An instruction that attempts to read either of these SPRs is invalid.

2.3.4.4.6 TLB Miss Update MRF Register (TLBMRF)

The TLBMRF register is used by the TLB store fault handler to update the C bit in the TLB entry that caused the most recent fault. The data written to this register contains the RPN, C, and PAGEIDX TLB fields and is formatted as the lower half of a TLB entry as shown in Figure 9.

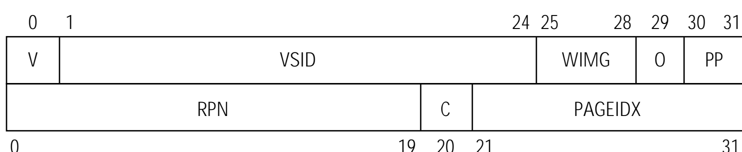


Figure 9: TLB Entry

The TLB entry accessed with this register is the one referenced by MISR(7:8) in the set indexed by the address saved in MAR. Knowledge of the format of this register is not usually necessary: the TLB store interrupt handler normally reads this register, *ORs* in the C bit, and writes it back. The TLB store interrupt handler should also set the C bit in the corresponding PTE.

2.3.4.5 Debugging Registers

The X^{704} implements instruction and data address breakpoints through the use of the IABR, DABR, and BPTCTL registers. Two formats of DABR are implemented: one that conforms to the definition suggested in Appendix A of Book III and an extended definition that provides additional functionality. Accesses to the compatible DABR register affect both the extended XDABR register and the BPTCTL register.

See Book III for the behavior of data breakpoints if the BPTCTL register is left as initialized by a hard reset and all accesses to XDABR and BPTCTL are done through the DABR register.

2.3.4.5.1 Breakpoint Control Register (BPTCTL)

The BPTCTL register contains enables and control information for both instruction and data breakpoints. The format of this register is shown in Figure 10.

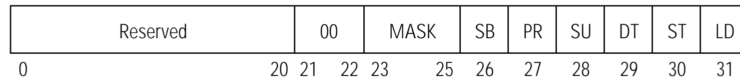


Figure 10: Breakpoint Control Register

The fields are defined as follows:

MASK selects which of the low 3 bits of xdabr are used in data address breakpoint comparisons. A '0' bit in MASK(0:2) prevents the corresponding bit in xdabr(29:31) from participating in the comparison. An address match occurs if

```
((EADDR<0:28> == DABR<0:28>) &&
((EADDR<29:31> & MASK) == (DABR<29:31> & MASK)))
```

If MASK is 7, the effective address and XDABR must match exactly. If MASK is 0, the effective address and XDABR need only refer to the same double-word. This field does not affect instruction breakpoints.

SB is the strobe bit. If this bit is set, instruction and data address breakpoints cause a strobe of a pin instead of an exception. See Section 5.2 on page 109 for a description of the STROBE pin.

PR is the problem state bit. If this bit is set, data address breakpoints occur on problem state references. This bit does not affect instruction address breakpoints.

SU is the supervisor state bit. If this bit is set, data address breakpoints occur on supervisor state references. This bit does not affect instruction address breakpoints.

DT is the data translation bit. If this bit is set, data translation must be enabled in order to trigger a data address breakpoint. If it is clear, data translation must be disabled in order to trigger a data address breakpoint.

ST is the store enable bit. If this bit is set, data address breakpoints may occur on stores. If it is clear, stores will not cause data address breakpoints. The ***dcbz*** instruction is considered to be a 32-byte store.

LD is the load enable bit. If this bit is set, data address breakpoints may occur on loads. If it is clear, loads will not cause data address breakpoints.

A data breakpoint address match occurs if any byte in the reference matches the breakpoint address in XDABR as modified by the MASK field. For references which span multiple doublewords, part of the reference may have completed before the trap is taken. The ***dcbz*** instruction is defined to reference all 32 bytes in a cache block. Cache management instructions other than ***dcbz*** do not cause instruction or data address breakpoints.

A data address breakpoint occurs if all of the following conditions are met:

- The effective address matches the value in XDABR as modified by the MASK field.
- MSR[DR] has the same value as BPTCTL[DT].
- BPTCTL[PR] and MSR[PR] are both set, or BPTCTL[SU] is set and MSR[PR] is clear.
- The reference is a load and BPTCTL[LD] is set, or the reference is a store and BPTCTL[ST] is set.

If neither LD nor ST is set, or if neither PR nor SU is set, the data address breakpoint is disabled.

A ***stwcx.*** instruction that does not perform a store may still take a data address breakpoint.

A data address breakpoint with SB clear causes a data storage interrupt with DSISR(9) set and the address saved in DAR.

If SB is set and a breakpoint is triggered, no exception occurs. Instead, the value of the L2CTL.STROBE bit is inverted for one bus cycle before being placed on the STROBE pin. The elapsed time between the triggering of the breakpoint and the pulse on the pin is not defined precisely, but will be a small number of bus cycles.

Multiple breakpoints triggered in a small number of processor cycles can appear on the pin as one pulse because of the ratio between processor cycles and bus cycles. The strobe does not occur unless the exception would have occurred: a higher-priority trap suppresses the strobe. The SB bit permits references to be detected without affecting the performance of the processor in almost all cases. Instruction breakpoint hits occurring near other instruction fetch traps may cause a slight change in processor timing.

Breakpoint exceptions have lower priority than TLB misses, TLB store faults and all other data storage interrupts.

In order to ensure that changes to the breakpoint registers take effect, software should execute a ***sync*** instruction after writing BPTCTL or XDABR/DABR and before the first data reference that could cause a break-

point. An ***isync*** instruction following the ***sync*** is required if a change in BPTCTL[SB] affects subsequent instruction breakpoints.

2.3.4.5.2 Instruction Address Breakpoint Register (IABR)

The IABR register contains an effective word address that is compared with the program counter. The format of this register is shown in Figure 11.

Figure 11: Instruction Address Breakpoint Register

The fields are defined as follows:

ADDR — is bits (0:29) of the instruction breakpoint address.

IE — is the instruction breakpoint enable. If this bit is set, an instruction address breakpoint occurs when the processor attempts to issue the instruction fetched from the effective address in the ADDR field and MSR[IR] has the same value as the IT field.

IT — is the instruction translation enabled bit. If this bit is set, instruction translation must be enabled in order to trigger an instruction address breakpoint. If it is clear, instruction translation must be disabled in order to trigger an instruction address breakpoint.

An instruction breakpoint match occurs if the address of the first byte of the instruction matches the ADDR field and MSR[IR] has the same value as the IT field. Instruction breakpoint exceptions are taken before the instruction is executed. An instruction breakpoint exception causes a trace interrupt and sets SRR1(11).

If instruction breakpoints are being enabled, disabled, or changed, a ***sync*** instruction followed by an ***isync*** instruction should be executed after BPTCTL or IABR writes.

2.3.4.5.3 Extended Data Address Breakpoint Register (XDABR)

The XDABR register contains a 32-bit effective address that is compared with effective addresses used by loads, stores, and cache operations. When the two addresses match, and the appropriate enables are set in the BPTCTL register, a data storage interrupt occurs. See the description of the BPTCTL register on page 42 for more information on data breakpoints.

2.3.4.5.4 Data Address Breakpoint Register (DABR)

This is a PowerPC-compatible version of the DABR register. It accesses both the XDABR and BPTCTL registers in a way that mimics the behavior of the data breakpoint register as suggested in Appendix A of Book III. The DABR is not a separate register from the XDABR; it merely provides an alternate way of accessing the same data. The format of this register is shown in Figure 12.

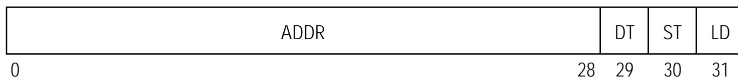


Figure 12: Data Address Breakpoint Register

The fields are defined as follows:

ADDR — is bits (0:28) of the doubleword breakpoint address.

DT — is the data translation bit. If this bit is set, data translation must be enabled in order to trigger a data address breakpoint. If it is clear, data translation must be disabled in order to trigger a data address breakpoint.

ST — is the store enable bit. If this bit is set, data address breakpoints occur on stores to addresses that match XDABR as modified by the MASK field. The ***dcbz*** instruction is considered to be a store.

LD — is the load enable bit. If this bit is set, data address breakpoints occur on loads from addresses that match XDABR as modified by the MASK field.

On writes, the contents of the ADDR field are written to bits (0:28) of XDABR, and bits (29:31) of XDABR are cleared. The contents of the DT, ST, and LD fields are written to the corresponding fields in the BPTCTL register. In addition, writes to DABR also set the PR and SU bits in the BPTCTL register to one.

On reads, the contents of XDABR are returned, with bits 29, 30, and 31 replaced by BPTCTL[DT], BPTCTL[ST], and BPTCTL[LD], respectively.

2.3.4.5.5 Event Register (EVENT)

The X^{704} processor can use the TBU and TBL time base registers to count the occurrence of performance-related events. The EVENT register controls which of those events are counted. The format of the EVENT register is shown in Figure 13.

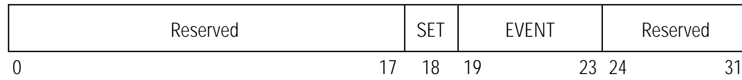


Figure 13: Event Register

When the EVENT field contains 0, the TBU and TBL registers form a 64-bit counter that increments every four bus cycles. Any other setting of this field causes these registers to count other events and to no longer maintain the time base value. When the EVENT field contains 1, the TBU and TBL registers hold their current values. When the field contains a value greater than 1, the TBU register counts instructions completed, and the TBL register counts the events shown in Table 3 depending on the value of the SET and EVENT fields.

Table 3: Event Counter Selections

EVENT	SET = 1	SET = 0
0	time base upper	time base lower
1	do not count	do not count
4	L1 instruction cache accesses	L1 instruction cache misses
5	ITLB accesses	ITLB misses
8	L1 data cache accesses	L1 data cache misses
9	TLB accesses	TLB misses
10	L2 cache accesses	L2 cache misses
11	snoop accesses	snoop misses
16	decode buffer empty	memory operand hold
17	multi-step X holds	misaligned accesses
18	ALU valid in A	ALU valid in C
19	ALU valid in M	reserved
20	0 flows issued	0 flows completed
21	1 flow issued	1 flow completed
22	2 flows issued	2 flows completed
23	3 flows issued	3 flows completed
24	branch correctly predicted taken	branch incorrectly predicted taken
25	branch correctly predicted not taken	branch incorrectly predicted not taken
26	pipe restarts	finder invalids
27	traps from ***tw***/***twi*** instructions	all other traps
28	mispredicts in A	mispredicts in C

Table 3: Event Counter Selections (Cont.)

EVENT	SET = 1	SET = 0
29	mispredicts in M	mispredicts in W
30	processor clocks	flows completed
31	processor clocks	flows issued
All others	reserved	reserved

Familiarity with the material in Chapter 3 and Chapter 4 on the X^{704}'s pipeline structure, performance, and branch prediction is needed for a full understanding of most of these events. In particular, the event counters may not agree with the values expected for a program run using the sequential execution model because speculative instruction issues and cache accesses and cache misses may be counted.

The MODES[TBD] bit disables counting only when the EVENT register contains zero.

When the TBU and TBL registers reach their maximum value, they increment to zero. When gathering performance statistics, they should be read frequently to ensure that data is not lost. A 32-bit counter incrementing at 500 MHz will wrap once in approximately 8.5 seconds.

Important Note: The definition of the EVENT register changed from the definition used in the prototype version of the X^{704}.

2.3.4.6 Processor Control Registers

The X^{704} contains several miscellaneous registers that control various processor functions such as resource enabling and disabling, error reporting, and multiprocessor support.

2.3.4.6.1 Modes Register (MODES)

The MODES register, depicted in Figure 14, controls several functions related to instruction decoding and dispatching.

Figure 14: Modes Register

The fields are defined as follows:

BPE — is the branch prediction enable bit. If this bit is clear, the branch prediction hardware is ignored and all branches will be predicted to be not taken. If it is set, the processor predicts branch directions and targets as described in Section 3.8 on page 81.

TBD — is the time base disable bit. If this bit is set, the time base register does not increment. If it is clear, the time base register increments every fourth bus clock. This bit controls the time base register increment only while the EVENT register contains zero.

DE — is the diagnostic access enable bit. If this bit is set, the ***lwdx*** and ***stwdx*** instructions execute. If it is clear, attempts to execute those instructions will result in illegal instruction program interrupts.

POE — is the pipeline overlap enable bit. If this bit is clear, instructions are issued only when the execution pipeline is empty. If it is set, instructions are issued when other instructions are in the pipeline. This bit does not affect superscalar issue or instruction queuing in the fetch unit.

SSE — is the superscalar enable bit. If this bit is clear, superscalar issue is disabled and only one instruction may be issued on each cycle. If it is set, up to three instructions may be issued on each cycle as described in Section 4.4 on page 92.

2.3.4.6.2 Machine Check Register (CHECK)

The CHECK register, shown in Figure 15, contains temperature status information and the enables for various conditions that can cause machine checks or checkstop conditions. Not all conditions that cause machine checks have an enable. If the machine check condition is detected, and that condition has no enable, or if the enable bit corresponding to the error is set, a machine check interrupt occurs. If MSR[ME] is set, execution continues at the machine check trap vector. If MSR[ME] is clear, the processor enters the checkstop state and halts.

See Section 5.2.3 on page 113 for more information on the processor's operating temperature range and the use of the status bits in the CHECK register.

Reserved	WT	HT	R	IF	DC	L2C	L2B	BP	TLB	SW	Reserved
0 5	6	7	8	9	10	11	12	13	14	15	16 31

Figure 15: Machine Check Register

The fields are defined as follows:

WT — is the warm temperature indication. When this bit is set, the processor is near the upper end of its operating temperature range.

HT — is the over-temperature indication. When this bit is set, the processor has exceeded its maximum operating temperature.

R — is the reset type bit. This bit is initialized by hardware to one on a hard reset and zero on a soft reset. It may not be written by software.

IF — is the instruction fetch machine check enable. If this bit is set, multiple IBAT or ITLB hits on the effective address of an issuing instruction are reported as machine checks.

DC — is reserved for future use. The value of this bit is ignored.

L2C — is the invalid level 2 state machine check enable. If this bit is set, an invalid state in a level 2 cache tag is reported as a machine check. This includes cache LRU state errors and multiple hits in a set for the same address.

L2B — is the invalid bus controller state machine check enable. If this bit is set, an invalid state in the bus controller is reported as a machine check. This includes detecting invalid snoop types on the bus and errors in filling, pushing, or evicting cache blocks.

BP — is the bus parity machine check enable. If this bit is set, bus address and data parity errors are reported as a machine check. Machine checks caused by the assertion of the $\overline{\text{TEA}}$ or $\overline{\text{MCP}}$ interface signal are always enabled.

TLB — is the TLB machine check enable. If this bit is set, an address that matches multiple TLB entries or multiple DBATs is reported as a machine check.

SW — is the software-initiated machine check bit. A single machine check occurs when software changes the value of this bit from zero to one. If machine check interrupts are enabled, the interrupt handler should clear this bit in order to re-enable software machine checks.

Failure to enable a machine check can result in undefined behavior should the disabled machine check condition occur. For example, a masked parity

error could result in the use of corrupted data and a reference through an address that matches multiple TLB or BAT entries could result in referencing unexpected or non-existent physical memory.

2.3.4.6.3 L2/Bus Control Register (L2CTL)

The L2CTL register controls the behavior of all three on-chip caches and the bus interface. The format of this register is shown in Figure 16.

00000	CLOCK	BC	BS	BE	BT	SB	IE	DE	L2E	000	IP	DP	CM	TM	IC
0 10	11 15	16	17	18	19	20	21	22	23	24 26	27	28	29	30	31

Figure 16: L2/Bus Control Register

The fields are defined as follows:

CLOCK — is the current ratio between the external (bus) clock and the processor clock. This value is set from the external PLL_CFG(2:6) pins on power on or hard reset and cannot be changed by software. See the description of the PLL_CFG pins in Section 5.2.1 on page 111.

BC — is the broadcast cache operations bit. If this bit is clear, cache management operations are broadcast on the bus and global bus transactions are snooped. These operations include write kills, ***dcbf***, and ***dcbi***. If this bit is set, the X^{704} does not broadcast coherence operations and ignores all incoming snoop transactions. Setting this bit will not prevent broadcasts of kill operations caused by ***dcbz*** instructions.

BS — is the broadcast synchronization bit. If this bit is clear, ***sync*** instructions are broadcast on the bus. If it is set, ***sync*** instructions are not broadcast, and the X^{704} assumes that nothing external to the chip affects the completion of ***sync*** instructions.

BE — is the broadcast ***eieio*** bit. If this bit is clear, ***eieio*** instructions are broadcast on the bus. If it is set, ***eieio*** instructions are not broadcast on the bus.

BT — is the broadcast TLB operations bit. If this bit is clear, ***tlbie*** and ***tlbsync*** instructions are broadcast on the bus. If it is set, TLB operations are not broadcast on the bus, and the X^{704} assumes that nothing external to the chip affects the completion of ***tlbsync*** instructions.

SB — is the strobe bit. The value of this bit is copied to the STROBE pin. Data and instruction address breakpoints can optionally invert this bit for one bus cycle. See the discussion of the BPTCTL register in Section 2.3.4.5.1 on page 42 for more information on the use of the strobe facility by breakpoints.

IE — is the instruction cache enable bit. If this bit is set, the instruction cache is enabled. If it is clear, instruction fetches bypass the instruction cache, and the level 2 cache does not write data into the instruction cache.

DE — is the data cache enable bit. If this bit is set, the data cache is enabled. If it is clear, load and store accesses bypass the data cache, and the level 2 cache does not write data into the data cache.

L2E — is the level 2 cache enable bit. If this bit is set, the level 2 cache is enabled. If it is clear, all instruction and data accesses not satisfied by the instruction and data caches will be satisfied from off-chip memory.

IP — is the instruction prefetch enable. If this bit is set, the level 2 cache can prefetch additional blocks from off-chip into the level 2 cache in response to instruction cache misses.

DP — is the data prefetch enable. If this bit is set, the level 2 cache can prefetch additional blocks from off-chip into the level 2 cache in response to data cache misses.

CM — is the column mask update enable bit. If this bit is set, updates to the level 2 column disable register (L2CDR) are allowed. If it is clear, writes to that register are ignored. See Section 2.3.4.6.4 on page 52 for more information on the L2CDR register.

TM — is the tag block valid update enable bit. If this bit is set, the block valid field in the level 2 cache use records can be updated with diagnostic stores to the use records. If it is clear, diagnostic writes to the use records may not alter the field valid mask. See Section 3.4.3 on page 73 for more information on use records and the field valid mask.

IC — is the instruction cache coherency bit. If this bit is set, the processor ensures that the instruction cache stays coherent with respect to data stores. If it is clear, software is responsible for ensuring that updates to the instruction stream are reflected in the instruction cache by executing an instruction sequence similar to that shown in Section 2.2.6 on page 32. Setting this bit degrades the performance of the processor, but may be useful for applications that emulate instruction execution for other architectures that enforce instruction cache coherency.

