SN 8756

MC88110

32-bit Microprocessor Design Specification

Revision 2.0

MC88110

Motorola Confidential Proprietary
DO NOT COPY

CONFIDENTIAL INFORMATION TRANSMITTAL RECORD (Attachment "A")

Pursuant to Non-Disclosure Agreement # 89010052MD, Motorola Inc. has transmitted the following confidential information, MC88110 Design Specification

To: APPLE
Company name

By: ______________________
Motorola Representative

Received: ALLEN BAUM.

Title: ______________________

Date: ______________________

Date: ______________________

This document contains information on a new product. Specifications and information herein are subject to change without notice. Motorola reserves the right to make changes to any products herein to improve functioning or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under its patent rights nor the rights of others.

3.8 Memory Management Unit 91
3.8.1 Address Translation Caches 92
3.8.2 MMU Control 96
3.8.3 Translation Descriptors 97
3.8.4 PATC Miss and Refill 100
3.8.5 Cache/MMU Fault Conditions 106
3.8.6 Probe transactions 107
3.8.7 Breakpoints on logical addresses 109
3.8.8 Cache and MMU Control Registers 111
3.9 Bus Interface 127
3.9.1 Signal Description 127
3.9.2 Packaging 134
3.9.3 Bus Operation 135
3.9.4 Bus Timing Examples 141
A. - Appendix 163
A.1. - Exception Vector Table 163
A.2. - Control Registers 164
A.3. - Opcode Assignments 167

LIST of FIGURES

Figure 2.1.1.2. - cmp Extension for String Operations....9
Figure 2.1.3.4.a - Byte Ordering Mode....15
Figure 2.1.3.4.b - Heterogeneous Byte Order Environment....16
Figure 2.2.1 - Floating Point Data Formats....17
Figure 2.2.2.2.2 - Memory Storage Alignment....20
Figure 2.2.4.1 - Floating Point Compare Results....22
Figure 2.2.5.1 - IEEE Exceptions....25
Figure 2.2.5.2 - Floating Point Arithmetic Status and Control Registers....26
Figure 3.1.a - 88110 Block Diagram....27
Figure 3.1.b - Master Instruction Pipeline....28
Figure 3.2.4 - Processor Status Register....34
Figure 3.2.6.2.1 - Load Store Unit Block Diagram....38
Figure 3.3.1 - Floating Point Architecture....43
Figure 3.3.2.1 - Floating Point Exception Cause Register....44
Figure 3.3.3.1 - FPCR w/SLZ Control....48
Figure 3.3.4.3 - Floating Point Divide Latency....52
Figure 3.4.1 - 3-D Image Rendering....53
Figure 3.4.3 - Example Graphics Data Types....55
Figure 3.4.4.a - punpk.n....57
Figure 3.4.4.b - punpk.b....57
Figure 3.4.4.c - punpk.h....57
Figure 3.4.4.c - ppack.8....58
Figure 3.4.4.d - ppack.16....59
Figure 3.4.4.e - ppack.16....59
Figure 3.4.4.f - ppack.32....60
Figure 3.4.4.g - ppack.32.b....60
Figure 3.4.4.h - ppack.32.h....61
Figure 3.4.4.i - pmul....62
Figure 3.4.4.1.2 - Arbitrary Saturation Limits....65
Figure 3.5.1 - 3-D Graphics Transform....70
Figure 3.6.4.1 - Exception Latency....74
Figure 3.6.6.2 - NMI....76
Figure 3.7.1 - Instruction Cache Organization....79
Figure 3.7.2 - Data Cache Organization....81
Figure 3.7.2.1.a - Data Cache State Diagram....82
Figure 3.7.2.1.b - Data Cache Operation....83
Figure 3.8.1 - Address Translation Block Diagram....91
Figure 3.8.1.1 - Block ATC Entry....93
Figure 3.8.1.2 - Page ATC Entry....94
Figure 3.8.3.1 - Area Descriptor Format....98
Figure 3.8.3.2 - Segment Descriptor Format....98
Figure 3.8.3.3 - Page Descriptor Format....99
Figure 3.8.3.4 - Indirect Page Descriptor Format....100
Figure 3.8.4.1 - Hardware Tablewalk....101
Figure 3.8.4.1.1 - Page Descriptor Indirection....102
Figure 3.8.4.1.2 - Masked Protection Indirection....103
Figure 3.8.7 - Breakpoint Register Entry....110
Figure 3.8.8.a - Instruction MMU/Cache/TIC Command Register....111
Figure 3.8.8.b - IMMU/ICache Control Register....112
Figure 3.8.8.c - Instruction MMU ATC Index Register....114
Figure 3.8.8.d - IMMU BATC Read/Write Port....115

Figure 3.8.8.e - Instruction Access Status Register 117
Figure 3.8.8.f - Data MMU/Cache Command Register 119
Figure 3.8.8.g - DMMU/DCache Control Register 120
Figure 3.8.8.h - DMMU ATC Index Register 122
Figure 3.8.8.i - DMMU BATC Read/Write Port 123
Figure 3.8.8.j - Data Access Status Register 125
Figure 3.9.1 - 88110 Pinout 127
Figure 3.9.4.1.1 - Fast Single Beat Reads 141
Figure 3.9.4.1.2 - Fast Single Beat Writes 142
Figure 3.9.4.1.3 - Single Beat Reads with Waits 143
Figure 3.9.4.1.4 - Single Beat Writes with Waits 144
Figure 3.9.4.1.5 - Single Beat Reads with Data Bus Grant 145
Figure 3.9.4.1.6 - Single Beat Writes with Data Bus Grant 146
Figure 3.9.4.1.7 - Single Beat Read/Write/Read Sequences 147
Figure 3.9.4.1.8 - Non-pipelined xmem 148
Figure 3.9.4.1.9 - Invalidation vs. Read vs. Write 149
Figure 3.9.4.2.1 - Fastest Burst Transfers 150
Figure 3.9.4.2.2 - Burst Read with Waits/DBG* 151
Figure 3.9.4.2.3 - Burst Write with Waits/DBG* 152
Figure 3.9.4.2.4 - Burst Read in Half-Speed Mode 153
Figure 3.9.4.2.5 - Non-pipelined Burst Transfers 154
Figure 3.9.4.2.6 - PTA* Examples 155
Figure 3.9.4.2.7 - Pipelined using DBB* Data Tenure Hand-off 156
Figure 3.9.4.3.1 - Non-Pipelined Error Termination 157
Figure 3.9.4.3.2 - Non-Pipelined TRTRY* Termination 158
Figure 3.9.4.3.3 - Pipelined TRTRY* Termination 159
Figure 3.9.4.4.1 - Non-Pipelined Snoop Hit ARTRY* 160
Figure 3.9.4.4.2 - Pipelined Snoop Hit ARTRY* 161
Figure 3.9.4.4.3 - Open Pipe Snoop Hit and Collision ARTRY* 162
Figure A.3.a - Immediate Opcodes 167
Figure A.3.b - Control Register Access and SFU1 Opcodes 168
Figure A.3.c - SFU2 Graphics Opcodes 169
Figure A.3.d - Branch, Bit Field, and Memory Opcodes 170
Figure A.3.e - Triadic Operations Opcodes 171

LIST of TABLES

Table 2.1.1.1.6 - Summary of 88000 Integer Arithmetic....8
Table 2.1.1.3 - bcnd Prediction....10
Table 3.1.1.3 - Simultaneous Instruction Issue....30
Table 3.3.2.2 - Floating Point Exceptions....45
Table 3.3.2.3 - Floating Point Exception Mapping....46
Table 3.3.3.1 - SLZ Mode Actions....49
Table 3.3.3.2.a - SLZ Mode Default Results....50
Table 3.3.3.2.b - Recognized and Generated Values....50
Table 3.4.4.a - ppack Field Size and Rotation....58
Table 3.4.4.b - pcmp Results....61
Table 3.4.4.1.1 - 8-bit Saturation Examples....64
Table 3.5.a - SFU0 Instruction Timing Summary....67
Table 3.5.b - SFU1 Instruction Timing Summary....67
Table 3.5.c - SFU2 Instruction Timing Summary....68
Table 3.5.d - SFU0 Flow Control Instruction Timing Summary....68
Table 3.5.e - SFU0 Memory Instruction Timing Summary....69
Table 3.5.f - Control Register Instruction Timing Summary....69
Table 3.8.2 - MMU Address Translation Modes....96
Table 3.8.8.a - Instruction BATC Block Size Selection....113
Table 3.8.8.b - Data BATC Block Size Selection....121
Table 3.9.1 - Pin Descriptions....133
Table 3.9.3.7 - Termination Cases....139
Table A.1 - Exception Vector Table....163
Table A.2.1 - SFU0 Control Registers....164
Table A.2.2 - SFU1 Control Registers....165

1. INTRODUCTION

This document specifies the extensions being made to the 88000 architecture in general and the implementation details of the 88110 advanced RISC processor specifically. Features not addressed herein can be assumed to be the same as those defined for the 88100/200 microprocessor in the 88100 and 88200 User's Manuals. It is assumed that the reader is familiar with those documents.

1.1. 88110 OVERVIEW

The 88110 is a second-generation Motorola 88000 family RISC microprocessor incorporating new architectural and implementation features. The processor utilizes a high level of integration and an advanced microarchitecture to deliver sustained performance better than one instruction per clock. It is a complete stand-alone microprocessor capable of performing all system and application level operations, many of which previously required dedicated processors (e.g. graphics). Emphasis has been placed on achieving a balanced design to maximize overall system performance. Hardware multiprocessor features make the 88110 ideal for systems employing multiple tightly coupled processors. Bus bandwidth is sufficient to directly support 2 processors into DRAM memory. Larger numbers of processors can be configured using external hardware to provide additional memory bus bandwidth.

1.2. 88110 FEATURES

- Two instructions issued per clock
- Multiple independent function units:
  - Two 32-bit integer arithmetic and logical units
  - One 32-bit bit-field unit
  - Three 80-bit extended precision floating point units (add, multiply, divide)
  - Two 64-bit 3-D graphics units (add, pack)
- 32, 32-bit general purpose registers and 32, 80-bit extended registers
- 8K Byte, 2-way set associative, physically addressed instruction cache
- 8K Byte, 2-way set associative, physically addressed data cache
- Dynamic reordering of loads and stores at runtime
- 64-bit pipelined external data bus with burst transfer capability
- Hardware enforced data cache coherency (bus snooping) for multiprocessors
- Critical-word-first burst cache line fills with instruction and data streaming
- 32-entry branch target instruction cache and static branch prediction
- 40-entry instruction address translation cache
- 40-entry data address translation cache
- Page address translation - for demand paged virtual memory
- Block address translation - for mapping large contiguous blocks of memory
- Hardware or software address translation cache refill
- Logical Address Breakpoint registers for software debugging
- JTAG boundary scan for in-system testability
- TAB and PGA packaging

The 88110 is a true "superscalar" design capable of executing multiple instructions per clock from a conventional linear instruction stream. The machine attempts to fetch, decode, issue, and execute two instructions on every clock with no instruction type or alignment restrictions imposed on software to achieve two instruction-per-clock throughput.

The design consists of multiple concurrent and independently pipelined function units which share a common set of registers. All data dependency and function unit contention is fully interlocked and resolved by hardware. Instructions from different execution units may finish out of program order but register file updates are automatically reordered into strict program issue sequence. Internal machine pipelines are not exposed to the program. This avoids the need to insert NOP instructions which needlessly waste instruction bandwidth and cache entries. It also assures object code compatibility with past and future generations of 88000's.

All integer, logical, bit-field, and graphics function units execute instructions in a single clock cycle. Operands for these function units come from and return to the 32-word general purpose register file (GRF). The GRF has sufficient bandwidth to support a sustained execution rate of two instructions per clock.

In the 88110, floating point operations are performed nearly as fast as integer operations. There are three independent and concurrent floating point execution units; one for multiplication, one for addition, and one for division. The floating point multiply and add function units are fully pipelined so that new instructions can be issued to each unit on every clock. The floating point units run concurrently with all other function units on the chip and any mix of integer, memory, and floating point operations can be issued together in the same clock without restriction. An extended register file has been added to provide additional storage for floating point operands. The extended register file (XRF) holds 32 values of any precision (single, double, or double-extended). The XRF has sufficient bandwidth to support a continuous throughput of two floating point operations per clock.

The 88110 also implements a set of graphics instructions to accelerate rendering of 3-D images. Graphics instructions perform operations on multiple fields within their 64-bit operands in parallel. There are two independent and concurrent graphics execution units; one for pixel addition, subtraction, and comparison and another for pixel packing, unpacking, and rotation. Both units execute instructions in a single clock. Pixel multiplication is performed using the multiplier and can run concurrently with pixel addition and packing.

The instruction cache delivers two instructions per clock to the instruction unit on a hit and streams instructions directly from the bus to the IU on a miss. The data cache can accept or deliver a 64-bit operand to the register files every clock on a hit. Cache misses are serviced by filling the missed cache line with a burst bus transaction which begins with the address of the missed data (or instruction). Data cache coherency is maintained among multiple processors by snooping all bus transactions for potential conflicts. 88110 caches employ a separate set of cache tags for snooping so that it does not interfere with normal processor cache accesses.

Load and store instruction buffers preceding the data cache and the bus interface unit help hide memory latencies on cache misses. Store instructions can issue into the store buffers even before store data is available from a previous computation. Load instructions can bypass store instructions which are stalled in the buffers

awaiting data. Potential memory hazards due to this dynamic reordering of loads and stores is prevented by hardware interlocks. The adverse effects of long memory latency can be further avoided by explicitly scheduling data into the cache ahead of when it will be needed using the "touch" instruction provided for that purpose.

The 88110 implements a precise exception model to simplify and accelerate exception handling. Although instructions finish out of program sequence, the machine will automatically empty all internal pipelines and back up to the precise machine state that existed when the faulting instruction issued.

2. 88000 ARCHITECTURE

This section should be thought of as an addendum to the "MC88100 RISC Microprocessor User's Manual." It describes changes and extensions to the base 88000 architecture as specified in that manual. Features specified herein are to be considered permanent changes to the architecture and will be used as the basis of all future implementations of the 88000, beginning with the 88110.

The *Architecture* is defined to be: "the instruction set and programming model visible to user mode assembly language programs." The *architecture* defines the lowest common denominator for all 88000 processors, i.e. the minimum set of features which must be present in any implementation in order for that implementation to qualify as member of the 88000 family. Intentionally excluded from the definition of *architecture*, are supervisor mode instructions, internal machine registers, the exception handling model, caches, and memory management facilities. These are not considered part of the base architecture and are subject to change from processor to processor.

An *Implementation* is a specific embodiment of the architecture, e.g., the 88100 and 88110 are implementations of the 88000 architecture. An implementation of the architecture can be done in hardware, software, or a combination of both, but all implementations must support all architectural features as a minimum. If a hardware implementation chooses to implement an architectural feature in software, it must do so in a way which is not functionally apparent to user mode programs in any way other than performance.

An *Implementation Dependent User Mode Feature* is a new class of features introduced with the 88110. Such features are visible directly to user mode code but are NOT considered part of the 88000 architecture. It was recognized that some performance features need to be directly accessible to user mode code but these features may be too implementation specific or subject to change on the next generation to include in the base architecture. Therefore, a distinction is drawn between architecture and implementation dependent user mode features as a warning to confine the use of such features to library routines which can be unique for each implementation. Their existence should not be exposed to general purpose compilers or used in object code which must be portable across multiple implementations. Implementation dependent user mode features will be clearly identified as such in this document.

The 88000 is designed around the concept of Special Function Units (SFU's). A special function unit is an independent design module with reserved instruction opcodes and a defined control mechanism which allows it to share resources and communicate efficiently with the main processor. Typically, an implementation will construct an SFU as one or more independently pipelined execution units which run concurrently with other SFU's and sharing the common set of registers and the existing instruction interlock facilities (scoreboard). There are 8 SFU's defined. SFU0 implements the "base architecture" and is present in all implementations. Other SFU's are optional and may or may not be included in a given implementation.

Each SFU has its own "architecture." In the 88100, SFU1 was dedicated to floating point operations. Since any SFU, other than SFU0, is optional, floating point operations are not strictly a component of the base architecture. However, floating point operations are becoming increasingly important, so it is anticipated that many, if not most, future implementations of the 88000 will support floating point in hardware. To preserve software compatibility, it is recommended that an implementation which does provide floating point, do so in the manner prescribed by the 88100 User's manual. In order to facilitate compatibility, we define the 88000 *floating point architecture* and henceforth reserve Special Function Unit One (SFU1) in the base architecture, exclusively for it.

As the 88000 architecture evolves over time, extensions may be added to the base architecture which are not present in earlier implementations. As such extensions are added, we will strive to provide "user-code forward compatibility" to insure an uninterrupted growth path for our customers; but we do not insist on "user-code backward compatibility" so that the 88000 architecture has the flexibility necessary to grow with the technology.

This section specifies extensions and modifications to the 88000 base architecture and the SFU1 architecture. These will be embodied in the 88110 and all future 88000 implementations. The extensions provide additional flexibility and functionality to the architecture. The modifications have been made to facilitate the architectural extensions; the modifications are minor and in most cases will not cause code incompatibility with the 88100.

2.1. BASE ARCHITECTURE

2.1.1. BASE ARCHITECTURE EXTENSIONS

Features specified in this section are all completely upward compatible extensions to the original 88000 architecture and should not exhibit any incompatibility with 88100 programs.

2.1.1.1 New Arithmetic Instructions

The following instructions, or instruction extensions, are added to the 88000 base architecture:

2.1.1.1.1 Signed Integer Multiply

The original 88000 architecture provided unsigned 32x32⇒32 bit multiply in both 16-bit immediate and triadic register addressing forms. An unsigned multiply is sufficient for address arithmetic but a signed multiply is more useful for general integer arithmetic. Therefore, a signed 32x32⇒32 bit multiply is added to the architecture.

muls rD, rS_1, rS_2

The signed 32-bit value in register rS_1 is multiplied by the signed 32-bit value in rS_2. If the product cannot be represented as a signed 32-bit result, an overflow exception is taken and no result is written into rD.

2.1.1.1.2 Unsigned Integer Multiply Mnemonic

The official 88100 mnemonic **mul** will be changed to **mulu** to be consistent with the convention that unsigned operations (those which do not generate overflow exceptions) end in **u**, e.g. **addu**, **subu**, **divu**. The **mul** mnemonic will continue to exist as a synonym (alias) of **mulu** for some period of time so that code generators need not change overnight.

2.1.1.1.3 Unsigned Integer Multiply Double

To further facilitate high precision integer arithmetic a 32x32⇒64 bit unsigned multiply is added to the architecture. This instruction simplifies general integer arithmetic because the result is guaranteed to fit in the destination.

mulu.d rD, rS_1, rS_2

The 32-bit unsigned value in register rS_1 is multiplied by the 32-bit unsigned value in rS_2 and the 64-bit unsigned product is placed in register rD:rD+1.

2.1.1.1.4 Signed Integer Divide Mnemonic

The official 88100 mnemonic **div** will be changed to **divs** to be consistent with the mnemonic for signed integer multiply. The **div** mnemonic will continue to exist as a synonym (alias) of **divs** for some period of time so that code generators need not change instantaneously.

2.1.1.1.5 Unsigned Integer Divide Double

As a complement to newly provided 64-bit integer multiply a $64 \div 32 \Rightarrow 64$ bit unsigned divide is also defined.

divu.d rD, rS_1, rS_2

The unsigned 64-bit value in register $rS_1{:}rS_1{+}1$ is divided by the unsigned value in rS_2 and the 64-bit unsigned quotient is placed in register rD:rD+1. If the divisor in rS_2 is equal to zero then no result is written into rD:rD+1 and a Divide-by-Zero exception is taken.

2.1.1.1.6 Integer Arithmetic Summary

The following table summarizes the 88000 integer arithmetic options:

Arithmetic	Unsigned								Signed †							
Addressing	Immediate				Triadic				Immediate				Triadic			
Result Size	32		64		32		64		32		64		32		64	
Carry-out	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Add and Sub		✓			✓	✓				✓			✓	✓		
Multiply		✓				✓		✚						✚		
Divide *		✓				✓		✚		✓				✓		

✓ Exists in original architecture
✚ New architectural extension
† May generate overflow
Y/N Generates/does not generate carry
* May generate divide-by-zero

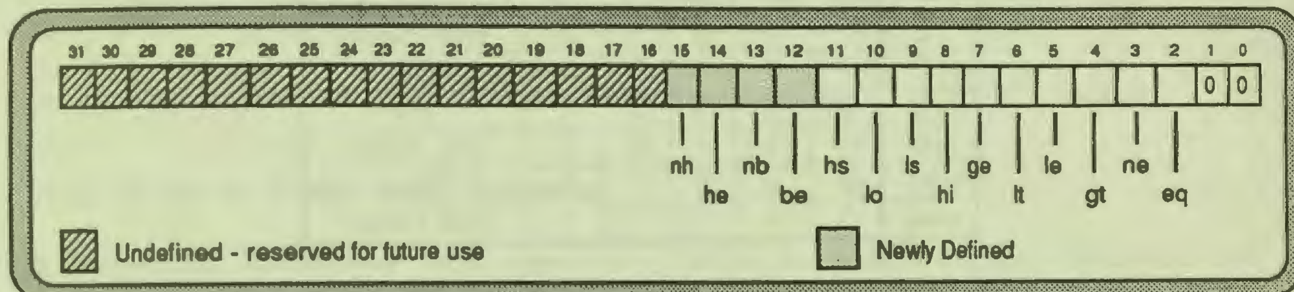
Table 2.1.1.1.6 - Summary of 88000 Integer Arithmetic

2.1.1.2 String Processing Operations

Two new conditions are evaluated by the 88000 compare (**cmp**) instruction in order to improve string handling. In addition to the result bits currently returned, it will also return an "any byte equal" and an "any half-word equal" bit (and their complements). This facilitates rapid searching of long strings for a given character.

cmp rD,rS1,rS2
cmp rD,rS1,IMM16

The data contained in rS_1 is compared with the data in rS_2 or the zero-extended 16-bit immediate and returns a bit string of evaluated results into rD. The format of the returned bit string is shown below:



nh:	no half-word equal	hi:	unsigned greater than
he:	any half-word equal	ge:	signed greater than or equal
nb:	no byte equal	lt:	signed less than
be:	any byte equal	le:	signed less than or equal
hs:	unsigned greater than or equal	gt:	signed greater than
lo:	unsigned less than	ne:	not equal
ls:	unsigned less than or equal	eq:	equal

Figure 2.1.1.2. - cmp Extension for String Operations

2.1.1.3 Branch Prediction

Static branch prediction is the assignment of a preferred direction for branch instructions (i.e. taken or not taken) at compile time. The compiler may use programmer or profiler supplied data to predict whether or not a branch is likely to be taken. Static branch prediction was not defined for the 88100 but is defined for use in future parts. This is not a change from the 88100 instruction set but is simply a convention which the compiler can use to optimize branch performance consistently across all implementations.

The 88000 has three conditional branch instructions; branch on bit clear - **bb0**, branch on bit set - **bb1**, and branch conditional - **bcnd**. The instructions **bb0** and **bb1** are primarily used in conjunction with the integer and floating point compare instructions. The compare instructions place a bit string which is the result of a variety of comparisons made on the source operands. The bit string returned has bits for both the true and complement of each comparison, e.g. both equal and not equal. Therefore, when used in concert with the compare instruction, **bb0** and **bb1** are functionally redundant. This redundancy can be exploited to allow the compiler to indicate to the machine which way the branch is likely to go. By convention, a **bb0** is interpreted as a suggestion that the branch is not likely to be taken while a **bb1** suggests that the branch will most likely be taken[1].

Static branch prediction convention has also been added to the **bcnd** instruction. **bcnd** compares the data in the source register to zero. The sign bit and zero bit of

---

1 Statistics show that most conditional branches are taken - roughly 70% of them. Most existing 88100 compilers generate **bb1** branches more frequently than **bb0's**. Thus an implementation which employs static branch prediction should get better than 50% prediction on existing 88100 code without recompiling.

the comparison are used to index a five bit field in the instruction. If the indexed bit is a 1 then the branch is taken. The table below shows the definition of the bits in the instruction field:

rS_1	bcnd opcode bits					Prediction
	25	24	23	22	21	
=0	0	0	0	1	0	Not Taken
≠0	0	1	1	0	1	Taken
>0	0	0	0	0	1	Taken
<0	0	1	1	0	0	Not Taken
>=0	0	0	0	1	1	Taken
=<0	0	1	1	1	0	Not Taken

Table 2.1.1.3 - bcnd Prediction

By convention, if bit 21 is a one (1) in the **bcnd** instruction then the branch is predicted to be taken. Thus, specifying the not-equal-zero, greater-than-zero, and greater-than-equal-zero conditions indicate the branch will likely be taken. Specifying the equal-zero, less-than-zero, and less-than-equal-zero conditions indicate that the branch will not likely be taken.

2.1.1.4 Signed Immediate Mode

The 88000 architecture previously defined all 16-bit immediate fields to be unsigned values. This mode of operation will continue to exist and in addition a new mode is added which allows immediates to be signed 2's complement values. The mode is selectable only from supervisor mode. In unsigned mode the operation of all instructions is the same as previously defined in the 88100.

2.1.1.4.1 Signed Immediate Addressing

In signed-immediate mode, the immediate offset of load and store instructions is sign extended to a 32-bit value before being added to the base register.

2.1.1.4.2 Signed Constants

In signed-immediate mode, the immediate constants for signed integer arithmetic operations are sign extended to 32-bits before being used as an operand. Sign extension is performed on **add, sub, cmp**, and **div** instructions. Sign extension is not performed on unsigned integer arithmetic (**addu, subu, divu, mulu**), logical operations (**and, mask, xor**, and or), or bounds check (**tbnd**) - operation of these instructions is the same in both signed and unsigned-immediate modes and the same as previously defined in the 88100.