The BC, BS, and BT bits should be cleared in multiprocessor systems and should be set in uniprocessor systems. The BE bit should be cleared in all multiprocessor systems or in any uniprocessor system with an external device that is sensitive to the order in which writes are completed.

For more information on prefetching, see Section 3.4.7 on page 78.

2.3.4.6.4 L2 Column Disable Register (L2CDR)

The L2CDR register provides one of two ways to mark damaged cache RAM entries as unusable. By setting the appropriate bit in this register, an entire associativity class consisting of 4KB is removed from the level 2 cache. The register is formatted as shown in Figure 17.

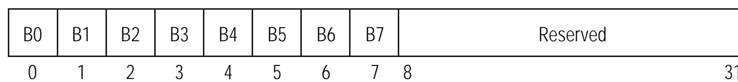


Figure 17: L2 Column Disable Register

The fields are defined as follows:

Bn	is the block *n* disable bit. If this bit is set, block *n* in each set of the level 2 cache is marked as unusable. Any valid data in a cache block marked disabled is lost.

This register can be written only when the CM bit in the L2CTL register is set. It is intended to be set by a power-on-self-test cache checker when it discovers multiple failures in a column. Because the level 2 cache data RAM is 2-way interleaved, physical RAM errors that affect an entire column require the paired column from the other bank (the paired banks are 0–4, 1–5, 2–6, and 3–7) to be disabled. When a column is disabled, it must also be marked as recently used in the PLRU field of each level 2 cache use record. See Section 3.4.3 on page 73 for information on cache use records and the cache replacement strategy.

Disabling more than four columns may cause level 2 cache controller machine checks to occur when the level 2 cache is enabled.

2.3.4.7 Processor Identification Register (PIR)

The PIR register is a 32-bit register that can be read or written by privileged programs. It is neither interpreted nor used by the hardware. Its intended use is holding a unique processor identification number for each CPU in a multiprocessor system.

2.3.5 Storage Control

The X^{704} provides ordinary storage segments as described in Book III; direct-store segments are not supported. An attempt to reference data with a fixed-point load or store instruction through a segment register with the T bit set causes a data storage interrupt. An attempt to reference data with a floating-point load or store instruction through a segment register with the T bit set causes an alignment interrupt. Cache management instructions that reference direct-store segments are treated as ***nop***s.

2.3.5.1 Translation Lookaside Buffer (TLB)

The X^{704} contains a 128-entry, 4-way set-associative TLB. In addition, the instruction fetch unit contains a 4-entry, fully associative instruction TLB (ITLB). When an instruction fetch misses in the ITLB, the X^{704} attempts to resolve the miss by searching the main TLB. If it finds a matching entry in the TLB, it copies translation and protection information into the least recently used ITLB entry and it marks the TLB entry as the most recently used entry in its set. The translations present in the ITLB are always a subset of the translations contained in the main TLB.

The hardware does not search the page table or update the TLB if either the ITLB miss or any data access fails to find a matching TLB entry. Instead, a TLB miss interrupt occurs and a software handler performs the page table search and TLB refill.

The hardware also does not update the storage access recording bits in page table entries. When it places an entry in the TLB, the TLB miss interrupt handler should set the Reference (R) bit in the associated PTE. The Change (C) bit in the PTE is copied to the TLB entry. When the hardware detects a store access through a TLB entry with the C bit clear, a TLB store interrupt occurs. The handler for this interrupt should set the C bit in the TLB entry and the C bit in the associated PTE before restarting the faulting instruction.

See Section 2.3.6.2 on page 56 for detailed descriptions of the TLB miss and TLB store interrupts, and Appendix A for sample handlers.

The TLB miss handler can write either a specific entry or the least recently used entry in the target set. Writing multiple TLB entries that translate the same effective address is an error and may cause a machine check or boundedly undefined results.

2.3.5.2 Block Address Translation

The X^{704} implements block address translation as described in Book III of the *PowerPC Architecture Specification*. Because the two halves of each IBAT or DBAT register must be loaded separately, software must ensure that inconsistencies caused by a partially loaded IBAT or DBAT do not affect program execution. To do this, load these registers only when the associated relocation enable bit is clear.

Loading an IBAT or DBAT register with an invalid BL field or with either BEPI or BRPN fields that are inconsistent with the BL field can cause boundedly undefined results.

2.3.5.3 Storage Access Modes

When the caching inhibited storage access mode is enabled, the state of the write through required access mode is assumed to be off. Thus, the two unsupported access mode combination (WIM = 110 or 111) are treated as caching inhibited, write through not required storage.

2.3.5.4 Reference and Change Recording

The X^{704} hardware does not set the PTE Reference and Change bits directly; they are set by the TLB miss and TLB store interrupt handlers as discussed in Section 2.3.5.1 on page 53.

Because TLB misses must be satisfied before the hardware can determine that an access is permitted, it is likely that the PTE Reference bit will be set in cases where read or write permission is denied and no storage access occurs. Similarly, a TLB store fault may occur before it is known whether a ***stwcx.*** instruction will succeed. Thus, it is likely that the PTE changed bit will be set in these circumstances. The TLB store handler is invoked only when the reference has write permission to the target page.

A TLB miss on an instruction fetch occurs only when that instruction is required by the sequential execution model and any exceptions related to previous instructions have been resolved.

2.3.5.5 Storage Control Instructions

The X^{704} implements all of the storage control instructions specified in Book III of the *PowerPC Architecture Specification* except for the optional ***tlbia*** instruction. The ***tlbia*** instruction may be emulated by a sequence of ***tlbie*** instructions as described in Section 2.3.5.5.3 on page 55.

2.3.5.5.1 Data Cache Block Invalidate (***dcbi***)

Executing the ***dcbi*** instruction invokes the TLB miss handler if data translation is enabled and no translation for the effective address is found in the TLB or DBAT. If a translation is found, but write access is not allowed, a data storage interrupt occurs.

If either data address translation is disabled or a translation is found, and the addressed block is marked as valid in the level 2 cache, the block is invalidated in the data cache, the instruction cache, and the level 2 cache. Any modifications to the data in the cache block are discarded. If the storage is marked as coherence required, the kill operation is broadcast on the bus.

The operation of this instruction is independent of the state of the cache enables. If the block is not present in the level 2 cache of any processor, the instruction is treated as a ***nop***.

2.3.5.5.2 TLB Invalidate Entry (***tlbie***)

The ***tlbie*** instruction invalidates all four TLB entries in the set addressed by RB(15:19) and flushes the ITLB. The TLB entries are invalidated without regard to their contents. If the broadcast TLB bit is set in the L2CTL register, a TLB invalidate operation is broadcast on the bus to other processors.

In a multiprocessor system, software must guarantee that only one processor in the system is executing ***tlbie*** instructions at any given time, or undefined behavior including a deadlocked system occurs.

2.3.5.5.3 TLB Invalidate All (***tlbia***)

The ***tlbia*** instruction is not implemented on the X^{704}, instead the entire TLB can be invalidated by executing a sequence of 32 ***tlbie*** instructions—one for each of the 32 sets in the TLB. For example, a subroutine containing the following loop invalidates all entries in the TLB:

```
for (addr = 0; addr < 32 * 0x1000; addr += 0x1000)
        tlbie (addr);
```

2.3.5.5.4 TLB Synchronize (***tlblsync***)

The ***tlbsync*** instruction completes only when all ***tlbie*** instructions previously issued by this processor are complete. If the broadcast TLB bit in the L2CTL register is set, the ***tlbsync*** operation is broadcast on the bus and

the instruction does not complete until all processors have executed all ***tlbie*** instructions issued on this processor before the ***tlbsync***.

In a multiprocessor system, software must guarantee that only one processor in the system is executing ***tlbsync*** instructions at any given time, or undefined behavior including a deadlocked system occurs.

2.3.6 Interrupts

The following sections discuss the X^{704} implementation of interrupts.

2.3.6.1 Interrupt Classes

The X^{704} does not have any imprecise interrupts. All floating-point exceptions are precise.

2.3.6.2 Interrupt Definitions

In addition to the interrupt types defined in the *PowerPC Architecture Specification*, the X^{704} implements the TLB miss and TLB store interrupts. The X^{704} interrupt vector is shown in Table 4, with implementation-dependent interrupts shown in shaded rows

Table 4: Interrupt Vector Offsets

Vector Offset	Interrupt Type
0x0000	Reserved
0x0100	System Reset
0x0200	Machine Check
0x0300	Data Storage
0x0400	Instruction Storage
0x0500	External
0x0600	Alignment
0x0700	Program
0x0800	Floating-Point Unavailable
0x0900	Decrementer
0x0A00	Reserved
0x0B00	Reserved
0x0C00	System Call
0x0D00	Trace
0x0E00	Floating-Point Assist

Table 4: Interrupt Vector Offsets (Cont.)

Vector Offset	Interrupt Type
0x1000	TLB Miss
0x1100	TLB Store

2.3.6.2.1 System Reset Interrupt

On the X^{704}, the system reset interrupt is caused by the assertion of either the HRESET or the SRESET input pin. The interrupt handler determines which pin was asserted by examining the R bit in the CHECK register.

A system reset interrupt always clears SRR1(30), indicating that the interrupt is not recoverable.

2.3.6.2.2 Machine Check Interrupt

A machine check condition occurs when one of the conditions enabled in the CHECK register is detected or when the TEA interface signal is asserted. A machine check condition causes a machine check interrupt if the MSR[ME] bit is set. If MSR[ME] is clear, the processor enters checkstop mode instead. In checkstop mode, the processor stalls until either the HRESET or the SRESET external reset pin is asserted. In some circumstances, it may not be possible to take a machine check interrupt even when MSR[ME] is set; instead, the processor enters checkstop mode.

In general, machine check conditions cannot be precisely related to the execution of any particular instruction and cannot be restarted. A machine check interrupt can result in corrupted data being placed in general registers or in any one of the caches.

The following registers are set:

SRR0 Set to the effective address of the last instruction that completed. For some machine checks, that instruction may have caused boundedly undefined results.

SRR1

1:4 Set to 0.

10 Set to 0.

11:15 Set to a nonzero value according to the following:

00001an invalid level 2 cache tag or use record state was detected.

00010an address hit more than one tag in the level 2 cache.

00011the bus controller detected an internal state machine error.

00100an invalid snoop type (TT) was detected.

00101a bus data parity error was detected.

00110a bus address parity error was detected.

00111a qualified assertion of the $\overline{TEA}$ signal was received.

01001software requested a machine check by writing a one to CHECK[SW].

01010a multiple DBAT or TLB hit was detected on a single data reference.

10000the external machine check pin ($\overline{MCP}$) was asserted.

10100a multiple IBAT or ITLB hit was detected on a single instruction reference.

All other nonzero values are reserved. If multiple machine check conditions are detected between instruction issues, this field may not be meaningful.

30 Set to zero, indicating that the interrupt is not recoverable.

Others: Loaded from the MSR register.

The machine check interrupt handler should set the MSR[ME] bit so that additional machine checks that occur while the handler is executing do not cause the processor to checkstop.

Software that tests for the existence of physical memory by issuing loads and observing whether a machine check results should adhere to the following guidelines:

- Instruction references should not be used to probe for memory.
- The storage being probed should be marked as caching inhibited, or the probe references should be executed with caches disabled.
- The probe references should be preceded and followed by ***sync*** instructions.

Warning: Failure to follow these guidelines may result in checkstops caused by multiple machine checks.

2.3.6.2.3 Data Storage Interrupt

The X^{704} implements data storage interrupts as described in the *PowerPC Architecture Specification* with the following notes:

- Any attempt to access fixed-point data in a direct-store segment causes a data storage interrupt with DSISR(0) set and DAR set to the effective address of the direct-store reference.
- An ***stwcx.*** instruction with an effective address for which a normal store would cause a data storage interrupt causes a data storage interrupt even if the processor does not perform the store.
- Data address breakpoints are supported: DSISR(9) is set for data breakpoints and cleared for other data storage interrupts. On a data address breakpoint, the DAR register may not contain the effective address computed by the instruction that triggered the breakpoint. For load/store multiple, move assist, or unaligned elementary accesses where the breakpoint address is not in the same doubleword as the effective address, DAR contains an address in the doubleword that triggered the breakpoint.
- An ***lwarx*** or ***stwcx.*** instruction that addresses a location that is write through required completes correctly and does not cause a data storage interrupt.

2.3.6.2.4 Instruction Storage Interrupt

The X^{704} implements instruction storage interrupts as defined in the architecture specification.

2.3.6.2.5 External Interrupt

The X^{704} implements external interrupts as defined in the architecture specification.

The X^{704} expects the external interrupt signal ($\overline{\text{INT}}$) to be asserted until the external interrupt handler software acknowledges the interrupt to the device signalling it.

2.3.6.2.6 Alignment Interrupt

The X^{704} causes an alignment interrupt when any of the following conditions occur:

- The effective address of a ***lwarx*** or ***stwcx.*** instruction is not word-aligned.
- A ***lswi***, ***lswx***, ***stswi***, ***stswx***, ***lmw***, or ***stmw*** instruction is executed in power-endian mode.
- Any unaligned access that crosses a doubleword boundary is attempted in power-endian mode.
- The effective address of a floating-point load or store instruction references a location in a direct-store segment.

The SRR0, SRR1, DAR, and DSISR registers are set as described in the architecture specification, with the additional note that alignment interrupts caused by ***lmw***, ***lswi***, and ***lswx*** instructions set DSISR(27:31) to the RA field of the instruction.

2.3.6.2.7 Program Interrupt

The X^{704} implements program interrupts as defined in the architecture specification.

2.3.6.2.8 Floating-Point Unavailable

The X^{704} implements floating-point unavailable interrupts as defined in the architecture specification.

2.3.6.2.9 Decrementer Interrupt

The X^{704} implements decrementer interrupts as defined in the architecture specification.

2.3.6.2.10 System Call Interrupt

The X^{704} implements system call interrupts as defined in the architecture specification.

2.3.6.2.11 Trace Interrupt

The X^{704} implements trace interrupts as shown in Appendix A of Book III.

In addition, instruction address breakpoints cause trace interrupts. See Section 2.3.4.5 on page 41 for more information on instruction breakpoints.

When a trace interrupt is taken, SRR1 is set as follows:

1:4	is set to 0.
10	is set to 0.
11	is set to 1 for instruction breakpoints or 0 for single-step or branch trace interrupts.
12:15	are set to 0.
Others:	are loaded from the MSR register.

2.3.6.2.12 Floating-Point Assist Interrupts

The floating-point assist interrupt is not used by the X704.

2.3.6.2.13 TLB Miss interrupt

The TLB miss interrupt is a X704 implementation-dependent interrupt. The interrupt vector offset of this trap is 0x1000.

A TLB miss interrupt occurs when MSR[IR] is set and no translation for an instruction fetch address is present in either an IBAT or the TLB, or when MSR[DR] is set and no translation for an effective address is present in either a DBAT or the TLB.

The following registers are set:

MSR

14	is set to 1, blocking system TLB invalidates.
Others:	as described in the architecture specification.

SRR0 — is set to the effective address of the instruction that caused the interrupt.

SRR1

0:3	are loaded from CR0.
4	is set to 0.
10:15	are set to 0.
Others:	are loaded from the MSR register.

MAR — is set to the instruction or data effective address that caused the TLB miss interrupt.

MISR

1	is set to 1 for loads or stores, indicating a possible page fault, and is set to 0 for instruction fetches.
2	is set to 1 for instruction fetches, and set to 0 for loads or stores.
6	is set to 1 for stores, and set to 0 for loads or instruction fetches.
Others:	are set to 0.

A TLB miss interrupt changes the contents of the MAR and MISR registers. Changes to these registers can alter the contents of the CMP, HASH1, and HASH2 registers and can change the TLB entry accessed by the TLBLRU0, TLBLRU1, and TLBMRF registers.

The definition of the MISR allows a common TLB miss handler to handle both instruction and data TLB misses. If the miss is really a page fault (no matching PTE is found in either PTEG searched) the handler looks at MISR(2). If it is clear, the handler copies MAR and MISR to DAR and DSISR and jumps to the data storage interrupt vector. This works because MISR(1) and MISR(6) are set correctly for a page fault. If MISR(2) is one, the handler sets SRR1 to 0x40000000, indicating an instruction page fault, clears the MSR[TW] bit set by the trap, and then jumps to the instruction storage interrupt vector.

The TLB miss handler should take advantage of the TLB assist SPRs described in Section 2.3.4.4 on page 38. The handler can safely alter CR0 without first saving it because the hardware has already saved CR0 in SRR1. General registers must be saved to SPRGs. The handler must restore the condition register and any altered general registers before exiting. A handler that exits without executing an ***rfi*** instruction must clear MSR[TW], which was set by the trap. See Appendix A for a sample TLB miss handler.

2.3.6.2.14 TLB Store Interrupt

The TLB store interrupt is a X^{704} implementation-dependent interrupt. The interrupt vector offset of this trap is 0x1100.

A TLB store interrupt occurs when a store instruction executes with MSR[DR] set, a valid TLB entry translates the effective address computed by that instruction, the PP field of that TLB entry permits the store access, and the changed (C) bit of that TLB entry is clear.

The following registers are set:

MSR

14	is set to 1, blocking system TLB invalidates.

Others:	as described in the architecture specification.
SRR0	is set to the effective address of the instruction that caused the interrupt.
SRR1	
0:3	are loaded from CR0.
4	is set to 0.
10:15	are set to 0.
Others:	are loaded from the MSR register.
MAR	is set to the effective address of the data referenced by the instruction that caused the TLB store interrupt.
MISR	
4	is set to 1, indicating a possible protection fault.
6	is set to 1, indicating a fault caused by a store instruction.
7:8	are set to the TLB element number of the entry that translated the access.
Others:	are set to 0.

A TLB store interrupt changes the contents of the MAR and MISR registers. Changes to these registers can alter the contents of the CMP, HASH1, and HASH2 registers and can change the TLB entry accessed by the TLBLRU0, TLBLRU1, and TLBMRF registers.

The TLB store handler should take advantage of the TLB assist SPRs described in Section 2.3.4.4 on page 38. The handler can safely alter CR0 without first saving it because the hardware has already saved CR0 in SRR1. General registers must be saved to SPRGs. The handler must restore the condition register and any altered general registers before exiting. A handler that exits without executing an ***rfi*** instruction must clear MSR[TW], which was set by the trap. See Appendix A for a sample TLB store interrupt handler.

2.3.6.3 Exception Ordering

The X^{704} processor adds the following interrupt priority conditions for implementation-dependent interrupts:

- Data TLB miss interrupts have a higher priority than TLB store interrupts, and both of those have a higher priority than data storage interrupts, but a lower priority than alignment interrupts.
- Instruction fetch TLB interrupts have a higher priority than instruction storage interrupts.
- A trace interrupt caused by an instruction breakpoint is of lower priority than an instruction storage interrupt.
- A single-step or branch trace interrupt occurs before an instruction breakpoint trace interrupt on the following instruction.
- Data breakpoints are the lowest priority data storage interrupt.