2.1.2. BASE ARCHITECTURE CLARIFICATIONS

2.1.2.1 Integer Multiply and Divide with SFU1 Disabled

In the 88100, execution of integer multiply (**mul**) or divide (**div,divu**) instructions while SFU1 is disabled causes a Floating Point Unimplemented exception to be taken. These instructions are changed to be independent of SFU1 status[1], i.e., they should not generate a Floating Point Unimplemented exception when SFU1 is disabled.

2.1.2.2 r0 as a Destination Register

In the 88100, instructions which specified r0 as a destination register performed the indicated operation but did not modify any registers. Any side-effect, such as generating an exception, performing a memory cycle, or updating a status flag, occurred normally but the result was discarded. This definition is changed slightly: it will continue to be true that the register file will not be modified, however, the occurrence of side-effects is now defined to be implementation dependent. It is no longer guaranteed that all expected side-effects will occur; for example, a **ld** r0,r2,r3 may not actually cause a memory reference or an **add.co** r0,r1,r2 may not affect the carry flag or generate an overflow.

2.1.2.3 NOP's

The 88000 instruction set architecture is designed so that NOP (No OPeration) instructions are never required in the instruction stream. However, should the need for a NOP arise under some special circumstance, three instructions are recommended:

1) **bcnd** lt0,r0,x - a conditional branch which will not be taken and is predicted not to be taken. This is a non-serializing NOP which will never have a data dependency, will never generate an exception, and is likely to be fast in most implementations. Since it is an untaken branch it is also not likely to have adverse affect on branch acceleration hardware - such as requiring an entry in a branch cache.
2) **add** rX,rX,r0 - a non-serializing NOP; rX can be chosen to either force or avoid a data dependency. rX may be specified as r0; **add** r0,r0,r0 is guaranteed never to have any user visible side-effects but may experience a data dependency in some implementations.
3) **tb1** 0,r0,X - a serializing NOP; will force all instructions currently in execution to complete before issuing.

2.1.2.4 Operating Modes

The 88100 defined several "operating modes" which were selectable from supervisor mode by control bits in the Processor Status Register (PSR). The PSR itself is not an

---

1 The intent of this change is to make integer multiply and divide an official part of the base architecture. It is desired that the operation of base architecture instructions not be affected by an unrelated implementation dependent operation such as disabling a special function unit. Implementations which choose to implement integer multiply and divide in software rather than hardware, should do so by taking the SFU0 Unimplemented Opcode exception.

architectural feature since it is not visible from user mode code. However, some of the operating modes controlled by it *are* architectural; qualifying on the basis of user mode code visibility or as components of the minimum guaranteed feature set (lcd features).

The features controlled by the 88100 PSR are reiterated here to clarify their status as architectural features. Also, a new mode, signed-immediate mode, is added to the architecture. In most cases the exact mechanism used to activate the various modes is not defined architecturally but is implementation specific.

2.1.2.4.1 User/Supervisor Mode

When the machine is in "user mode," only architecturally defined instructions can be executed. Any attempt to execute privileged instructions from user mode will result in a Privilege Violation exception. When the machine is in "supervisor mode," all instructions - user and privileged - can be executed freely. The mechanism used to enter supervisor mode from user mode is defined architecturally; any trap, exception, or external interrupt while in user mode, causes an immediate switch to supervisor mode. The mechanism used to go from supervisor mode to user mode is implementation dependent (the 88100 uses a privileged return-from-exception (**rte**) instruction).

2.1.2.4.2 Byte Ordering

The 88000 native memory addressing mode is "Big-Endian", i.e., an address points to the most significant byte of multibyte data items in memory. However, some other architectures have selected a "Little-Endian" mode of addressing, i.e., the address points to the least significant byte of multibyte data items. In order to peacefully coexist with such systems, the 88100 provided an alternate mode of operation which permitted the 88100 to access Little-Endian data.

In order to improve support for dynamic switching of addressing modes and to support a wider range of bus sizes, the operation of Little-Endian byte order mode will be changed. This change will allow BCS compliant Big-Endian systems to more easily access Little-Endian data bases, will simplify the construction of heterogeneous tightly-coupled systems, and will allow more efficient emulation of Little-Endian instruction sets. The new definition is described in the subsection on Operating Modes found in the Architecture Modifications section below.

2.1.2.4.3 Signed Immediate Mode

The new signed immediate mode is also considered an architectural feature. The operation of this mode is explained in the previous section on Architectural Extensions.

2.1.2.4.4 Misaligned Access Exception Mode

In most cases this type of exception would not be considered architectural, i.e., the architecture would define the result and an implementation could either do it directly in hardware or take an exception and synthesize the result in software. But

the misaligned accesses is a special case because there is no one correct result suitable for all applications. Therefore, the flexibility to choose the appropriate action is given to the system by architecturally guaranteeing that misaligned accesses can always be forced to cause an exception. Thus all 88000's will provide a misaligned access exception mode. In this mode, any memory access to a location which is not aligned on its respective size boundary will cause a Misaligned Access exception. Implementations may wish to allow this mode to be disabled under supervisor program control - in which case a misaligned access will simply ignore the appropriate number of lsb's required to align the access.

2.1.2.5 Code, Address, and Data Modification

There are three general categories of code and data modification which can cause difficulties in highly parallel implementations:

1) Instruction stream modification (self-modifying code)
2) Code relocation
3) Logical address aliasing

Certain restrictions are placed on these operations to simplify construction of future machines.

2.1.2.5.1 Self-modifying Code

In highly parallel machines, multiple instructions may be in various stages of internal pipelines, prefetch queues, and caches. Thus, modification of an impending instruction in memory may not be seen at all places of interest by the processor. This means that the time interval over which code modification will be effective cannot be guaranteed.

Therefore, altering the currently executing instruction stream from user mode is not supported architecturally. It cannot be assumed that programs which do this will work correctly on any current or future implementation of the 88000.

However, implementations must have mechanisms, possibly available only through supervisor mode routines, which can guarantee the effectiveness of "code modification" as described in the following section on code relocation.

2.1.2.5.2 Code Relocation

Code relocation, e.g. movement by a garbage collector, can have the same difficulty as modifying the instruction stream. Therefore, code relocation is also not architecturally supported from user mode. However, it is architecturally required that all implementations provide some mechanism to allow code relocation.

As a minimum, code modification and relocation must be supported by a trap to a supervisor routine. The trap itself is architecturally defined to synchronize the machine and empty all internal pipelines. The supervisor mode routine should perform the requested modification or relocation and then take any additional action required to bring instruction caches into coherence. The supervisor would then return to the user program which must, once again, synchronize the machine and

empty the internal pipelines. At this point the code modification or relocation is guaranteed to be completely in effect.

2.1.2.5.3 Logical Address Aliasing

A similar condition can arise on the data stream. If a machine allows two logical addresses to map to the same datum, then modification of that datum via one logical address may not be seen throughout the machine if it has already been fetched under a different address (alias). The time interval over which the modification will be effective may not be same across all implementations. Therefore, systems which utilize logical address aliasing must be aware of the characteristics of the specific implementation being used.

All implementations must either fully support logical address aliasing or provide a mechanism, possibly at reduced performance, which can be used to guarantee that logical address aliasing will work correctly.

2.1.3. BASE ARCHITECTURE MODIFICATIONS

This section describes the modifications which have been made to the 88000 architecture as originally defined by the 88100 processor. Some of the modifications *could* cause incompatibility with 88100 code. However, such changes have been limited to infrequently used or special purpose features and therefore should not cause compatibility problems in practice. No functionality has been removed from the architecture as a result of these changes.

2.1.3.1 lda Unscaled Instructions

All immediate and triadic-register forms of unscaled **lda** are eliminated from the architecture. These instructions are functionally redundant with the **addu** instructions[1] and are removed to recover opcode space for instruction set extensions. The scaled triadic register form of **lda** is retained because its functionality is not redundant.

2.1.3.2 xmem Immediate Instructions

The immediate form of **xmem** is eliminated from the 88000 architecture[2]. The triadic register form of **xmem** is retained and provides all necessary functionality with little loss in generality. These instructions have also been removed to recover opcode space for instruction set extensions.

---

1 88100 programs which used unscaled **lda** will not work correctly on future 88000's (including the 88110). However, a warning was printed in the 88100 User's Manual to use **addu** instead of scaled **lda**. Therefore, incompatibility due to this change is not expected to be a problem.

2 88100 programs which used immediate **xmem**'s will not work correctly on future 88000's (including the 88110). Since these operations are infrequent and not generally accessible from high level languages, their elimination should cause incompatibility in only a few isolated routines.

2.1.3.3 Illegal Opcodes

In the 88100, some unimplemented opcodes and opcode fields were not detected and did not cause a Unimplemented Opcode exception. This makes it difficult to use these opcode fields as the architecture is extended because the new functionality cannot be emulated on older implementations. Therefore, an architectural requirement is imposed on future implementations that *all* unimplemented major (I<31:26>) and minor (I<15:5>) opcode fields as well as all unused register specifier fields (I<25:21>, I<20:16>, and I<4:0>), must cause an exception. Unimplemented SFU0 opcodes will cause the Unimplemented Opcode exception and unimplemented SFU1-7 minor opcodes will cause the respective SFU exception.

2.1.3.4 Byte Ordering Mode

Two modes of addressing, Big-Endian (the native mode) and Little-Endian, are supported[1]. The two modes of operation are illustrated below:

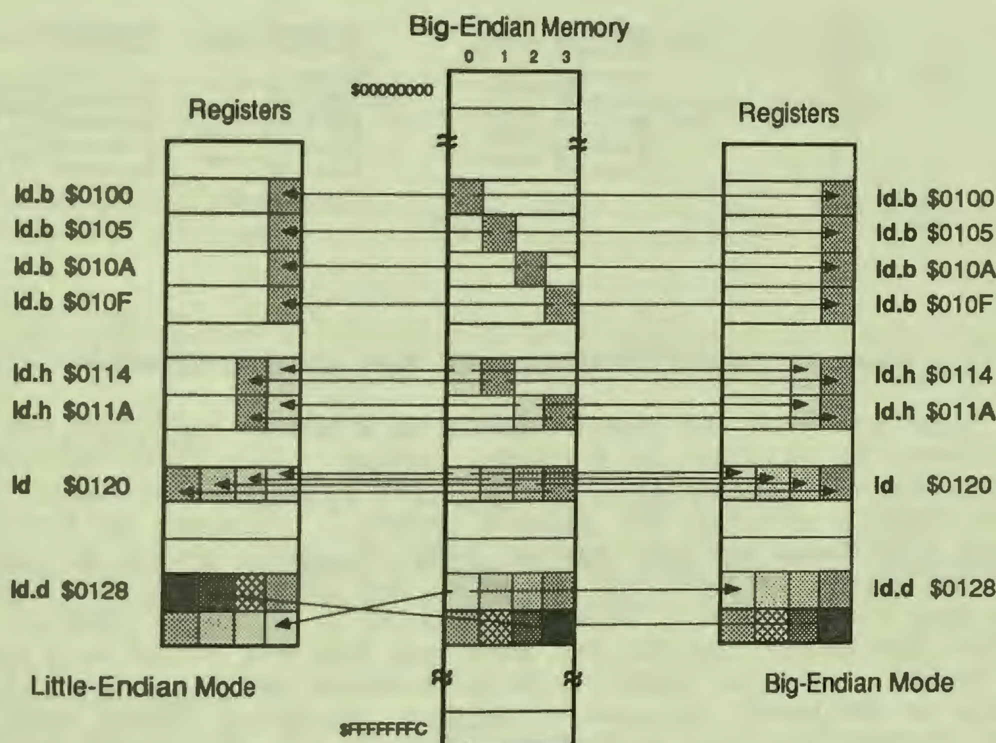


Figure 2.1.3.4.a - Byte Ordering Mode

Although not shown in the diagram, quad-word transfers into the extended register file (described later) are also supported in both modes. Bytes are swapped in a manner analogous to that shown for doubles.

Instruction addressing is always Big-Endian regardless of the byte ordering mode.

1 In the 88100, byte ordering altered the address at which a data item was found in memory but the order of bytes within a data item was not affected. This definition is changed and in future 88000's an address will point to the same byte in memory regardless of the mode but the order of bytes within a data item is swapped between registers and memory.

An example of using the byte ordering mode in a heterogeneous environment is shown below:

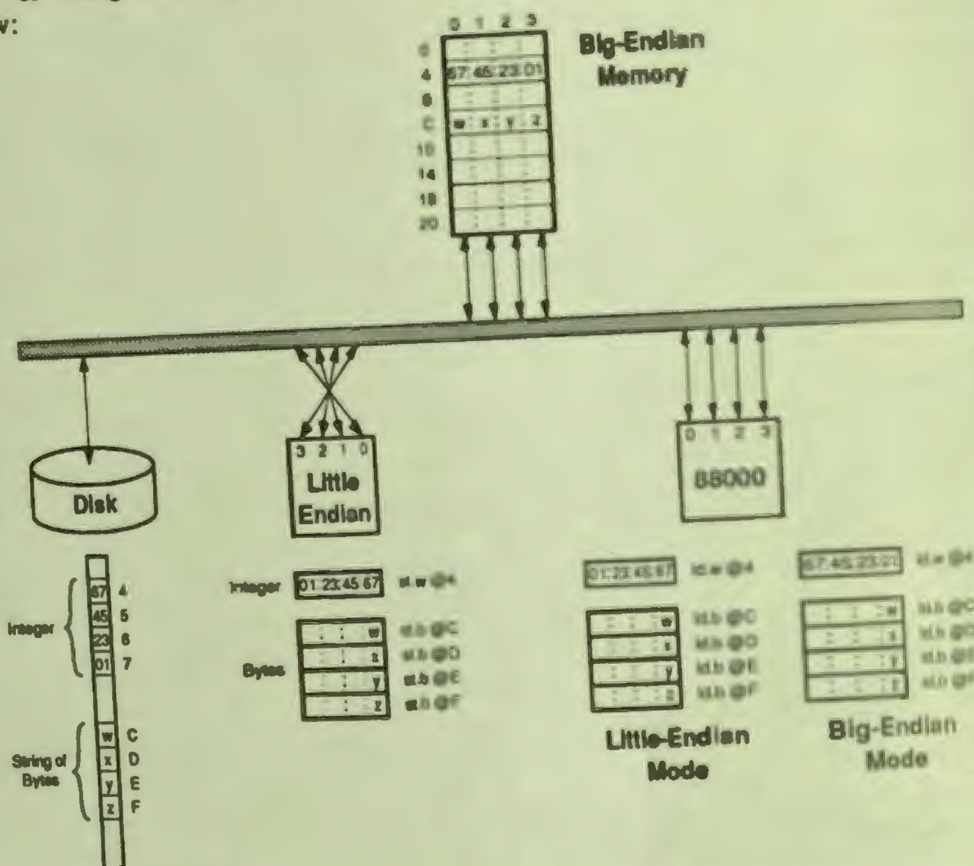


Figure 2.1.3.4.b - Heterogeneous Byte Order Environment

In this figure four devices are shown connected to a common bus. The first is main memory which is organized in Big-Endian fashion. Then there are two 32-bit processors, one a Big-Endian 88000 and the other a generic Little-Endian processor. Since the 88000 is the same byte gender as memory, it connects up directly to the bus. The Little-Endian processor however, being of opposite gender, is connected to the bus by reversing its bytes left to right so that processor byte zero connects to memory byte zero. This byte reversing connection to the bus is necessary so that data placed into memory from the disk, which may have been created on a true Little-Endian system, would be legible to the Little-Endian processor. (A disk has no knowledge of the internal structure of the data; therefore it cannot know how to correctly "unswap" bytes into a Big-Endian memory - it must simply lay the bytes into memory in the order they were written to it.)

Below each processor is shown some data as it might appear in internal registers. The example shows the Little-Endian processor storing a 32-bit integer (01234567) to memory at address 4. The byte reversal ends up storing data "backward" in memory as seen by the 88000 in Big-Endian addressing mode. However, if the 88000 is placed in Little-Endian mode, a byte-swap correction is applied to the integer as it is read into the register. Also illustrated is the storage of a character string "wxyz"; notice here, however, than no byte swapping is performed and data is read the same in both modes.

2.2. FLOATING POINT ARCHITECTURE EXTENSIONS

Extensions to the floating point (SFU1) architecture, as originally defined by the 88100, are described in this section. All extensions are upward compatible with the 88100. The two major extensions are: 1) support for IEEE extended precision floating point data, and 2) addition of a new extended register file to provide more register namespace for floating point data.

2.2.1. FLOATING POINT DATA TYPES

Architecturally the 88000 supports four IEEE standard 754 floating point data formats; single precision, double precision, double-extended precision, and quad precision. The formats are shown in the figure below:

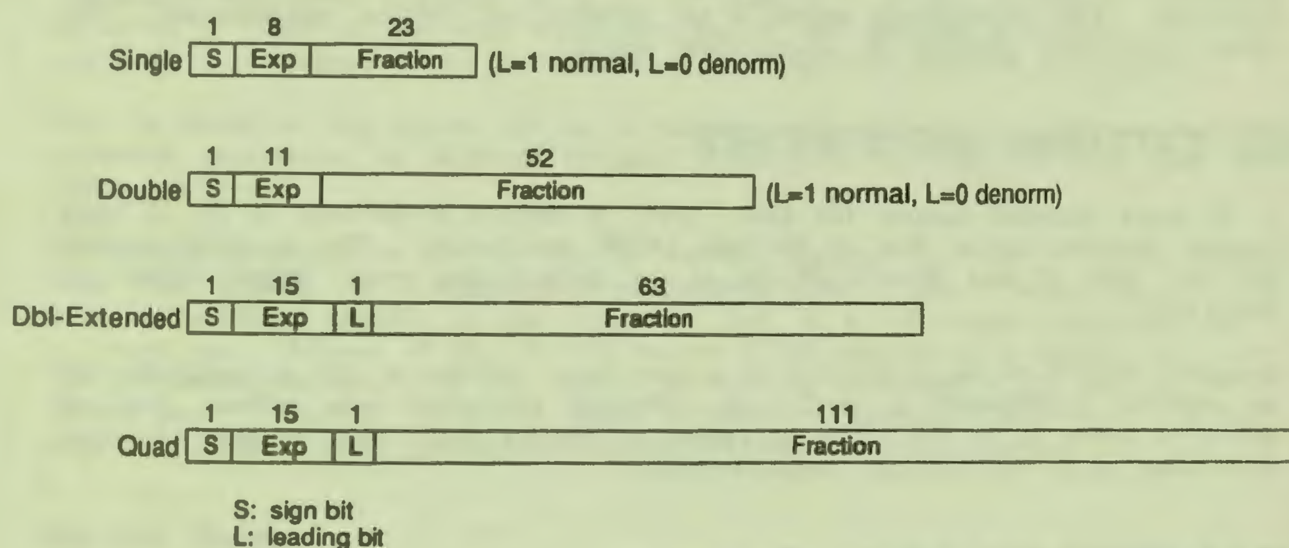


Figure 2.2.1 - Floating Point Data Formats

If an implementation of the 88000 provides floating point support, it must implement (possibly in software) one of the following combinations of formats:

1. single
2. single and double
3. single, double, and double-extended
4. single, double, and quad

The IEEE floating point specification recommends that a given implementation not support an IEEE extended precision format which is smaller than the highest precision format supported. Thus, either double-extended or quad format may be supported in a given implementation - but not both. The 88000 SFU1 instructions encode double-extended and quad precision the same. The memory format of double-extendeds and quads is also the same so that a double-extended precision processor is forward compatible with a quad precision processor. Therefore, the term "extended" in this document will be used as a generic term referring to either double-extended or quad format.

2.2.1.1 Unnormalized Extended Precision Numbers

IEEE single and double precision numbers have an implied leading bit (single digit to left of radix point) of 1 for normalized numbers and 0 for denormalized numbers. However, extended precision numbers have an explicit leading bit which can be either 1 or 0. This leads to the possibility of redundant encodings, e.g. $1.1001 * 2^{011} = 0.1101 * 2^{100}$ where the first number is normalized and the second number is unnormalized. Note that unnormalized numbers are distinct from denormalized by the fact the unnormalized numbers have a non-zero exponent. The IEEE spec requires that redundant encodings either be disallowed or that they be indistinguishable.

The 88000 floating point architecture will utilize the second alternative. When an unnormalized source operand is used, it will be normalized before being used in the operation. The unnormalized source is not affected and remains unnormalized. The 88000 will never generate an unnormalized result.

2.2.2. EXTENDED REGISTER FILE

A 32 entry extended register file (xr0 - xr31) is defined in addition to the 32 entry general purpose register file in the base 88000 architecture. The extended register file can hold 32 data objects of any of the defined data types: single, double, or extended.

Extended register xr0 is always read as positive zero. Writing to xr0 is permissible but no register modification is performed. Normal instruction side effects, such as setting a status bit in the FPSR or causing a memory cycle, may or may not occur depending upon the specific implementation.

2.2.2.1 Moving Data between Registers

Data can be moved between extended registers and back and forth between register files with the **mov** instruction. No support is provided for moving operands within the general registers or for moving extended precision operands between register files. If it is necessary to move extended precision operands between register files, it should be done through memory using stores and loads.

2.2.2.2 Moving Data To and From Memory

The functionality of the base architecture load and store instructions is expanded to allow transfer of data between memory and the extended register file. All load and store source operands (addresses), regardless of the destination register file, are taken from the general registers.

2.2.2.2.1 Loads

Data can be loaded directly from memory into an extended register using either the immediate or triadic form of the **ld** instruction. The mnemonic for general register

and extended register loads can be the same. The assembler will detect that an extended register is being used as the destination register and generate the appropriate opcode.

ld.t xrD,rS_1,rS_2	.t = {blank} (single), .d (double), .x (extended)
ld.t xrD,rS_1[rS_2]	scale factor = 4x, 8x, and 16x respectively
ld.t xrD,rS_1,I16	

A register load performs a memory fetch of the appropriate number of words required to transfer the data type specified by the ".t" field into extended register xrD. General registers rS_1 and rS_2 (or I16), are used to compute the memory address. One word is transferred for a single, two words for a double, and 4 words for extended.

Data transfers are required to be aligned on their size boundary in memory else a Misaligned Reference exception is taken (if enabled). If the Misaligned Reference exception is disabled, the least significant bits of the address are ignored, i.e., the reference is performed to the next lower address boundary which is size aligned.

Data is stored in the register file as a memory image, i.e. no data type conversion, exponent conversion, or reserved operand checking is performed on the load and store operation.

When loading singles or doubles into an extended register, the value given to unused bits is not defined. It is considered a programming error if data is loaded (or **moved**) into an extended register as one type and used in a subsequent calculation as a different type. Misuse of data in this manner is not guaranteed to operate the same on all implementations. All data should be explicitly converted to the desired format before use. Explicit conversion does not have an associated performance penalty since all floating point instructions support full mixed mode operations.

2.2.2.2.2 Stores

Data can be stored from the extended register file directly to memory using either the immediate or triadic register form of the **st** instruction by simply specifying an extended register as the destination.

st.t xrD,rS_1,rS_2	.t = {blank} (single), .d (double), .x (extended)
st.t xrD,rS_1[rS_2]	scale factor = 4x, 8x, and 16x respectively
st.t xrD,rS_1,I16	

A store takes the data in extended register xrD and transfers it to memory. General registers rS_1 and rS_2 (or I16), are used to compute the memory address. The appropriate number of words are transferred to handle the data type as specified in the ".t" field.

Data transfers are required to be aligned on their size boundary in memory else a Misaligned Reference exception is taken if enabled. If the Misaligned Reference exception is disabled, the least significant bits of the address are ignored and the reference is performed to the next lower evenly aligned boundary.

Single precision values are stored in memory on word boundaries. Double precision values are stored on even word boundaries. Extended precision values (both 80 and

128-bit) are stored on quad word boundaries. The figure below shows how data is stored in memory. 80-bit double-extended floating point values are stored left justified and zero filled in the quad word memory field so that future implementations using quad precision will be compatible with data stored in double-extended format.

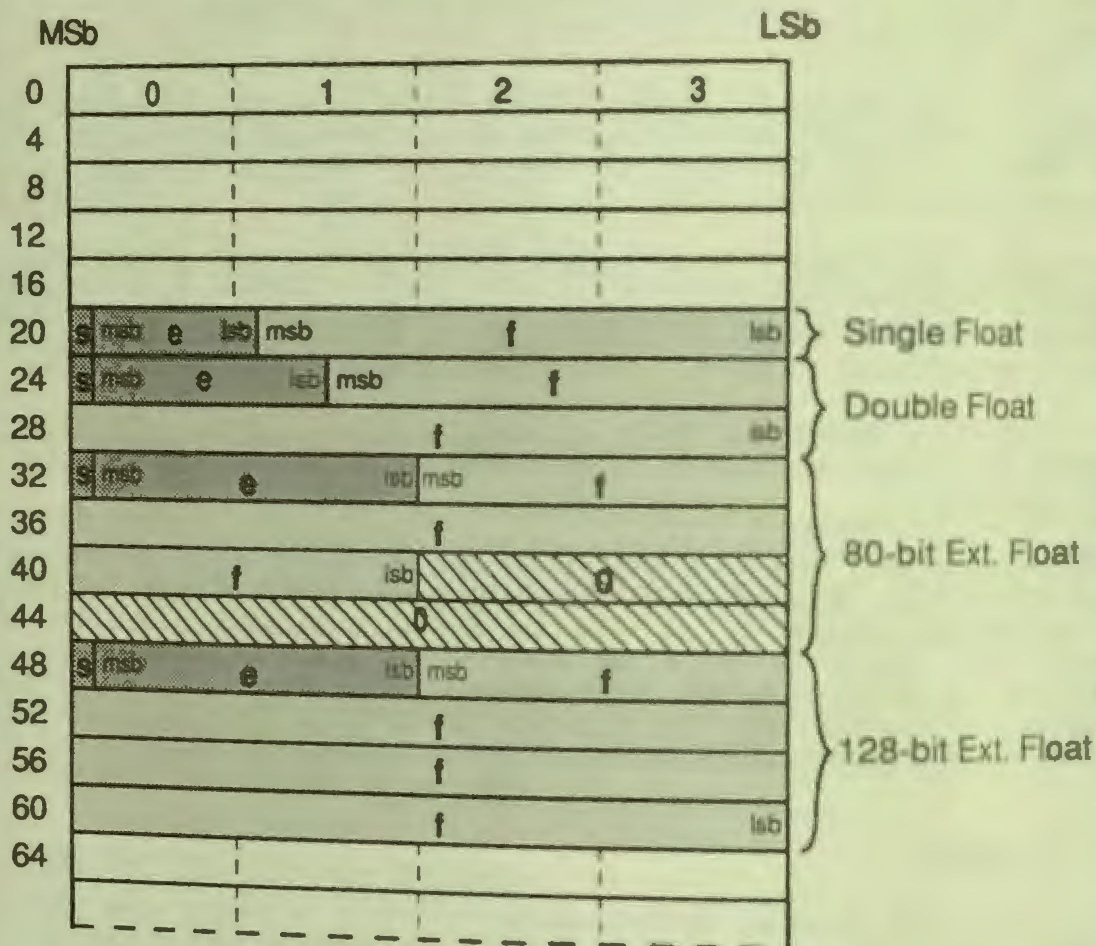


Figure 2.2.2.2.2 - Memory Storage Alignment

2.2.2.3 Extended Register File Modified

Some programs may not perform any floating point operations. For these programs there is no need for the operating system to save and restore the extended registers on a context switch. Therefore, a bit (XMOD) is maintained in the FPSR to indicate whether or not a process has modified an extended register. The operating system may interrogate this bit on a context switch to see whether it must save and restore the extended registers. Any instruction which specifies the XRF as a destination will set this bit and it will remain set until explicitly cleared by software.

2.2.3. FLOATING POINT DATA IN THE GENERAL REGISTERS

The following restrictions are imposed on floating point operands (or any operands with greater precision than a single word) in the general register file[1]:

2.2.3.1 Double-Word

When double precision is specified in a general register, then the double register rS_1:rS_1+1 or rD:rD+1 is used. The 88000 architecture allows double-word (double precision) data to be stored in any arbitrary general register pair without restriction on alignment. All implementations will continue to provide this flexibility but some implementations may perform better when double-words are aligned in even numbered register pairs, e.g. r4:r5, rather than r5:r6. Therefore, to get maximum performance out of double-precision data in the general registers, it is strongly recommended that new software and compilers allocate double-word data into even numbered register pairs.

2.2.3.2 Quad-Word

Extended precision is not supported as a data type in the general registers. No floating point operations, mov's, loads, or stores are provided for extended precision (quad-words) in the general registers.

2.2.4. FLOATING POINT OPERATIONS

The arithmetic operations are basically the same as those provided by the 88100 but extended, in an upward compatible fashion, to also operate out of the extended register file. The mnemonics for general register and extended register operations can be the same. The assembler will detect that an extended register is being used and generate the appropriate opcode. Mixed register file operands in the same instruction are not provided in general but can occur in some instances (float⇔integer conversion, data movement, and comparisons).

2.2.4.1 SFU1 Instructions

The following floating point instructions are defined (the .t extension used here is a generic type specifier suffix; it would actually be a three letter extension specifying the type of the destination and both source operands, e.g. .t = .ssd, .xds, ... etc.):

1) **fmul.t** xrD,xrS_1,xrS_2
   fmul.t rD,rS_1,rS_2
   Multiplies the floating point value in xrS_1 (rS_1) by the floating point value in xrS_2 (rS_2) and places the floating point product in xrD (rD).

---

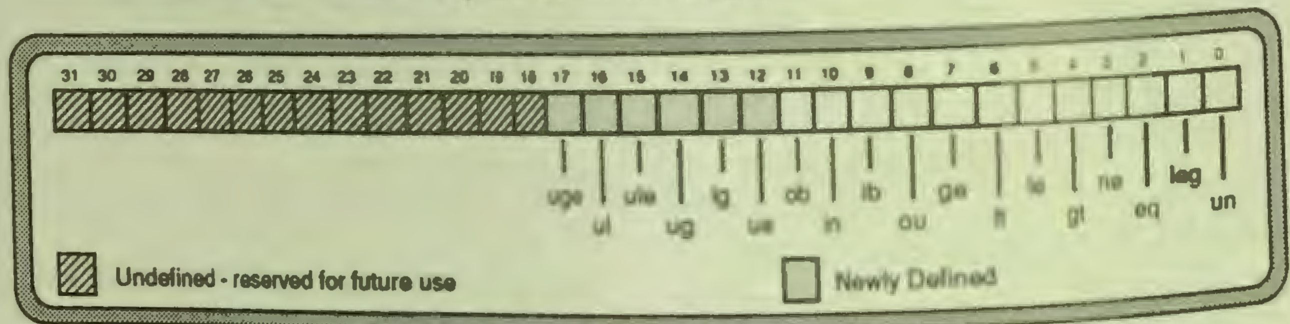
1 Floating point operands continue to be supported out of the general register file. However, some implementations may provide better performance when the operands are in the extended file. Therefore, to assure smooth transition to future generation parts it is recommended that floating point data be allocated in the extended registers whenever possible.

2) **fcvt.t** xrD,xrS_2
fcvt.t rD,rS_2
Converts the floating point value in xrS_2 (rS_2) from the precision specified by the source type specifier to the precision specified by the destination type specifier and places the result in xrD (rD). The operation is similar to performing an **fadd** $rD,r0,rS_2$ using the currently specified rounding mode, but the sign of the source operand is strictly preserved.

3) **fadd.t** xrD,xrS_1,xrS_2
fadd.t rD,rS_1,rS_2
Adds the floating point value in xrS_1 (rS_1) to the floating point value in xrS_2 (rS_2) and places the floating point sum in xrD (rD).

4) **fsub.t** xrD,xrS_1,xrS_2
fsub.t rD,rS_1,rS_2
Subtracts the floating point value in xrS_2 (rS_2) from the floating point value in xrS_1 (rS_1) and places the floating point difference in xrD (rD).

5) **fcmp.t** rD,xrS_1,xrS_2
fcmp.t rD,rS_1,rS_2
Compares the floating point value in xrS_2 (rS_2) with the floating point value in xrS_1 (rS_1) and places an 18-bit result string in general register rD as shown in the figure below. The comparison operation sets the result bits as if the operation performed were a subtraction of rS_2 from rS_1 but does not generate the arithmetic exceptions (such as overflow) possible with a subtraction. No exception is generated if either of the operands are unordered (any NaN) unless one of them is a signaling NAN. If either operand is a signalling NaN, then invalid (AFINV) is signaled in the FPSR and a trap is taken if enabled by EFINV in the FPCR.



un:	unordered. Previously "nc" (not comparable)	ib:	in bounds
leg:	less than, equal or greater than. Previously "cp"	in:	in bounds
eq:	equal	ob:	unordered or out of bounds
ne:	not equal or unordered	ue:	unordered or equal
gt:	greater than	lg:	less than or greater than
le:	less than or equal	ug:	unordered or greater than
lt:	less than	ule:	unordered or less than or equal
ge:	greater than or equal	ul:	unordered or less than
ou:	unordered or out of range	uge:	unordered or greater than or equal

Figure 2.2.4.1 - Floating Point Compare Results

6) **fcmpu.t** rD,xrS_1,xrS_2
   fcmpu.t rD,rS_1,rS_2
   Compares the floating point value in xrS_2 (rS_2) with the floating point value in xrS_1 (rS_1) and places a result string in general register rD just as **fcmp** does. Unlike **fcmp**, however, **fcmpu** signals an invalid exception in the FPSR if either operand is unordered (NaN). A trap will be taken if enabled by EFINV in the FPCR.

7) **mov** xrD,xrS_2
   mov.t xrD,rS_2 .t = {blank} (single), or .d (double)
   mov.t rD,xrS_2 .t = {blank} (single), or .d (double)
   The data in the source register is moved to the destination register. The ".t" suffix specifies the size of the operand to be moved. When data movement is within the extended register file, the entire register is moved regardless of the size specification. For double precision operands in the general registers, two consecutive registers are used (rD:rD+1 or rS_2:rS_2+1). When single or double data is moved from the general register file to the extended register file, the value of unused bits is not defined, i.e, they are implementation dependent. No reserved operand checks are performed for **mov**.

8) **flt.t** xrD,rS_2
   flt.t rD,rS_2
   Converts the 32-bit signed integer value in general register rS_2 into a floating point value of type .t and stores the result in register xrD (rD).

9) **int.t** rD,xrS_2
   int.t rD,rS_2
   Converts the floating point value of type .t in xrS_2 (rS_2) into a 32-bit signed integer and stores the result in general register rD. The rounding mode specified in the Floating Point Control Register is used to perform the conversion.

10) **nint.t** rD,xrS_2
    nint.t rD,rS_2
    Converts the floating point value of type .t in xrS_2 (rS_2) into a 32-bit signed integer and stores the result in general register rD. The IEEE 754 "round to nearest" rounding mode is used to perform the conversion.

11) **trnc.t** rD,xrS_2
    trnc.t rD,rS_2
    Converts the floating point value of type .t in xrS_2 (rS_2) into a 32-bit signed integer and stores the result in general register rD. The IEEE 754 "round to zero" rounding mode is used to perform the conversion.

12) **fdiv.t** xrD,xrS_1,xrS_2
    fdiv.t rD,rS_1,rS_2
    Divides the floating point value in xrS_1 (rS_1) by the floating point value in xrS_2 (rS_2) and places the floating point quotient in xrD (rD).

13) **fsqrt.t xrD,xrS_2**
    fsqrt.t rD,rS_2
    Takes the square root of the floating point value in xrS_2 (rS_2) and places the floating point result in xrD (rD).

2.2.4.2 Mixed-Mode Operations

All arithmetic operations support a full mixed-mode data type model, allowing each operation to specify the type of each source and destination operand individually. All 27 permutations of source types and destination types are supported. Exponent debiasing, reserved operand checking, and data type conversion are performed as a part of each arithmetic operation.

Mixed mode operations are first carried out to infinite precision, then converted, with a single rounding, to the result precision. This approach gives the highest accuracy result attainable.

2.2.4.3 Data Type Conversion

The mixed mode operations performed by the 88110 are used to perform conversions from one IEEE floating point precision to another. Conversion can be performed either as the final operation of an expression evaluation or as a separate operation explicitly for the purpose of conversion.

When carried out as an explicit operation, the new floating point convert (fcvt) instruction should be used rather than addition or subtraction to r0 or xr0. This is because IEEE addition or subtraction of +0 does not correctly preserve the sign of zero in all rounding modes as required for conversion (e.g. $-0 + +0 = +0$ w/rnd→near, and $+0 - +0 = -0$ w/rnd→$-\infty$).

2.2.5. FLOATING POINT EXCEPTIONS

2.2.5.1 Floating Point Exception Model

The 88000 floating point architecture is defined by the *IEEE 754 Standard for Binary Floating Point Arithmetic.* That standard specifies 5 different exceptions[1] which must be signaled to the user program. The 88000 floating point architecture defines two registers for interfacing to user programs: the Floating Point Status Register (FPSR) is used to signal IEEE exceptions to the user; and the Floating Point Control Register (FPCR) allows the user to enable or disable the trap for each of the IEEE exceptions. Conceptually the operation is as shown in the following diagram:

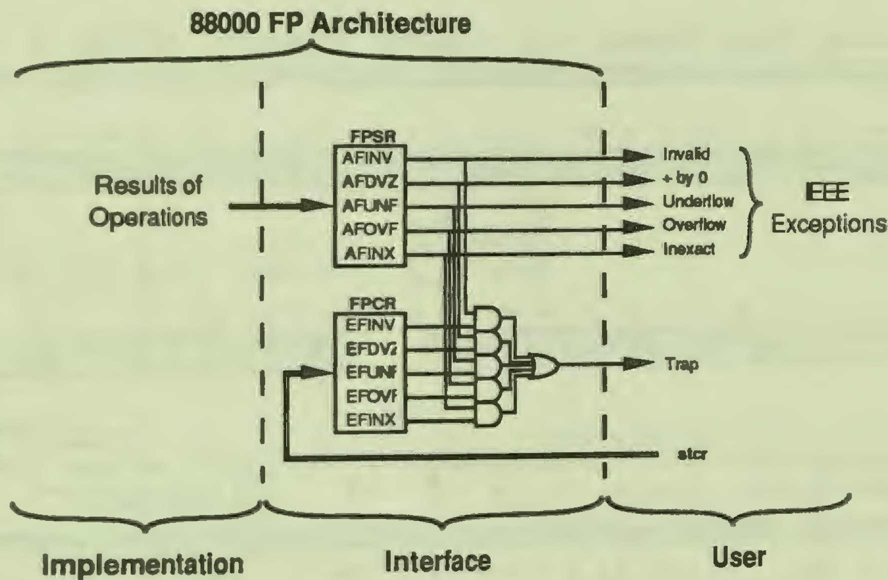


Figure 2.2.5.1 - IEEE Exceptions

The results of arithmetic operations map to IEEE exceptions which are reported to the user in the FPSR. If an exception is generated and the user has disabled the associated trap in the FPCR then a default numeric result is returned to the user as specified in the standard.

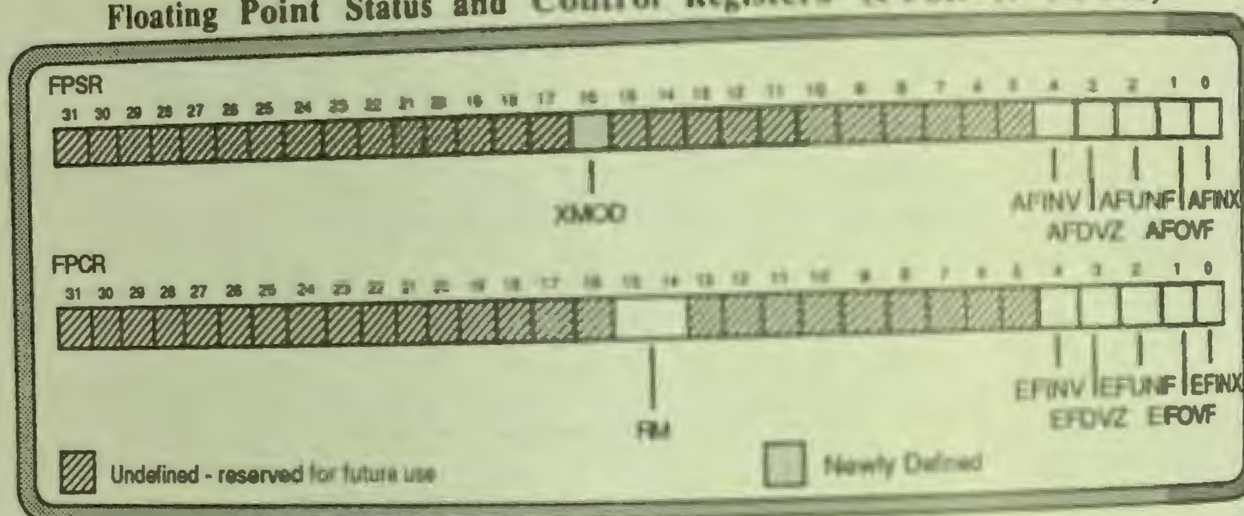
1 "Exceptions" in the IEEE sense should not be confused with 88110 instruction exceptions. An IEEE exception is the occurrence of an IEEE specified event which must be reported (signaled) to the user program. Signally involves reporting status flags visible to the user program and optionally trapping to a user trap handler.

2.2.5.2 Floating Point Arithmetic Status and Control Registers

The Floating Point Status Register (FPSR) and Floating Point Control Register (FPCR) are the registers used to communicate between the User program and the 88000 implementation. As such, they are defined in the 88000 floating point architecture and should be functionally identical among all implementations. These registers are located in SFU1 control registers fcr62 and fcr63. They may be tested, saved, restored, or initialized using the **fldcr, fstcr**, and **fxcr** instructions from either user or supervisor mode.

All status bits in the FPSR register are sticky, i.e. they hold accumulated status and once set are cleared only by writing directly to the register. The layout of these two registers is shown in the figure below.

Floating Point Status and Control Registers (FPSR & FPCR)



AFINX:	Accumulated Inexact Flag	EFINX:	Inexact Exception Enable
AFOVF:	Accumulated Overflow Flag	EFINX:	Inexact Exception Enable
AFUNF:	Accumulated Underflow Flag	EFUNF:	Underflow Exception Enable
AFDVZ:	Accumulated Divide-by-Zero	EFDVZ:	Divide-by-Zero Exception Enable
AFINV:	Accumulated Invalid Operation	EFINV:	Invalid Exception Enable

XMOD: Extended Register File Modified

RM:
- 00 - Round to nearest
- 01 - Round toward Zero
- 10 - Round toward negative infinity
- 11 - Round toward positive infinity

Figure 2.2.5.2 - Floating Point Arithmetic Status and Control Registers

3. 88110 IMPLEMENTATION

This section covers the implementation details of the 88110.

3.1. OVERVIEW AND BLOCK DIAGRAM

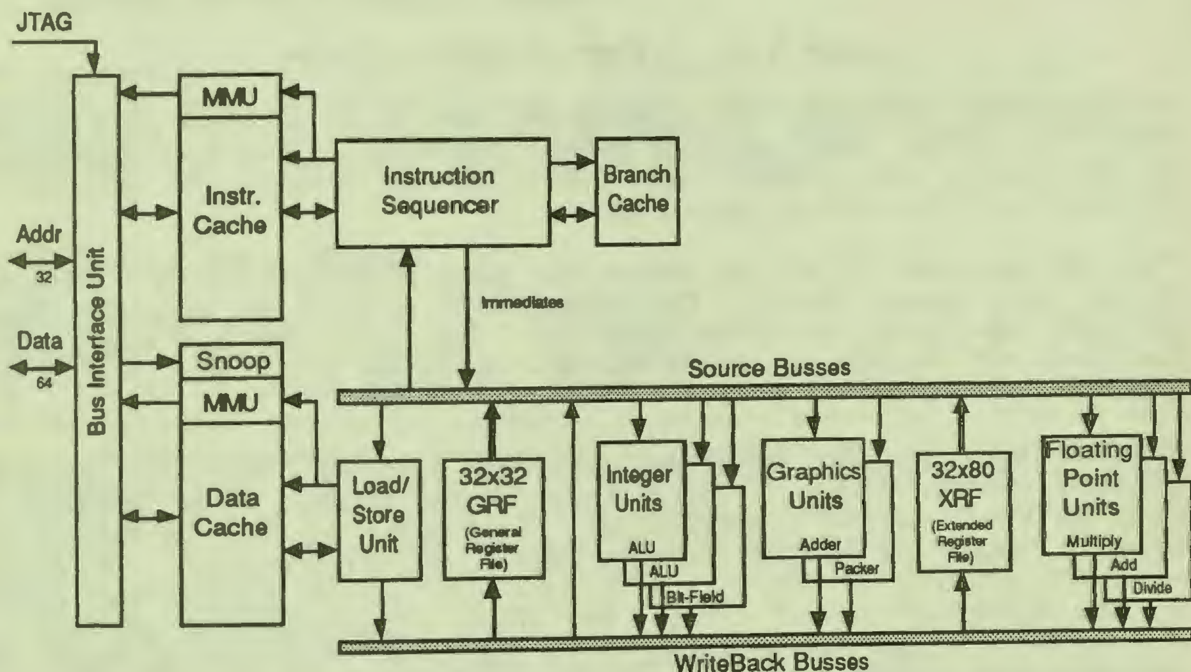


Figure 3.1.a - 88110 Block Diagram

The instruction sequencer is the heart of the 88110. It provides centralized control over data flow among execution units and the register files. The sequencer implements the master instruction pipeline, enforces data interlocks, dispatches (issues) instructions to available execution units, directs data from the register files onto and off of the busses, and maintains a state history so it can back the machine up in the event of an exception.

The master instruction pipeline is a 4 stage pipeline. A simplified diagram is shown below:

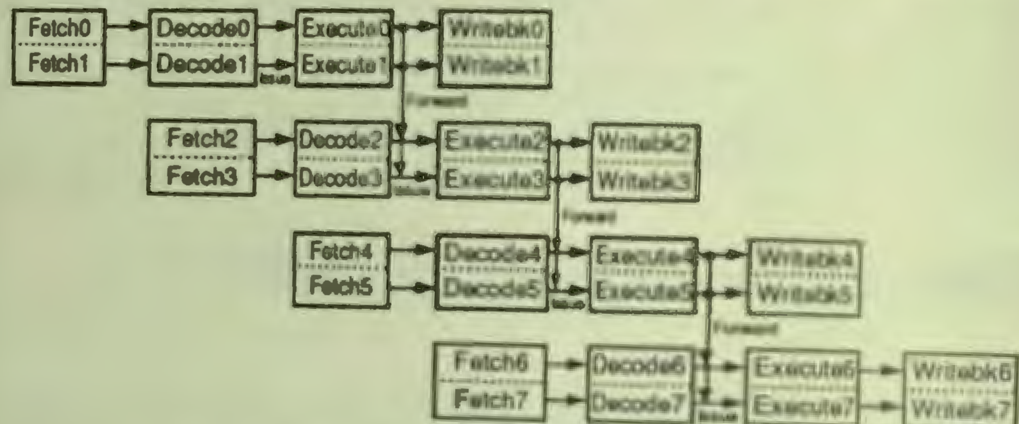


Figure 3.1.b - Master Instruction Pipeline

Two instructions are decoded and considered for issue to the execution units on each clock. The pipeline stages are fully hardware interlocked so that data dependencies will automatically stall instruction issue without software assistance. A maximum of two instructions are issued on each clock.

When an instruction "issues", the register files place source data for that instruction on the source operand busses. The execution unit to which the instruction issues then reads data off the appropriate source bus. The register files and source busses have sufficient bandwidth to sustain execution throughput at the peak rate of two instructions per clock.

Execution units are each independent functional units with their own internally controlled pipelines. When an execution unit finishes execution of an instruction it places the result data onto a writeback bus. The register files take the data off the writeback busses and store it into the correct destination register. If another instruction is waiting for this data, it is "forwarded" past the register files directly into the appropriate function unit(s). This allows a data dependent instruction to issue on the next clock without waiting for the data to be written into the register file and read out back again.

There are two writeback busses available to the execution units. Since different execution units have different pipeline lengths, it is possible for more than two instructions to be completing in a given clock cycle. Therefore, execution units arbitrate for an available slot on a writeback bus. Highest writeback priority is granted to single cycle execution units (integer and graphics units) so that single cycle instructions are always guaranteed a writeback slot while multistage pipeline units (floating point units and load/store units) arbitrate for writeback slots. Pipelined execution units which are denied a writeback slot, will continue to advance their internal pipeline stages and accept new instructions until all pipeline stages are full.

The sequencer attempts to issue a pair of instructions on each clock. Instructions are issued to the execution units in strict program sequence. If the first instruction in an "issue pair" cannot issue, then neither instruction in the pair is issued. If the first instruction in the pair issues but the second cannot, then the second instruction

is moved into the vacated first instruction issue position and a new one is fetched from the instruction cache to replace it. If both instructions in the pair issue, then two new instructions are fetched from the instruction cache.

On an instruction cache hit, a pair of instructions are fetched and returned to the instruction unit on each clock - regardless of their address alignment. Thus, there are no instruction address alignment restrictions imposed on dual instruction issue to avoid pipeline interruptions, and no NOP instructions are required in the code stream to fill empty issue slots. On an instruction cache miss, if the miss is caused by the fetch of the first instruction in an issue pair, then a minimum (ideal memory) three clock latency is incurred which produces six instruction bubbles. If the miss is to the second instruction in an issue pair (pair straddles a cache line), then a 4 clock latency producing seven instruction bubbles is incurred.

3.1.1. INSTRUCTION SEQUENCING

The sequencer attempts to issue a pair of instructions on each clock. In order for an instruction to issue, the sequencer must determine that the required source data is available, that no other instruction still in execution targets the same destination register, and that an appropriate execution unit is available.

3.1.1.1 Source Data Hazards

If an instruction attempts to use a source operand which has not yet finished being computed by a previous instruction, a data dependency or "data hazard" is said to exist. The 88110 resolves data hazards by blocking issue of the dependent instruction until all its source operands are available. A register "scoreboard" is used to keep track of operand availability. Conceptually, the scoreboard is a bit-vector with one bit associated with each register in the register files. Whenever an instruction issues, the scoreboard bit associated with its destination register is set thus marking the register "busy." When that instruction finishes execution and writes back its result, the scoreboard bit is cleared thus making the register and the data being written to it "available" for use. A necessary condition for any instruction to issue is that the scoreboard bits for all its source registers must be clear. If they are not, it means that one or more of its source operands is not yet available and instruction issue is suspended until the scoreboard bits for those registers are cleared by the instruction(s) which are computing the operands.

3.1.1.2 Destination Register Hazards

Since the 88110 allows instructions to complete out of order, there is a potential for an instruction's result to be overwritten by an instruction which issued earlier but completed later. To preclude this possibility, the scoreboard bit of the destination register is also checked as a condition of instruction issue. This insures that updates to any given register are always completed in the order specified by the program and thus no data is ever incorrectly overwritten.