2.3.7 Synchronization Requirements for Special Registers

Several of the X^{704}'s implementation-dependent SPRs can alter the context in which addresses are interpreted and in which instructions are executed. The side effects caused by these context-altering instructions may not occur in program order, and can require explicit software synchronization.

Table 5 shows the type of synchronization required before and after an instruction that changes the contents of each SPR. As in Book III, the notation *CSI* in the table means any context-synchronizing instruction or any interrupt other than a non-recoverable reset or machine check.

Table 5: Synchronization Requirements for Implementation-Dependent SPRs

Register	Required Before	Required After
MAR	none	***sync***[1]
MISR	none	***sync***[1]
SPRG	none	none
EVENT	***sync***[2]	CSI
TLBLRU0–TLBLRU1	none[3]	CSI[4]
TLBMRF	none[3]	CSI[4]
BPTCTL	none	***sync***
IABR	none	CSI
DABR/XDABR	none	***sync***
CHECK	none	CSI

Table 5: Synchronization Requirements for Implementation-Dependent SPRs (Cont.)

Register	Required Before	Required After
MODES	none	CSI
L2CTL[IE]	none	***isync***[5]
L2CTL[L2E]	***sync***	***isync***
L2CTL (other)	none	***sync***
L2CDR	none[6]	***sync***
PIR	none	none
TLB entries	none	CSI[4]

1. Required only when the write is followed by an access to the TLBLRU0, TLBLRU1, or TLBMRF registers.
2. The ***sync*** instruction ensures that all storage-related events are counted before the value of EVENT changes.
3. These registers should not be written while address translation is enabled.
4. A context-synchronizing event is required before translation is re-enabled. Accesses to the new translation should not be made until after the CSI following the instruction that sets MSR.IR or MSR.DR.
5. If the cache line containing the instruction that modifies L2CTL is already in the instruction cache, its contents must be the same as the contents of that line in memory. If the two lines differ, the results of continued execution are boundedly undefined.
6. This register should not be written while the level 2 cache is enabled.

Accesses to special purpose registers using ***stwdx*** instructions to the diagnostic address space are also subject to these synchronization requirements. Additional synchronization rules for some MSR bits are given in Section 2.3.3.2 on page 34.

3. Processor Operation

This chapter presents a detailed description of the X^{704} microarchitecture and implementation, including the execution pipeline, caches, TLB, and branch prediction units.

3.1 Execution Pipeline

The X^{704} pipeline consists of a fetch stage (F) followed by five execution stages used by all instructions: decode (D), address generation (A), cache access (C), tag match (M), and writeback (W). These stages are normally denoted by the initials F, D, A, C, M, and W. The fetch stage is usually omitted from pipeline diagrams because it does not participate in instruction interlock or operand bypass operations; however, it is shown in diagrams including branch mispredicts to demonstrate the cause of the performance penalty.

The terms group, flow, and step are frequently used in describing the pipeline. A *group* is a set of zero to three instructions that are issued on a single cycle and travel down the pipeline together. Individual instructions proceed down the pipeline in *flows*. Most instructions need only a single flow, but some complicated instructions require multiple flows. For example, the move assist and load and store multiple instructions use one flow for each register transferred, misaligned load accesses require two flows, and misaligned store accesses require three flows. Most flows require a single *step* in each pipe stage, but instructions such as integer multiplies and divides require multiple steps in the ALU.

The following sections describe each pipeline stage. Additional information, including detailed pipeline diagrams, appears in Chapter 4.

3.1.1 Fetch Stage (F)

In the fetch stage, the instruction fetch unit reads the instruction cache and finder and, if the fetch PC hits in the instruction cache, places either one or two instructions into the six-element instruction buffer.

3.1.2 Decode Stage (D)

In this stage, the decode unit reads instructions from the decode buffer, determines how many instructions can be issued on this clock, reads any general registers needed by any instructions on this cycle, and calculates the branch target address for any branch being issued on this cycle.

3.1.3 Address Generation Stage (A)

In this stage, the decode unit generates the effective address for storage access instructions.

3.1.4 Cache Access Stage (C)

In this stage, the decode unit presents the effective address for memory references to the data cache and to the TLB. Cache read data is available at the end of this stage and can be bypassed to other parts of the pipeline. This bypassing occurs before the load/store unit determines if the address hit in the cache.

3.1.5 Tag Match Stage (M)

In this stage, the TLB and data cache determine if memory accesses hit. In the event of a data cache miss, the pipeline is held until the referenced data is available. If the TLB misses, an exception is raised. Information on any other exceptions occurring on any instruction in this stage is combined and prioritized. If an exception is detected, the instruction causing the exception and all following instructions in this stage or in the D, A, or C stages are cancelled.

3.1.6 Writeback Stage (W)

In this stage, instruction results are written back to the register file and store data is transferred to the store queue. Instructions are considered to be complete once they reach the W stage.

3.1.7 ALU Operations

The X[704] contains a single ALU that can be used in the A, C, or M pipe stage. This sliding ALU stage, known as X, is normally located in the A stage, but will slide out to the C or M stage if an operand is not available in A. Placing the ALU farther down the pipeline reduces the load-use penalty but increases the penalty for mispredicted branches; the X[704] pipeline dynamically adjusts to minimize these penalties.

The relocating X stage also reduces the complexity of the instruction grouping logic by eliminating a number of group breaks that would otherwise need to be detected in the D stage. For example, the instruction dispatcher need not hold an ALU instruction with an operand that is the target of a load being issued in the same cycle.

3.1.8 Floating-Point Operations

Although floating-point operations typically take longer than ALU operations, the floating-point pipeline can be viewed as operating in lock-step with the integer pipeline. Floating-point exceptions are detected or predicted in or before the M stage. If an exception cannot be ruled out, the integer and load/store pipelines stall in M until the exception status of the floating-point operation is known.

3.2 Instruction Cache

The 2KB instruction cache consists of 64 direct-mapped, 32-byte blocks. Because the cache size is smaller than the page size, the cache can be viewed as being either physically or virtually addressed. The tags contain physical addresses.

The instruction cache supplies one doubleword of data to the instruction fetch unit on each cycle. When an instruction cache miss occurs, the cache is filled at one doubleword per cycle from the level 2 cache in critical-word-first order. Cache validity is maintained on a doubleword basis, and the level 2 cache might not supply all four doublewords in a block—particularly in the case where the cache miss occurs in the middle of a block. In that case, the level 2 cache gives a low priority to the doublewords from the start of the block through the doubleword before the miss address, and frequently does not send these doublewords to the instruction cache.

The contents of the instruction cache are a subset of the level 2 cache contents.

When the instruction cache is disabled (L2CTL.IE is clear), all instruction fetch requests are handled as if they were targeted at caching-inhibited storage. When executing with the instruction cache disabled and the level 2 cache enabled, instruction fetch accesses are not satisfied from the level 2 cache, and data brought in from off-chip memory in response to instruction fetches is not placed in the level 2 cache. This mode is intended for use by cache diagnostics only.

Note: Operating in this mode is not recommended.

The instruction cache data and tags can be read and written with the diagnostic access instructions at the addresses shown in Table 7 on page 84. The instruction cache tags are formatted as shown in Figure 18.

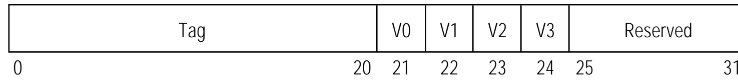


Figure 18: Instruction Cache Tags

The fields are defined as follows:

Tag	is physical address bits (0:20) of the entry present in this block.
V0	is the valid bit for the first doubleword in the block. If this bit is set, the doubleword is present in the instruction cache.
V1	is the valid bit for the second doubleword in the block. If this bit is set, the doubleword is present in the instruction cache.
V2	is the valid bit for the third doubleword in the block. If this bit is set, the doubleword is present in the instruction cache.
V3	is the valid bit for the fourth doubleword in the block. If this bit is set, the doubleword is present in the instruction cache.

3.3 Data Cache

The 2KB data cache consists of 64 direct-mapped, 32-byte blocks. Because the cache size is smaller than the page size, the cache can be viewed as being either physically or virtually addressed. The tags contain physical addresses. The data cache is a write through cache.

The data cache can supply or receive up to one doubleword of data to or from the load/store unit on each cycle. When a data cache miss occurs, the cache

is filled from the level 2 cache at one doubleword per cycle in critical-word-first order. Cache validity is maintained on a doubleword basis, and the level 2 cache might not supply all four doublewords in a block. Data cache fills of the non-critical word have a higher priority than the low-priority instruction cache fills described in the previous section.

The contents of the data cache are a subset of the level 2 cache contents.

When the data cache is disabled (L2CTL.DE is clear), all data storage requests are handled as if they were targeted at caching-inhibited storage. When executing with the data cache disabled and the level 2 cache enabled, data accesses are not satisfied from the level 2 cache, and data brought in from off-chip memory in response to load or store instructions is not placed in the level 2 cache. This mode is intended for use by cache diagnostics only; operating in this mode is not recommended.

The data cache data and tags can be read and written with the diagnostic access instructions at the addresses shown in Table 7 on page 84. The data cache tags are formatted as shown in Figure 19.

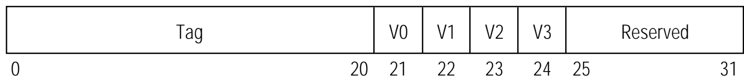


Figure 19: Data Cache Tags

The fields are defined as follows:

Tag	is physical address bits (0:20) of the entry present in this block.
V0	is the valid bit for the first doubleword in the block. If this bit is set, the doubleword is present in the data cache.
V1	is the valid bit for the second doubleword in the block. If this bit is set, the doubleword is present in the data cache.
V2	is the valid bit for the third doubleword in the block. If this bit is set, the doubleword is present in the data cache.
V3	is the valid bit for the fourth doubleword in the block. If this bit is set, the doubleword is present in the data cache.

3.4 Level 2 Cache

The level 2 cache is a 32 KB unified instruction and data cache organized as a set-associative cache with 128 sets of eight 32-byte blocks.

The level 2 cache data RAM is arranged as two interleaved banks. Each read or write has a two-cycle access time, but sequential accesses to alternating banks allow one operation to be started on every cycle. Up to a doubleword of data can be written to the level 2 cache from either the bus or the store queue in one operation. When data is not being written, a doubleword of data can be read out of the cache in order to load either of the level 1 caches or to supply data to the system bus for cache evictions and snoop pushes. When satisfying level 1 cache misses, the level 2 cache supplies the data and tag values to the level 1 caches.

The level 2 cache also implements the multiprocessor MESI cache coherency protocol. It snoops bus operations, updating its cache tags and invalidating primary cache blocks as necessary. The level 2 cache also supports the data cache block store, flush, invalidate, touch, touch for store, and block zero operations, and the instruction cache block invalidate operation.

In addition to tags recording which blocks it contains, the level 2 cache contains *use records* (see Figure 20) that record which blocks are present in either level 1 cache. This allows the cache to determine which coherency operations on the bus affect the level 1 caches and also allows some cache operations (data cache block zero, for example) to be implemented almost entirely in the level 2 cache, preventing them from delaying processor accesses to the level 1 caches.

The level 2 cache data, tags, and use records may be read and written with the diagnostic access instructions at the addresses shown in Table 7 on page 84.

3.4.1 Level 2 Cache Tags

Each cache block has a tag formatted as shown in Figure 20.

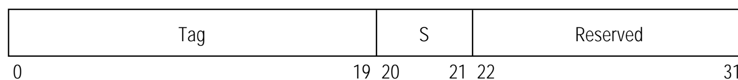


Figure 20: Level 2 Cache Tags

The fields are defined as follows:

Tag — is physical address bits (0:19) of the entry present in this block.

S — is the MESI state for this cache block. The cache state is encoded as shown in Table 6.

Table 6: Level 2 Cache Tag MESI State Values

Value	Cache Line State
00	Invalid
01	Shared
10	Exclusive
11	Modified

3.4.2 Address Translation and the Level 2 Cache

When data address translation is enabled, bits (0:19) of the effective address must be translated from virtual to physical addresses. As soon as the effective address is available, address bits (20:28) are used to index into the cache. This allows a cache tag lookup to proceed in parallel with the corresponding TLB accesses. By the time the high-order physical address bits are needed to determine if any of the tags in the set matched, the TLB will have supplied them to the cache.

3.4.3 Level 2 Cache Replacement Policy

Each cache set has a use record containing information about which blocks have been recently used, which blocks are present in the level 1 caches, and which blocks are not functional. A use record is formatted as shown in Figure 21.

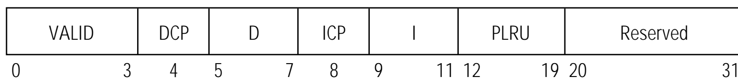


Figure 21: L2 Cache Use Record

The fields are defined as follows:

VALID — is the block valid field. This field can be used by a cache test initialization program to mark one or more blocks as unusable. This field is encoded as follows:

0000	all eight blocks in this set are valid.
1*xxx*	block *xxx* is invalid and is not used.
0100	blocks 0–3 are invalid and are not used.
0101	blocks 4–7 are invalid and are not used.

All other encodings are reserved.

DCP — is the data cache present bit. If this bit is set, one of the blocks in this set can be present in the data cache.

D — is the data cache block field. If the DCP bit is set, this field contains the index of the block that can be present in the data cache.

ICP — is the instruction cache present bit. If this bit is set, one of the blocks in this set can be present in the instruction cache.

I — is the instruction cache block field. If the ICP bit is set, this field contains the index of the block that can be present in the instruction cache.

PLRU — is the pseudo-LRU information field. This field records information on which blocks have been recently accessed. If bit n of this field is a one, then the level 2 cache controller considers block n of this cache set to have been recently used. The contents of this field is updated on all cache accesses and is used to determine which block should be replaced when a new block is brought into the cache.

Because of the cache geometry, there are two locations in the level 2 cache where each level 1 cache block may reside. If a level 1 cache block is replaced, and the new block comes from a different set than the old block, the level 2 cache use records for both sets must be updated. The use record for the replacement block is updated immediately, but there can be some delay before the use record for the block being removed from the level 1 cache can be updated, creating a temporary inconsistency where the D/DCP or I/ICP fields indicate that a block is present in a level 1 cache when it has already been replaced. It is possible that this use record update is never made. This may cause some unnecessary level 1 cache invalidates and subsequent misses, but it never causes incorrect behavior.

The pseudo-LRU algorithm used to select a block to be replaced is deterministic. If the use records are initialized to an identical state before the level 2 cache is enabled, the same memory reference pattern results in the same series of cache block replacements.

All of the fields in the use record except VALID and PLRU should be initialized to zero at reset time. Diagnostic accesses to the cache data and tags can be used to identify any bad blocks that must be recorded in the VALID field. This field is intended to record information about isolated bad blocks. If an entire column is bad, it can be disabled by using the L2CDR register described in Section 2.3.4.6.4 on page 52. Any block disabled with the VALID field must also be marked as recently used by setting the appropriate bit in the PLRU field.

Disabling more than four blocks in a single set can cause level 2 cache controller machine checks to occur on cacheable accesses to that set.

3.4.4 Disabling the Level 2 Cache

When the level 2 cache is disabled (L2CTL.L2E is clear), the processor does not maintain the level 2 cache tags or use records, and does not act upon or respond to any snooped requests on the bus. While it is possible to execute with the level 2 cache disabled and either or both level 1 caches enabled, storage coherence with other processors is not maintained, and some cache management instructions do not function correctly. In addition, care must be taken when re-enabling the level 2 cache. When executing in this state, level 1 cache misses continue to be satisfied with burst reads from off-chip memory, but the level 2 cache does not maintain inclusion. Before enabling the level 2 cache, the contents of the level 1 caches should be invalidated, synchronizing them with memory, and then the level 1 caches should be disabled. Finally, both the level 1 and level 2 caches should be enabled simultaneously.

Disabling an enabled level 2 cache must be done carefully. All prefetching should be disabled while the level 2 cache is still enabled. The L2CTL write that disables the cache must be preceded by a ***sync*** instruction and followed by an ***isync*** instruction. If the ***isync*** instruction is not the last instruction in a cache block, the remainder of the instructions in that block can be fetched as though the cache were still enabled.

3.4.5 Flushing the Level 2 Cache

The restriction that the level 1 caches are always a subset of the level 2 cache presents a complication to any program that must invalidate all blocks or flush all modified data from the level 2 cache. When programs write to the instruction stream, a single cache block can be marked as modified and also be present in both the instruction and data caches. A routine that attempts to flush the cache by touching 32KB worth of data replaces that

cache block in the data cache, but will not necessarily evict it from the instruction cache, and therefore it can remain modified in the level 2 cache.

While good programming practices dictate that writes to the instruction stream be done in the coherent fashion suggested in Section 2.2.6 on page 32, an operating system cannot guarantee that all application software is well-behaved. The following algorithm, which uses the ***lwdx*** instruction to access the L2 cache tags directly, can be used by privileged software to ensure that all modified data has been written back to main memory.

```
for (i = 0; i < L2_N_SETS; i++)
    for (j = 0; j < L2_ASSOC; j++)
        {
            tag_addr = MAKE_DIAG_L2_TAG_ADDR (i, j);
            tag = LWDX (tag_addr);
            if ((tag & L2_TAG_STATE_MASK) == L2_TAG_MODIFIED)
                                DCBST (L2_TAG_TO_ADDR (tag, i));
        }
```

In this example, the MAKE_DIAG_L2_TAG_ADDR macro creates the diagnostic address that accesses the level 2 cache tag for block *j* in set *i*. The L2_TAG_TO_ADDR macro returns the physical address of the block described by the cache tag, and the LWDX and DCBST macros invoke the ***lwdx*** and ***dcbst*** instructions, respectively. This routine must run with data address translation disabled so that the physical address of the cache block can be used as the effective address argument to ***dcbst***.

If application software is expected to flush the cache reliably, this routine should be provided as an operating system service. Alternatively, an application can guarantee that all modified lines are written back to memory by flushing the instruction cache (by executing code from 64 consecutive cache blocks, for example) and then flushing the level 2 cache by loading data from 1024 consecutive cache blocks known not to be modified in the cache.

Because it was optimized for zeroing large blocks of memory that are not expected to be referenced immediately, the ***dcbz*** instruction does not place the block containing the target storage address in the level 1 cache. It also does not invalidate the level 1 cache entries indexed by the target address if they contain blocks from a different storage address. Because of this, ***dcbz*** cannot be used to flush the entire level 2 cache.

3.4.6 Cache Coherency Protocol

The X^{704} uses the 4-state MESI protocol to maintain data coherency among its caches, the caches in other processors in a multiprocessor system, I/O devices, and main memory. This section describes the cache states, the operations that change cache block states, and the transitions that those operations cause. Some of the mechanisms used to detect and perform state transitions are part of the external bus protocol and are described in the *PowerPC 60x Microprocessor Interface Definition*.

The MESI states are:

Invalid — The block is not valid in the level 2 cache.

Exclusive — The block is valid in the level 2 cache, is not modified, and is not present in the cache of any other processor in a multiprocessor system.

Shared — The block is valid in the level 2 cache, is not modified, but can be present in the caches of other processors in a multiprocessor system.

Modified — The block is valid in the level 2 cache, has been modified with respect to the contents of main memory, and is not present in the cache of any other processor in a multiprocessor system.

In order to guarantee correct operation of the cache coherence scheme, the memory coherence storage control attribute (M bit) should be set for all pages that may be shared between processors. If the M bit is not set for a shared page, software must use cache management and synchronization instructions to ensure that separate processors have a consistent view of the data on that page.

The operations that cause changes in the MESI state of a cache block are:

Read miss — The block is changed from the invalid state to either exclusive or shared, depending on whether another processor has the line cached.

Write miss — The block is changed from the invalid state to modified.

Evict/Flush — The block is changed from the exclusive, shared, or modified states to invalid because it is being replaced in the cache, because it is the target of a ***dcbf*** instruction or bus flush operation, or because another processor is requesting exclusive access to it. If the block was in the modified state, the contents are written back to memory.

Write hit — The block is changed from the exclusive or shared states to modified because it was the target of a store instruction that hit in the cache.

Bus read hit	The block is changed from the exclusive state to the shared state because another processor requested read access to the block.
Clean	The block is changed from the modified state to the exclusive state because it was the target of a ***dcbst*** instruction or a bus operation that requested a clean. The modified data is written back to memory.
Invalidate	The block is changed from the exclusive, shared, or modified state to invalid because it was the target of a ***dcbi*** instruction or a bus operation that requested an invalidate. If the block was in the modified state, the modified data is discarded.

3.4.7 Cache Prefetching

When prefetching is enabled, the level 2 cache controller uses spare resources to move data into the level 2 cache by having each miss that completes start a prefetch reference on the cache block at the next higher address. Prefetches have lower priority than demand misses or stores when accessing the system bus and internal busses, but are otherwise implemented in a nearly identical fashion to demand misses—including sharing the same level 2 cache tag access resources. At any time, only one data address (demand or prefetch) and one instruction address can access the level 2 cache tags. If a new address arrives for a demand miss before the prefetch address completes its tag access, the prefetch request is dropped.

Prefetching stops when the requested data is already in the level 2 cache, after the last line on a physical memory page is fetched, when the L2CTL register is written, and when any TLB invalidate occurs. In addition, prefetches are never performed on guarded data pages, and all cache prefetching is disabled when the processor clock to bus clock ratio is less than 6:1.

The ***dcbt*** and ***dcbtst*** touch instructions are treated as prefetch requests that can be made even when data prefetching is disabled. Touch instructions are executed at the same priority as other prefetches. The touched data is never placed in the level 1 data cache. Unlike either demand misses or other prefetches, touch instructions that complete never cause further prefetch requests. Because touch instructions overwrite the previous contents of the level 2 cache tag access register, a sequence of touch instructions rarely results in all of the requested lines being brought into the cache. The most likely result is for the last reference in the sequence to be the only successful one.

3.5 Translation Lookaside Buffer (TLB)

The TLB contains 128 entries, organized as a 4-way set-associative cache; each can be used to map a virtual page address to a physical address. A total of 512KB of storage can be covered by TLB translations. Each TLB entry is a doubleword formatted as shown in Figure 22.

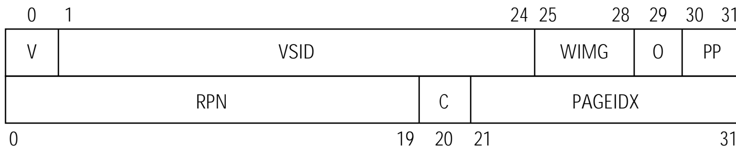


Figure 22: TLB Entry

The fields are defined as follows:

V	is the valid bit. The translation entry is valid if this bit is set, and invalid if it is clear.
VSID	is the virtual segment ID associated with this translation.
WIMG	are the storage access control bits for the page associated with this translation.
PP	are the page protection bits for the page associated with this translation.
RPN	is the page number of the physical page frame associated with this translation. The physical address for memory references using this translation is produced by appending bits (20:31) of the effective address to RPN.
C	is the page changed bit. If an instruction attempts to store to this page and this bit is clear, a TLB store interrupt occurs.
PAGEIDX	is bits (24:34) of the virtual address (bits (4:14) of the effective address) associated with this translation.

TLB entries may be written with diagnostic accesses at the addresses shown in Table 7 on page 84, or by using the TLBLRU and TLBMRF registers described in Section 2.3.4.4 on page 38.

A TLB match with an effective address occurs when both of these conditions are true:

1. The virtual segment ID in the segment register referenced by bits (0:3) of the effective address matches the VSID field of the TLB entry.
2. Bits (4:14) of the effective address match the PAGEIDX field of the TLB entry.

Effective address bits (15:19) index into the TLB, and do not participate further in the determination of a match. If no match is found for an effective address, a TLB miss interrupt occurs.

The TLB miss handler can write a specific entry in the set indexed by bits (15:19) of the effective address of an ***lwdx*** instruction, or it can use the TLBLRU0 and TLBLRU1 registers to write the least recently used entry in the set addressed by DAR(15:19). See Appendix A for an example of a TLB miss handler.

Software that writes a TLB entry should also set the Reference bit in the associated PTE.

Writing multiple TLB entries that translate the same effective address is an error and can cause a machine check or boundedly undefined results.

The TLB tracks usage history in each set using a 3-bit pseudo-LRU algorithm that works as follows:

- Bit 0 is set when entries 0 or 1 are used, and cleared when entries 2 or 3 are used.
- Bit 1 is set when entry 0 is used, and cleared when entry 1 is used.
- Bit 2 is set when entry 2 is used, and cleared when entry 3 is used.

Application of these rules yields the following state transition table:

Current State	State After Access to Entry			
	0	1	2	3
000	110	100	001	000
001	111	101	001	000
010	110	100	011	010
011	111	101	011	010
100	110	100	001	000
101	111	101	001	000
110	110	100	011	010
111	111	101	011	010

When choosing the LRU entry to replace, the TLB uses the following rules:

- If bit 0 is set, choose entry 3 if bit 2 is set, or entry 2 if bit 2 is clear.
- If bit 0 is clear, choose entry 1 if bit 1 is set, or entry 0 if bit 1 is clear.

These rules are embodied in the following table:

State	LRU Block
00x	0
01x	1
1x0	2
1x1	3

3.6 Instruction TLB (ITLB)

The ITLB consists of four 8-byte entries used to translate instruction addresses. Unlike the main TLB, the ITLB translates directly from effective addresses to physical addresses, skipping the virtual stage. As a result, the ITLB must be flushed each time a segment register or TLB entry is modified, including each time entries are modified by writes to the TLBLRU0, TLBLRU1, and TLBMRF registers or by a local or broadcast ***tlbie*** operation. The ITLB is maintained automatically by the hardware, which flushes it and refills it from the TLB as necessary.

3.7 Block Address Translation

The X[704] supports block address translation as defined in the *PowerPC Architecture Specification*. The instruction fetch unit contains four pairs of IBAT registers, and the load/store unit contains four pairs of DBAT registers. These registers can be accessed with ***mfspr*** and ***mtspr*** instructions using the defined SPR numbers or with diagnostic accesses.

3.8 Branch Prediction

The X[704] instruction fetch unit maintains branch prediction and branch target information in the finder. There is one finder entry for each doubleword in the instruction cache. Finder entries are written to a default value (as described in Section 3.8.3 on page 83) when a block is brought in to the instruction cache, and are updated by the decode unit as necessary. A finder entry holds information on the direction and target for a maximum of one branch instruction; if an aligned doubleword contains two branches, the finder entry describes only one branch at any given time. This condition can cause poor prediction performance as prediction information for each branch continually overwrites the information for the other branch.

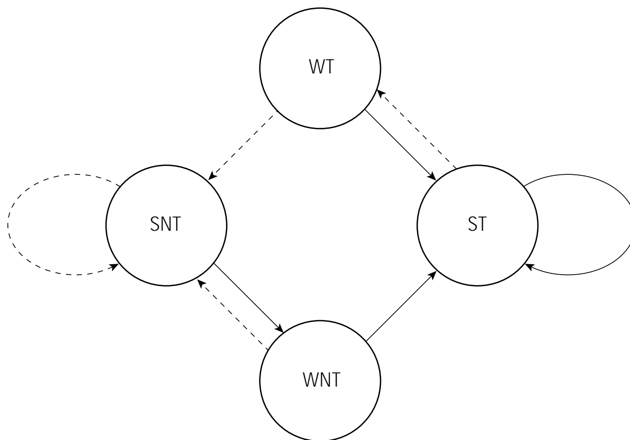
Branch prediction information is lost when a block is evicted from the instruction cache. Finder information can also be corrupted if a block containing a branch is evicted from the instruction cache while that branch is in the instruction buffer or the pipeline. If this happens, the decode unit can update the finder entry for the replacement block, even if there is no branch in that block, because its finder target address is a cache block index rather than a complete physical address. This can result in spurious mispredicted branches, but does not affect the correct execution of programs.

The finder can be accessed using the diagnostic address space so that the finder RAM can be tested by a power-on-self-test program. The width of the finder RAM is 14 bits and the data is right justified in bits (18:31) when read or written. The test program need not leave any particular value in the finder, but clearing each entry is recommended.

3.8.1 Branch Direction Prediction

The finder maintains two bits of branch direction information that represents four states: strong taken (ST), weak taken (WT), weak not taken (WNT), and strong not taken (SNT). When the state is either ST or WT, the branch is predicted taken, and when the state is either WNT or SNT, the branch is predicted not taken. Each time a branch is executed, the finder entry is updated as depicted in Figure 23.

Finder entries for absolute branches always predict the branches to be not taken.



Solid lines indicate taken branches and dashed lines indicate not taken branches.

Figure 23: Predicted Branch Direction State Transitions

3.8.2 Branch Target Prediction

For instructions that alter the program counter and are predicted to be taken, the finder value may specify that the new fetch PC comes from the link register, from the count register, from SRR0, or directly from the finder for branches with targets within the same 2KB block as the branch instructions or from the 2KB blocks immediately before and immediately after that block.

Relative branches whose targets are too far away for the finder to address are never predicted to be taken.

3.8.3 Finder Initialization

When a block is loaded into the instruction cache, the corresponding finder entries are initialized according to the following rules:

- If neither instruction is a branch, the finder is initialized to WNT.
- If both instructions are branches, only the first one is examined.
- ***b*** and ***bc*** instructions are initialized to WT if bits (16:21) of the instruction are either all zeros or all ones, and to WNT otherwise.
- ***rfi***, ***bclr***, and ***bcctr*** instructions are initialized to WT.
- The *y* bit in the BO field of conditional branch instructions is ignored.

For information on WT and WNT settings, see Section 3.8.1 on page 82.

3.9 Diagnostic Accesses

The X[704] allows diagnostic access to all of its internal RAM structures using the ***lwdx*** and ***stwdx*** instructions. These instructions use an alternate address space to address individual resources such as the caches, TLB, finder, and BATs. The diagnostic address space is defined in Table 7.

Table 7: Diagnostic Address Space

Address	Structure Accessed
`0000 0000 ---- ---- ---- xxxx xxxx xx00`	SPR[1]
`1000 0100 ---- ---- ---- -xxx xxxx x-00`	Finder
`1000 1000 ---- ---- ---- -xxx xxxx xx00`	Instruction Cache Data
`1000 1010 ---- ---- ---- -xxx xxx- --00`	Instruction Cache Tags
`1001 1000 ---- ---- ---- -xxx xxxx xx00`	Data Cache Data
`1011 1110 ---- ---- ---- -xxx xxx- --00`	Data Cache Tags
`1010 1000 -AAA ---- ---- xxxx xxxx xx00`	Level 2 Cache Data
`1010 1010 -AAA ---- ---- xxxx xxx- --00`	Level 2 Cache Tags
`1010 1110 ---- ---- ---- xxxx xxx- --00`	Level 2 Cache Use Records
`1011 1000 ---- ---x xxxx ---- ---A AL00`	TLB
`1011 1010 ---- ---x xxxx ---- ---- -L00`	LRU TLB entry

1. The entry address is the SPR number as used in ***mfspr*** and ***mtspr*** instructions.

In this table, an *x* represents a bit used to address an individual entry within a larger structure, an *A* selects among elements of a set in associative structures, and *L* is clear to select the more-significant half of a doubleword entry and set to select the less-significant half. Finally, a hyphen (**–**) represents an address bit that is ignored.

Only those SPRs implemented in the instruction fetch, load/store, and level 2 cache units can be accessed through the diagnostic address space. See Table 2 on page 36 for a list of SPRs implemented in those units.

When the cache is enabled, avoid diagnostic writes to any of the three caches, including the tags. An instruction that writes to the data or tags of an enabled cache has boundedly undefined results. Diagnostic writes to the data cache data RAMs must be followed by a ***sync*** or ***eieio*** instruction in order to ensure that the data is visible to subsequent load instructions and to ensure that the writes are done in order.

Diagnostic writes to the instruction cache can be suppressed by an ITLB miss. Hence, diagnostic writes of the instruction cache should be done with instruction address translation disabled (MSR[IR] clear). This conflict does not affect diagnostic writes to the IBATs, the finder, or other fetch unit structures.

A single diagnostic write to the TLB only writes half of a TLB entry. The two diagnostic writes required to write an entire entry must be executed with both instruction and data translation disabled (MSR[IR] and MSR[DR] clear). Diagnostic write accesses to the TLB with address translation enabled have boundedly undefined results.

Diagnostic writes to the TLB LRU space modify the least recently used TLB entry in the addressed set. The 3-bit LRU information described in Section 3.5 on page 79 is not directly accessible to software. If either data or instruction translation is enabled (MSR[IR] or MSR[DR] set), TLB LRU space accesses must be preceded by ***sync*** instructions in order to ensure that previous references have updated the TLB LRU information. If instruction translation is enabled, TLB LRU space accesses must be followed by ***isync*** instructions.

Diagnostic accesses to the finder should be performed only when the instruction cache is disabled (L2CTL.IE is clear) and branch prediction is disabled (MODES[BPE] is clear).

Diagnostic accesses to addresses not defined in Table 7 or to undefined entries in the diagnostic SPR space cause data storage interrupts. Undefined addresses are those where bits (0:3) differ from all of the entries in the table.

Diagnostic accesses never trigger data breakpoints, even if the diagnostic address matches the effective address in the DABR.

3.10 Power-On Reset and Hard Reset Initialization

When the $\overline{\text{HRESET}}$ signal is asserted, most of the processor state becomes undefined. The registers listed in Table 8 are initialized as shown. Execution begins at the system reset interrupt vector at address `0xfff00100`. All other processor state, including the general registers, cache tags, level 2 cache use records, and TLB entries, is undefined and must be initialized before use. The X^{704} has no power-on-reset circuitry, so the $\overline{\text{HRESET}}$ signal must be asserted after power is applied to the chip.

Table 8: Hard Reset State Initialization

Resource	Setting
MSR	0x00000040
SRR1	0x00000040
DEC	0xffffffff
PVR	0x0060rrrr[1]
BPTCTL	0x00000000
IABR[IE]	0[2]
EVENT	0x00000000
CHECK	0x00800000
MODES	0x00000000
L2CTL	0x00cc0000[3]
ITLB	All entries are invalid

1. The contents of the revision field is implementation-dependent.
2. All other bits of this register are undefined at reset.
3. The clock field is set from the PLL_CFG pins.

A soft reset, taken in response to the assertion of the $\overline{\text{SRESET}}$ signal, causes the same actions as a hard reset, except the CHECK[R] bit is cleared, the remaining fields of the CHECK register and BPTCTL are left unchanged. The $\overline{\text{SRESET}}$ signal is examined only if $\overline{\text{HRESET}}$ is not asserted.

For all resets, the cache tags and the level 2 cache use records must be initialized before the caches are enabled. The level 2 cache initialization software can also determine if any blocks are unusable, and should set the L2CDR register or the VALID and PLRU fields of use records to reflect any errors found. The TLB entries must be initialized before address translation is enabled. To ensure a deterministic reset, the finder should be initialized to all zeros before branch prediction is enabled. The TLB LRU information is initialized such that the behavior is deterministic and identical on each reset.

The MODES register is reset to a value that specifies the most conservative mode of operation. The system reset interrupt handler should enable pipeline overlap and superscalar execution as soon as possible. Branch prediction should be enabled as soon as the finder has been initialized.

4. Instruction Execution

This chapter describes the performance-related characteristics of the X[704]. There are four major components affecting the execution time of instructions:

- the inherent execution time of each instruction in the X[704], usually one cycle
- the parallelism available in the X[704]
- the interactions between instructions
- the ability of the caches to supply instructions and data to the pipelines.

These subjects are all covered in the following sections.

4.1 Pipeline Diagrams

The examples in this chapter make extensive use of simple pipeline diagrams. The progress of an instruction flow is shown in a horizontal line with a letter in each column indicating the current pipe stage. In the simplest case, a load instruction fetched and executed with no pipeline delays is depicted like this:

```
lbz   r1, (r2)        F   D   A   C   M   W
```

Instructions issued in the same group are represented by identical lines, since any pipe stall holds all instructions in a group. Instructions issued on succeeding cycles are offset to the right by one column for each cycle of delay. The F stage is omitted here; it is shown only in those diagrams where it is key to understanding the example.

```
lbz   r1, (r2)        D   A   C   M   W
addi  r3, r4, 1       D   A   C   M   W
sthu  r5, 4(r2)           D   A   C   M   W
```

In this example, the ***lbz*** and ***addi*** instructions are issued together, followed one cycle later by the ***sthu*** instruction.

Each column in a pipeline diagram represents the state of the machine at any point in time, so one can see that the store instruction is computing its address in A while the previous load is accessing the cache in C. Multiple occurrences of a stage in single lines of pipeline diagrams represent pipeline stalls. Even though instructions can spend several cycles in the instruction decode buffer waiting to be dispatched, only one D is shown except when demonstrating dispatch grouping rules.