3.1.1.3 Execution Unit Hazards

The third condition for instruction issue is that the required execution unit be available to begin execution of the instruction. The sequencer monitors all execution unit availability and suspends instruction issue in the event of a conflict. An execution unit may be unavailable for one of several reasons:

1) A multicycle non-pipelined unit can have only one instruction in execution at a time. Such a unit goes busy when an instruction issues to it and can not accept another instruction until the previous one finishes. (The divide unit is the only such unit on the 88110.)
2) An execution unit which can take more clocks to execute than the number of pipeline stages available in the unit can go busy when all its pipeline buffers fill up. Such units cannot accept another instruction until an instruction completes and frees up a buffer. (The load/store unit is the only such unit on the 88110.)
3) Some execution units (all except the Integer Unit) can accept only a single instruction per clock. Issuing two instructions to these units on the same clock is prohibited. The following table enumerates instruction pairs which cannot issue simultaneously due to this execution unit conflict:

	N \ N+1	arithmetic	logical	bit field	multiply	divide	branch	load	store	fp multiply	fp add/sub	fp divide	graph. mult.	graph. add	graph. pack
Integer	arithmetic	■	■	■	■	■	■	■	■	■	■	■	■	■	■
	logical	■	■	■	■	■	■	■	■	■	■	■	■	■	■
	bit field	■	■	□	■	■	■	■	■	■	■	■	■	■	■
	multiply	■	■	■	□	■	■	■	■	□	■	■	□	■	■
	divide	■	■	■	■	□	■	■	■	■	■	□	■	■	■
Flow Control	branch	■	■	■	■	■	□	■	■	■	■	■	■	■	■
Memory	load	■	■	■	■	■	■	□	□	■	■	■	■	■	■
	store	■	■	■	■	■	■	□	□	■	■	■	■	■	■
Floating Point	multiply	■	■	■	□	■	■	■	■	□	■	■	□	■	■
	add/sub/cmp	■	■	■	■	■	■	■	■	■	□	■	■	■	■
	divide	■	■	■	■	□	■	■	■	■	■	□	■	■	■
Graphics	multiply	■	■	■	□	■	■	■	■	□	■	■	□	■	■
	add & sub	■	■	■	■	■	■	■	■	■	■	■	■	□	■
	packing	■	■	■	■	■	■	■	■	■	■	■	■	■	□

■ Can be issued simultaneously
□ Cannot be issued simultaneously

Table 3.1.1.3 - Simultaneous Instruction Issue

NOTES: *1)* Integer arithmetic instructions which affect the carry flag (**add**, **sub**, **addu**, or **subu** with **.co** or **.cio** suffix) will prevent issue of any other instruction in the same clock when they are issued as the first instruction in an issue pair.

2) Floating Point add/sub/cmp includes **fadd**, **fsub**, **fcmp**, **fcvt**, **flt**, **int**, **nint**, and **trnc**.

3) The **lda** instructions have the same restrictions as **ld**.

4) Any instruction in the delay slot of a **bsr.n** which specifies r0 or r1 as a destination, will not issue in the same clock as the **bsr**.

5) All instructions which cause the machine to serialize (**xmem**, **tb0**, **tb1**, **tcnd**, **rte**, **ldcr**, **xcr**, **stcr**, **fldcr**, **fxcr**, and **fstcr**), will not issue in the same clock with any other instruction,

6) If the serialize bit in the PSR is set, then simultaneous instruction issue is disabled and at most only one instruction issues per clock.

3.1.1.4 Instruction Flow Control

Flow control operations (branches) are expensive to execute because they disrupt normal flow in the instruction pipeline. A change in program flow creates bubbles in the pipeline due to the time required to fetch the new target instruction stream. In typical code, with 4 or 5 sequential instructions between branches, the machine could waste up to 25% of its execution bandwidth waiting on branch latency.

The 88000 architecture has an optional branch delay slot after each branch instruction so that the compiler can try to schedule useful work into at least one of the bubbles. However, the high instruction issuing bandwidth of the 88110 can produce up to three bubbles for each branch. Thus a single architectural delay slot is insufficient to hide all branch latencies.

To minimize instruction bubbles caused by branch latency, the 88110 accelerates fetching the new target instruction stream by employing a branch Target Instruction Cache (TIC). The TIC is automatically filled the first time the target of a taken branch is fetched from the instruction cache. The next time the branch instruction is fetched, its target instructions are also fetched from the TIC thus allowing instruction flow to proceed unimpeded.

The 88110 implements a branch reservation station and static branch prediction to allow branches to issue as early as possible. The reservation station allows a branch instruction to issue even before its source operand is ready. With the branch issued and out of the way, instruction issue can continue while the branch operand is being computed and the condition is being evaluated. Static branch prediction is used to determine which instruction stream is executed while the branch is being resolved. When the branch operand becomes available it is forwarded to the branch unit and the condition is evaluated. Branch instructions whose source data is not available and must issue to the reservation station are said to be "predicted." Predicted branches which later turn out to have followed the wrong path, are said to be "mispredicted." Branch instructions which issue with source data already available are "unpredicted." Instructions issued under a predicted branch are said to be issued "conditionally." The 88110 will automatically back out the effect of conditionally executed instructions issued under a mispredicted branch.

3.2. SFU0 (BASE PROCESSOR) IMPLEMENTATION

3.2.1. r0 AS A DESTINATION REGISTER

In the 88110, using r0 as a destination register works as it did in the 88100 except for the special case of **ld** r0,rS_1,rS_2. In all other cases, instruction side effects, such as setting the carry flag or causing an exception, occur as if the destination register were a register other than r0. **ld** r0 is defined as a "touch" operation (described later) which has special characteristics in this regard.

3.2.2. CODE, ADDRESS, AND DATA MODIFICATION

3.2.2.1. Self-Modifying Code

Although not supported transparently from user mode, it is possible to run self-modifying code on the '110. To alter the executing instruction stream the user must perform a trap to a supervisor program which will clear the instruction pipelines, make the requested code change, assure all caches are coherent, and return to the user program. Any attempt to directly alter the instruction stream from the user program is considered a programming error and the results are unpredictable.

3.2.2.2. Code Relocation

Code relocation is supported in the same manner as self-modifying code described in the previous paragraph.

3.2.2.3. Logical Address Aliasing

Logical Address aliasing is fully supported on the 88110 with no associated performance penalty.

3.2.3. DOUBLE-WORD ALIGNMENT IN THE GENERAL REGISTERS

As suggested in the floating point architecture section, not all implementations will support misaligned double-word operands in the general register file at full performance. The 88110, in fact, will implement misaligned doubles in software. Any SFU0 instruction which specifies an odd numbered register for a double-word operand will cause a Unimplemented Opcode exception. Floating point instructions will cause an SFU1 Floating Point Unimplemented exception and graphics instructions will cause an SFU2 exception.

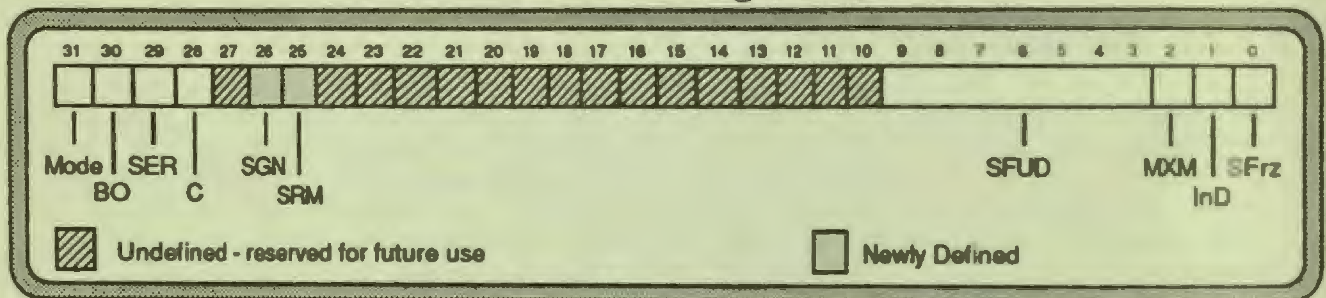
Software handlers will be provided to emulate the SFU0 and SFU1 instructions which use misaligned double operands. However, these handlers are provided only to insure code compatibility with existing applications. Applications which use misaligned double-word operands extensively should be recompiled to realize the full

performance potential of the 88110. Some 88000 compilers already enforce even register alignment; those which do not, should begin doing so immediately.

3.2.4. PROCESSOR STATUS REGISTER

The PSR is not directly visible from user mode code and is therefore not an architectural feature. However, certain features, required by the architecture, are controlled by the PSR. (The PSR is provided as a central repository for controlling many of these features anticipating that the PSR will be maintained by the operating system as a part of each process's state which is saved and restored on context switches.) The format chosen for the PSR is upward compatible with that used on the 88100. Extensions to the 88100 PSR are highlighted in gray in the diagram below.

Processor Status Register (PSR)



Mode:	0 - user mode
	1 - supervisor mode
BO:	0 - big endian Byte Order
	1 - little endian Byte Order
SER:	0 -concurrent instruction execution
	1 - SeRial Instruction execution
C:	0 - no carry out of add/addu/sub/subu.co (or .cio)
	1 - Carry out of add/addu/sub/subu.co (or .cio)
SGN:	0 - Immediate offsets and constants are unsigned
	1 - Immediate offsets and constants are SiGNed 2's complement.
SRM:	0 - concurrent memory instruction execution
	1 - SeRialize Memory instructions
SFUD:	0000000 - SFU7-SFU1 enabled
	1111111 - SFU7-SFU1 disabled
MXM:	0 - Misaligned access eXception Mode enabled
	1 - Misaligned access eXception Mode disabled
InD:	0 - external interrupt enabled
	1 - external INterrupt Disabled
SFrz:	0 - shadowing enabled
	1 - Shadow registers Frozen
////:	Undefined[1] - reserved for future use

Figure 3.2.4 - Processor Status Register

---

1 Throughout this document, control register bits labeled "Undefined - reserved for future use" should be treated in the following manner:
1) When the control register is read, undefined bits will appear as a zero. However, to insure compatibility with future machines the value of an undefined bit should never be depended upon by software. Software should explicitly test only those bits it needs to see.
2) The machine will ignore any data written to an undefined bit. However, once again for future compatibility sake, undefined bits should always be set to zero.

The PSR is accessed as SFU0 control register 1 (cr1) using the **ldcr, xcr** or **stcr** instructions from supervisor mode only. Any change to the PSR via **stcr** or **xcr** will take affect prior to the next instruction. A program change to the carry bit in the PSR will be seen by an immediately following **ldcr**. Any attempt by user mode code to access this register will result in an Privilege Violation Exception.

3.2.4.1. Operating Modes

Supervisor/user mode, byte-order mode, signed-immediate mode, and misaligned access exception mode are all implemented in the 88110 as described in the architecture section. These modes are each controlled by single bit enables in the PSR. The operation of the two serial execution modes, however, are unique to particular implementations; their operation in the 88110 is described in this section.

3.2.4.1.1. Serial Instruction Execution Mode

The 88110 normally has multiple instructions in execution at any point in time and these instructions may complete out of program order. This concurrency and out of order execution is usually of no functional consequence to the program because the machine has full hardware facilities to correctly reorder the results into program order. If it is necessary, or convenient, to force instructions to issue and complete in the exact order specified by the program, the 88110 can be forced to do so by setting the SER bit in the PSR. When this bit is set, the machine will be forced to completely "serialize" before each new instruction is issued. In this mode, the inherent concurrency in the machine is lost, and a commensurate reduction of performance will be experienced.

3.2.4.1.2. Serial Memory Reference Mode

The 88110 data memory unit has a series of instruction buffers which hold memory instructions that are either waiting for data from a previous computation or waiting for access to the cache. Load instructions in these buffers are sometimes allowed to run ahead of store instructions which actually issued earlier. The hardware prevents this reordering when it would violate the logical intent of the program but otherwise load and store instructions may execute - and cause bus transactions - out of program order. Normally this has no functional impact on the program, but there are some instances where it is not permissible. For example, some I/O device control registers must be accessed in the exact sequence specified by the program. The 88110 has no way of detecting this situation dynamically. Therefore, a mode is provided which allows the program (operating system) to force memory references to be completed in strict program sequence. The mode is enabled and disabled by the SRM bit in the PSR so that it can be maintained as part of the process state.

Whenever a **ld** or **st** instruction is executed with the SRM bit set, the machine will completely serialize (as described previously) before it is issued. This ensures that all load and store transactions on the bus will run in strict program sequence.

3.2.4.2. Carry Flag

The carry out from base architecture add and subtract instructions which specify a **.co** or **.cio** extension must be held in the processor for use by subsequent add and subtract instructions which specify the **.ci** or **.cio** extension. As a result the carry must be considered a part of the processor state and its value must be preserved across context switches. The 88110 holds the carry flag in the PSR along with other process specific state information so that it may quickly be saved and restored on context switches.

The carry flag is not sticky, i.e., any instruction which specifies **.co** or **.cio** will set or clear it depending on the carry out of the operation. It is modified on each occurrence of an **add.co**, **sub.co**, **addu.co**, **subu.co**, **add.cio**, **sub.cio**, **addu.cio**, or **subu.cio** instruction and is used as the carry-in to the **add.ci**, **sub.ci**, **addu.ci**, **subu.ci**, **add.cio**, **sub.cio**, **addu.cio**, and **subu.cio** instructions. No other user mode instructions look at or modify this bit.

3.2.4.3. SFU Disable

One disable bit for each of the seven SFU's (SFU1-7; SFU0 cannot be disabled) is allocated in the PSR. The 88110 implements only two of these bits (PSR bits 3 and 4) for its two function units (SFU1 and SFU2). The remaining SFU disable bits are hardwired to a "one." SFU1 and SFU2 are automatically disabled on any exception or reset. Any attempt to execute an SFU instruction for an SFU that is disabled will result in an exception being taken through the corresponding SFU exception vector.

3.2.4.4. External Interrupt Enable

The 88110's maskable external interrupt will be ignored while the InD bit in the PSR is set. The InD bit is automatically set to disable interrupts when any exception occurs and must be explicitly cleared by supervisor mode code to enable interrupts. This bit does not affect the operation of the non-maskable interrupt (NMI).

3.2.4.5. Shadowing Frozen

The 88110 implements a precise exception model. On an exception, the hardware backs the machine up to its state at the point of the exception and saves the instruction pointer (XIP) of the excepting instruction and corresponding the Processor Status Register (PSR) in a set of shadow registers (the EIP and EPSR). The exception also causes the SFrz bit in the new PSR to be set, which freezes the shadow registers until the exception handler can save them off to memory. If any exception occurs while the SFrz bit is set, the processor will take the Unrecoverable Error exception. Once the shadows are safely saved away in memory, the exception handler can turn off SFrz, thus allowing another exception to be handled.

3.2.5. SFU0 CONTROL REGISTERS

Each SFU has it own control register space of 64 registers which are accessible using the **ldcr**, **stcr** and **xcr** instructions. All SFU0 control registers (cr0-cr63) are

accessible only from supervisor mode - any attempt to access them from user mode will result in a Privilege Violation exception.

All control registers accesses via **ldcr**, **xcr**, or **stcr** serialize the machine, i.e. all instructions in execution are forced to complete before the control register access is performed.

All 64 control registers are not completely implemented. Reading an unimplemented control register causes either an Unimplemented Opcode or a Privilege Violation exception. Unused bits within an implemented register are always read as zeros and writing to them has no affect. However, unused bits should always be written as zeros to assure compatibility with future implementations which use might these bits.

The control register assignment is shown in the table in Appendix A.2.

3.2.6. SFU0 EXECUTION UNITS

3.2.6.1. Integer Unit

The integer unit (INTU) in the 88110 is composed of three subunits - two identical arithmetic/logic units (ALU's) and one bit-field unit (BFU). Each subunit has a one clock execution phase and can process instructions at a throughput rate of one instruction per clock. Since the maximum issue rate of the machine is 2 instructions per clock and there are two ALU's, an integer arithmetic instruction is never delayed due to integer ALU unavailability; it is said to have a "blockage" of zero clocks.

Bit-field instructions may issue to the BFU from either the first or second instruction issue slot but since there is only a single BFU only one bit-field instruction can issue per clock, i.e., the BFU has a blockage of one clock.

Instructions executed by the ALU's include: **and, mask, xor, or, addu, subu, add, sub,** and **cmp**. Instructions executed by the BFU include: **clr, set, ext, extu, mak, rot, ff1,** and **ff0**.

3.2.6.2. Load/Store Unit

3.2.6.2.1. Overview and Block Diagram

The 88110 load/store unit (LD/ST) handles all the traffic between the on-chip data cache and the register files. The load/store unit is implemented as an independent execution unit so that stalls in the memory pipeline do not cause the master instruction pipeline to stall (unless of course there is a data dependency). The unit is fully pipelined so that memory instructions of any size may be issued on back-to-back cycles.

There is a 64-bit wide data path between the cache and load/store unit and a full 80-bit wide data path between the load/store unit and the register files. Single-word and double-word data require one clock to access the cache while extended precision floating point numbers require two clocks. All data transfers between the load/store

unit and the register files occur in a single clock cycle; thus load and store extended do not stall instruction issue. A conceptual block diagram of load/store unit operation is shown below:

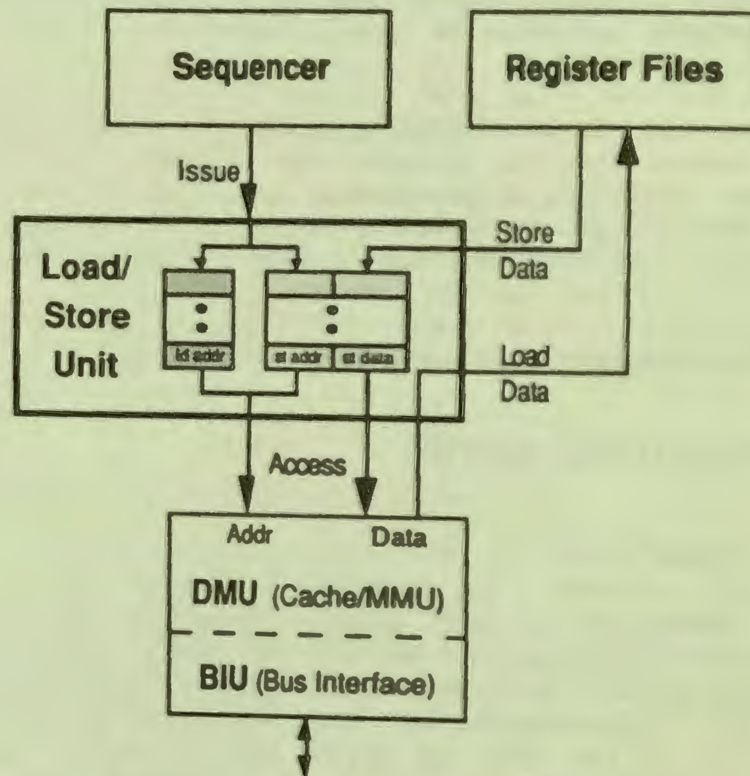


Figure 3.2.6.2.1 - Load Store Unit Block Diagram

The load/store unit has a series of load address buffers (dark shading) and store address/data buffers (light shading) each set of which operate as independent FIFO queues. The store buffers allow store instructions to issue before store data is available from a previous computation, i.e., scoreboard holds on the destination register do not delay the issue of stores. The sequencer forwards data directly into the store data buffer as soon as it becomes available. The load buffers allow multiple additional loads to be issued while a previous load is stalled waiting for data from the cache or bus. Loads are allowed to bypass stores which are stalled in the store buffers waiting for data. Memory hazards are prevented by comparing load addresses to store addresses and disallowing loads to run ahead of stores for which there is an address match.

3.2.7.2.1.1 Load/Store Instruction Issue

When a load or store instruction is encountered, the sequencer first checks the scoreboard to determine if the address operands are available and, for a load only, makes sure there is no destination register conflict. It then checks the load/store unit to verify that there is an available instruction buffer. If so, the instruction is issued to the appropriate load or store queue.

If there are no prior instructions waiting in the respective instruction queue and the cache is not busy servicing a prior request, then the load or store address falls

through the queue directly to the MMU and cache. If the cache is busy or if there are already instructions pending in the buffers then the new instruction is issued to the queue.

3.2.7.2.1.2 Load/Store Run Time Re-Ordering

The load/store unit always runs loads in program order with respect to other loads and stores in order with respect to other stores, but does allow loads to run out of order with respect to stores. In the event that a store is stalled in the store buffers waiting for data from a previous computation, subsequent load instructions are allowed to bypass them and get access to the cache. This allows run time overlapping of tight loops; loads from the top of a loop can pass stores from the previous loop iteration which are still waiting for a computation to complete.

In order to guarantee correct program functioning, the only restriction that must be imposed on load/store re-ordering is that a load which is allowed to run out of order cannot access the same memory location as any store it is allowed to bypass. To enforce this, the load/store unit compares the low order bits of each load's address with all store instructions waiting in the store buffers. If a match is detected, the load's access to the cache is delayed until the aliased store completes.

3.2.7.2.1.3 Cache Access

Once load and store instructions have been issued to a buffer in the load/store unit, the sequencer is free to continue issuing other instructions. The load/store unit then executes these buffered instructions as the cache and memory become available. When both load and store buffers have instructions queued up, the load store unit gives first priority to stores then loads.

3.2.7.2.1.4 Load/Store Instruction Timing

A load instruction which hits the data cache has a latency of 2 clocks. This means that an instruction with a data dependency on the load will experience a delay of 2 clocks with respect to the earliest it could have executed had it not had the data dependency. The latency for a load which misses the cache, assuming zero wait state memory references, is four clocks.

3.2.7.2.1.5 Serialization of Memory References

Correct operation of some I/O device control registers require that memory cycles to them run in the exact order specified by the program. Two alternatives are provided for forcing this strict ordering: 1) the trap not taken (**tb1** 0,r0,vector) instruction will cause the machine to completely serialize, i.e, all pending instructions will be allowed to complete and store buffers emptied before the next instruction is allowed to execute, and 2) setting the I/O Serialization (SRM) bit in the PSR will cause all load and store instructions to serialize the machine before they issue.

3.2.7.2.1.6 Store-through

An option is provided on the triadic register form of stores called "store-through." The store-through option serves two basic purposes: first, it provides a mechanism to force data through the cache into memory on a per instruction basis, which can be useful in cases such as writing to a display screen (frame buffer); and second, it provides a way to prevent the storage of certain data which the program knows will not be reused, from replacing a potentially more useful line in the cache. This not only avoids the wasted time of moving a line out and back in again, but also improves the hit rate of subsequent loads to those lines.

When specified, this option unconditionally causes the store to write through the cache directly into memory. If a store-through hits the cache on its way through, the cache is updated but the line is not marked dirty unless it is already dirty. If the store-through misses the cache then no line is allocated in the cache, i.e., no dirty line copyback is forced, no new line is brought into the cache, no existing line is replaced, and no data is written into the cache. The access simply goes directly through to memory, bypassing the cache completely.

Store through is specified by a **.wt** (for Write-Through) on any triadic register form of the **st** instruction. All operand sizes and both register files are supported:

st.wt.t	rD,rS_1,rS_2	.t = .b (byte), .h (half), {blank} (word), or .d (double)
st.wt.t	$rD,rS_1[rS_2]$	.t = .b (byte), .h (half), {blank} (word), or .d (double)
st.wt.t	xrD,rS_1,rS_2	.t = {blank} (single), .d (double), or .x (extended)
st.wt.t	$xrD,rS_1[rS_2]$	.t = {blank} (single), .d (double), or .x (extended)

3.2.7.2.1.7 Touch Load

A new feature is implemented in the 88110 to allow data to be scheduled into the cache under program control. Normally, data is brought into the cache automatically on demand, i.e. when it is needed. This can lead to instruction execution stalls due to dependencies on data which must come all the way from main memory. In many cases, however, it can be predicted, well in advance, which data is going to be needed by the program. By requesting data be brought into the cache ahead of it actual use, the latency of the memory system can be overlapped with useful work and stalls due to long latency cache misses can be minimized. The touch load is provided for this purpose. A touch is specified by a *word* load to r0:

ld $r0,rS_1,rS_2$
ld $r0,rS_1[rS_2]$

> If the cache line containing the data specified by the effective address of the load is not already in the cache (valid), then it is brought into the cache replacing an existing line if necessary. No user visible machine state is modified and no exceptions (e.g. MMU or bus error) are recognized.

A touch load is similar in most respects to other loads but has a couple of important distinctions. First, it never generates an exception and therefore the machine need never recover from it. This means that touch need not block retirement of instructions from the history buffer like a normal load can. Second, although it may bring data into the cache, it does not write a result into the register files. Thus, loads

executed under a touch need not run in program sequence with the touch operation, and can be allowed access to the cache while waiting on the touch's line fill to begin.

Past and future implementations which do not support this feature remain compatible because this instruction does not affect the user program visible state. Therefore, whether it actually performs the memory reference or not is irrelevant to the functionality of the program - although it may indeed affect its performance.

3.3. SFU1 (FLOATING POINT) IMPLEMENTATION

3.3.1. IEEE CONFORMANCE

Some features of the IEEE 754 specification are important functionally but occur rarely in practice, e.g. NaN's, invalid operations, denormalized numbers, etc. Since precise compliance with the specification in many of these cases is difficult to achieve in hardware, the 88110 implements these features in a software envelope wrapped around the hardware. The relative frequency with which these conditions occur is so low that this does not introduce a significant performance penalty.