4.2 Sliding ALU Stage

As described in Section 3.1.7 on page 69, the ALU, also known as X, pipe stage can be located in the A, C, or M pipe stages. In pipeline diagrams, the letter *x* is appended to a pipe stage name to denote the current location of the ALU stage. For example, a flow that executes an ALU operation in the C stage is depicted like this:

```
D    A    Cx   M    W
```

The X stage is initially located in A. This allows condition register flag values to be computed early in the pipe and thus reduces the penalty for mispredicted branches. When an ALU operand is not available in the A stage, X moves to a later pipe stage. This can happen with operands such as SPRs that cannot be bypassed, but occurs most frequently when load data is used by the following instruction, as in this example:

```
lwz   r1, (r2)      D    A    C    M    W
add   r3, r2, r1         D    A    Cx   M    W
```

The load instruction result is not available until the end of the C stage, and therefore cannot be used in the A stage of the ***add*** instruction. Instead, the ALU moves to the C stage of the ***add***. If the load and use had been issued in the same cycle, the X stage would move to M, as shown in this pipeline diagram:

```
lwz   r1, (r2)      D    A    C    M    W
add   r3, r2, r1    D    A    C    Mx   W
```

This example shows the effective load-use penalty of zero cycles. Once X has moved out to M, successive load-use pairings have no further effects. Consider this example:

```
lwz   r1, (r2)      D    A    C    M    W
add   r3, r2, r1    D    A    C    Mx   W
lwz   r4, 4(r2)          D    A    C    M    W
add   r3, r4, r3         D    A    C    Mx   W
```

The X stage is also moved out to M on all accesses to SPRs located in the decode or branch units, including the link register, count register, and condition register. For the condition register, this occurs only for accesses with instructions such as ***mtcrf*** and ***mfcr***, and does not occur for compare instructions or those that set CR0 or CR1 because Rc is set. For example:

```
add    r1, r2, r3     D   Ax  C   M   W
mtlr   r4                 D   A   C   Mx  W
addi   r1, r1, 4              D   A   C   Mx  W
```

After it has moved later in the pipe, the X stage remains where it is as long as instructions continue to use the ALU. At the first opportunity when the ALU is not being used, X returns to A. Compare the following two examples:

```
li     r0, 4          D   Ax  C   M   W
lwz    r1, (r2)       D   A   C   M   W
add    r1, r2, r1         D   A   Cx  M   W
subf   r3, r3, r1             D   A   Cx  M   W
```

and

```
li     r0, 4          D   Ax  C   M   W
lwz    r1, (r2)       D   A   C   M   W
add    r1, r2, r1         D   A   Cx  M   W
lwz    r3, 4(r2)              D   A   C   M   W
subf   r1, r4, r1                 D   Ax  C   M   W
```

In the second example, the second load instruction does not use the ALU, so the X stage can be moved back to A in time to execute the subtract instruction. (Assume that the ***lwz*** and ***subf*** instructions are not grouped because the instruction buffer was empty after the ***lwz*** was issued.) Notice that the ALU is used in consecutive cycles by the ***add*** and ***subf*** instructions; no ALU cycles are wasted.

The rules for moving the X stage back to an earlier stage in the pipeline are:

1. If the next group does not require the ALU and X is not already in A, move X back one pipe stage either from M to C or from C to A. If the next two flows do not require the ALU, X can move back from M to A in a single cycle.
2. On a mispredicted branch, X moves back to A.

The X stage is most likely to move back toward the A stage when the pipeline is empty because of a branch mispredict or an empty instruction decode buffer.

The later in the pipe X is located, the longer it takes for results to be available to other execution units. For performance, the most important factor is the impact of the availability of flag results on the cost of branch mispredicts, though the availability of ALU results for use as address generation operands is also important. If a branch is predicted correctly, there is no visible penalty even when X is all the way out in the M stage. For example, a correctly predicted compare and branch to a load instruction could look like this:

```
cmp   r1, r2        D   A   C   Mx  W
beq                 D   A   C   M   W
lwz   r3, (r4)          D   A   C   M   W
```

There is always a penalty for mispredicted branches; the important factor is that the penalty increases by one cycle for each cycle it takes to discover that the branch was mispredicted, which occurs no earlier than the cycle after the flag value is computed. If the branch in the previous example had been predicted incorrectly, this sequence would incur the maximum five-cycle branch mispredict penalty, as shown in the following pipeline diagram:

```
cmp   r1, r2        D   A   C   Mx  W
bcc                 D   A   C   M   W
<mispredict>                        ---
lwz   r7, (r8)                          F   D   A   C   M   W
```

If the X stage of the compare instruction had been in the A stage, the pipeline diagram would have looked like this, and the mispredict penalty would have been only three cycles.

```
cmp   r1, r2        D   Ax  C   M   W
bcc                 D   A   C   M   W
<mispredict>                ---
lwz   r7, (r8)                  F   D   A   C   M   W
```

If the flag value had been known when the branch arrived in A, the mispredict penalty would have been the minimum two cycles.

4.3 Branch Resolution

A conditional branch is *unresolved* until the value of the condition register flag it depends on is known. *Resolving* a branch consists of determining the direction of the branch, determining whether the branch direction and branch target address were predicted correctly, and flushing the pipeline and redirecting the fetch unit if either was incorrectly predicted. Each pipeline stage can contain an unresolved branch in any position in the instruction group. Each unresolved branch in the pipeline can be predicted to be either taken or not taken.

Unconditional branches must also be resolved. Even though the branch direction is known, the branch target address still needs verification. In this case, resolution need not wait for any particular flag to become available.

If a flag has not been set in some time, a conditional branch depending on that flag can be resolved in the A stage. In other cases, including the common case where the branch immediately follows the instruction that modifies the condition register, the branch can be resolved in the stage after the new flag value is computed. Only one branch, the oldest unresolved branch in the pipeline, can be resolved on each cycle.

Most flags are set by arithmetic instructions, and their values are available in the cycle after the X stage. Condition register logical instructions are executed in the branch unit and their results are not available until the W stage.

The following example illustrates branch resolution:

```
                              0   1   2   3   4   5   6
cmpwi  cr1, r3, 0             D   Ax  C   M   W
lwz    r2, (r1)                   D   A   C   M   W
cmpwi  r2, 0                      D   A   C   Mx  W
beq    .+40                       D   A   C   M   W
bgt    cr1,.+20                       D   A   C   M   W
```

Even though the value of CR1 is known before the second branch enters A in cycle 3, that branch cannot be resolved until it is in W during cycle 6. This is because the value of CR0 needed to resolve the first branch is not known until the end of cycle 4, preventing the first branch from being resolved until cycle 5; a later branch cannot be resolved before an earlier one.

Conditional branches that decrement and test the count register can be predicted unless they are preceded by an explicit load of the count register. In that case, the conditional branch cannot be resolved until the cycle after the ***mtctr*** instruction reaches the W stage.

4.4 Instruction Grouping Rules

The X^{704} is a superscalar processor. The decode unit can issue up to three instructions on each cycle, one to each of the following three pipelines:

ALU/Float	This pipeline executes all instructions that do not access memory and do not fall in the branch group. These are either integer arithmetic, logical, or shift operations, floating-point operations other than loads and stores, and some SPR accesses.
Load/Store	This pipeline executes instructions that access memory, including cache operations, synchronization, and diagnostic accesses that go through the data cache (***dcbz***, ***eieio***, diagnostic cache tag accesses, and so on). Most SPR accesses are handled by the load/store pipeline.
Branch	This pipeline executes conditional and unconditional branches including ***rfi***, ***sc***, condition-register logical, and ***isync***. These instructions have primary opcodes equal to 0b0100xx.

The only instruction that can be the third in a group is a PC-relative branch. There are no other position restrictions. Floating-point and integer operations cannot be executed in a single group. The load/store with update instructions use both the load/store and the ALU pipelines, thus preventing an ALU or floating-point instruction from issuing in the same group.

No instructions can be issued if the pipeline is stalled and there is at least one valid instruction in the A stage that cannot proceed down the pipe. In the absence of such a stall, the decode unit places instructions in the group to be issued until one of the following conditions occurs:

1. The instruction decode buffer is empty.
2. There are three instructions in the group being issued.
3. There are two instructions in the group being issued, and the next instruction in the instruction decode buffer is not a PC-relative branch.
4. The next instruction in the decode buffer uses the same pipeline as an instruction already in the group being issued.
5. The next instruction in the decode buffer writes the same register as an instruction already in the group being issued.
6. Either the next instruction in the decode buffer or an instruction placed in the issuing group is one of a class of instructions that must execute by itself.

This class comprises the ***dcbst***, ***dcbtst***, ***dcbz***, ***sync***, ***isync***, ***tlbie***, ***tlbsync***, ***lmw***, ***stmw***, ***lswx***, ***stswx***, ***lswi***, ***stswi***, ***lwarx***, ***stwcx.***, ***sc***, ***rfi***, ***mtcrf***, ***mfcr***, ***mfmsr***, ***mtmsr***, ***mftb***, ***mtsr***, ***mtsrin***, ***mfsr***, ***mfsrin***, ***mcrxr***, as well as ***mtspr*** and ***mfspr*** instructions that access privileged registers.

mtspr and ***mfspr*** instructions referencing the LR, CTR, and XER registers execute as ALU instructions. All other ***mtspr*** and ***mfspr*** instructions execute in a group by themselves.

7. The group being issued contains an ***mtspr*** instruction and the next instruction in the decode buffer is a conditional branch.
8. The next instruction in the decode buffer is a floating-point divide, any floating-point instruction with the Rc bit set, or a floating point status and control register instruction, and the group being issued is not empty.
9. The integer pipeline is not empty and the next instruction in the decode buffer is an ***lswx*** or ***stswx***.
10. The group being issued contains an instruction and either branch tracing or single-step tracing is enabled (MSR[SE] or MSR[BE] are set), or the processor is in single-issue mode (MODES[SI] is set).
11. The group being issued contains an instruction that takes an instruction storage interrupt, instruction fetch TLB miss interrupt, or instruction breakpoint trace interrupt or strobe pulse.

Multi-flow instructions use one D stage for each flow. No other instructions issue during these additional D stages. The load and store multiple and move assist instructions are multi-flow; they have one flow for each register accessed. An ***lswx*** or ***stswx*** instruction with a length of zero requires one flow. Each memory reference in a misaligned multi-flow instruction incurs additional performance penalties as described in Section 4.8 on page 99.

Only one instruction can be issued on the cycle following a mispredicted branch that occurred because of an invalid finder entry. Consider this pair of instructions issued in a single group:

```
14  beq . +40
54  add r0,  r1,  r2
```

where the finder indicates that the branch is taken and that the ***add*** instruction is also a taken branch. The invalid finder entry causes a mispredict back to PC 14. When the branch instruction is reissued, a group break occurs

before the ***add*** instruction. This break is required because only one finder entry can be updated on each cycle, and both instructions require finder entry modification. This situation is rare, and this group break has a negligible performance impact.

Because the fetch unit predicts branch target addresses and places instructions fetched from the branch target into the instruction buffer, there is no requirement that instructions executed in the same group be from sequential addresses.

4.5 Fetch Stalls

No instructions can be issued until they have been fetched into the processor. The fetch unit loads instructions into the instruction buffer unless any of the following conditions are present:

1. The fetch buffer portion of the instruction buffer is not empty.
2. The current fetch causes an ITLB miss.

   If the ITLB miss can be satisfied from the main TLB, there is a minimum four cycle penalty. If the miss cannot be satisfied from the main TLB and the processor attempts to issue the instruction at the offending address, a TLB miss interrupt occurs.
3. The current fetch PC misses in the instruction cache.

   An instruction cache miss has a minimum penalty of four cycles if the level 2 cache speculative access succeeds. The minimum penalty is five cycles if there is no speculative access, or if the speculative access fails.
4. The value in the appropriate register (CTR, LR, or SRR0) is not current, and a ***bcctr***, ***bclr*** or ***rfi*** indirect branch instruction in the instruction buffer is predicted to be taken.

   In this case, the fetch unit stalls until the instruction buffer contains three or fewer instructions and all instructions in the pipeline or instruction buffer, if any, that could modify the register containing the target address have completed.
5. An ***isync*** or ***rfi*** instruction is issued, the L2CTL register is written, or an interrupt is taken.

   In this case, the fetch unit resumes fetching instructions when all cache operations, synchronization instructions, and diagnostic writes have

been removed from the store queue. Diagnostic writes include modifications to the TLB or to SPRs that affect the context in which instructions addresses are interpreted. This stall ensures that these events are context synchronizing.

6. The instruction buffer has an associated four-entry queue that provides an entry for each instruction in the instruction buffer that is either marked as a branch in the finder or causes a trap known to the fetch unit.

   If this queue is full, an instruction requiring a queue entry cannot be put in the instruction buffer until the cycle after another entry is freed. Entries are removed from this queue when the associated instruction is issued. This queue rarely fills, and should not cause any performance degradation.

4.6 Decode Stalls

After the decode unit has determined how many instructions can be moved from the instruction buffer into the execution pipeline, it may discover that one or more of those instructions cannot be issued because of a resource conflict. Rather than attempt to determine if a smaller instruction group could be issued, the decode unit prevents the entire group from moving to the A stage.

Any of the following events cause decode stalls:

1. The instruction group contains a load or store with an address register operand that is being written by either a load instruction in the A stage or an ALU instruction that has not yet reached the X stage. This is known as an *address generation dependency*. For example:

   ```
   lwz   r1, (r0)     D  A  C  M  W
   lwz   r2, (r1)     D  D  D  A  C  M  W
   ```

   In this case, the second load instruction is delayed for two cycles: once by a group break because of a pipeline conflict with the previous instruction, and once by a stall because it may not reach the A stage until the result of the previous instruction can be bypassed from its C stage.

   Normally, the X stage occurs in A, so a sequence like the following does not cause an empty instruction group to be issued:

```
lwz    r1, (r0)       D   A   C   M   W
add    r3, r0, r7     D   Ax  C   M   W
lwz    r2, (r3)           D   A   C   M   W
```

If the X stage is farther out in the pipeline, multiple address generation stalls occur, as in the extra two-cycle delay in this example:

```
lwz    r1, (r0)       D   A   C   M   W
add    r3, r0, r1     D   A   C   Mx  W
lwz    r2, (r3)           D   D   D   A   C   M   W
```

2. A group containing an indirect branch instruction (***bcctr***, ***bclr***, or ***rfi***) cannot be issued while any instruction that modifies the register holding the target address is in the pipe. This is apparent in a sequence like the following:

```
mtlr   r0             D   A   C   M   W
blr                   D   D   D   D   D   A   C   M   W
```

3. A group containing a floating-point instruction cannot be issued unless the ALU X stage is in the A stage.
4. A group containing a floating-point instruction with the Rc bit set or a floating point status and control register instruction cannot be issued while there is a floating-point divide instruction in the execution pipeline.
5. A group containing a floating-point computational instruction cannot be issued while an instruction that sets FPSCR explicitly (***mtfsf***, ***mtfsfi***, ***mtfsb0***, and ***mtfsb1***) is in the pipeline but has not yet reached the W stage. This can cause a delay of as long as four cycles in issuing the next instruction.
6. A group containing a floating-point instruction cannot be issued while there is a single-precision floating-point load of a denormalized value in the W stage, and any instruction using the load/store pipeline is in the M stage.
7. No instructions can be issued while an ***stwcx.*** or ***sync*** instruction is waiting to complete in the W stage. See the entries for these instructions in Section 4.7 on page 97.

As shown in Section 4.2 on page 88, the sliding X stage eliminates most group breaking because of read-after-write register dependencies; only address operand dependencies cause group breaks. In addition, an ALU instruction that writes CR and a condition-register logical instruction (for example, ***cror***) can execute in the same group.

4.7 Pipe Stalls

Certain conditions cause an instruction to require multiple cycles in a single pipe stage, causing the pipeline to stall. When the pipeline stalls, instructions already in the pipe advance and instructions can be issued while there are empty stages behind the stalled instruction.

The following conditions cause the pipeline to stall:

1. A multi-step instruction uses multiple X stages. The various multiply and divide instructions are the only multi-step instructions.

   The ***mulhw*** and ***mulhwu*** instructions always take five steps and the ***mullwo*** instruction always takes six steps, but ***mullw*** and ***mulli*** take between three and five steps depending on the number of leading zeroes in the (RB) operand to ***mullw*** or the sign-extended immediate operand of ***mulli*** according to this table:

Number of Leading Zeros	Steps
16 or more	3
8 to 15	4
fewer than 8	5

   The ***divw*** and ***divwu*** instructions always take 37 steps.

2. If the X stage is in M, ***tw*** and ***twi*** instructions stall in X for a single cycle.

3. A load instruction in the C stage stalls for a single cycle if it addresses the same doubleword as a store in the M or W stage, or if it addresses the same cache block being supplied from the level 2 cache to the data cache on that cycle.

4. A load instruction in the C stage stalls while a store queue entry is being written to the data cache. A store instruction in the C stage stalls in this situation only if the store queue entry updates the data cache tags.

   This situation occurs when the store queue is full or when the hardware cannot determine whether it will become full. If the store queue were not full, advancing the pipeline would take precedence over writing a store queue entry to the data cache. This case is rare.

5. A store instruction in either the C stage or the W stage stalls for a single cycle if it hits the cache block being supplied from the level 2 cache to the data cache on that cycle.

6. Load instructions that cause data cache misses stall in the M stage until the target word is accessible. The minimum data cache miss penalty is three cycles if the level 2 cache speculative access succeeds. The minimum penalty is four cycles if there is no speculative access or if it fails.
7. If the store queue is full, a store instruction in the W stage stalls until a store queue entry becomes available.
8. A load or store instruction that uses the result of a misaligned load or load algebraic instruction as an address operand stalls in the A stage until that misaligned load or load algebraic instruction has exited the M stage. An ALU instruction that uses the result of a misaligned load or load algebraic instruction may also stall in X. See Section 4.8 on page 99.
9. A caching inhibited or diagnostic load stalls in the M stage until the target data is returned. Caching inhibited loads must also wait for all caching inhibited stores to be drained from the store queue, and diagnostic loads other than data cache data accesses must wait for all diagnostic stores to be drained from the store queue. Diagnostic accesses include reads and writes of SPRs implemented in the load/store, level 2 cache, and fetch units. A list of those SPRs can be found in Table 2 on page 36.
10. A ***dcbf***, ***dcbi***, or ***dcbz*** instruction in the M stage, W stage, or store queue stalls a subsequent load or store instruction to the same cache block index in the C stage until two cycles after the cache operation is removed from the store queue. Any load or store instruction in the C stage stalls for one cycle on the cycle after a ***dcbf***, ***dcbi***, or ***dcbz*** instruction is removed from the store queue. This stall also affects caching inhibited loads and stores.
11. The **sync** instruction stalls in the W stage until the store queue is empty and the level 2 cache has no operations in progress. All instructions in the pipe behind the ***sync*** will be re-issued when the ***sync*** completes.
12. The ***eieio*** instruction stalls in the W stage and holds subsequent load/store instructions in the C stage until the store queue is empty and the level 2 cache reports that all previous ***tlbie*** and ***tlbsync*** instructions have been broadcast on the bus.
13. The ***lwarx*** and ***stwcx.*** instructions stall in the M stage until all branches ahead of them have been resolved and they are known to be on the execution path.

14. The ***stwcx.*** instruction stalls in the M stage until the store queue is empty. When the store queue is empty, the conditional store operation is sent to the level 2 cache and the instruction stalls until the level 2 cache reports whether the store succeeded.
15. Branches that are not the last instruction in their group stall in the M stage until they are resolved.
16. The ***mfcr*** instruction stalls in the M stage for one cycle if the W stage is not empty.

Stalls caused by store instructions in the W stage are visible to the rest of the pipeline only when there is another load/store instruction in M. In that case, it appears as though the load/store instruction in M is stalling.

TLB misses are interrupts and do not cause pipe stalls. Pipeline stalls caused by floating-point instructions are covered in Section 4.9 on page 101.

4.8 Penalties for Algebraic and Misaligned Loads and Stores

Most load results can be bypassed from the cache output in the C stage directly to any stage that might need them. Some load instructions require extra processing that prevents this efficient bypassing. The load algebraic instructions require additional time to perform sign extensions, and cannot bypass their results immediately. Some misaligned loads require multiple accesses to the cache and must introduce pipe stalls. The following sections illustrate these penalties.

4.8.1 Pipeline Diagrams for Algebraic Loads

Load algebraic instructions access the cache as efficiently as other load instructions, but they cannot bypass their result from the C stage as other loads do. A subsequent instruction that uses the result of a load algebraic can stall, or the ALU can move out to a later stage. There is no decrease in the bandwidth of these instructions, so if the following instructions do not use their results, no penalty is incurred. For example,

```
lha   r1, (r2)      D   A   C   M   W
lwz   r3, 4(r2)         D   A   C   M   W
lwz   r4, 8(r2)             D   A   C   M   W
```

Similarly, a sequence of ***lha*** instructions executes with no stalls:

```
lha   r1, (r2)      D   A   C   M   W
lha   r3, 2(r2)         D   A   C   M   W
lha   r4, 4(r2)             D   A   C   M   W
lha   r5, 6(r2)                 D   A   C   M   W
```

If an ALU instruction needs the result of an algebraic load, the X stage stalls for one cycle waiting for the result, as though it were a one-cycle cache miss penalty:

```
lha   r1, (r2)      D   A   C   M   W
or    r3, r4, r5    D   Ax  C   M   W
add   r1, r1, r2        D   A   Cx  Cx  M   W
```

For a non-algebraic load, the result would have been available at the end of the C stage and could have been wrapped into the first C stage of the ***add***. The ***or*** instruction forces a pipeline conflict group break and also demonstrates that two cycles of possible ALU usage were lost. If the result of an algebraic load is needed to generate an address for the following instruction, that instruction is held in A for an additional cycle:

```
lha   r1, (r2)      D   A   C   M   W
lwz   r4, (r1)          D   D   A   A   C   M   W
```

These examples demonstrate that algebraic loads have the same bandwidth as other loads, but are encumbered by an additional cycle of latency.

4.8.2 Pipeline Diagrams for Misaligned Loads

The X704 executes most unaligned loads with no performance penalty. Unaligned loads that cross a doubleword (eight-byte) boundary require two instruction flows: one for each doubleword access to the data cache. This impacts performance. In this section, the term *misaligned* refers only to accesses that cross doubleword boundaries.

In pipeline diagrams, misaligned loads are shown with two A stages. The additional A stage represents both the C stage of the first cache access flow and the A stage of the second one. This extra flow can be viewed as an A stage stall that reduces the bandwidth of misaligned loads to one every two cycles. A sequence of misaligned loads looks like this:

```
lwz   r2, (r1)      D   A   A   C   M   W
lwz   r3, 4(r1)             D   A   A   C   M   W
lwz   r4, 8(r1)                     D   A   A   C   M   W
```

If an ALU instruction needs the result of a misaligned load, the X stage stalls for one cycle waiting for the result, as though it were a one-cycle cache miss penalty:

```
lwz    r2, (r1)        D   A   A   C   M   W
or     r3, r4, r5      D   Ax  Ax  C   M   W
add    r2, r2, r1              D   A   Cx  Cx  M   W
```

Again, the ***or*** instruction forces a group break for illustrative purposes only. If the load result is needed to generate an address for the following instruction, that instruction is held in A until the load result is available at the end of the M stage:

```
lwz    r2, (r1)        D   A   A   C   M   W
lwz    r4, (r2)                D   A   A   A   C   M   W
```

4.8.3 Pipeline Diagram for Misaligned Stores

Misaligned store instructions that cross a doubleword (eight-byte) boundary require two cache accesses and two A stages just as misaligned loads do. In addition, they require two M stages because the store data must be supplied to the data cache for each write. This causes a delay of one cycle to the following instructions, requiring an extra cycle in the A stage to complete their execution. The additional stalls reduce the bandwidth of misaligned stores to one every three cycles.

A pipeline diagram of a misaligned store followed by an aligned store and an unrelated ALU operation looks like this:

```
stw    r1, (r2)        D   A   A   C   M   M   W
stw    r3, (r4)                D   A   A   C   M   W
addi   r5, r6, 4               D   Ax  Ax  C   M   W
```

4.9 Floating-Point Execution

Unlike integer operations, all of the non-load/store floating-point instructions have multiple-cycle latencies. Portions of the floating-point unit are not pipelined, preventing some floating-point operations from issuing on every cycle. The following table shows the bandwidth and latency for all of the floating-point instructions.

Table 9: Floating-Point Instruction Bandwidth and Latency

Instruction	Bandwidth	Latency
fabs	1	4
fadd	1	4
fadds	1	4
fcmpo	1	4[1]
fcmpu	1	4[1]
fctiw	1	4
fctiwz	1	4
fdiv (typical)	34	35
fdiv (worst case)	140[2]	141
fdivs (typical)	20	21
fdivs (worst case)	66[2]	67
fmadd	2	5
fmadds	1	4
fmr	1	4
fmsub	2	5
fmsubs	1	4
fmul	2	5
fmuls	1	4
fnabs	1	4
fneg	1	4
fnmadd	2	5
fnmadds	1	4
fnmsub	2	5
fnmsubs	1	4
frsp	1	4
fsel	1	4
fsub	1	4
fsubs	1	4

1. This is the latency until a branch depending on the resulting flags can be resolved.
2. Worst-case divides are those requiring the maximum normalization of an operand and the maximum denormalization of the result.

The bandwidth column lists the minimum number of cycles between issues, where a value of one indicates that one instruction can issue every

cycle. The latency column shows the number of cycles that must elapse before the result of the instruction can be used as an input to another floating-point instruction.

4.9.1 Floating-Point Computational Instructions

The floating-point execution pipeline has four execution stages, called F1, F2, F3, and F4, in addition to the normal decode and writeback stages. The W stage of a floating-point operation normally occurs one cycle after the W stage of load/store or branch instructions issued in the same group. The latency of floating-point instructions is shown in this pipeline diagram:

```
fadd   fr1, fr2, fr3   D   F1   F2   F3   F4   W
fmul   fr4, fr1, fr5                      D    F1   F1   F2   F3   F4   W
```

The extra cycle of latency on double-precision multiplies is manifested by an additional F1 stage. This double use of a pipe stage means that double-precision multiply and multiply-add operations can be issued only every other cycle, as shown in this diagram:

```
fmul   fr1, fr2, fr3   D   F1   F1   F2   F3   F4   W
fmul   fr4, fr5, fr6            D    F1   F1   F2   F3   F4   W
fadd   fr7, fr8, fr9                      D    F1   F2   F3   F4   W
```

The ***fadd*** instruction in this diagram shows that any floating-point instruction following an instruction with a two-cycle bandwidth is delayed.

The result of a floating-point divide instruction cannot be bypassed when used as an operand to a following instruction; it must be written into the floating-point register file and then read out. Thus, the latency of the floating-point divider is one cycle longer than the issue rate. When all floating-point exceptions are disabled, the Rc bit in a divide instruction is clear, and the MODES[POE] bit is set, the divider operates asynchronously. In this case, a subsequent floating-point instruction can be delayed by one cycle to allow the divider to write a result into the register file. When the divider is operating synchronously, the entire pipeline stalls waiting for the divide result. In this case, the latency and issue bandwidth are identical.

4.9.2 Floating-Point Compare Instructions

A branch instruction that depends on a condition register field set by a floating-point instruction cannot be resolved until the W stage of the instruction that sets CR. A mispredicted branch in a group with a floating-point compare incurs a six-cycle penalty, as shown in the following pipeline diagram:

```
fcmpu  fr4,fr5        D   F1  F2  F3  F4  W
bcc    cr1            D   A   C   M   M   W
<mispredict>                              ---
add    r1, r2, r3                             F   D   A   C   MW
```

4.9.3 Floating-Point Load and Store Instructions

Floating-point load instructions have an extra cycle of load-use penalty when compared to fixed-point loads. In addition, there is no equivalent of the sliding X stage to absorb some of the load-use penalty. The load-use relationship is shown in this pipeline diagram:

```
lfd    fr0, (r4)      D   A   C   M   W
fadd   fr0, fr1, fr2              D   F1  F2  F3  F4  W
```

Single-precision floating-point loads of denormalized data incur an additional penalty of up to 23 cycles while they are converted to the double-precision format used in the floating-point register file. The normalization occurs in the W stage and stalls the entire pipeline. A single-precision floating-point load of the value zero requires one extra cycle of latency because the result cannot be bypassed until the processor can determine that zero is not a denormalized value. This extra latency does not affect the one issue per cycle bandwidth of floating point loads.

Floating-point store data must come directly from the floating-point register file; neither load data nor computational results can be bypassed. The result-store relationship is shown in this pipeline diagram:

```
fadd   fr0, fr1, fr2      D   F1  F2  F3  F4  W
stfd   fr0,(r1)               D   A   C   C   C   C   M   W
```

The result is read from the register file in the last C stage of the store, one cycle after it was written by the ***fadd*** instruction. If the result had come from a load of a denormalized single-precision value, the pipeline diagram would look like this (the lowercase *w* pipe stage represents a shift of a single bit of a normalization):

```
lfs    fr0, (r1)      D   A   C   M   w   w   w   W
stfd   fr0, (r2)          D   A   C   C   C   C   C   C   M   W
```

Single-precision floating-point stores of denormalized values are performed without any performance penalty.

4.9.4 Floating Point Exceptions and Condition Register Updates

When inexact, underflow, and overflow exceptions are enabled, an instruction group containing a floating-point computational instruction stalls in the M stage until the floating-point unit can determine whether an exception has occurred. An instruction group containing a floating-point computational instruction with the Rc bit set also stalls.

4.9.5 Floating-Point and Integer Pipeline Synchronization

The floating-point and integer pipelines are not tightly coupled; floating-point and load/store or branch instructions that issue in the same group do not necessarily proceed down the pipeline together. Table 10 shows the valid alignments of the two pipelines.

Table 10: Floating-Point and Integer Pipeline Alignments

	D	F1	F2	F3	F4	FW
D	•					
A		•	•	•	•	
C		•	•	•	•	
M		•	•	•	•	
W				•	•	•
—					•	•
—						•

This loose coupling allows one pipeline to proceed while the other is stalled. In the absence of stalls, paired instructions proceed on the diagonal path from D/D through A/F1, C/F2, M/F3, and W/F4; floating-point instructions continue on to the floating-point write (FW) stage, which is one cycle after the W stage for a fixed-point instruction. If any floating-point exceptions are enabled and there is a possibility that the floating-point instruction may take an exception, or if a floating-point instruction updates CR because the Rc bit is set, the group must pass through the M/F4 point to coordinate exception processing and condition register updates. This will always cause the integer pipe to stall for at least one cycle. Condition register updates resulting from floating-point compare instructions can be handled at the M/F3 point and do not cause a stall.

4.9.6 Optimizing Floating-Point Performance

To obtain the best floating-point performance on the X704, follow these guidelines:

- Disable all exceptions by clearing the five exception enable bits in the FPSCR.
- Do not set the Rc bit in any floating-point instructions.
- Explicit reads and writes of FPSCR with the floating-point status and control register instructions should be used sparingly.
- If programs are expected to generate denormalized numbers, they should be run in non-IEEE mode by setting FPSCR[NI].
- In floating-point code, fixed-point instructions should be scheduled so that the ALU X stage remains in the A stage. In particular, fixed-point loads should be separated from uses by two cycles.
- Instructions should be scheduled so that floating-point instructions do not immediately follow double-precision floating-point multiply or multiply-add instructions.

5. Signal Descriptions

The X^{704} supports the Basic Transport Protocol described in the *PowerPC 60x Microprocessor Interface Specification*. The X^{704} does not support the Extended Transfer Protocol described in that document. The 60x bus provides a 64-bit data bus and a separate 32-bit address bus, each with byte parity. The following sections describe the X^{704} processor interface in more detail.

5.1 Bus Interface Signals

Figure 24 illustrates the X^{704} bus interface signals grouped according to their functions. All interface signals except the clocks and those listed as configuration and test are described in the *PowerPC 60x Microprocessor Interface Specification*. The X^{704} does not support the $\overline{\text{XATS}}$ extended address transfer start pin, the $\overline{\text{SMI}}$ interrupt pin, and the CKSTP_IN and CKSTP_OUT check stop pins described in that specification.

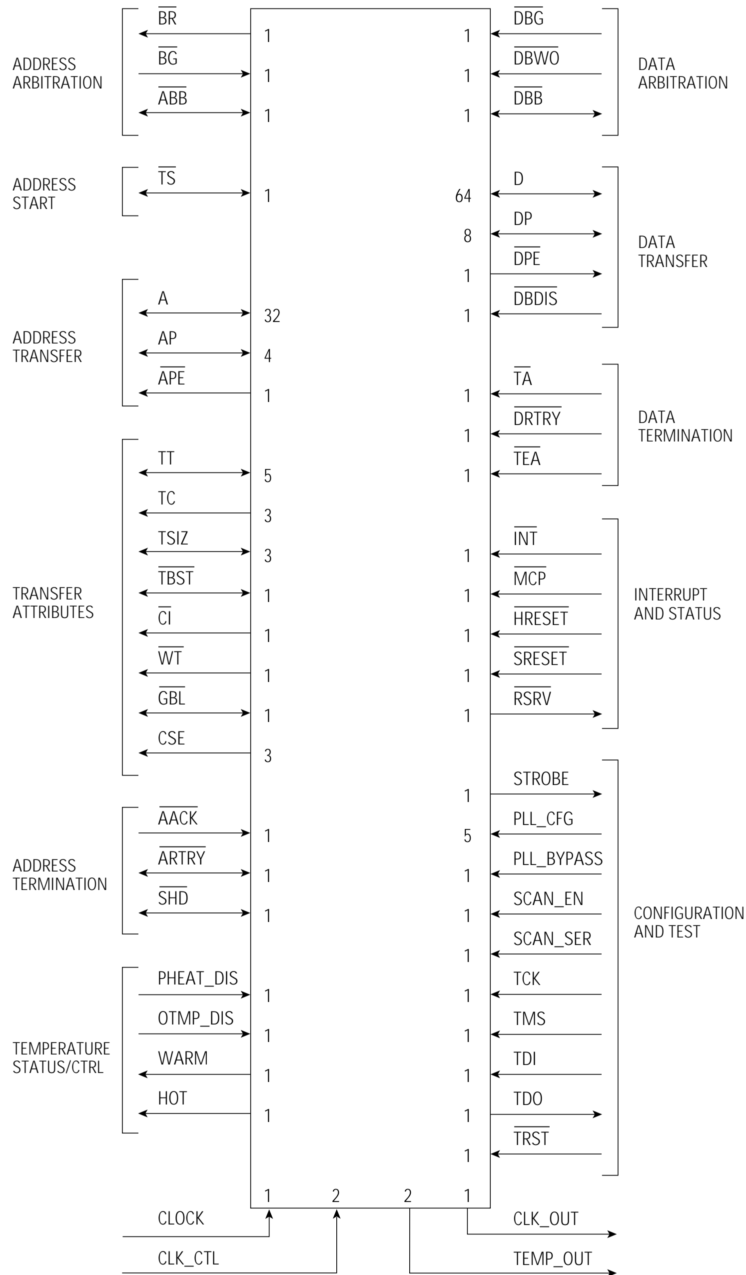


Figure 24: X704 Bus Interface Signals

5.2 Signal Descriptions

The following table describes the X^{704} processor-dependent bus interface signals.

Table 11: Processor-Dependent Signal Descriptions

Signal Name	Pins	Active	I/O	State Meaning	Timing Comments
STROBE Breakpoint strobe	1	N/A	O	Asserted/Negated—when breakpoint strobes are used, this pin indicates breakpoint hits. The signal is either asserted or negated on a breakpoint depending on the value of the L2CTL.SB bit.	Asserted/Negated—pulsed for one bus clock within three bus clocks after the breakpoint is triggered.
PLL_CFG PLL and clock configuration	5	high	I	Asserted/Negated—configures the ratio between processor and bus clocks. See Section 5.2.1 on page 111 for more information.	These pins may be changed only while the $\overline{\text{HRESET}}$ input is asserted.
PLL_BYPASS PLL disable	1	high	I	Asserted—disables the PLL and causes the CLOCK input to be passed directly to the internal clock signal.	These pins can be changed only while the $\overline{\text{HRESET}}$ input is asserted.
CLOCK System clock	1		I	Standard input clock sent to the PLL.	
CLK_CTL Clock control	2	high	I	Selects the internal clock tree source among the internal clock signal, TCK, and a speed test clock. See Section 5.2.1 on page 111 for more information.	See Section 5.2.1 on page 111 for more information on this signal.
CLK_OUT PLL test clockout	1		O	This pin is a 50% duty cycle clock that runs at half the frequency of the internal system bus clock PLL output. It is intended to be used as a heartbeat test.	
SCAN_EN Scan enable	1	high	I	Asserted—enables scan mode on all scannable flip-flops and disables all internal RAM write enables. Negated—All flip-flops and RAM write enables are in the normal operating mode.	See Section 7.2 on page 119 for more information on this signal.

Table 11: Processor-Dependent Signal Descriptions

Signal Name	Pins	Active	I/O	State Meaning	Timing Comments
SCAN_SER scan serial mode	1	high	I	Asserted—all scan elements are configured as a single scan chain. Negated—the scan elements are configured as multiple parallel scan chains.	See Section 7.2 on page 119 for more information on this signal.
HOT	1	high	O	Asserted—the die temperature has exceeded the maximum operating temperature. If the OTMP_DIS pin is not asserted, the processor has shut down. Negated—the die temperature is below the maximum operating temperature.	See Section 5.2.3 on page 113 for more information on this signal.
WARM	1	high	O	Asserted—the die temperature is near the top of the operating range. Negated—the die temperature is below the top of the operating range.	See Section 5.2.3 on page 113 for more information on this signal.
TEMP_OUT	2	analog	O	The voltage across these two pins is a measurement of the die temperature. Bit 1 of this signal is a ground reference for bit 0.	
OTMP_DIS	1	high	I	Asserted—the processor will continue operating when the maximum operating temperature is exceeded. Negated—the processor will shut down when its maximum operating temperature is exceeded.	See Section 5.2.3 on page 113 for more information on this signal.
PHEAT_DIS	1	high	I	Asserted—the processor will process reset interrupts as soon as they are detected. Negated—the processor will hold reset interrupts pending until the die temperature is above a minimum threshold.	See Section 5.2.3 on page 113 for more information on this signal.