The SFU1 design provides for the hardware/software partitioning of functionality by taking an SFU1 exception whenever one of the IEEE spec features which is not implemented in hardware is encountered. The hardware reports the cause of the exception in the Floating Point Exception Cause Register (FPECR) so that the software can take the appropriate action. When the SFU1 exception cause dictates that an IEEE exception should be signaled, the software envelope maps the status reported in the FPECR to the correct IEEE exception and signals it to user code through the FPSR. If the software sees that the trap corresponding to the signaled IEEE exception is enabled in the FPCR it will vector to the User Trap Handler. If the trap is disabled, then the software envelope will place the correct IEEE default result in the destination register and return to the user program. When the cause of the SFU1 exception is an unimplemented feature, like square-root or denormalized operand, then the software envelope will perform the operation, put the result in the destination register, and return to the user program without the user ever being aware that the operation was actually performed in a software routine. The process is illustrated pictorially in the following figure:

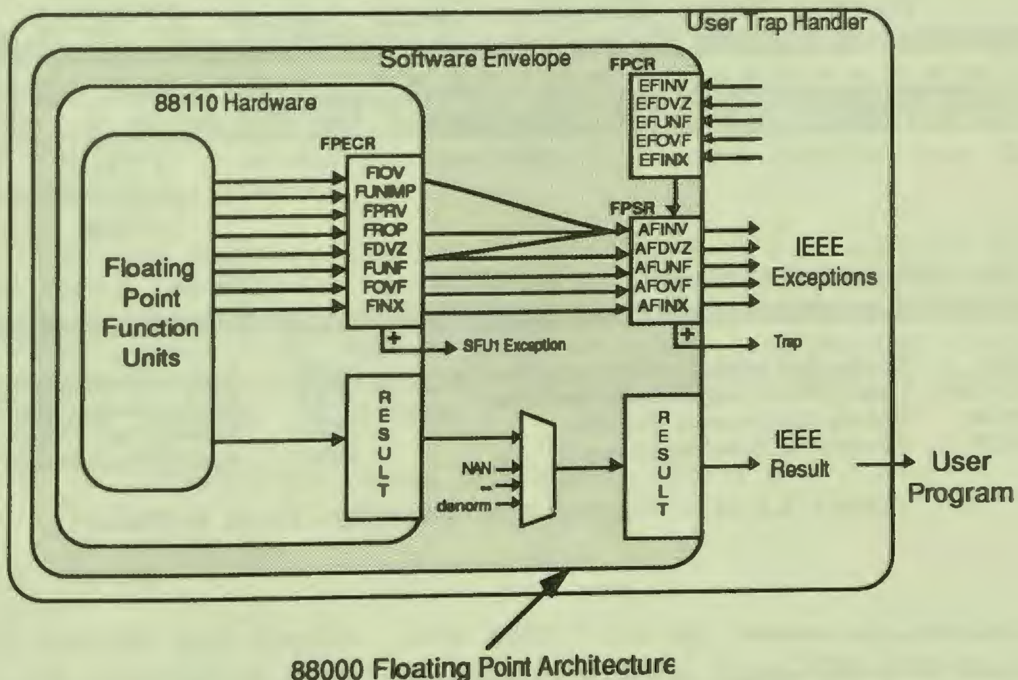


Figure 3.3.1 - Floating Point Architecture

Motorola Confidential Proprietary - DO NOT COPY

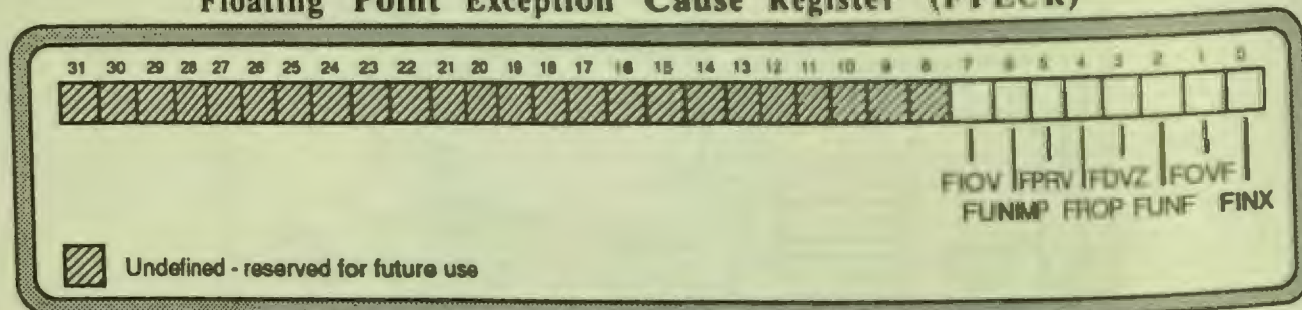
3.3.2. IEEE EXCEPTIONS

The 88110 implements the IEEE exceptions with the assistance of a software envelope as described above. When the hardware detects an invalid operation, it generates an SFU1 exception using the SFU1 exception vector[1]. The hardware reports the cause of the exception in the Floating Point Exception Cause Register (FPECR). The software envelope then analyzes the operands and FPECR and, if necessary, reports the appropriate IEEE exceptions to the user through the FPSR (described in the architecture section). Operands and operations which cause SFU1 exceptions are enumerated in a table later in this section.

3.3.2.1. Floating Point Exception Cause Register

The FPECR is an internal control register used to communicate between the hardware and a user invisible software envelope. An implementation which implements the entire IEEE spec in hardware would have no need for this register. Therefore, it is not considered a component of the floating point architecture. Indeed, any implementation which employs a register like the FPECR, must insure that that it is not visible to user mode code. The 88110 implements this register as SFU1 floating point control register 0 (fcr0) which is accessible only from supervisor mode by the **fldcr**, **fxcr**, and **fstcr** instructions. Any attempt to access it from user mode will cause a Floating-Point Privilege Violation exception.

The FPECR used by the 88110 is identical with the FPECR used in the 88100 (however, not all flags are set in precisely the same manner as in the 88100). The register format is shown in the next figure.



FIOV:	Floating-Point to Integer Conversion Overflow	FDVZ:	Floating-Point Divide-by-Zero
FUNIMP:	Floating-Point Unimplemented Instruction	FUNF:	Floating-Point Underflow
FPRV:	Floating-Point Privilege Violation	FOVF:	Floating-Point Overflow
FROP:	Floating-Point Reserved Operand	FINX:	Floating-Point Inexact

Figure 3.3.2.1 - Floating Point Exception Cause Register

1 Throughout this document references are made to SFU exceptions such as the Floating Point Unimplemented exception or Floating Point Overflow exception. Actually, all SFU exceptions trap through the SFU1 exception vector and the cause of the exception is recorded in the Floating Point Exception Cause Register. For purposes of convenience in the documentation, each FPECR cause will be referred to as a separate exception; for example, an SFU exception with the FUNIMP bit set in the FPECR is referred to as a Floating Point Unimplemented exception.

3.3.2.2. Exception Causes

The following table shows the conditions which will generate an SFU exception and cause bits in the FPECR to be set.

	FUNIMP	FROP	FOVF	FUNF	FDVZ	FINX	FIOV
fmul	SFU1 off	NaN, invalid, denorm, or unnorm	Overflow	Underflow		Inexact	
fadd	SFU1 off	NaN, invalid, denorm, or unnorm	Overflow	Underflow		Inexact	
fsub	SFU1 off	NaN, invalid, denorm, or unnorm	Overflow	Underflow		Inexact	
fcvt	SFU1 off	NaN, invalid, denorm, or unnorm	Overflow	Underflow		Inexact	
fcmp	SFU1 off	NaN, invalid, denorm, or unnorm					
fcmpu	SFU1 off	NaN, invalid, denorm, or unnorm					
flt	SFU1 off					Inexact	
int	SFU1 off	NaN, invalid, denorm, or unnorm				Inexact	$rS_2<-2^{31}$, $rS_2> 2^{31}-1$
nint	SFU1 off	NaN, invalid, denorm, or unnorm				Inexact	$rS_2<-2^{31}$, $rS_2> 2^{31}-1$
trnc	SFU1 off	NaN, invalid, denorm, or unnorm				Inexact	$rS_2<-2^{31}$, $rS_2> 2^{31}-1$
fdiv	SFU1 off	NaN, invalid, denorm, or unnorm	Overflow	Underflow	$rS_2=0$	Inexact	
fsqrt	Always						

Table 3.3.2.2 - Floating Point Exceptions

Definitions of the terms and acronyms used in the above table:

FUNIMP: Floating Point Unimplemented. This FPECR bit is set whenever an SFU1 class opcode which is not implemented in hardware (e.g. fsqrt) is executed. If a FUNIMP is signaled then the values of all other exception cause flags in the FPECR are undefined.

FROP: Floating Point Reserved Operand. This FPECR bit is set whenever either of the source operands is a "reserved operand" or the operation being performed on the given operand is invalid according to the IEEE specification. A reserved operand is any NaN (signalling or quiet), a denormalized number, or an extended precision unnormalized number. Invalid operations include the following operations on infinity:

a) magnitude subtraction of infinities, such as $(+\infty) + (-\infty)$,
b) $0 \times \infty$,
c) ∞ / ∞, and
d) 0/0

FOVF: Floating Point Overflow. This FPECR bit is set whenever the rounded result of the operation is too large to be represented as a finite number in the destination format (sgl, dbl, or ext).

FUNF: Floating Point Underflow. This FPECR bit is set whenever the rounded result of the operation is too small to be represented as a finite normalized number in the destination format. Any time the hardware attempts to generate a denormalized result, it will instead set FUNF and take the SFU1 exception.

FDVZ: Floating Point Divide by Zero. This FPECR bit is set whenever the denominator (rS_2) operand of an **fdiv** instruction is a zero and the numerator is a non-zero finite normalized number. FDVZ is set even if the numerator (rS1) is also zero.

FINX: Floating Point Inexact. This FPECR bit is set whenever rounding of the result of a floating point arithmetic operation or floating point to integer conversion causes a loss of accuracy. FINX can occur by itself or along with either FOVF or FUNF.

FIOV: Floating Point to Integer conversion Overflow. This FPECR bit is set whenever source operand of a floating point to integer conversion (**int**, nint, trnc) is too large to be represented as a signed 32-bit integer.

FPRV: Floating Point Privilege Violation. This FPECR bit is set by an **fldcr**, an **fxcr**, or an **fstcr** which attempts to access floating point control registers fcr0 - fcr61 from user mode.

3.3.2.3. Exception Mapping

When the hardware sets a bit in the FPECR it also generates an SFU1 exception which invokes the software envelope (except in the special case of FINX described later). The software envelope maps the FPECR status flags to the IEEE accumulated exception flags (AFxxx) in the FPSR according to the following table:

FPECR		FPSR	
FIOV	⇒	AFINV	
FROP	⇒	AFINV	only if cause of FROP was: a) signalling NaN, b) +∞ + -∞, c) 0 * ∞, d) ∞/∞, or e) 0/0
FDVZ	⇒	AFDVZ	
FUNF	⇒	AFUNF	only if ENUNF is enabled
FOVF	⇒	AFOVF	
FINX	⇒	AFINX	
FOVF && FINX	⇒	AFOVF and AFINX	
FUNF && FINX	⇒	AFUNF and AFINX	

Table 3.3.2.3 - Floating Point Exception Mapping

In cases not explicitly called out in this table, for example FROP due to quiet NaN, the software envelope will simply return the correct IEEE result and not signal any AFxxx exceptions to the user. Any time an AFxxx bit is set in the FPSR by the software envelope (even if it's already set), and the corresponding EFxxx trap is enabled in the FPCR, then a branch is taken to a User Trap Handler. If the EFxxx trap is disabled, the

software envelope will place the specified IEEE 754 default result in the destination register and return to the user program leaving the AFxxx bit set in the FPSR.

3.3.2.4. Floating Point Infinity Arithmetic

In the 88100, infinity arithmetic was performed in software. An infinity was always treated as a reserved operand, i.e., a Floating Point Reserved Operand exception (FROP) was generated if either source operand was an infinity. The 88110 performs infinity arithmetic in hardware. The correct IEEE result is generated for all valid operations on infinity. FROP is asserted only if the operation being performed is invalid according to the IEEE specification. In any case where a new infinity would be produced, the 88110 will generate an Floating Point Overflow (FOVF) exception.

Invalid operations on infinity are:

a) magnitude subtraction of infinities, such as $(+\infty) + (-\infty)$,
b) $0 \times \infty$, for all combinations of signs of zero and infinity,
c) ∞ / ∞, for all combinations of signs of infinities.

3.3.2.5. Floating Point Inexact

The FPCR and the FPSR are simply software registers, i.e. user code loads the FPCR and the software envelope reads the FPCR and updates the FPSR. These registers have no direct affect on the hardware. There is one exception to this - Floating Point Inexact (FINX). If the floating point unit signals an inexact condition, the hardware will check the Floating Point Inexact Enable (EFINX) bit in the FPCR. If it is disabled, the processor will not generate an SFU1 exception but does set the AFINX bit in the FPSR. If EFINX is enabled, then the FINX bit in the FPECR and the AFINX bit in the FPSR are both set and the processor takes the SFU1 exception.

3.3.2.6. Floating Point to Integer Conversion Overflow

In the 88100, if the operand being converted to an integer has an exponent greater than or equal to 30 then a Floating Point Integer Conversion Overflow exception is taken. In the 88110, the conversion range is extended. Only if the operand being converted will not fit in a signed 32-bit integer is a Floating Point Integer Conversion Overflow exception taken.

3.3.2.7. Unnormalized Extended Precision Numbers

The 88100 did not support IEEE double-extended numbers and therefore did not need to deal with unnormalized operands. The 88110 handles unnormalized extended precision numbers in the software envelope rather than in hardware. If an unnormalized operand is encountered, the machine will take a Floating Point Reserved Operand exception (FROP in FPECR). The software envelope will normalize the number, perform the operation, and return the result in the destination register as if the hardware had performed the operation. The original unnormalized source operand remains unnormalized.

3.3.3. TIME-CRITICAL FP. OPERATIONS (SLZ MODE)

THIS IS AN IMPLEMENTATION DEPENDENT USER MODE FEATURE. Do not depend on it to be supported in the same manner on future 88000 implementations.

The 88110 depends on a software envelope to fully implement the IEEE floating point specification - overflows, underflows, NaN's, and denormalized numbers, all cause SFU1 exceptions which invoke a software routine to deliver the correct IEEE result. To accelerate time critical operations and make them more deterministic, the 88110 provides a mode of operation which avoids invoking the software envelope and attempts to deliver "sensible", if not strictly IEEE correct, results in hardware. In this mode, denormalized numbers, quiet NANs, and IEEE invalid operations are treated as legitimate, returning default results rather than causing SFU1 exceptions. When this mode is enabled, the hardware assumes the job normally performed by the software envelope.

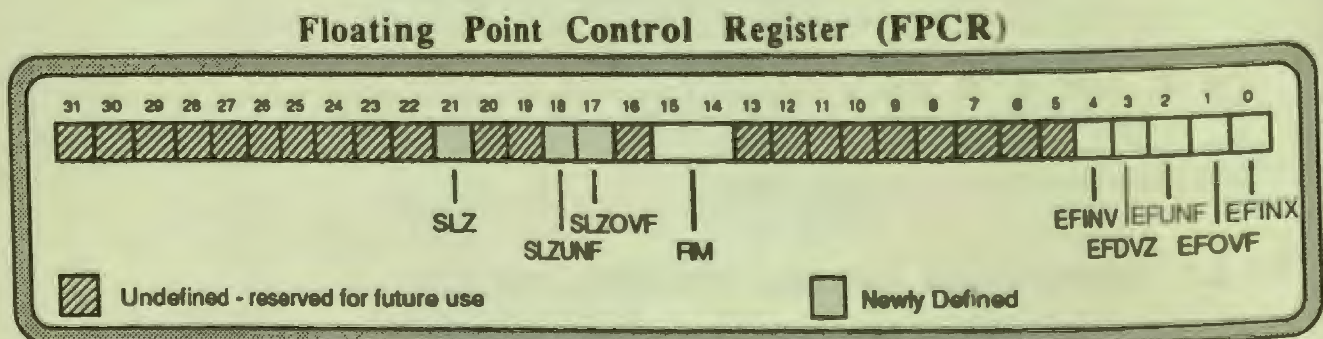
SLZ mode operation is provided individually for overflow and underflow, or as a global mode covering all arithmetic exceptions. To activate SLZ mode, the user would set the appropriate SLZ bit in the FPCR. Setting a SLZ bit causes the hardware to block corresponding SFU1 exceptions and deliver default results. If the SFU1 exception is not disabled by a SLZ bit in the FPCR, the SFU1 exception is taken and the bits in the FPECR indicate the cause of the exception. The value of any IEEE exception flag in the FPSR which is under the influence of SLZ mode, is undefined.

The Floating Point Privilege Violation (FPRV) exception, and the Floating Point Unimplemented (FUNIMP) exception are never blocked by any SLZ bit. The EFxxx trap enable bits (except EFINX) in the FPCR have no effect on the hardware.

3.3.3.1. SLZ Mode Control

SLZ mode is activated by three bits which have been added to the FPCR. The layout of the FPCR with SLZ control bits is shown below:

Floating Point Control Register (FPCR)



SLZOVF:	Flush result to correctly signed infinity on overflow
SLZUNF:	Flush result to correctly signed zero on underflow
SLZ:	Deliver default results for all operations as described in the sections below.

Figure 3.3.3.1 - FPCR w/SLZ Control

Three bits are provided to selectively activate SLZ mode operation. SLZOVF enables default results for floating point overflow (only). SLZUNF does the same for floating

point underflow. SLZ is a global enable which causes default results to be delivered for all exceptional conditions (except FPRV and FUNIMP) as described in the section on default results below. A summary of the action taken for different settings of the FPCR trap enable (ENxxx) and SLZ mode control (SLZxxx) bits is shown in the table below.

Exceptional Condition Exists (Fxxx in FPECR)	FPCR SLZxxx	FPCR ENxxx	Signal IEEE Exception in FPSR	Take SFU1 Exception Vector	S/W Delivers IEEE Result	H/W Delivers IEEE Result	H/W Delivers Default Result	S/W Traps to User Handler
0	x	x				✓		
1	0	0	✓	✓	✓			
1	0	1	✓	✓				✓
1	1	x					✓	

Table 3.3.3.1 - SLZ Mode Actions

3.3.3.2. SLZ Mode Default Results

In SLZ mode, if the SFU1 exception corresponding to a given FPECR flag is disabled by a SLZxxx bit in the FPCR, then a default result is written to the destination register. The default result returned is a function of the source operands and the operation being performed. The following table shows the conditions under which default results are generated. The next table shows the formats of the source operands which may cause default results to be generated and the formats of the default results generated by the 88110.

Condition (FPECR)	Source Operand(s)	Default Results			
		Compare	Convert to Int.	Add/Sub	Mul/Div
FIOV	± real	N/A	Large ± integer	N/A	N/A
	±∞	N/A	Large ± integer	N/A	N/A
FROP	±Signaling NaN	Unordered	Large ± integer	+qNaN	±qNaN
	± Quiet NaN	Unordered	Large ± integer	+qNaN	±qNaN
	± Unnormalized	Unordered	0	+qNaN	±qNaN
	± Denormalized	Unordered	0	+qNaN	±qNaN
	Invalid (∞-∞, 0×∞, ∞/∞)	N/A	N/A	+qNaN	±qNaN
FDVZ	±0 / ±0	N/A	N/A	N/A	±qNaN
	±real / ±0	N/A	N/A	N/A	± Infinity (±∞)
FUNF	±reals	N/A	N/A	± Zero (±0)	± Zero (±0)
FOVF	±reals	N/A	N/A	± Infinity (±∞)	± Infinity (±∞)
FINX	All	N/A	± Inexact result	± Inexact result	± Inexact result
FINX && FIOV	All	N/A	Same as FIOV	Same as FIOV	Same as FIOV
FOVF && FINX	All	N/A	N/A	Same as FOVF	Same as FOVF
FUNF && FINX	All	N/A	N/A	Same as FUNF	Same as FUNF

± For conversion to integer, the sign of the result is the same as the sign of the source operand. For addition and subtraction the result is correctly signed except for qNaN's which are always positive. For multiplication and division the result is correctly signed, i.e., the exclusive-or of the signs of the two source operands.

Table 3.3.3.2.a - SLZ Mode Default Results

Number	Recognized Values				Generated Values			
	Sign	Exponent	Leading	Fraction	Sign	Exponent	Leading	Fraction
Large + Integer	N/A	N/A	N/A	N/A	0	N/A	N/A	100...0
Large - Integer	N/A	N/A	N/A	N/A	1	N/A	N/A	100...0
+0	0	00...0	0	000...0	0	00...0	0	000...0
-0	1	00...0	0	000...0	1	00...0	0	000...0
+∞	0	11...1	x	000...0	0	11...1	0	000...0
-∞	1	11...1	x	000...0	1	11...1	0	000...0
+qNaN †	0	11...1	x	1yy...y	0	11...1	1	110...0
-qNaN †	1	11...1	x	1yy...y	1	11...1	1	110...0
+sNaN †	0	11...1	x	0yy...y	N/A	N/A	N/A	N/A
- sNaN †	1	11...1	x	0yy...y	N/A	N/A	N/A	N/A
+real	0	zz...z	1	xxx...x	0	zz...z	1	xxx...x
-real	1	zz...z	1	xxx...x	1	zz...z	1	xxx...x
+ denorm	0	00...0	x	yyy...y	N/A	N/A	N/A	N/A
- denorm	1	00...0	x	yyy...y	N/A	N/A	N/A	N/A
+ unnorm	0	zz...z	0	yyy...y	N/A	N/A	N/A	N/A
- unnorm	1	zz...z	0	yyy...y	N/A	N/A	N/A	N/A

* The leading bit is only meaningful for extended precision.

† qNaN in an IEEE quiet (non-signalling) NaN. A Universal qNaN is generated in all situations calling for a NaN result. sNaN is an IEEE signalling NaN and is never generated by the 88110 in SLZ mode.

x Don't Care, i.e. either 0 or 1.

y Don't Care, but yy...y is not all 0's.

z Don't Care, but zz...z is not all 0's or all 1's.

Table 3.3.3.2.b - Recognized and Generated Values

3.3.3.3. SLZ Mode Rounding

In SLZ mode, all rounding is performed in the current rounding mode as specified in the FPCR. In the case of overflow, rounding is performed in IEEE round-to-nearest mode regardless of the current rounding mode, i.e., the default results are always infinity (rather than the largest finite number called for by the IEEE spec in some rounding cases). **Int**, **nint**, and **trnc** round as specified by the instruction itself.

3.3.3.4. SLZ Mode Compare

In SLZ mode, both **fcmpu** and **fcmp** generate the correct IEEE results in all comparisons between normalized and ordered operands, i.e.:

a) $-\infty < -\text{real} < \pm 0 < +\text{real} < +\infty$,
b) $+0 = +0 = -0 = -0$,
c) $+\infty = +\infty$, and $-\infty = -\infty$

If any of the operands are unnormalized, denormalized, or unordered (NaN) then **fcmp** will set the unordered result bits (un, ne, ou, ob, ue, ug, ule, ul, uge) and clear all others (leg, eq, gt, le, lt, ge, ib, in, lg) - FROP is not set, no SFU1 exception is generated, and invalid (AFINV) is not signaled. However, **fcmpu** *will* set FROP and take the SFU1 exception to a software envelope in these cases. The software envelope will determine if the operands were unordered or not normalized. If they were not normalized it will normalize them and perform the comparison, returning the correct result to the program. If they were unordered it will signal invalid (AFINV) in the FPSR; if the invalid trap is disabled by the FPCR (EFINV=0), then it will return the result string with the unordered bits set and the others cleared; otherwise (EFINV=1) it will trap to the user handler.

3.3.3.5. SLZ Mode Handling of Denormalized Numbers

Whenever the result computed in SLZ mode underflows, the result returned is a correctly signed zero. When a denormalized number is detected as a source operand, a universal quiet NaN is returned for floating point results or zero for conversion to integer. Thus, the 88110 flushes denormalized results to zero but does not flush denormalized source operands to zero. Since the 88110 will never itself produce a denormalized result in SLZ mode, it should not need to deal with them as source operands unless they were contained in an original input data set.

The extended precision case of exponent=0, leading=1, is considered denormalized and will cause a qNaN to be returned. The 88110 will never generate this number.

3.3.4. SFU1 EXECUTION UNITS

3.3.4.1. Floating Point Add Unit

The floating point adder (FPADD) is a three stage fully pipelined design, capable of accepting one single, double, or extended precision addition every clock and requiring 3 clocks to complete execution, i.e., it has a latency of 3 clocks. The instructions executed by the floating point adder include **fadd**, **fsub**, **int**, **nint**, **trnc**, and **flt**.

3.3.4.2. Multiplier

The multiplier unit (MUL) is also a three stage fully pipelined design, capable of accepting one single, double, or extended precision multiply every clock with a latency of 3 clocks. The multiplier is shared between floating point, integer, and graphics operations. The instructions executed by the floating point multiplier unit include **fmul**, **mulu**, **mulu.d** , **muls**, and **pmul**.

3.3.4.3. Floating Point Divider Unit

The floating point divide unit (DIV) is a non-pipelined iterative design. The results produced are exact IEEE results with no software fix-up required. The instructions executed by the floating point divide unit are **fdiv**, **divs**, and **divu**.

The performance of the divide unit is dependent on the precision and type of the operands. Latency for each of the cases is shown in the following table. Since the divide unit is not pipelined, it cannot accept another divide instruction until it finishes with the one in progress. Thus, it has a blockage equal to its latency.

Instruction	Latency
divs, divu	19
divu.d	30
fdiv.sss (sgl)	13-14
fdiv.ddd (dbl)	23-24
fdiv.xxx (ext)	26-27

Figure 3.3.4.3 - Floating Point Divide Latency

3.3.4.4. Square Root

The 88110 does not implement square root in hardware. Executing the **fsqrt** instruction will cause a Floating Point Unimplemented exception. A software handler will be provided to emulate the square root operation as specified in the architecture.

3.4. SFU2 (GRAPHICS) IMPLEMENTATION

THIS IS AN IMPLEMENTATION DEPENDENT USER MODE FEATURE. Do not depend on this feature to be supported in the same manner on future 88000 implementations.

In the 88110, SFU2 is dedicated to a set of operations for accelerating 3-D graphics rendering algorithms. The graphics extensions are considered an implementation dependent feature as opposed to a fundamental architectural feature so that we retain the freedom necessary to track the rapidly evolving graphics technology.

3.4.1. GRAPHICS OVERVIEW

The graphics ***task*** is to convert an image, stored as an abstract representation of primitive shapes, and render that image as a realistic scene on a computer display. The graphics ***problem*** is to do it with dispatch. The process of rendering realistic animated 3-D images in real time is computationally intensive. The process has five major steps: 1) viewpoint transformation, 2) lighting, 3) shading, 4) image processing, and 5) display.

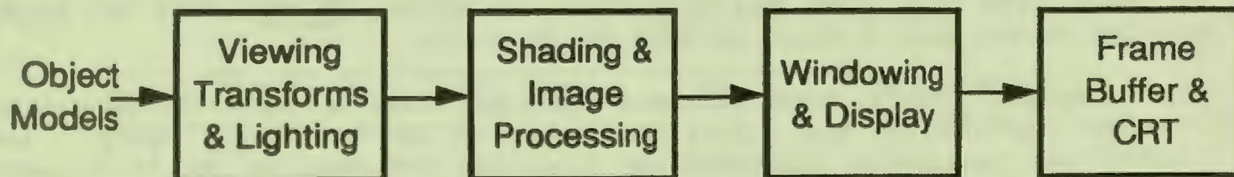


Figure 3.4.1 - 3-D Image Rendering

1) Viewpoint transformation. An image is typically represented as a group of complex polygons specified by vertex coordinates, normal vectors, surface properties, and color values. The vertices and normals must undergo a series of mathematical transformations (matrix multiplications) to orient the objects in space relative to the viewer, scale them to the right size, adjust for perspective foreshortening, and clip to the desired display volume. Coordinates are almost exclusively maintained and manipulated as floating point numbers.