Table 11: Processor-Dependent Signal Descriptions

Signal Name	Pins	Active	I/O	State Meaning	Timing Comments
TRST_ JTAG test reset	1	low	I	Asserted—resets the JTAG TAP controller.	This signal must be held asserted during normal chip operation. This signal must be asserted synchronously with TCK.
TCK JTAG test clock	1		I	JTAG scan and test clock	
TMS JTAG test mode select	1	high	I	Asserted/Negated—causes the TAP controller to change states as defined in the JTAG specification.	
TDI JTAG test data in	1	high	I	Asserted/Negated—carries the serial data input to the TAP controller.	
TDO JTAG test data out	1	high	O	Asserted/Negated—carries the serial data output from the TAP controller.	

5.2.1 Clock and Phase-Locked Loop Signals

The X^{704} receives an external system clock on the CLOCK input pin. The system clock frequency must be between 40MHz and 100MHz.

The X^{704} contains a phase-locked loop (PLL) referenced to the external system clock that generates the internal processor and bus clocks. The internal processor clock frequency is an integral multiple of the system clock frequency and can range from 350MHz to 650MHz in normal system operation. The internal bus clock is a copy of the system clock output by the PLL for use on-chip.

The internal processor clock to system clock ratio configuration information is encoded in the 5-bit PLL_CFG input. The values 1, 2, and 17 through 31 are reserved and may not be used. In all other cases, the system bus clock frequency is multiplied by one more than the value of this field to produce the processor clock frequency. For example, a value of 5 in PLL_CFG denotes a processor clock to bus clock ratio of 6:1.

Representative settings of PLL_CFG for typical system and processor clock rates are shown in Table 12.

Table 12: Typical PLL_CFG Settings

System Clock (MHz)	Processor Clock (MHz)	PLL_CFG Value
40	400	01001
50	500	01001
50	600	01011
60	420	00110
60	600	01001
66.7	400	00101
66.7	600	01000
80	400	00100
80	640	00111
100	400	00011
100	600	00101
100	700	00110

The use of a bus clock to processor clock ratio of 1:1 is allowed only for chip testing; the bus interface is not logically functional in this configuration.

If the PLL_BYPASS input pin is asserted, the PLL is disabled and the system clock input is passed directly to the internal processor clock distribution tree. The PLL configuration inputs are still used to create an internal bus clock, but system logic that requires a functional bus must include a clock divider to create an external bus clock that matches the internal one. Use PLL bypass mode for chip testing only.

The two-bit CLK_CTL signal selects alternate clock sources during test and scan operations. This signal is encoded as shown in the following table:

Table 13: CLK_CTL Settings

CLK_CTL	Clock distributed throughout chip
00	TCK, speed-test trigger enabled
01	scan speed test (See Section 7.3 on page 120)
10	PLL output or PLL bypass
11	TCK

In normal operation, CLK_CTL is set to 10. During scan testing, it is set to 11 to distribute the TCK JTAG test clock pin. When CLK_CTL is set to 00, the scan speed testing trigger is enabled. In this state, when a rising edge is detected on CLK_CTL(1) the internal clock is switched from TCK to the PLL output (or PLL bypass) clock for two internal clock cycles. The CLK_CTL(1)

edge is sampled on the rising edge of the CLOCK input. During scan speed testing, CLK_CTL(1) should be asserted for at least two external clock cycles. See Section 7.3 on page 120 for more information on scan speed testing.

5.2.2 Test Signals

The X^{704} provides the five interface signals needed to implement the IEEE 1149.1 JTAG standard. That standard should be consulted for information on the JTAG protocol. The X^{704} deviates from the standard by requiring the optional TRST test reset pin to be asserted synchronously with the TCK test clock.

The CLK_OUT pin provides a basic check of chip functionality. This pin outputs a 50% duty cycle clock running at half the frequency of the internal bus clock signal generated by the PLL. The PLL generates a bus clock even when PLL_BYPASS is asserted.

The SCAN_EN and SCAN_SER signals support manipulation of the internal scan chains. Use of these signals is described in Section 7.2 on page 119.

The STROBE pin indicates that instruction or data breakpoints have been triggered. This pin is intended to be used as a trigger for a logic analyzer. For more information on breakpoints, see Section 2.3.4.5 on page 41.

5.2.3 Thermal Monitoring and Control Signals

The X^{704} contains an internal temperature sensor unit that constantly monitors the internal die temperature. This unit supplies an analog output representing the current temperature and two digital outputs indicating whether the die temperature has exceeded either a *warm* or *hot* threshold. The hot threshold represents the maximum operating temperature of the part, and the warm threshold is set approximately 10°C below that point.

In order to prevent physical damage to the processor, the temperature sensor unit turns off the voltage reference generators when the hot threshold is exceeded, effectively cutting power to the chip and causing it to cease operating. This automatic cutoff can be disabled by asserting the OTMP_DIS input pin. Assertion of this pin is not recommended for other than testing purposes. Once an over-temperature shutdown has occurred, the processor cannot be restarted until the temperature drops below the warm threshold and either the $\overline{\text{HRESET}}$ or $\overline{\text{SRESET}}$ pin is asserted.

The WARM output is intended to be used by systems that can provide additional cooling capacity in high temperature situations, or as an indication to the system that an over-temperature shutdown may occur.

The digital threshold indications are provided to external system hardware on the WARM and HOT output pins and to system software as the WT and HT bits in the CHECK register.

The temperature sensor provides a *preheat* period before processing a reset interrupt. At lower temperatures, the processor requires higher voltages. By delaying a reset interrupt until the processor has reached a minimum operating temperature, a lower voltage can be used, reducing the amount of power dissipated by the processor. The preheat delay can be suppressed by asserting the PHEAT_DIS input pin.

The exact values of the preheat, warm, and hot temperature thresholds, the correlation between the TEMP_OUT analog outputs and die temperature, power dissipation, and the processor's voltage requirements will be supplied in a later version of this document.

6. Processor Interface

The X^{704} processor uses the PowerPC 60x processor interface standard. This standard contains several features that are implementation dependent. This section contains a summary of ways that the X^{704} interface may differ from other PowerPC processor interfaces.

- The X^{704} supports the read with no intent to cache (RWNITC), ***lwarx*** reservation set, ICBI, TLB invalidate, TLBSYNC, and EIEIO bus operations.
- The X^{704} does not support the external control word read and external control word write bus operations produced by the ***ecwix*** and ***ecowx*** instructions.
- The X^{704} does not support the extended transfer protocol (PIO).
- The X^{704} does not provide any power management signals, the $\overline{\text{SMI}}$ signal, or any checkstop signals.
- The X^{704}'s 8-way set-associative level 2 cache requires three cache set element (CSE) output signals.
- The X^{704} supports the multiprocessing features of the interface definition including the $\overline{\text{SHD}}$ signal, reservation cancellation on snooped read with intent to modify (RWITM) operations, and the suppression of snoops for transactions that do not assert the $\overline{\text{GBL}}$ signal.
- The X^{704} supports non-cacheable and write through ***dcbz*** operations.
- The X^{704} supports write through write atomic transactions.
- The X^{704} does not support a timebase enable input signal; the timebase is enabled by a field in the MODES register.

The following sections elaborate on the X^{704} bus interface implementation.

6.1 Address Bus

In order to maximize available address bus bandwidth, the X^{704} always asserts $\overline{\text{TS}}$ coincidentally with $\overline{\text{ABB}}$ and deasserts $\overline{\text{ABB}}$ on the cycle following $\overline{\text{AACK}}$.

Address and data parity errors detected by the X^{704} cause $\overline{\text{APE}}$ or $\overline{\text{DPE}}$ to be asserted even if the CHECK.BP bus parity machine check enable bit is clear.

The X^{704} will neither generate nor snoop the external control word read or external control word write TT encodings. The X^{704} does not generate the read with no intent to cache (RWNITC) TT encoding, but it will snoop global transactions of that type. When an RWNITC snoop is received, the $\overline{\text{ARTRY}}$ and $\overline{\text{SHD}}$ pins are asserted if necessary, and the cache state is changed to exclusive as for a clean bus operation.

6.2 Data Bus

On burst reads, the X^{704} can present any doubleword-aligned address in the block and expects to receive the addressed doubleword of data first, followed by the remaining doublewords in increasing address order, wrapping back to the beginning of the block if required. On burst writes, the X^{704} always transfers data beginning at the start (lowest address) of the block.

6.3 Coherency Protocol

On cycles where it is not driving the bus, the X^{704} snoops all bus transactions where the $\overline{\text{GBL}}$ signal is asserted. The X^{704} never snoops its own transactions or asserts $\overline{\text{ARTRY}}$ in response to its own transactions.

The $\overline{\text{ARTRY}}$ signal is asserted in response to the following conditions:

- A snoop hits a modified block, causing that block to be written back to memory.
- A snoop that might require a writeback arrives while an earlier snoop writeback is in progress. This snoop is retried even if no writeback is required.
- A bus SYNC operation arrives when a snoop writeback is in progress.
- A bus TLBSYNC operation arrives while the X^{704} has a pending operation based on a TLB translation that occurred before the most recent snooped TLB invalidation.
- A bus TLBIE operation arrives while the MSR[TW] bit is asserted.
- Resource contention prevented a snoop operation from accessing the cache tags in time to determine how to assert $\overline{\text{ARTRY}}$ and $\overline{\text{SHD}}$ accurately.

6.4 Features for Improved Bus Performance

The X^{704} implements the following performance-enhancing bus protocol extensions allowed by the bus specification:

- Optional disabling of the $\overline{\text{DRTRY}}$ signal, decreasing the minimum read latency by one cycle.
- Optional data-streaming mode, increasing the maximum bandwidth.
- Optional elimination of the $\overline{\text{ABB}}$ and $\overline{\text{DBB}}$ signals.

The $\overline{\text{DRTRY}}$ signal allows external cache and memory controllers to cancel a data transfer after it has already been sent to the processor. The processor must buffer data for a cycle to prevent it from being used before a transfer is canceled. This buffering adds a cycle to the read latency. Disabling the $\overline{\text{DRTRY}}$ signal eliminates this cycle of latency. There is a performance cost, however. In this mode, the earliest data transfer cannot occur until the first cycle of the $\overline{\text{ARTRY}}$ window, and not on the cycle before that as it can using the standard protocol.

Data streaming allows consecutive burst reads to appear on the bus without an intervening dead cycle. Do not use this feature unless $\overline{\text{DBB}}$ is disabled and the system arbitration logic asserts $\overline{\text{DBG}}$ for only the single cycle before a data transfer must start on the bus.

The $\overline{\text{DRTRY}}$ feature is disabled and data streaming is enabled when $\overline{\text{DRTRY}}$ is asserted along with $\overline{\text{HRESET}}$ in the hardware reset interval.

X^{704} processors recognize address tenures by tracking the $\overline{\text{TS}}$ and $\overline{\text{AACK}}$ signals and do not depend on $\overline{\text{ABB}}$ assertions. Assertions of $\overline{\text{ABB}}$ are always recognized and prevent a X^{704} from using a bus grant. The X^{704} will drive $\overline{\text{ABB}}$ during its address tenures.

The X^{704} does not require the $\overline{\text{DBB}}$ input if the system guarantees that $\overline{\text{DBG}}$ will only be asserted for the one cycle before its data tenure should start. $\overline{\text{DBB}}$ assertions are always recognized and prevent a X^{704} from using a data bus grant.

7. Test Interface

This chapter briefly describes the X^{704}'s test interface.

7.1 JTAG Interface

The X^{704} supports an IEEE 1149.1-compliant JTAG TAP interface that can be used to perform board-level testing. The TAP controller supports the JTAG boundary scan SAMPLE/PRELOAD, EXTEST, INTEST, and BYPASS instructions. The JTAG ID register is not supported. All X^{704} I/O pins except CLOCK, CLK_OUT, SCAN_EN, SCAN_SER, and the five JTAG interface signals appear on the boundary scan chain.

The TAP controller is not used to access the internal scan chains described in the next section.

7.2 Scan Chains

The X^{704} supports a single serial scan chain that includes every flip flop in the design. This scan chain can also be configured as 32 separate scan chains that can be accessed in parallel. The serial scan mode is intended for functional debug of prototype systems, while the parallel scan interface can be used for both functional debug and for manufacturing testing where high bandwidth scan is required.

The scan interface is driven entirely from input pins and does not use the JTAG TAP controller. It does make use of the TCK test clock and the TDI and TDO scan data input and output pins. To enable scan, the test device should assert SCAN_EN to place all internal flip flops in the scan configuration. Asserting the SCAN_EN pin also disables the write enables on all of the internal RAMs.

The SCAN_SER pin selects between the serial and parallel scan interfaces. When this pin is asserted, the flip flops are treated as one long scan chain with an input on the TDI pin and an output on the TDO pin. When SCAN_SER is not asserted, the flip flops are treated as 32 scan chains with inputs on one set of D bus data pins and outputs on another set of D bus data pins.

Two clocking mechanisms can be used during scan. If the PLL is in bypass mode, the CLK_CTL input can be set to 10 to use the CLOCK input pin to clock the scan chain. Alternatively, CLK_CTL can be set to 11 to select TCK as the source of the scan clock. In this case, the scan interface can be used while the PLL is running and synchronized to the CLOCK input.

7.3 At-Speed Testing

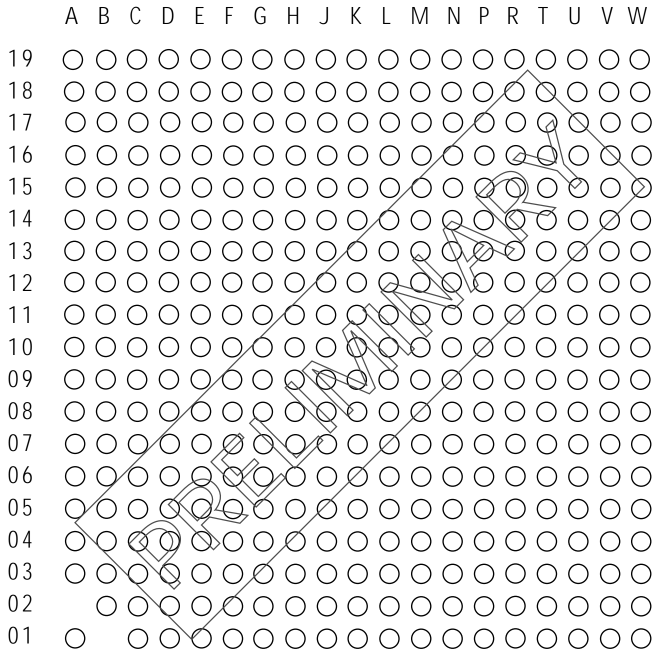
Because the X^{704} runs at internal clock rates greater than the speed at which a tester can supply vectors to the pins, some method other then simple external test vectors must be used to do speed fault grading. In order to meet this requirement, the X^{704} provides a special speed test feature. With the PLL running (PLL_BYPASS deasserted), the CLK_CTL input can be set to 00 to select the TCK clock and enable the scan speed test trigger.

After loading a test vector while TCK is selected as the clock, the tester clears SCAN_EN and sets CLK_CTL to 01 for at least two cycles of the CLOCK input, restoring the 00 value after that time. This causes the internal clock distribution logic to switch to the PLL output clock for two internal clocks, effectively running one processor clock cycle at the internal processor clock rate rather than the tester clock rate. The scan chain can then be scanned out of the chip by asserting SCAN_EN and clocking TCK to see if any faults occurred at speed.

8. Package Description

Part A

Part A shows the pinout of the P1 package as viewed from the bottom surface.



NOT TO SCALE

Part B

Part B shows the side profile of the P1 package to indicate the direction of the bottom surface view.

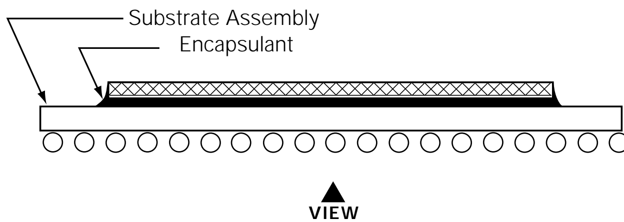


Figure 25: Pinout Diagram for the X^{704} Package

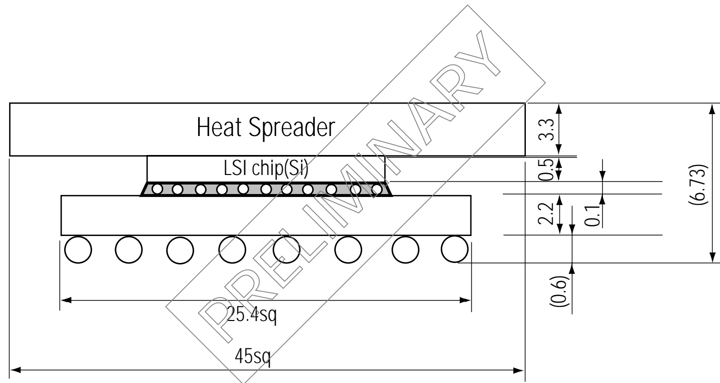
Table 14: Pinout Listing for the X^{704} Package

Signal Name	Pin Number
A0—A31	L01, M01, M03, K02, L02, L03, K01, N01, J01, H02, K03, J03, N02, G02, J02, M02, G03, F01, H03, E01, D02, F02, H01, F03, G01, E02, D01, E03, C01, C02, D03, C03
$\overline{\text{AACK}}$	C06
$\overline{\text{ABB}}$	V03
AP0—AP3	W04, V04, T09, T01
$\overline{\text{APE}}$	V08
$\overline{\text{ARTRY}}$	W08
$\overline{\text{BG}}$	A07
$\overline{\text{BR}}$	U09
$\overline{\text{CI}}$	W06
CLK_CTL0—CLK_CTL1	B16, A17
CLK_OUT	W03
CLOCK	C09
CSE0—CSE2	U06, P03, U01
$\overline{\text{DBB}}$	W07
$\overline{\text{DBDIS}}$	A16
DBG	A06
$\overline{\text{DBWO}}$	A05
DH0—DH31	U13, W14, V12, V16, W15, U15, R17, U16, V15, T17, U12, W17, R18, T18, M17, V17, V18, U17, P18, N18, P17, N19,U18, N17, T19, M18, K17, L18, L17, R19, R01, J17
DIODE_A	B06
DIODE_C	B07
DL0—DL31	K18, P19, K19, L19, M19, J19, J18, H18, G18, G19, F18, F19, F17, D19, C18, C19, T11, P02, U11, P01, V10, U14, V09, U08, V07, G17, D18, D17, E19, E17, E18, C17
DP0—DP7	V14, W16, V11, U19, H17, H19, U07, U10
$\overline{\text{DPE}}$	W10
$\overline{\text{DRTRY}}$	B04
$\overline{\text{GBL}}$	N03
HOT	C04
$\overline{\text{HRESET}}$	C11
$\overline{\text{INT}}$	B09
$\overline{\text{MCP}}$	A04
Not Connected	A01, A18, B03, B17, B18, B19, C05, C14, C15, C16, T07, T13, V01, V19, W02, W18
OTMP_DIS	A09

Table 14: Pinout Listing for the X^{704} Package (Cont.)

Signal Name	Pin Number
PHEAT_DIS	B05
PLL_BYPASS	C12
PLL_CFG0—PLL_CFG4	B14, C08, B11, A10, C07
$\overline{\text{RSRV}}$	T03
SCAN_EN	A12
SCAN_SER	A11
$\overline{\text{SHD}}$	W09
$\overline{\text{SRESET}}$	B15
$\overline{\text{STROBE}}$	U05
$\overline{\text{TA}}$	A08
$\overline{\text{TBST}}$	V13
TC0—TC2	T10, W12, U04
TCK	C10
TDI	B08
TDO	R03
$\overline{\text{TEA}}$	A03
TEMP_OUT	A14, C13
TMS	B10
$\overline{\text{TRST}}$	A13
$\overline{\text{TS}}$	W05
TSIZ0—TSIZ2	V02, U03, T02
TT	W11, W13, V05, V06, U02
Vc	D08, D11, E07, E10, E13, F09, F12, G08, G11, H07, H10, H13, J06, J09, J12, K08, K11, K14, L07, L10, L13, M09, M12, N08, N11, P07, P10, P13, R09, R12, T08
Vdd	D05, D14, D16, E04, E06, E15, F05, F14, F16, G04, G06, G15, H05, H14, H16, J04, J15, K05, K16, L04, L15, M05, M14, M16, N04, N06, N15, P05, P14, P16, R04, R06, R15, T05, T14, T16
VGc	D09, D12, E08, E11, F07, F10, F13, G09, G12, H08, H11, J07, J10, J13, K06, K09, K12, L08, L11, L14, M07, M10, M13, N09, N12, P08, P11, R07, R10, R13, T12
VGf	D07, D10, D13, E09, E12, F08, F11, G07, G10, G13, H09, H12, J08, J11, J14, K07, K10, K13, L06, L09, L12, M08, M11, N07, N10, N13, P09, P12, R08, R11
Vrd	B12
Vrn	B13
Vrp	A15
Vss	D04, D06, D15, E05, E14, E16, F04, F06, F15, G05, G14, G16, H04, H06, H15, J05, J16, K04, K15, L05, L16, M04, M06, M15, N05, N14, N16, P04, P06, P15, R05, R14, R16,T04, T06, T15
WARM	B02
$\overline{\text{WT}}$	R02

X^{704} Package Structure

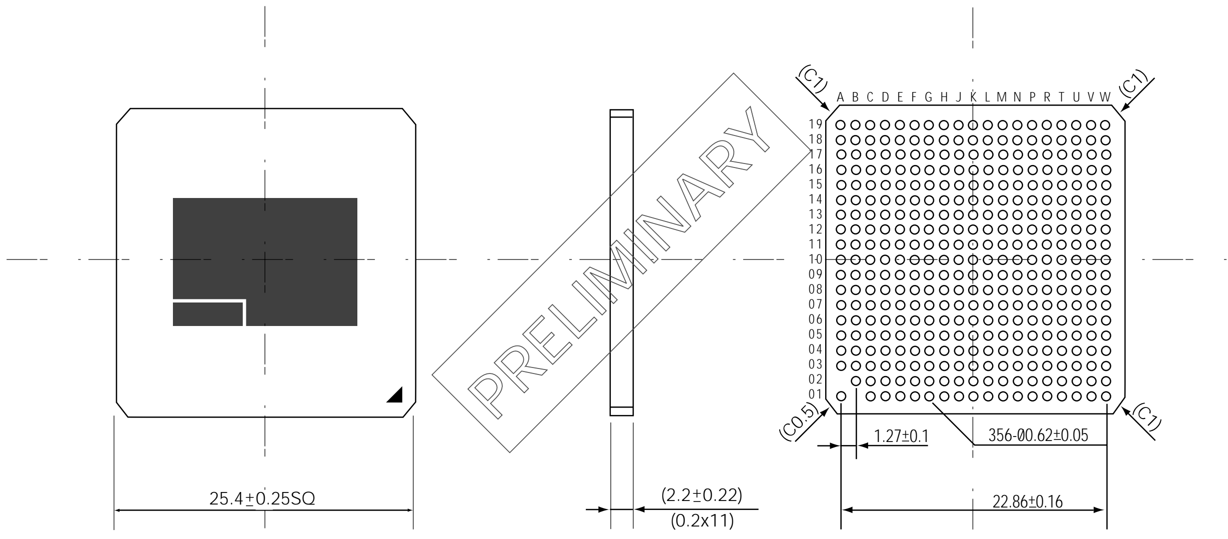


Note: All values are mm. All dimensions are nominal ± 10% tolerance.

Figure 26: X^{704} Package Structure

Figure 27: X704 Package Drawing

X704 C-BGA Base Design



359 pin C-BGA

Appendix A. Sample TLB Interrupt Handlers

This appendix contains sample handlers for the TLB miss and TLB store interrupts. These examples demonstrate the use of the X^{704}'s TLB miss SPRs and were written with more attention to clarity than to performance.

The TLB miss handler performs the following steps:

1. Saves some general registers and the counter register.
2. Initializes the PTE search from the CMP and HASH1 registers.
3. Searches both possible PTEG groups for the requested translation.
4. Uses the TLBLRU0 and TLBLRU1 registers to write the new translation into the TLB if the matching PTE is found.
5. Converts the TLB miss interrupt into the proper instruction storage or data storage interrupt if the matching PTE is not found.

```
! The TLB miss handler is called on both instruction and data TLB
! misses. The hardware writes MAR and MISR and saves CR0 in the
! upper bits of SRR1. The handler uses r29-31 and the counter
! register after saving them using information in SPRGs 4 and 5.

TLBMISS:                        ! handler at 0x1000
   mtsprg   5, r29              ! save r29 in SPRG5
   mfsprg   r29, 4              ! use r29 as pointer to save area
   stmw     r30, 0(r29)         ! save r30 and r31 in save area
   mfctr    r30                 ! save counter register
   stw      r30, 8(r29)         !     in save area

!  The magic CMP and HASH1 registers use the miss address saved in
!  MAR,the value in SDR1, and the contents of the segment register
!  indexed by the upper four bits of the address in MAR.

   mfspr    r30, cmp            ! upper half of desired PTE in r30
   mfspr    r31, hash1          ! addr of primary PTEG in r31
```

```
TLB_MISS_PTEG_SEARCH:
   li        r29, 8             ! loop over 8 PTEs in a PTEG
   mtctr     r29

TLB_MISS_LOOP:
   lwz       r29, 0(r31)        ! load upper half of this PTE
   cmplw     r29, r30           ! compare to desired entry
   beq       WRITE_TLB          ! found it!

   addi      r31, r31, 8        ! point to next PTE
   bdnz      TLB_MISS_LOOP      ! and try again

   andi.     r29, r30, 0x40     ! was this secondary search?
   bne       TLB_FAULT          ! if so, it's a storage interrupt
   mfspr     r31, hash2         ! otherwise, try secondary PTEG
   ori       r30, r30, 0x40     ! Set H=1 in target PTE upper half
                                !    and search secondary PTEG
   b         TLB_MISS_PTEG_SEARCH

! If we get here, the upper half of the PTE is
! in r29 and the PTE address is in r31.

WRITE_TLB_ENTRY:
   lwz       r30, 4(r31)        ! load lower half of PTE
   ori       r30, r30, 0x100    ! set referenced bit
   stw       r30, 4(r31)        ! and write it back

   mtspr     tlblru0, r30       ! write the TLB entry using CMP and
                                !    r30
   mtspr     tlblru1, r30

TLB_RETURN:                     ! restore registers
   mfsprg    r29, 4             ! get address of save area
   lwz       r30, 8(r29)        ! load old counter value
   mfsrr1    r31                ! SRR1 has saved CR0
   mtctr     r30                ! restore counter register
   mtcrf     0x80, r31          ! restore CR0

   lmw       r30, 0(r29)        ! load r30 and r31 from save area
   mfsprg    r29, 5             ! restore r29
   rfi                          ! return to faulting instruction

! If we get here, no translation is found, and we must convert
! this interrupt into either a data storage interrupt or an
```

```
! instruction storage interrupt.
! If it's a data storage interrupt, MAR and MISR must be copied to
! DAR and DSISR. If it's an instruction miss, we have to set up
! SRR1.

TLB_FAULT:
   mfmsr    r30               ! must reset MSR[TW]
   rlwinm   r30, r30, 0, 15, 13
   mtmsr    r30

   mfsprg   r29, 0            ! get address of save area
   lwz      r30, 8(r29)       ! restore counter register
   mtctr    r30

   mfspr    r30, misr         ! get MISR to look at type info
   andis.   r31, r30, 0x2000  ! was it instr TLB miss?
   mfsrr1   r31               ! get saved CR0 from SRR1
   bne      SETUP_ISI         ! it's an instruction storage fault

   mtdsisr  r30               ! copy MISR to DSISR
   mfspr    r30, mar          ! copy MAR
   mtdar    r30               ! to DAR
   mtcrf    0x80, r31         ! restore CR0
   lmw      r30, 0(r29)       ! restore r30 and r31
   mfsprg   r29, 5            ! and restore r29
   b        DATA_STORAGE_INTERRUPT

SETUP_ISI:
   mtcrf    0x80, r31         ! restore CR0
   lis      r30, 0x4000       ! set up SRR1 for page fault
   inslwi   r31, r30, 16, 0
   mtsrr1   r31
   lmw      r30, 0(r29)       ! restore r30 and r31
   mfsprg   r29, 5            ! and restore r29
   b        INSTR_STORAGE_INTERRUPT
```

The TLB store handler is similar to the TLB miss handler. An implementation that tried to minimize interrupt handler instruction cache usage could have the TLB miss and TLB store handlers share much of their code.

```
! The TLB store handler is invoked when a store references through
! a TLB entry with the C bit clear. The hardware guarantees that the
! faulting instruction had write permission to the page; if not, it
! would have invoked the data storage interrupt handler instead.
```

```
! The hardware writes MAR and MISR and saves CR0 in the
! upper bits of SRR1. The handler uses r29-31 and the counter
! register after saving them using information in SPRGs 4 and 5.

TLB_STORE:                      ! handler at 0x1100
   mtsprg   5, r29              ! save r29 in SPRG5
   mfsprg   r29, 4              ! use r29 as pointer to save area
   stmw     r30, 0(r29)         ! save r30 and r31 in save area
   mfctr    r30                 ! save counter register
   stw      r30, 8(r29)         ! in save area

! The CMP and HASH1 registers use the address saved in MAR,
! the value in SDR1, and the contents of the segment register
! indexed by the upper four bits of the address in MAR.

   mfspr    r30, cmp            ! upper half of desired PTE in r30
   mfspr    r31, hash1          ! addr of primary PTEG in r31

TLB_STORE_PTEG_SEARCH:
   li       r29, 8              ! loop over 8 PTEs in a PTEG
   mtctr    r29

TLB_STORE_LOOP:
   lwz      r29, 0(r31)         ! load upper half of this PTE
   cmplw    r29, r30            ! compare to desired entry
   beq      UPDATE_TLB          ! found it!

   addi     r31, r31, 8         ! point to next PTE
   bdnz     TLB_STORE_LOOP      ! and try again

   andi.    r29, r30, 0x40      ! was this secondary search?
   bne      TLB_STORE_FAULT     ! This shouldn't happen, O/S forgot
                                ! TLBIE after PTE invalidate?
   mfspr    r31, hash2          ! otherwise, try secondary PTEG
   ori      r30, r30, 0x40      ! Set H=1 in target PTE upper half
                                ! and search secondary PTEG
   b        TLB_STORE_PTEG_SEARCH

! If we get here, the upper half of the PTE is
! in r29 and the PTE address is in r31.

UPDATE_TLB:
   lwz      r30, 4(r31)         ! load lower half of PTE
   ori      r30, r30, 0x80      ! set changed bit
   stw      r30, 4(r31)         ! and write it back
```

```
    mfspr    r30, tlbmrf      ! load faulting TLB entry
    ori      r30, r30, 0x800  ! set changed bit
    mtspr    tlbmrf, r30      ! and write it back

    mfsprg   r29, 4           ! get address of save area
    lwz      r30, 8(r29)      ! load old counter value
    mfsrr1   r31              ! SRR1 has saved CR0
    mtctr    r30              ! restore counter register
    mtcrf    0x80, r31        ! restore CR0

    lmw      r30, 0(r29)      ! load r30 and r31 from save area
    mfsprg   r29, 5           ! restore r29
    rfi                       ! return to faulting instruction

! If we get here, we couldn't find the PTE that matches the TLB
! entry. This isn't supposed to happen. If it does, treat it like a
! TLB miss that turned into a page fault.

TLB_STORE_FAULT:
    mfmsr    r30              ! must reset MSR[TW]
    rlwinm   r30, r30, 0, 15, 13
    mtmsr    r30

    lis      r29, 0x4200      ! setup DSISR to be page fault on
                              ! store
    mtdsisr  r29
    mfspr    r29, mar         ! copy MAR
    mtdar    r29              ! to DAR

    mfsprg   r29, 0           ! get address of save area
    lwz      r30, 8(r29)      ! get saved counter
    mfsrr1   r31              ! get saved CR0
    mtctr    r30              ! restore counter register
    mtcrf    0x80, r31        ! restore CR0
    lmw      r30, 0(r29)      ! restore r30 and r31
    mfsprg   r29, 5           ! and restore r29
    b        DATA_STORAGE_INTERRUPT
```

Index

A

address translation 34, 73, 86
alignment 20, 27, 98, 100
ALU stage 88, 96

B

block address translation 54, 81
boundary scan 119
BPTCTL 42, 51
branch prediction 10, 19, 48, 81–83
branches
  indirect 94, 96
  mispredicted 88
  resolving 91
  unresolved 91
breakpoint
  data 21, 41–45, 59, 64
  instruction 41–44, 60, 64
  registers 41–45
bus interface 107–117
bus performance 117

C

cache operations 26, 50
caches
  coherency 32, 72, 77
  data 11, 25, 70–71
  enables 26, 51, 75
  flushing 75
  inclusion 26, 75
  instruction 8, 25, 69–70, 94
  level 2 14, 25, 72–76
  misses 94, 98
  prefetching 78
  replacement 26
change bit 27, 53, 54
CHECK 48, 57, 86
checkstop 57
CLK_CTL 112
clock 50, 111–113
CMP 38, 39
context synchronization 95
CR 89
CTR 10, 89

D

DABR 41, 45
DAR 59, 60
data cache. See caches, data
dcbf 30
dcbi 55
dcbst 29, 76
dcbt 78
dcbtst 78
dcbz 29, 42, 76
DEC 33
decode unit 10
denormalized numbers 23, 24, 104
diagnostic address space 84
direct-store segments 20, 53, 59

E

eciwx 18
ecowx 18
eieio 12, 32, 50, 98
exceptions
  floating-point 24, 69, 105
  inexact 24

F

fetch PC 9
fetch unit 8, 94
finder 8, 81–83
floating-point unit 13, 23, 101–105
flow, instruction 67
FPSCR 23, 96

G

group 67
guarded storage 19

H

HASH1 38, 40
HASH2 38, 40
HRESET 57, 85, 113

I

IABR 44
icbi 27
instruction buffer 8, 92, 94
instruction cache. See caches, instruction
instruction fetching 19, 26
instruction grouping 92–94
instruction issuing 10, 48
instructions
  dcbf 30
  dcbi 55
  dcbst 29, 76
  dcbt 78
  dcbtst 78
  dcbz 29, 42, 76
  diagnostic 30, 48, 84
  divide 97
  eciwx 18
  ecowx 18
  eieio 12, 32, 50, 98
  executing modified 32
  icbi 27
  indirect branch 96
  invalid 19, 21, 22, 25
  isync 28, 94
  lmw 21
  load algebraic 99
  lswi 20, 21, 60
  lswx 20, 21, 60
  lwarx 33, 98
  lwdx 18, 30, 48, 76
  mfspr 22, 35
  mtmsr 34
  mtspr 22, 35
  multi-flow 93
  multiply 97
  rfi 94
  stfiwx 24
  storage control 54
  stswi 20, 60
  stswx 20, 60
  stwcx. 59, 99
  stwdx 31, 48
  sync 12, 32, 50, 98
  tlbia 55
  tlbie 50, 55
  tlbsync 50, 55
INT 59
integer unit 10
interrupt 56–64
  alignment 60
  data storage 58
  decrementer 60
  external 59
  floating-point assist 61
  floating-point unavailable 60
  instruction storage 59
  machine check 57
  program 60
  system call 60
  system reset 57
  TLB miss 38, 53, 61, 127–129
  TLB store 38, 53, 62, 129–131
  trace 60, 64
interrupt priorities 64
interrupt vector 56
isync 28, 94
ITLB 8, 53, 81, 94

J

JTAG 113, 119

L

lmw 21
load/store unit 11, 20
LR 10, 89
LRU 26

lswi 20, 21, 60
lswx 20, 21, 60
lwarx 33, 98
lwdx 18, 30, 48, 76

M

machine check 48, 57
MAR 38, 61, 63
MCP 58
MESI 26, 72, 77
mfspr 22, 35
MISR 61
MODES 31, 33, 47
MSR 34
  BE 34
  DR 34, 61, 62
  IR 34, 61, 85
  LE 34
  ME 57
  SE 34
  TW 34
mtmsr 34
mtspr 22, 35

O

operand placement 27

P

performance 45, 87, 106
phase-locked loop 111
pipeline 48, 67–68, 87–101
  address generation 95
  load-use 88
  stalls 97–101
pipeline diagrams 87
PIR 52

R

reference bit 27, 53, 54, 80
register file
  floating-point 13, 23
  integer 10
registers
  BPTCTL 42, 51
  CHECK 48, 57, 86
  CMP 38, 39
  CR 89
  CTR 10, 89
  DABR 41, 45
  DAR 59, 60
  DEC 33
  FPSCR 23, 96
  HASH1 38, 40
  HASH2 38, 40
  IABR 44
  LR 10, 89
  MAR 38, 61, 63
  MISR 61
  MODES 31, 33, 47
  MSR 34
  PIR 52
  SDR1 38
  SPRG 35
  SRR0 34
  SRR1 34
  TBL 33
  TBU 33
  TLBLRU 38, 40
  TLBMRF 38, 41
  XDABR 44
reservation 21, 33
reserved fields 17, 33
reset 50, 85, 114
rfi 94

S

scan 119–120
SDR1 38
segment registers 38
SPRG 35
SRESET 57, 86, 113
SRR0 34
SRR1 34
stfiwx 24
store queue 11, 97
strobe 42, 51
stswi 20, 60
stswx 20, 60
stwcx. 59, 99
stwdx 18, 31, 48
superscalar 48, 92
sync 12, 32, 50, 98
synchronization 12, 32
  context 95

T

TBL 33

TBU 33

TEA 49, 57

temperature 113

TLB 53, 79–81
- invalidation 55
- replacement 80

TLB miss 38, 54

tlbia 55

tlbie 50, 55

TLBLRU 38, 40

TLBMRF 38, 41

tlbsync 50, 55

translation lookaside buffer. See TLB

U

use record 51, 72, 73, 86

X

XDABR 44