2) Lighting. Ambient, diffuse, and specular lighting models can be applied to the image using ray tracing or other lighting algorithms. Surface detail polygons may be added to simulate texture. Color and lighting information is resolved to an RGB triple at each polygon vertex which specifies the component intensities of the three additive primary colors, red, green, and blue. Normally these are fixed point values.

3) Shading. The image must be clipped, projected into 2 dimensions, and mapped from image coordinate space to display coordinates. It is flattened, or decomposed, into simple triangles or scan aligned trapezoids. Shading algorithms are applied to make polygon facets appear solid, to smooth polygonally approximated surfaces, and to convert polygons to an array of pixels suitable for display on a raster scan display device. Color values are interpolated from vertex normals by averaging surface normals of adjacent polygon facets. Then, either linear intensity (Gouraud) or normal-vector (Phong) interpolation is performed to get polygon boundary colors. Color slope for each scan line crossing the polygon is computed and used to calculate the color of each pixel on the scan line internal to the polygon. As pixels are

computed, Z-buffer depth information is applied to remove hidden surfaces. Anti-aliasing corrections may also be applied to remove discrete-pixel spatial sampling errors which cause object edges to appear jagged.

4) Image Processing. It is both convenient and efficient to manipulate individual objects in a scene separately. For example, if a foreground object is rotated, it is more efficient to simply re-render that object than the entire scene. Compositing algorithms, capable of smoothly blending multiple images, are used to accomplish this. Such algorithms are also capable of accurately rendering the effect of object transparency. Compositing algorithms utilize a fourth channel, called alpha (α), which is appended to the three (RGB) color channels of each pixel. α specifies the percentage of a pixel covered by an object. Using α, the net contribution of a foreground and background object can be computed by interpolation to give a composite color.

5) Display. Once the image has been created as an array of pixels it must be transferred to and from the display system's frame buffer within the context of the governing windowing system. Fast bitblt (Bit Block Transfer) algorithms, area fills, line drawing, etc., are required to get images displayed quickly. Once pixels are placed into the frame buffer, specialized display system hardware constantly scans the frame buffer using pixel data to modulate the intensity of the CRT's red, green, and blue electron guns to form the image on the screen.

The exceptional floating point performance of the 88110 is sufficient to perform viewpoint transformation and lighting calculations on complex images rapidly. The flexible data manipulation instructions and high data throughput of the 88110 allow efficient coding of the bitblt and other algorithms necessary to achieve good display system performance. The shading, raster conversion, and image processing phases of the problem, however, are very compute intensive and require hardware support beyond that found in most conventional microprocessors to achieve good interactive performance. The graphics extensions in the 88110 are targeted at improving performance on these phases of the rendering process. Primitive operations are provided to perform arithmetic operations for shading, z-buffering, and compositing on multiple values in a single instruction.

3.4.2. GRAPHICS REGISTERS

All graphics operands are located in the general register file. No special registers or temporary registers are needed. Thus the graphics operations provide the same generalized flexibility as the other instructions in the architecture.

3.4.3. GRAPHICS DATA FORMATS

Graphics data tends to be packed tightly in memory to reduce storage requirements and reduce memory bandwidth requirements. The 88110 reads graphics data in 64-bit quanta. The graphics instructions then allow individual fields within the 64-bit words to be processed in parallel - avoiding the need to pull them apart and operate on them individually. Some examples of the more common graphics data formats supported by the 88110 are illustrated below:

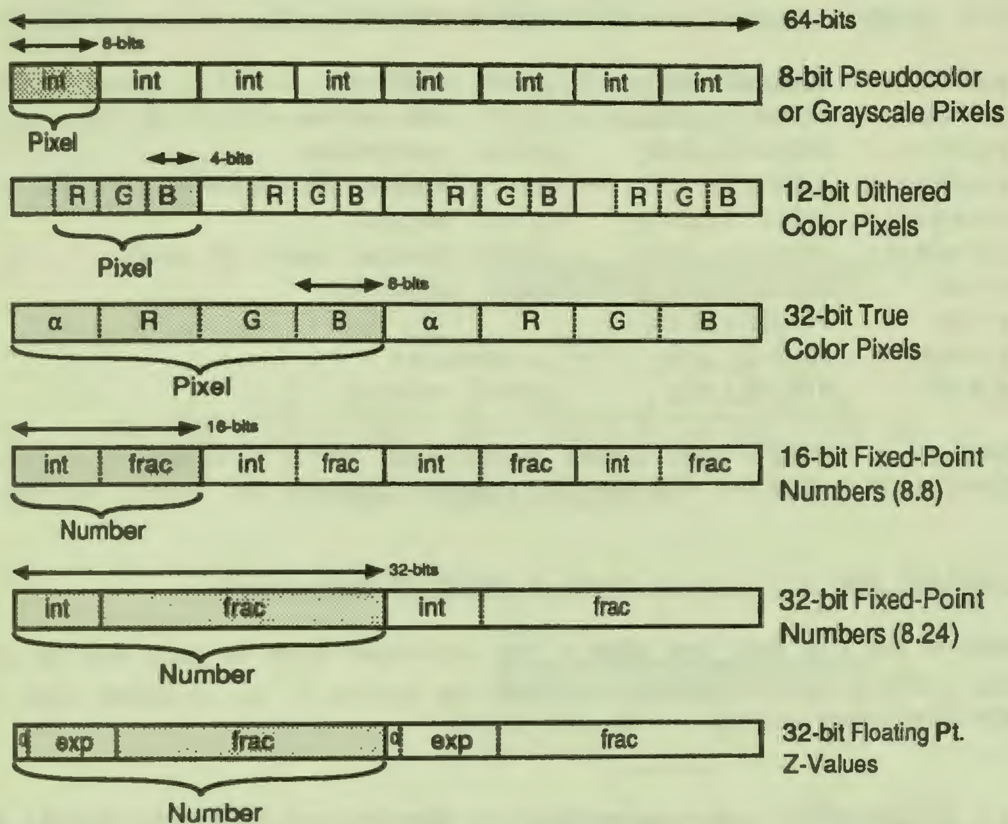


Figure 3.4.3 - Example Graphics Data Types

32-bit 8/8/8/8 αRGB true color pixels, 32-bit (8.24) fixed point intensity values, 32-bit floating point z-buffers, and 8-bit grayscale pixels are expected to be the most heavily used graphics data types.

3.4.4. GRAPHICS INSTRUCTION SET

An overriding goal in the design of the graphics assist hardware in the 88110 is architectural compatibility with the rest of the machine. The RISC philosophy has been adhered to - most (all but pixel multiplication) graphics operations are single cycle. Operands are in the general registers and data movement to and from memory relies on the base architecture's load/store instructions. Like other instructions in the machine, graphics instructions issue two at a time. They can be intermixed freely with other integer and floating point instructions with no restrictions on instruction alignment. Instruction pipelines are not exposed to the programmer - no NOP's are required to schedule pipeline delay slots. Data hazards are detected and interlocked by the same hardware scoreboard mechanism used for all other instructions. Special "modes" and dedicated special purpose registers which decrease chip efficiency, increase loop setup time, and generally increase algorithm overhead have been completely avoided.

The 88110 graphics instructions are summarized here:

1)	padd.t	drD,drS1,drS2	;pixel addition
2)	padds.x.t	drD,drS1,drS2	;pixel add and saturate
3)	psub.t	drD,drS1,drS2	;pixel subtraction
4)	psubs.x.t	drD,drS1,drS2	;pixel subtract and saturate
5)	punpk.t	drD,drS1,drS2	;pixel unpack
6)	ppack.r.t	drD,drS1,drS2	;pixel truncate, insert, & pack
7)	prot	drD,drS1,drS2	;pixel rotate left
	prot	drD,drS1,<O5>	
8)	pcmp	rD,drS1,drS2	;z-compare
9)	pmul	drD,rS1,rS2	;pixel multiply

where rD is a single-word general register and drD is a double-word general register (drD = rD:rD+1 (rD must be even)). Similarly for source registers.

padd.t drD,drS$_1$,drS$_2$ .t = .b (byte), .h (half), {blank} (word)

Fields of size t in drS$_1$ are added to the equivalent fields in drS$_2$ and the resulting fields placed in drD. Addition is carried out modulo 2^t, i.e. overflow (and underflow) wrap around within the fields.

padds.x.t drD,drS$_1$,drS$_2$ .x = .u, .us, or .s .t = .b (byte), .h (half), {blank} (word)

Fields of size t in drS$_1$ are added to the equivalent fields in drS$_2$ and the resulting fields placed in drD. Addition is carried out using unsigned, signed, or mixed saturation arithmetic as specified by .x (.u, .s, .us). (Saturation arithmetic is described in a subsection below.)

psub.t drD,drS$_1$,drS$_2$ .t = .b (byte), .h (half), {blank} (word)

Fields of size t in drS$_2$ are subtracted from the equivalent fields in drS$_1$ and the resulting fields placed in drD. Subtraction is carried out modulo 2^t, i.e., overflow (and underflow) wrap around within the fields.

psubs.x.t drD,drS$_1$,drS$_2$ .x = .u, .us, or .s .t = .b (byte), .h (half), {blank} (word)

Fields of size t in drS$_2$ are subtracted from the equivalent fields in drS$_1$ and the resulting fields placed in drD. Subtraction is carried out using unsigned, signed, or mixed saturation arithmetic as specified by .x (.u, .s, .us). (Saturation arithmetic is described in a subsection below.)

punpk.t drD,rS₁ .t = .n (nibble), .b (byte), or .h (half)

Fields of size t in rS_1 are placed into fields of size 2t with zero fill and concatenated to form a 64-bit result which is placed in drD.

The following diagrams illustrate the operation of **punpk**:

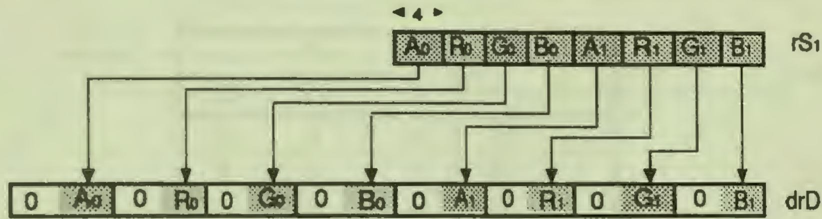


Figure 3.4.4.a - punpk.n

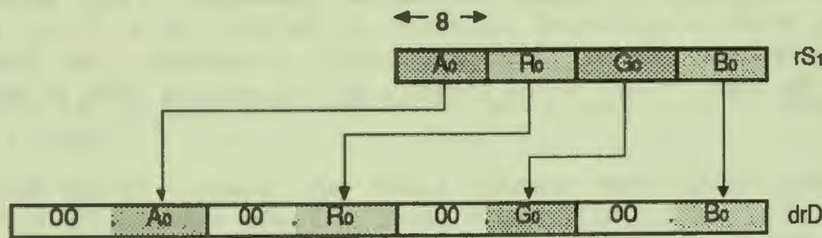


Figure 3.4.4.b - punpk.b

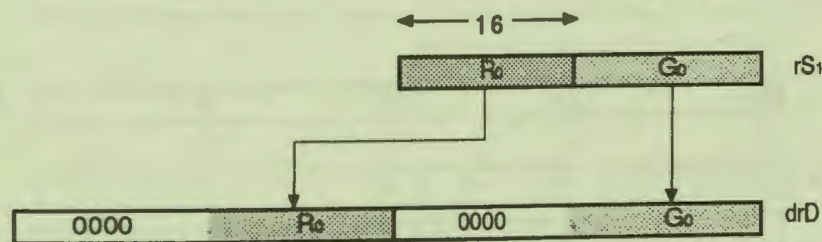


Figure 3.4.4.c - punpk.h

ppack.r.t drD.drS$_1$.drS$_2$ .r = .8, .16, .32 .t = .b (byte), .h (half), (blank) (word)

Fields of size t from drS$_2$ are truncated to t*r/64 bits and packed together. The resulting r bits replace the r most significant bits of drS$_1$ (but drS$_1$ is not actually modified). The resulting value is rotated to the left by r bits and placed in drD. The following table shows the truncated field width for the possible combinations of field size t and rotation size r.

PPACK		r		
		8	16	32
	8	x	x	4
t	16	x	4	8
	32	4	8	16

x = undefined operation

Table 3.4.4.a - ppack Field Size and Rotation

The following pictures illustrate the salient operations. They show the final operation, which is performed repeatedly in building up a 64-bit word containing multiple pixels. Note that initial, intermediate, and final steps are precisely the same. drS2 would normally be the same as drD to accumulate the result.

For shading, intermediate intensity values are normally 32-bit fixed point numbers. The position of the radix point is simply a matter of how one chooses to think about the numbers; the distinction is made here simply to distinguish the number of msb's which would be picked off and packed into the result.

Using ppack.8 to build 16-bit 4/4/4 RGB pixels from 4.28 bit fixed-point intensity values:

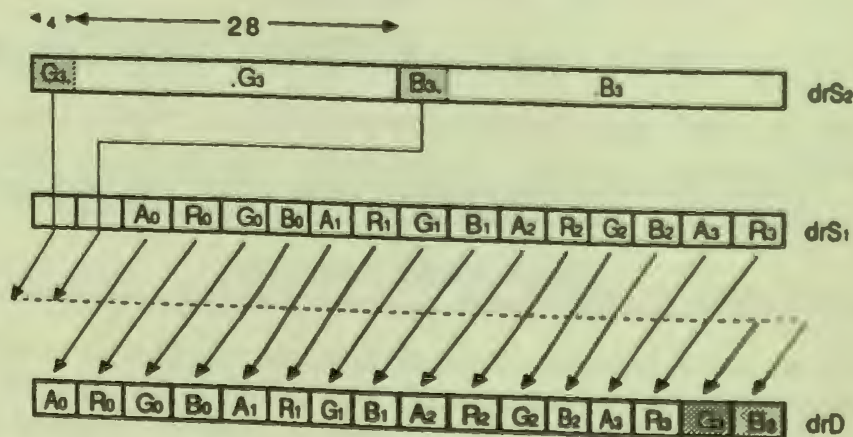


Figure 3.4.4.c - ppack.8

Using **ppack.16** to build 32-bit 8/8/8/8 ARGB pixels from 8.24 bit fixed-point intensity values:

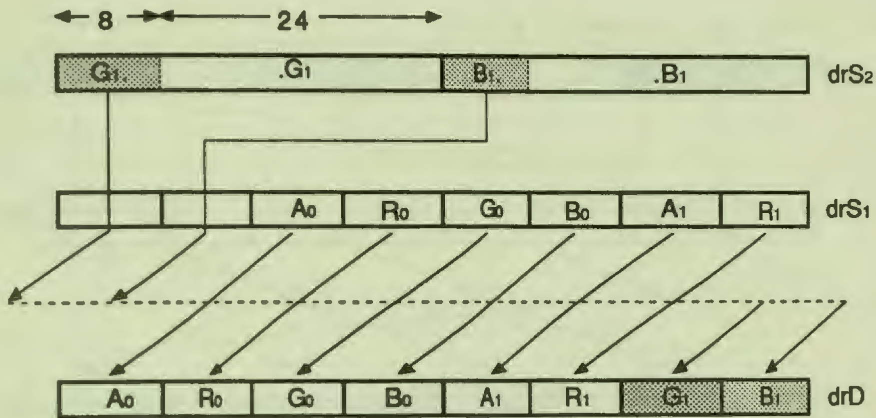


Figure 3.4.4.d - ppack.16

Using **ppack.16** to build 8-bit pseudocolor pixels from 8.24 bit fixed-point intensity values:

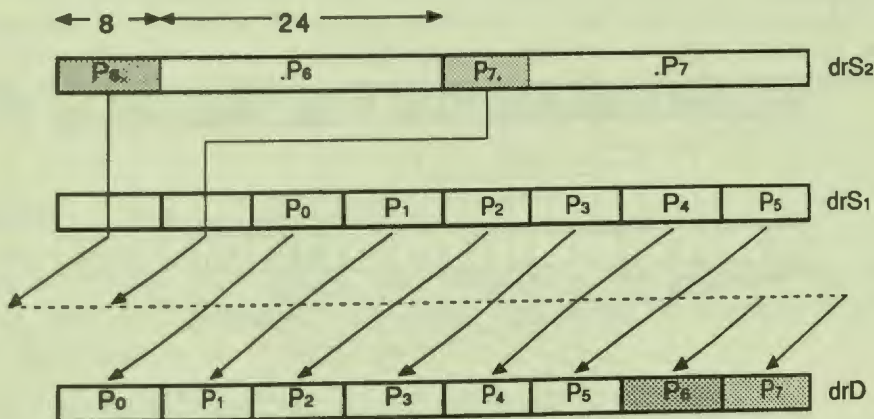


Figure 3.4.4.e - ppack.16

Using ppack.32 to build 64-bit 16/16/16/16 ARGB pixels from 16.16 bit fixed-point intensity values:

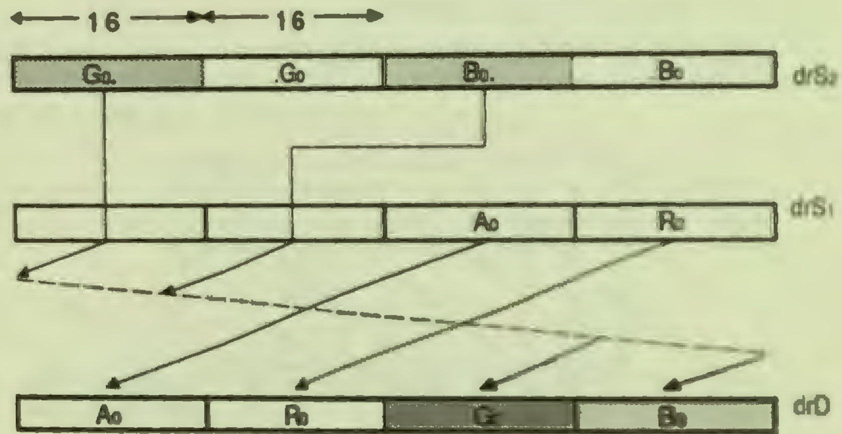


Figure 3.4.4.f - ppack.32

Using ppack.32.b to build 16-bit 4/4/4 RGB pixels from 4.4 bit fixed-point intensity values. This can also be used to convert 32-bit 8/8/8/8 ARGB pixels to 4/4/4/4 pixels.

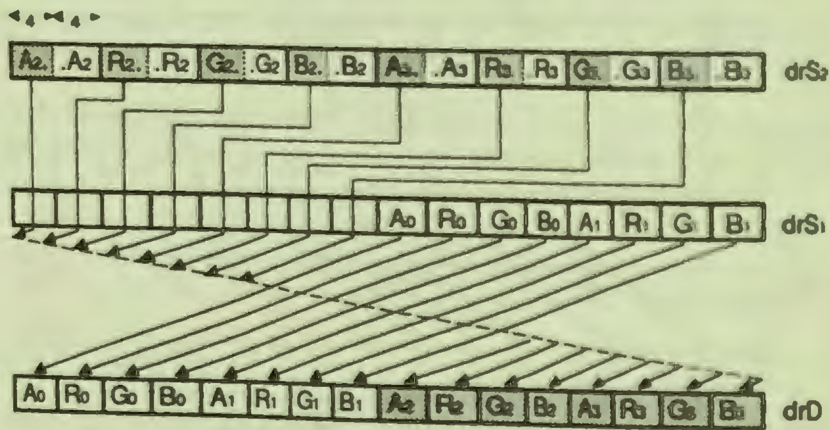


Figure 3.4.4.g - ppack.32.b

Using **ppack.32.h** to build 32-bit 8/8/8/8 ARGB pixels from 8.8 bit fixed-point intensity values:

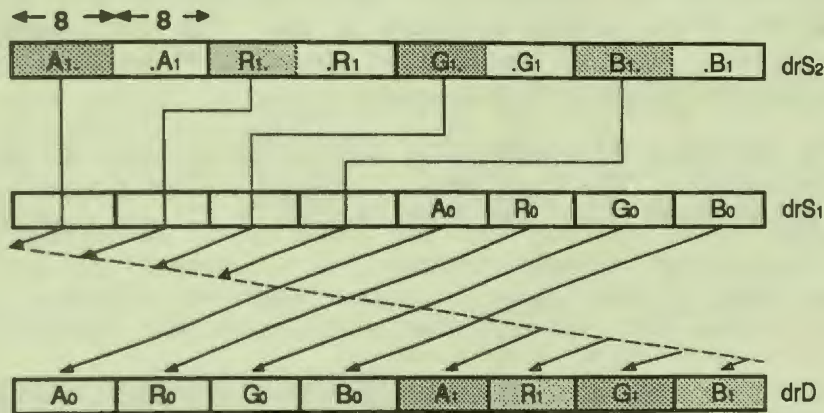


Figure 3.4.4.h - ppack.32.h

prot drD,drS_1,rS_2

prot drD,drS_1,<O6> <O6> or $rS_2 = 0,4,8,12, ..., 56,60$

The value in drS_1 is rotated to the left by the number of bits specified in register rS_2 or by the immediate value <O6> and the result is placed in drD. In this implementation, the rotation count is restricted to be an even multiple of 4 bits between 0 and 60 bits. Any odd multiple of 4-bits is simply truncated to the next lower even multiple of 4-bits. Any count greater than 60 bits is truncated to be less than or equal to 60 bits.

pcmp rD,drS_1,drS_2

Two fields of 32-bits each in drS_1 are compared to the corresponding fields in drS_2 using unsigned arithmetic. An 8-bit result string is returned in rD. The format of the result in rD is as follows:

rD[3:0]:	0
rD[4]:	$(drS_1[63:32] \geq drS_2[63:32])$ && $(drS_1[31:0] \geq drS_2[31:0])$
rD[5]:	$(drS_1[63:32] < drS_2[63:32])$ && $(drS_1[31:0] < drS_2[31:0])$
rD[6]:	$(drS_1[63:32] \geq drS_2[63:32])$ && $(drS_1[31:0] < drS_2[31:0])$
rD[7]:	$(drS_1[63:32] < drS_2[63:32])$ && $(drS_1[31:0] \geq drS_2[31:0])$
rD[8]:	!rD[4]
rD[9]:	!rD[5]
rD[10]:	!rD[6]
rD[11]:	!rD[7]
rD[31:12]:	0

Table 3.4.4.b - pcmp Results

pmul drD, rS_1, drS_2

The 32-bit value in rS_1 is multiplied by the 64-bit value in drS_2 and the 64 least significant bits of the product are placed in drD. The multiplication is unsigned. Any bits lost as a result of truncating the product to 64 bits are simply discarded with no indication of loss of significance.

The **pmul** instruction is intended to be used in concert with the **punpk** and the **ppack** instructions. The figure below illustrates illustrates how **pmul** might be used to multiply each of the 8-bit color channels in a 32-bit pixel by an 8-bit constant.

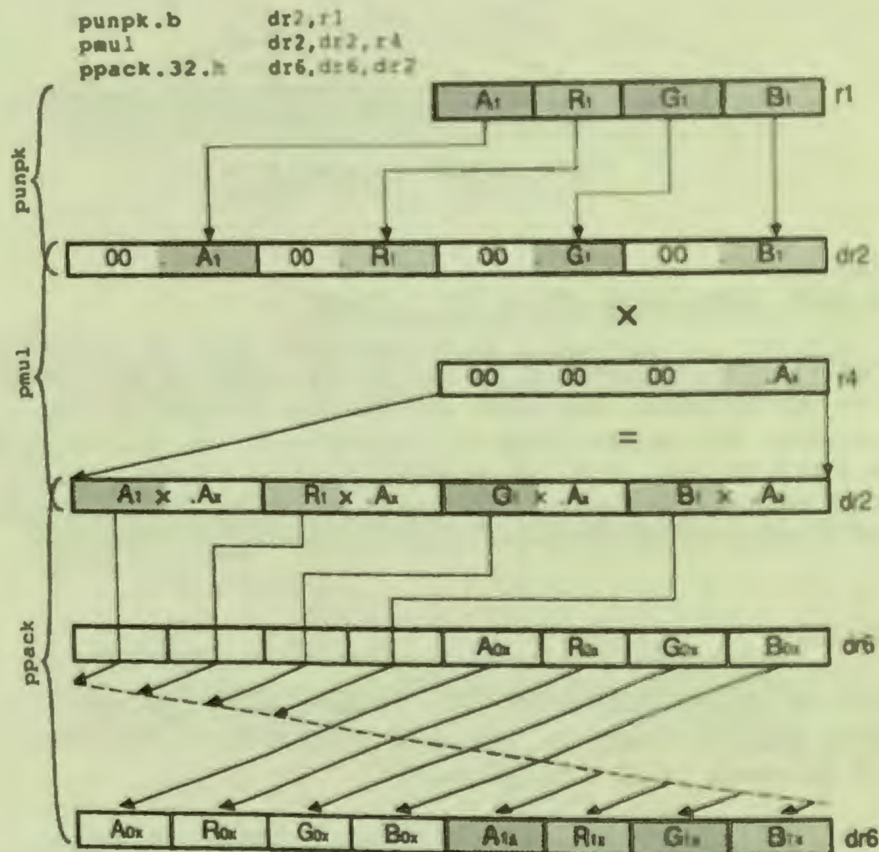


Figure 3.4.4.1 - pmul

The example starts out with a pixel in register 1. First the pixel is unpacked into double-register dr_2 which puts each color channel into a 16-bit field padded with zeros to the left. This is then multiplied by the 8-bit constant in r4 using **pmul**. The entire content of register dr2 is multiplied by the entire content of r4 just as if they were each a single number. However, since only 8-bit quantities are actually present in the multiplicand and multiplier, the zero padding prevents overflow between fields in the product. The result is as if 4 individual 8x8=16 bit multiplications had been performed. The **ppack** instruction can then be used to pick off the 8 most significant bit of each product and pack them back into pixel format. It is perhaps easiest to visualize this operation as four 8-bit integers each being multiplied by a single 8-bit fraction, yielding an intermediate result of four

16-bit (8.8) fixed point numbers, which are each truncated to their integer components.

3.4.4.1 Saturation

When adding and subtracting fields there is the possibility that the result will overflow (or underflow) the destination field size. For example, adding a 75% intensity value to a 50% intensity value would lose the most significant bits and alias to 25% intensity. This could produce an unacceptable visual anomaly in the resultant image. More appropriate for arithmetic on color intensity values would be for the addition to clamp, or "saturate," at the maximum intensity representable by the field. For the above example, saturation arithmetic would give a 100% intensity result which would be much more visually acceptable than the 25% result.

The mathematics of many graphics algorithms naturally preclude the possibility of overflow and therefore do not require saturation arithmetic. Other algorithms *depend* on the wrap-around nature of modulo arithmetic and they require non-saturating arithmetic. Still other algorithms perform intermediate calculations which may overflow but the final operation does not - thus some of the calculations need to be performed with modulo arithmetic and some with saturation arithmetic. To handle all these varied situations, the 88110 provides four forms of addition and subtraction. The most frequently used form, is simple addition without saturation. Then three forms of saturating addition are provided to handle the various data representations: 1) unsigned ± unsigned = unsigned, 2) signed ± signed = signed, and 3) unsigned ± signed = unsigned. The three forms of saturating addition are needed because overflow detection and maximum field value is different in all three cases. The binary arithmetic performed in the saturating forms is identical to the arithmetic performed in the non-saturating form. The only difference is that detection of overflow (or underflow) in the saturating forms causes the maximum (or minimum) field value to be substituted for the result in the saturating field(s).

3.4.4.1.1 Saturation Arithmetic

Unsigned ± unsigned = unsigned: saturation is indicated if there is a carry (or borrow in the case of subtraction) out of the most significant bit of the sum. The maximum field value is 2^t-1 and is substituted if an addition carries out. The minimum field value is 0 and is substituted if a subtraction borrows.

Unsigned ± signed = unsigned: saturation is indicated if the msb's of the two source fields are different, and the msb of the signed field is the same as the msb of the sum; for a+b=s, saturation=$((a_{t-1} \wedge b_{t-1})$ & $!(b_{t-1} \wedge s_{t-1}))$. The maximum field value is 2^t-1 and is substituted if addition of a positive number or subtraction of a negative number saturates. The minimum field value is 0 and is substituted if addition of a negative number or subtraction of a positive number saturates.

Signed ± signed = signed: saturation is indicated if the carry into the msb of the sum is different than the carry out of the msb of the sum. The max field value is $2^{(t-1)}-1$ and is substituted if the sum does not carry out. The minimum field value is $-2^{(t-1)}$ and is substituted if the sum does carry out.

The following table gives some examples for 8-bit fields (t=8):

s1	s2	padd.b	padds.u.b	padds.s.b	padd.us.b
00	00	00	00	00	00
00	55	55	55	55	55
00	7F	7F	7F	7F	7F
00	80	80	80	80	00
00	AA	AA	AA	AA	00
00	FF	FF	FF	FF	00
55	00	55	55	55	55
55	55	AA	AA	7F	AA
55	7F	D4	D4	7F	D4
55	80	D5	D5	D5	00
55	AA	FF	FF	FF	00
55	FF	54	FF	54	54
7F	00	7F	7F	7F	7F
7F	55	D4	D4	7F	D4
7F	7F	FE	FE	7F	FE
7F	80	FF	FF	FF	00
7F	AA	29	FF	29	29
7F	FF	7E	FF	7E	7E
80	00	80	80	80	80
80	55	D5	D5	D5	D5
80	7F	FF	FF	FF	FF
80	80	00	FF	80	00
80	AA	2A	FF	80	2A
80	FF	7F	FF	80	7F
AA	00	AA	AA	AA	AA
AA	55	FF	FF	FF	FF
AA	7F	29	FF	29	FF
AA	80	2A	FF	80	2A
AA	AA	54	FF	80	54
AA	FF	A9	FF	A9	A9
FF	00	FF	FF	FF	FF
FF	55	54	FF	54	FF
FF	7F	7E	FF	7E	FF
FF	80	7F	FF	80	7F
FF	AA	A9	FF	A9	A9
FF	FF	FE	FF	FE	FE

Table 3.4.4.1.1 - 8-bit Saturation Examples

3.4.4.1.2 Arbitrary Saturation Limits

Notice that saturation occurs only at the maximum and minimum field values It is sometimes desirable to be able to set arbitrary saturation limits. That is, it may be necessary to keep a result within a certain range which is smaller than the normal field range. The saturating forms of addition and subtraction provided by the 88110 can be used to synthesize this operation. If one has a result that needs to be clamped

to stay below a certain saturation level the following operation will produce the desired result. First, using **padds.u**, add the difference between the saturation limit and the maximum field value, then, using **psubs.u**, subtract that difference back out. If the result being clamped was already below the upper saturation limit then this operation is a NOP and the result is unchanged. If, however, the value being clamped was above the saturation limit then the first add would have saturated at the maximum field value and the subtract would have set the result to the saturation limit value. The analogous operation can be performed at the other end to clamp the value above a certain lower saturation level. The operation is illustrated in the following figure.

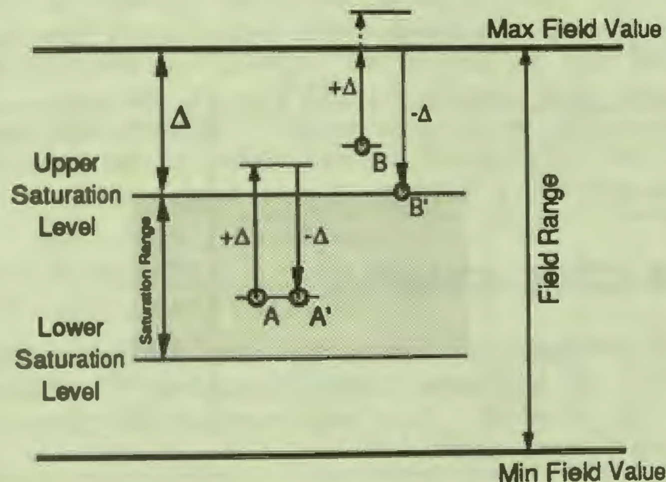


Figure 3.4.4.1.2 - Arbitrary Saturation Limits

In this example A was within the saturation limits and therefore retains it original value (A=A') but B becomes clamped to the upper saturation limit (B≠B'=UpperSaturationLevel).

3.4.5. GRAPHICS EXECUTION UNITS

There are two independent graphics units in the 88110; an adder and a pixel packing/unpacking unit.

3.4.5.1 Graphics Add Unit

The graphics adder is a dual 32-bit adder with controllable carry chains on each 8-bit boundary. Arithmetic is carried out using either modulo or saturation arithmetic. Saturation arithmetic is performed by substituting the appropriate maximum or minimum value for any field which overflows or underflows. Overflow and underflow detection and maximum and minimum field values are dependent on whether the operands are signed or unsigned.

Instructions executed by the graphics adder include **padd**, **padds**, **psub**, **psubs**, and **pcmp**. All forms of addition and subtraction (**padd**, **psub**, **padds**, **psubs**, **pcmp**) execute in a single clock.

3.4.5.2 Graphics Packing/Unpacking Unit

The pixel packing/unpacking unit is a specialized bit-field unit for packing, unpacking, and shifting pixel or fixed-point data. It operates on multiple bit-fields within 64-bit operands in parallel. The packing instruction accumulates pixel data as it is computed by truncating multiple fixed-point values, packing the resulting bit-fields tightly together, inserting them into a second source operand, and rotating the result into proper position for the next iteration - all in a single clock. The unit can also perform the inverse operation of unpacking bit-fields from a 32-bit operand and properly placing them into a 64-bit word for subsequent arithmetic calculations. Rotation to the left by any multiple of 4 bits from 0 to 60 bits is performed both as an individual instruction and as part of the packing operation. Instructions executed by the graphics packing unit include `ppack`, `punpk` and `prot`. Each of the graphics units execute instructions in a single clock. However, the graphics units are not identical and each can only accept one instruction, of the type it executes, per clock. Thus each graphics unit has a blockage of one clock.

3.4.6. GRAPHICS CODE EXAMPLE

The following code illustrates the use of the padd and ppack instructions. This code snip is the inner loop of a Gouraud shading algorithm. The loop computes two 32-bit αRGB pixels on each iteration. Each loop executes 12 instructions in 6 clocks thus generating a new pixel every three clocks, or 16.7 million pixels per second at 50MHz.

```
loop
   padd       AR,AR,ΔAR          ;add Δα to α and ΔRed to Red
   st.d       P0P1,r0,PPTR       ;store pixel pair computed on last loop
   ppack.16   P0P1,P0P1,AR       ;accumulate α and Red into next pair
   add        PPTR,PPTR,8        ;bump pixel array pointer two pixels
   padd       GB,GB,ΔGB          ;add ΔGreen to Green and ΔBlue to Blue
   sub        N,N,2              ;decrement loop counter two pixels
   ppack.16   P0P1,P0P1,GB       ;accumulate Green and Blue into next pair
   padd       AR,AR,ΔAR          ;add Δα to α and ΔRed to Red
   ppack.16   P0P1,P0P1,AR       ;accumulate α and Red into next pair
   padd       GB,GB,ΔGB          ;add ΔGreen to Green and ΔBlue to Blue
   ppack.16   P0P1,P0P1,GB       ;accumulate Green and Blue into next pair
   bcnd       ne0,N,loop         ;done?
end
```

The following pipeline diagram shows the execution of the this loop.

```
      0 1 2 3 4 5 6
loop  |-|       padd AR AR DAR
001C  |-|-|     st.d P0P1 R0 PPTR
0020    |-|       ppack.16 P0P1 P0P1 AR
0024    |-|       add  PPTR PPTR 8
0028      |-|       padd GB GB DGB
002C      |-|       sub  N N 2
0030        |-|       ppack.16 P0P1 P0P1 GB
0034        |-|       padd AR AR DAR
0038          |-|       ppack.16 P0P1 P0P1 AR
003C          |-|       padd GB GB DGB
0040            |-|       ppack.16 P0P1 P0P1 GB
0044            |-|       bcnd ne0  N loop
```

3.5. INSTRUCTION TIMING SUMMARY

The following is a summary of instruction execution timing parameters. The clock counts presented here are the latency[1] and blockage[2] induced by an instruction assuming it issues as the first instruction in an issue pair. These clock counts do *not* represent the average execution time of an instruction since the machine attempts to execute two instructions on every clock. For more details see the Appendix on Detailed Instruction Timing.

SFU0	Function Unit	Latency	Blockage
and,or,xor,mask	INTU (ALU)	1	0
clr,ext,extu, ff0,ff1,mak,rot,set	INTU (BFU)	1	1
add,addu, sub,subu,cmp	INTU (ALU)	1	0
add.car,addu.car sub.car,subu.car	INTU (ALU)	1	1
muls,mulu,mulu.d	MUL	3	1
divs,divu	DIV	19	19
divu.d	DIV	30	30

Table 3.5.a - SFU0 Instruction Timing Summary

SFU1	Size			Function Unit	Latency	Blockage
	32 .s	64 .d	80 .x			
fadd,fsub	●	●	●	FPADD	3	1
fcmp	●	●	●	FPADD	3	1
fmul	●	●	●	MUL	3	1
mov g ↔ x	●	●		N/A	1	1
mov x ← x			●	N/A	1	1
mov g ↔ x			●	trap	N/A	N/A
fdiv	●			DIV	13-14	13-14
fdiv		●		DIV	23-24	23-24
fdiv			●	DIV	26-27	26-27
fsqrt	●	●	●	trap	N/A	N/A

Table 3.5.b - SFU1 Instruction Timing Summary

1 Latency is the number of clocks which a data dependent instruction in the second slot of an issue pair, would be delayed with respect to the earliest it could possibly have issued had it not had the dependency on the first instruction.

2 Blockage is the number of clocks the second instruction in an issue pair would be delayed if it used the same function unit as the first instruction.

SFU2	Function Unit	Latency	Blockage
padd,psub	PADD	1	1
padds,psubs	PADD	1	1
pcmp	PADD	1	1
ppack	PPACK	1	1
punpk	PPACK	1	1
prot	PPACK	1	1
pmul	MUL	3	1

Table 3.5.c - SFU2 Instruction Timing Summary

Flow Control	Position in Issue Pair		Not Taken (bubbles)	Taken (bubbles[1])	
	1st	2nd		TIC Hit	TIC Miss
jmp, jsr	●		N/A	N/A	3
		●	N/A	N/A	2
jmp.n, jsr.n	●		N/A	N/A	2
		●	N/A	N/A	1
br, bsr	●		1	1	3
		●	0	0	2
br.n, bsr.n	●		0	0	2
		●	0	1	1
bb0, bb1, bcnd	●		1	1	3
		●	0	0	2
bb0.n, bb1.n, bcnd.n	●		0	0	2
		●	0	1	1
Traps			(clocks)	(clocks)	(clocks)
tb0, tb1, tcnd	●	●	Serialize+1	N/A	Serialize+3
tbnd	●	●	1	N/A	Serialize+3
rte	●	●	N/A	N/A	Serialize+3

Table 3.5.d - SFU0 Flow Control Instruction Timing Summary

[1] Bubbles are the number of opportunities to issue an instruction which were missed as a result of the branch

Memory	Transfer Size				Latency (clocks)		Blockage
	8/16 .b,.h	32	64 .d	80 .x	Cache Hit	Miss[1]	(clocks)
ld	●	●	●		2	4	1
				●	3	5	1
st	●	●	●	●	1	1	1
xmem	●	●			N/A	Serialize+7	Serialize+7

Table 3.5.e - SFU0 Memory Instruction Timing Summary

Control	Function Unit	Latency[2]	Blockage[2]
ldcr, stcr, xcr	SFU0	Serialize+2	Serialize+2
fldcr, fstcr, fxcr	SFU1	Serialize+2	Serialize+2

Table 3.5.f - Control Register Instruction Timing Summary

1 Best case miss, i.e. parked on the bus, no copyback for cache line replacement, and zero wait state memory.

2 Side effects of writing to a control register, such as causing a cache flush, may add to these times.

2

3.5.1. EXECUTION EXAMPLE

The following example demonstrates the 88110 instruction sequencing capability on a highly computational, floating point intensive process. The example is a 3D-transformation involving matrix multiplication. In this case, matrix **R**, a vertex, is multiplied by matrix **K**, the transform, and the resulting matrix is **R'** (R' = **R** * **K**).

$$[X',Y',Z',H'] = [X,Y,Z,H] * \begin{vmatrix} A_0 & B_0 & C_0 & D_0 \\ A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \end{vmatrix}$$

- Loop Unrolled Once
- 32 Floating Point Multiplications, 24 Floating Point Additions, 8 loads, 8 Stores
- 80 Instructions in 41 Clocks ⇒ 1.95 Instructions/Clock.
- @50 MHz ⇒ 2.4 Mpts/sec, 68 Double-precision MFLOPS, and 97.5 MIPS.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
|-|-|-|   fmul.ddd T2 Z0 A02                    |-|-|-|  fadd.ddd [illegible]
|-|-|     ld.d Y1 IPTR YNDX                     |-|-|-|  fmul.ddd [illegible]
  |-|-|-|   fmul.ddd T3 H0 A03                    |-|-|-|  fadd.ddd [illegible]
  |-|-|     ld.d Z1 IPTR ZNDX                     |-|-|-|  fmul.ddd [illegible]
    |-|-|-|   fadd.ddd T0 T6 T1                     |-|-|-|  fadd.ddd [illegible]
    |-|-|-|   fmul.ddd T1 X0 A10                    |-|-|-|  fmul.ddd [illegible]
      |-|-|-|   ld.d H1 IPTR HNDX                     |-|-|    st.d [illegible]
      |-|-|-|   fmul.ddd T4 Y0 A11                    |-|-|-|  fmul.ddd [illegible]
        |-|-|-|   fadd.ddd T2 T2 T3                     |-|-|-|  fadd.ddd [illegible]
        |-|-|-|   fmul.ddd T3 Z0 A12                    |-|-|-|  fmul.ddd [illegible]
          |-|-|-|   fmul.ddd T5 H0 A13                    |-|-|-|  fadd.ddd [illegible]
            |-|-|-|   fadd.ddd T1 T1 T4                     |-|-|-|  fmul.ddd [illegible]
            |-|-|-|   fmul.ddd T4 X0 A20                    |-|-|-|  fadd.ddd [illegible]
              |-|-|-|   fadd.ddd T0 T0 T2                     |-|-|-|  fmul.ddd [illegible]
              |-|-|-|   fmul.ddd T2 Y0 A21                    |-|-|    st.d [illegible]
                |-|-|-|   fadd.ddd T3 T3 T5                     |-|-|-|  fmul.ddd [illegible]
                |-|-|-|   fmul.ddd T5 Z0 A22                    |-|-|-|  fadd.ddd [illegible]
                  |-|-|     st.d T0 TPTR XNDX                     |-|-|-|  fmul.ddd [illegible]
                  |-|-|-|   fmul.ddd T6 H0 A23                    |-|-|-|  fadd.ddd [illegible]
                    |-|-|-|   fadd.ddd T4 T4 T2                     |-|-|-|  fmul.ddd [illegible]
                    |-|-|-|   fmul.ddd T2 X0 A30                    |-|-|-|   fadd.ddd [illegible]
                      |-|-|-|   fadd.ddd T1 T1 T3                     |-|-|-|-|  fmul.ddd [illegible]
                      |-|-|-|   fmul.ddd T0 Y0 A31                    |-|-|    st.d [illegible]
                        |-|-|-      fadd.ddd T5 T5 T6                     |-|-|-|   fmul.ddd [illegible]
                        |-|-|-|-|  fmul.ddd T6 Z0 A32                     |-|-|-|   fadd.ddd [illegible]
                          |-|-|      st.d T1 TPTR YNDX                     |-|       add [illegible] 16
                          |-|-|-|    fmul.ddd T3 H0 A33                      |-|-|-|   fadd.ddd [illegible]
                            |-|-|-|    fadd.ddd T0 T0 [illegible]            |-|-|-|   ld.d [illegible]
                            |-|        add  IPTR IPTR 16                       |-|-|-|   fadd.ddd [illegible]
                              |-|-|-|     fadd.ddd T4 T4 [illegible]           |-|       add [illegible] 0
                              |-|-|-|-|  ld.d X0 IPTR XNDX                        |-|-|-|   fmul.ddd [illegible]
                                |-|-|-|-|  fadd.ddd T6 T6 [illegible]            |-|-|     st.d [illegible]
                                |-|         add  TEMP TPTR 0                        |-|-|-|   fmul.ddd [illegible]
                                  |-|         add  TPTR TPTR 16                       |-|-|-|   fadd.ddd [illegible]
                                  |-|-|       st.d T4 TEMP ZNDX                       |-|-|-|-| st.d [illegible]
                                    |-|-|-|    fmul.ddd T2 X1 A00                         |-|       bcnd.n [illegible] loop
                                    |-|         sub  N N 2                                |-|       add [illegible] 16
                                      |-|-|-|   fmul.ddd T1 Y1 A01
                                      |-|-|      ld.d Y0 IPTR YNDX
                                        |-|-|-|   fmul.ddd T3 Z1 A02
                                        |-|-|      ld.d Z0 IPTR ZNDX
                                          |-|-|-|   fmul.ddd T4 [illegible] A03
                                          |-|-|      ld.d H0 IPTR HNDX
```

Figure 3.5.1 - 3-D Graphics Transform

3.6. EXCEPTIONS

88110 exceptions are generated as a result of unusual circumstances resulting from execution of an instruction or occurrence of some asynchronous external event. There are four things which cause exceptions in the 88110:

1. External interrupts.
2. Certain memory access conditions such as page faults and bus errors.
3. Internal errors such as an attempt to execute an unimplemented opcode or arithmetic overflow.
4. Trap instructions.

The handling of exceptions is transparent to user mode code and therefore the manner in which they are handled is not specified by the architecture. However, the architecture does specify a "vectored trap" mechanism which is invoked by the trap instructions **tb0, tb1, tbnd, and tcnd.** The 88110 utilizes this same mechanism to handle all types of exceptions. When a user mode instruction experiences an exceptional condition, the machine is placed into supervisor mode and control is transferred to a software exception handler routine located at some offset within a memory based vector table. Each exception generated in the machine transfers control to a different address in the vector table. When the exception has been dealt with, the handler can resume execution of the user program without its knowledge that such an event ever occurred.

The 88110 implements a precise exception model. This means that a) when an exception occurs, neither the faulting instruction or any instructions logically following it in the code stream will appear to have been executed, and b) the precise location (address) of the faulting instruction will be known to the exception handler. This precise exception model can simplify and speed up exception processing because software does not have to manually save the machine's internal pipeline states, unwind the pipelines, and cleanly terminate the faulting instruction stream; nor does it have to reverse the process to resume execution of the faulting stream.

3.6.1. ILLEGAL OPCODE EXCEPTIONS

The 88110 completely decodes all instruction opcode fields and register specifier fields. All undefined SFU0 instructions will cause an Unimplemented Opcode exception and all SFU1-7 instructions will cause the corresponding SFUx exception.

3.6.2. VECTORED EXCEPTIONS

When an exception is recognized, control is transferred to instructions located in the exception vector table. The vector table is located in memory at the address specified by the Vector Base Register - an SFU0 control register (cr7). The vector table is a 4k byte table which allocates two instructions for each exception. The lower 128 vectors are reserved for hardware use and are not accessible from user trap instructions. An attempt to specify vector 0-127 from a trap instruction will unconditionally cause Privilege Violation exception to be taken. The upper 384 vectors are allocated for software traps. Each two word vector contains the first two instructions of an exception handler. The assignment of exceptions to vector table entries is shown in Appendix A.1.

3.6.3. EXCEPTION PROCESSING

3.6.3.1 Exception Recognition

An exception is recognized when the faulting instruction and all instructions logically preceding it have completed execution. When this occurs, the following action is taken.

1. The logical address of the excepting instruction is saved in the Exception Instruction Pointer (EIP) along with a bit indicating whether the instruction was in the delay slot of a branch.
2. The PSR is saved in the Exception Processor Status Register (EPSR).
3. The excepting instruction and all instructions logically succeeding it are aborted. All user visible effects of these instructions are undone. The appearance is as if the excepting instruction had never executed, i.e., the destination register, status flags, etc. are all returned to their state prior to issue of the excepting instruction.
4. External interrupts are disabled (InD).
5. All special function units are disabled (SFUD).
6. Subsequent exceptions are directed to the error vector by setting the SFrz bit in the PSR.
7. The machine is placed into Supervisor Mode.
8. The vector address is computed by concatenating the Vector Base Register (SFU0 control register cr7) with 8 times the vector number. The table of vector assignments is shown above.
9. Execution is resumed at the instruction found at the vector address.

3.6.3.2 Exception Handling

Exception handling is a software process. Generally, an exception handler will first save the machine state, including all user visible registers (GRF, XRF, FPSR, and FPCR) and any other registers necessary to restart the user program, such as the EIP and EPSR. Once the machine state has been safely stored in memory, the exception handler may re-enable exceptions (SFrz=0) and interrupts (InD=0).

It is not always necessary for an exception handler to save the entire state of the machine. If the exception service required is simple and the system can tolerate execution of the handler with interrupts disabled, then the handler may elect to avoid the overhead of saving and restoring processor state from memory by guaranteeing that it will not generate any additional exceptions.

To simplify and speed up handling of exceptions, five control registers, accessible only in supervisor mode, are provided. These can be used to hold the supervisor stack pointer or other operating system specific data. At exception time, general purpose registers can be exchanged (using xcr) with these registers to minimize the amount of memory traffic needed by fast trap handlers. The registers may also be used as scratch storage by fast exception handlers to avoid saving general registers to memory.

3.6.3.3 Exception Recovery

When the exception handler has finished servicing the exception, it will restore the processor image which was saved in memory at the time of the exception. It will then execute an `rte` (return from exception) instruction. `rte` is the mechanism provided by both the 88100 and 88110 for termination of exception routines. `rte` first serializes the machine to guarantee that all exception handler instructions complete before returning control back to the user program. It then sets the machine into the mode (user or supervisor) it was in at the time of the exception, restores the executing instruction pointer (XIP) from the EIP, restores the PSR from the EPSR, fetches the instruction at the EIP, and then resumes execution.

3.6.4. PRECISE EXCEPTION MODEL IMPLEMENTATION

In order to achieve maximum performance, many things go on inside the 88110 in a sequence different than that specified by the executing program. Instructions execute in parallel, complete out of order, and some are even executed speculatively in anticipation of the instruction stream going in a certain direction. The hardware is careful to insure that this out of order operation never has an effect different than that specified by the program. This requirement is most difficult to assure in the event of an exception which occurs after instructions logically following the faulting instruction have already completed. At the time of an exception the machine state becomes visible to other processes and therefore must be in its correct architecturally specified condition. The 88110 takes care of this in hardware by automatically backing the machine up to the instruction which caused the exception and is therefore said to implement a "precise" exception model. The mechanism used to back the machine up in the event of an exception is the same mechanism used to recover from mispredicted branches.

To recover from an exception, a "history buffer" is used. This buffer is a FIFO queue which records relevant machine state at the time of each instruction issue. Instructions are placed on the head of the queue when they are issued and percolate to the tail of the queue while they are in execution. Instructions remain in the queue until they complete execution so that in the event of an exception the machine state necessary to recover the architectural state is available. As instructions complete execution they are released (retired) from the queue and the buffer storage reclaimed for new instructions entering the queue.

An exception is generated at any time during instruction execution and is recorded in the history buffer when the instruction finishes execution. The exception is not recognized until the faulting instruction reaches the tail of the history queue. When the exception is recognized, the queue is reversed and machine is restored to its state at the time the instruction issued. Machine state is restored at a constant rate of two instructions per clock.

To correctly restore the architectural state, the history buffer must record the value of the destination prior to instruction execution. The destination of a store, however, is in memory and it is not practical, from a performance standpoint, to always read memory before writing it. Therefore, stores issue immediately to store buffers, but do not update memory until all previous instructions have completed execution without exception, i.e. the store has reached the tail of the history buffer.

The history buffer has enough storage to hold 16 instructions worth of state. In the event of a long latency instruction, it is possible (if a data dependency does not occur first) that more than 15 instructions could issue before it finishes and fill up the history buffer. If so, instruction issue halts until the long latency operation, which is blocking retirement, finishes. There are two kinds of instructions which can potentially cause the history buffer to fill up; divide instructions and load/store instructions which miss the cache.

Two mechanisms are provided to help the programmer avoid history buffer induced stalls on memory operations: store-through and touch load. Store-through can be used to avoid unnecessarily dirtying cache lines and thus reducing the frequency of copybacks which increase load miss latency. The touch load can a) reduce average load latency by priming the cache and thus increasing the hit ratio of the real loads, b) allow subsequent loads to finish more quickly by giving them access the cache in parallel with the touch's line fill, and c) avoid history buffer induced stalls when loading data into the cache (touch never generates an exception and therefore does not block retirement of subsequent instructions from the history buffer).

3.6.4.1 Exception Timing

The following diagram illustrates the significant events in exception processing:

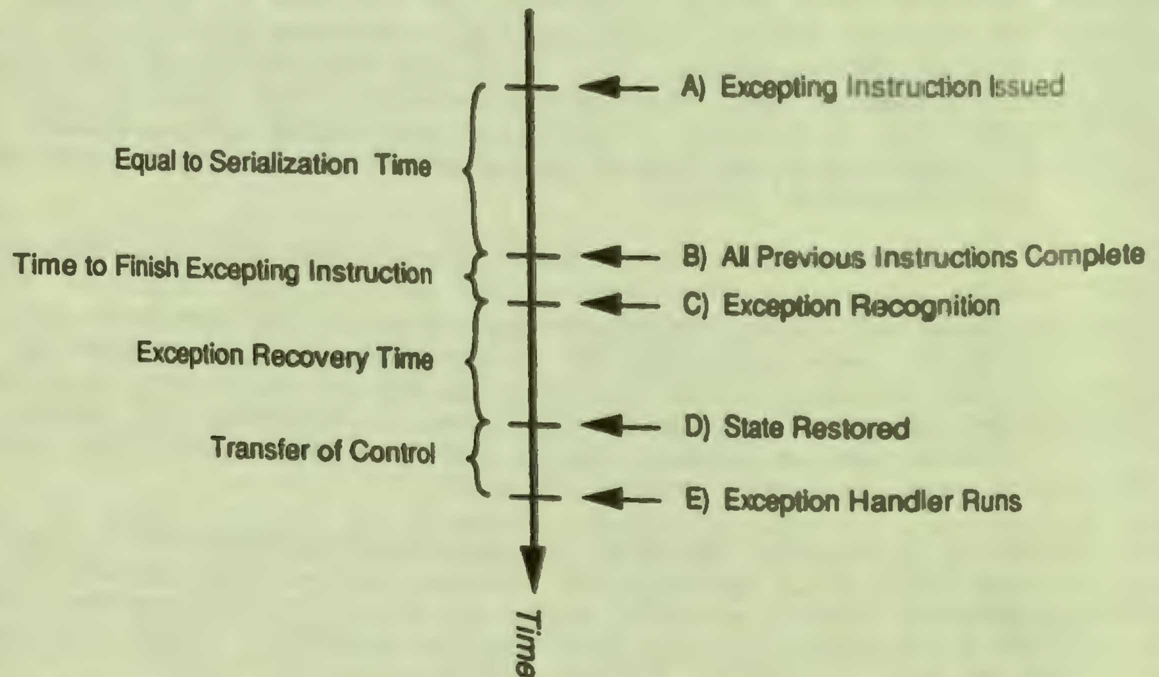


Figure 3.6.4.1 - Exception Latency

A) At time "A" an instruction which is destined to generate an exception issues.
B) At time "B" the excepting instruction has reached the tail of the history queue implying that all instructions preceding it in the code stream have finished execution without generating any exceptions.

C) By time "C" the excepting instruction has completed execution, the exception is "recognized", and exception processing begins. If at this point the instruction had not generated an exception it would have been retired.

D) By time "D" the state of the machine prior to the issue of the excepting instruction is restored, i.e., the machine is restored to its state at time A-ε.

E) At time "E" the PSR and instruction pointer of the executing process have been saved and control has been transferred to the exception handler routine.

At time "A" the excepting instruction issues and begins execution. During A-B previously issued instructions are finishing execution - the interval A-B is equivalent to the time required to serialize the machine (described below). B-C is the time required for the machine to complete execution of the excepting instruction. Frequently B-C will be zero because most instructions finish execution before they reach the tail of the queue. At time "C" the exception is recognized and during C-D the machine state is being restored. This time will depend on the number of instructions which have issued since the excepting instruction issued. At a maximum this could be 15 additional instructions and since state is restored at a rate of 2 instructions per clock this phase could require a maximum of 8 clocks. At time D all state has been restored and during interval D-E the machine is saving context information in the exception control registers (EIP and EPSR), computing the address of the exception handler, disabling interrupts, placing the machine in supervisor mode, and fetching the first instructions of the exception handler from the vector table. D-E will require 2 clocks plus the time required to fetch the target instructions.

3.6.5. SERIALIZATION

The 88110 has multiple function units each of which may be executing different instructions at the same time. This concurrency is normally transparent to the program, but in some special circumstances (e.g. debugging, I/O control, multiprocessor synchronization, etc) it may be necessary to force the machine to "serialize." Serialization in the 88110 means that instruction issue is halted until all instructions currently in progress have completed execution, i.e., all internal pipeline stages and instruction buffers have emptied, all outstanding memory transactions are complete, and the machine is completely synchronized. There are several ways to cause the 88110 to serialize:

1) An attempt to issue an **xmem, tbo, tb1, tcnd, ldcr, stcr, xcr, fldcr, fstcr,** or **fxcr** instruction will cause the machine to serialize before the instruction issues. Traps, even if not taken, cause serialization (this is the architecturally sanctioned method for forcing serialization from a user mode program).

2) Issue of a **ld** or **st** instruction when the SRM bit is set in the PSR will cause the machine to serialize. This provides a mechanism for forcing memory transactions to run in strict program order.

3) When the SER bit in the PSR is set, the machine will serialize on every instruction which is issued.

3.6.5.1 Serialization Latency

The time required to serialize the machine is the time required for all instructions currently in progress to complete. This time is heavily dependent on the instructions in progress and the memory system latency. It is impossible to put an absolute upper bound on this time because the memory system design is not under control of the '110. A worst case scenario for a "typical" system is that the history buffer is blocked by a cache miss load with copyback and 14 other instructions are pending, including 4 more load misses with copyback, 2 store misses with copyback, one extended precision floating point divide, and 7 other miscellaneous instructions. This is not a very likely situation but can be used to get an estimate of the upper limit.

3.6.6. INTERRUPTS

The 88110 provides 2 external interrupt lines - one maskable and the other non-maskable. Each interrupt has an unique vector in the exception vector table.

3.6.6.1 Maskable Interrupt (INT*)

The maskable interrupt is level sensitive (active low) and software acknowledged. It is software maskable by the InD bit in the PSR. InD is automatically set by hardware to disable INT interrupts on recognition of any exception. The maskable interrupt is of lower priority than both the non-maskable interrupt and internal exceptions.

3.6.6.2 Non-Maskable Interrupt (NMI*)

The other external interrupts is a non-maskable interrupt. This interrupt cannot be masked by software with an interrupt disable. If the NMI occurs in an exception handler with shadowing disabled the NMI vector will be taken rather than the Error Vector.

NMI is transition sensitive (active low) and software acknowledged. Once recognized by the 88110, NMI must transition to inactive and be reasserted before another NMI will be taken. The following diagram illustrates this operation.

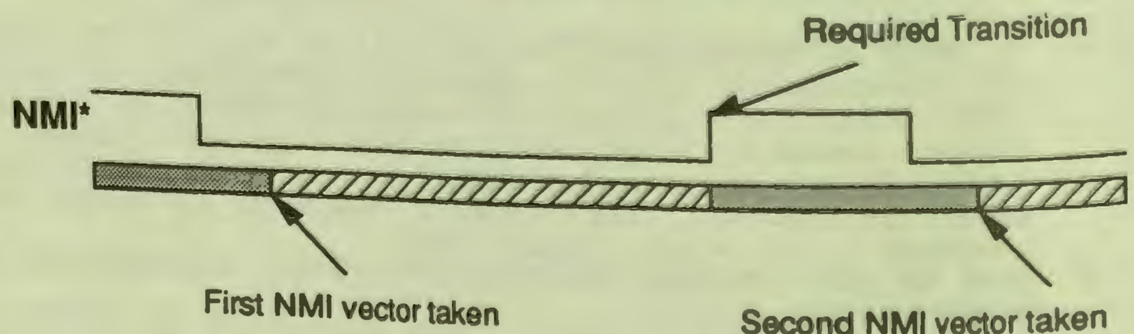


Figure 3.6.6.2 - NMI

3.6.6.3 Interrupt Latency

When an external interrupt is detected it is immediately assigned to the instruction at the tail of the history buffer, i.e., at point B in the exception latency diagram above. Thus, interrupt latency is measured as the interval B-E. The instruction at the tail of the queue is always allowed to complete execution before the interrupt is recognized.

Actual system level interrupt latency can be worse than just interval B-E. If the instruction to which the interrupt gets assigned also generates an exception, then the exception is given priority and is recognized first. If minimal interrupt latency is an important system parameter then exception handlers should save the machine context and re-enable interrupts as rapidly as possible so that the pending interrupt receives service quickly.

The allocation of the interrupt to the instruction at the tail of the history buffer minimizes interrupt response time. However, it does have one potentially confusing property. If a load instruction causes an interrupt, that interrupt might be assigned to an instruction which actually issued earlier. Thus when the machine is backed up, the instruction which caused the interrupt will appear to not yet have executed. If necessary this can be avoided by preceding an instruction which is destined to produce an interrupt by a trap-not-taken instruction to first serialize the machine.

3.7. CACHES

3.7.1. INSTRUCTION CACHE

The 88110 instruction cache is an 8Kbyte 2-way set associative cache. Cache organization is 128 sets, 2 lines per set, and 8 words per line. Cache lines are aligned on even 8-word boundaries in memory, thus a cache line will never cross a page boundary. Cache line fill begins with the missed instruction. The cache delivers the missed instruction and all subsequent instructions to the instruction unit as soon as they are received from the bus. New lines are allocated into empty cache lines if any are available. A pseudorandom replacement algorithm is used to select a line when no empty lines are available. The cache uses physical address tags; this obviates the need to flush the instruction cache on a process switch. Instruction cache coherency is maintained by software, supported by a fast hardware invalidation capability.

A block diagram of the instruction cache organization is shown in the figure below.

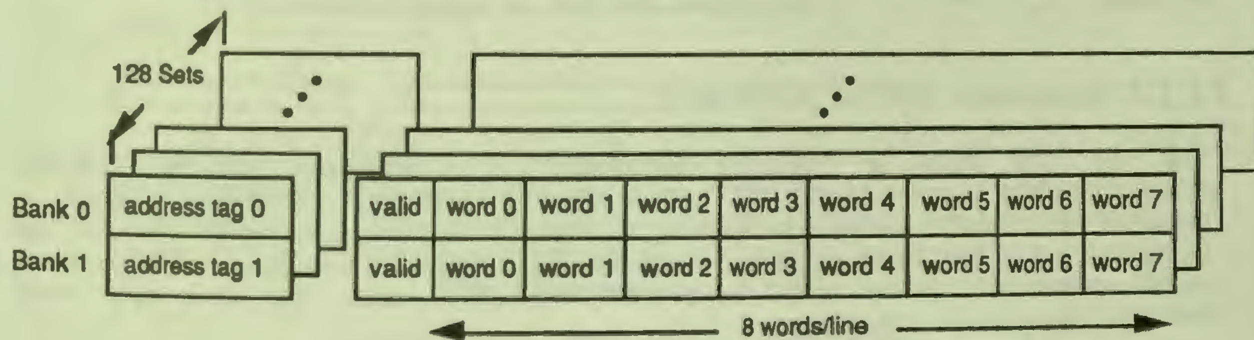


Figure 3.7.1 - Instruction Cache Organization

3.7.1.1 Instruction Cache Operation

On an instruction fetch, bits $<PA_{11}-PA_5>$ of the instruction's physical address (which are the same as logical address bits $<LA_{11}:LA_5>$) index into the cache to retrieve the tags and data for one line from each of the two banks. While the lines are being fetched, the 20 most significant bits of the logical address are translated to a physical address by the instruction address translation cache (ATC). The tags from both accessed lines are then compared against translated physical address bits $<PA_{31}-PA_{12}>$ from the ATC. If the tag from either line compares equal then it is a cache hit. If neither tag matches it is a cache miss.

Hit: If a tag match to the physical address is found and the matched entry is valid, then it is a cache hit. On a hit, three physical address bits $<PA_4:PA_2>$ are used to select one double-word from the cache line whose tag matched. Both instructions from the accessed double-word are immediately transferred to the instruction unit. When the access is within the cache line there is no address alignment restriction, i.e. a double-word is retrieved regardless of whether the address is to an odd or even instruction word. If the address accesses the last word in a cache line then only that one word is retrieved.

Miss: on a cache miss, the physical address of the missed instruction is sent to the BIU with a request to retrieve the cache line. In the event of a simultaneous data cache miss, the Bus Interface Unit (BIU) gives priority to the instruction cache. The BIU arbitrates for the bus and initiates an 8-word burst transfer read request. A cache line is then selected to receive the data which will be coming from the bus. The selection algorithm gives first priority to invalid lines. If neither of the two candidate lines in the selected set are invalid, then one of the lines is selected at random for replacement. The transfer begins with the aligned double-word containing the missed instruction (critical word first), followed by the remaining double-word(s) in the line, then (if necessary) by the double-word(s) at the beginning of the line (wrap around). As instructions are received from the bus they are written into the cache and also delivered (forwarded) directly to the instruction unit. As subsequent instructions are received from the bus interface unit they too are forwarded (streamed) to the instruction unit. As soon as the missed instruction is delivered to the instruction unit, execution is allowed to resume and proceeds in parallel with the remaining line fill. If a bus error is detected during the fetch of the instruction which caused the miss or on an instruction which is streamed to the instruction unit, then the line is marked invalid and an Instruction Access Exception is taken. If no bus error is encountered the line is marked valid.

3.7.1.2 Instruction Cache Coherency

The instruction cache is physically addressed and is therefore naturally coherent across multiple process contexts. However, no additional hardware support is provided to maintain coherency between multiple instruction caches or between the instruction cache and main memory. Thus in any situation, such as a virtual memory page replacement, which could cause the instruction cache to have stale data, software must force coherency.

A supervisor mode controlled mechanism is provided to "flash" invalidate the entire instruction cache. The invalidation is triggered by writing a "invalidate ICache" command to the Instruction MMU/Cache/TIC Command Register (ICMD). The invalidation operation requires 2 clocks plus the time required to serialize the machine imposed by the **stcr** to write the command to the ICMD.

A more selective mechanism is also provided which allows any particular line in the cache to be invalidated. This mechanism is invoked by first writing the physical address of the line to be invalidated into an address control register and then writing a "invalidate ICache line" command to the ICMD. Once again the operation requires two clocks plus serialization time.

3.7.2. DATA CACHE

The 88110 data cache is a 8Kbyte 2-way set associative cache. Cache organization is 128 sets, 2 lines per set, and 8 words per line. Cache lines are aligned on even 8-word boundaries in memory, thus a cache line will never cross a page boundary. Cache line fill begins on the missed data word. The missed word is transferred (forwarded) to the load store unit as soon as it is received from the bus. Subsequently received data is also transferred (streamed) to the load/store unit if it is needed. The cache supports both a write-through (w/write-no-allocate) and a write-back (w/write-allocate) policy, selectable on a page-by-page basis. Newly referenced data is allocated into an empty cache line when one is available and a pseudorandom selection algorithm is used to replace a line when no empty lines are available. Cache coherency is automatically maintained by hardware bus snooping. Cache tags are dual ported to prevent snooping traffic on the bus from interfering with processor operation and degrading performance. Cache tags hold physical addresses of cache entries and therefore the cache does not require flushing and invalidation on a process switch.

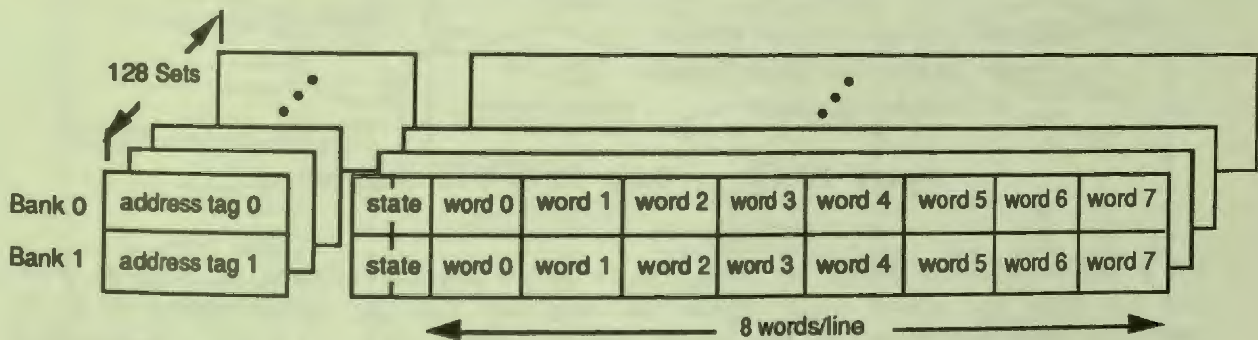


Figure 3.7.2 - Data Cache Organization

3.7.2.1. Data Cache Operation

The cache is a three state design. Two bits are included in each cache line to maintain the state information for that line. The status bits keep track of whether or not the line is valid and whether or not it is has been modified relative to memory. The state transition diagram for the write-back mode of the cache is shown below and described in the following paragraphs. All internal cache state transitions are visible on the external pins of the part to allow construction of coherent external secondary caches.

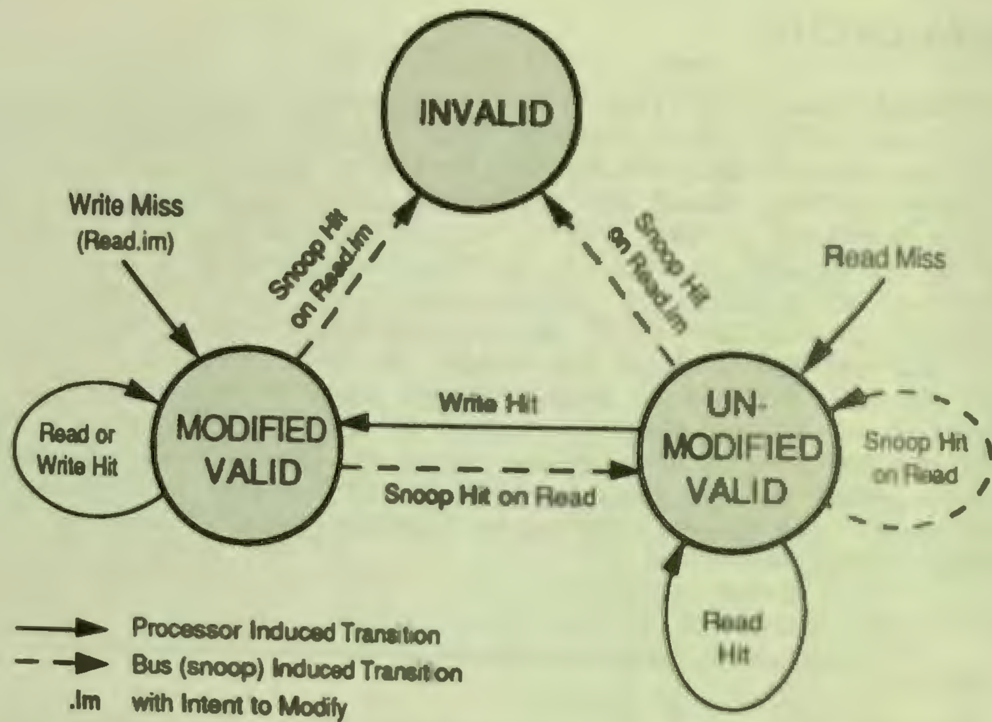


Figure 3.7.2.1.a - Data Cache State Diagram

On a load or store operation, bits <PA_{11}:PA_5> of the physical address are used to select one line from each cache bank. While the tags and data are being fetched from the cache, the most significant 20 bits of the logical address (<LA_{31}:LA_{12}>) are translated to a physical memory address by the data Address Translation Cache (ATC). The tags from both accessed lines are then compared against translated physical address bits <PA_{31}:PA_{12}>. If the tag from either selected line compares equal then it is a cache hit. If neither tag matches it is a cache miss. In the event of a simultaneous instruction cache miss, the BIU gives priority to the instruction cache and services it's miss first.

The following figure is a flowchart showing the conceptual operation of the cache. Cache operations are described in more detail in the paragraphs which follow.

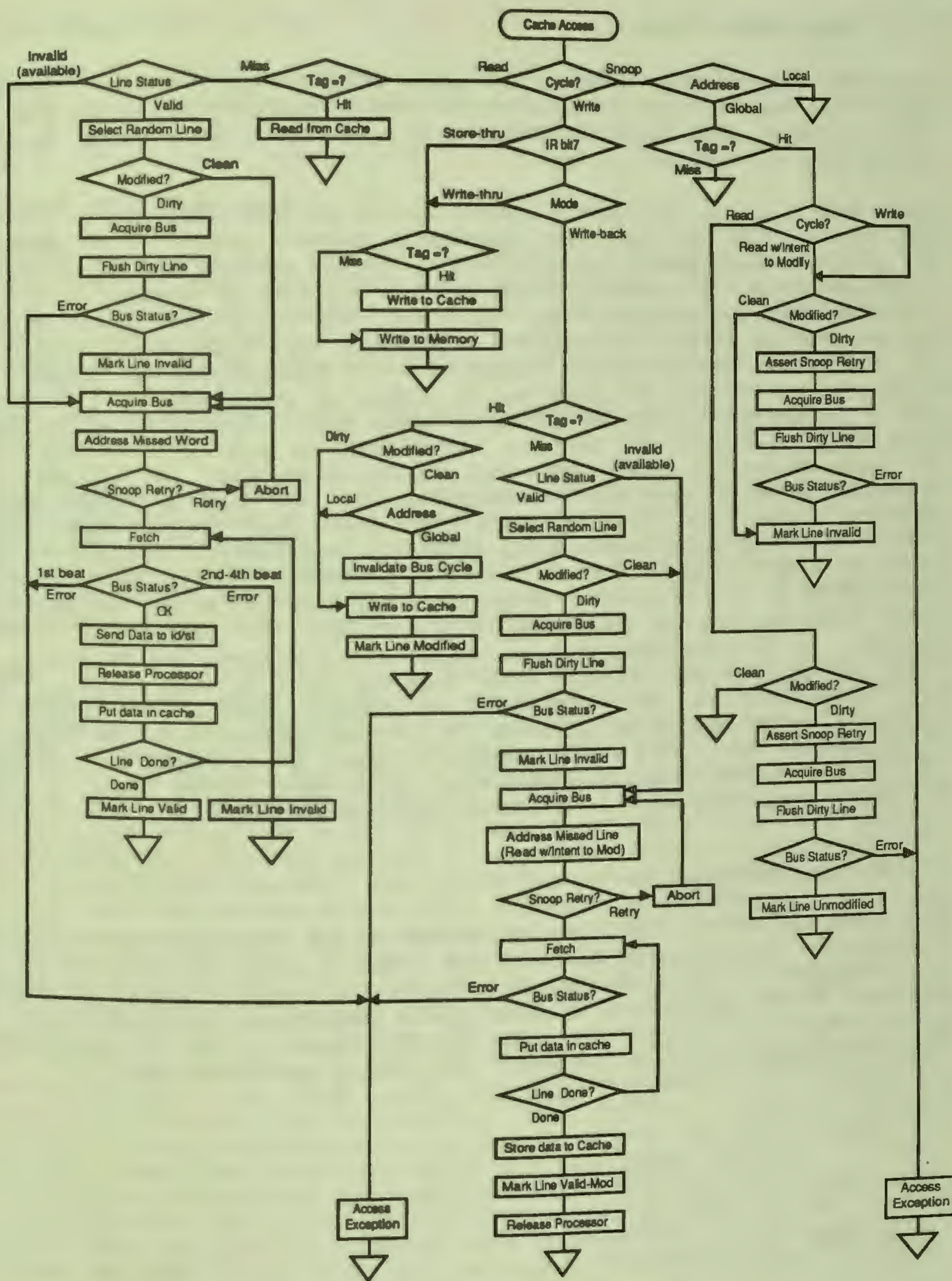


Figure 3.7.2.1.b - Data Cache Operation

3.7.2.2. Data Cache Read

Read Hit: data is simply read from the cache; no state transition occurs. Physical address bits <PA_4:PA_3> are used to select one aligned double-word out of the selected cache line and the double-word is transferred to the load/store unit. Access time of the data cache on a read hit is 1 clock.

Read Miss: A line in the cache is selected to hold the data which will be fetched from memory. The line replacement algorithm is described below. If the selected line is valid-modified (dirty), then it is sent to the Bus Interface Unit (BIU) to be written out (flushed) to memory. When the dirty line flush is complete, or if the selected line was not modified in the first place, then the physical address of the missed data is sent to the BIU along with a request to retrieve the missed cache line. The BIU arbitrates for the bus and initiates an 8-word burst transfer read request. If another processor on the bus recognizes the address as global and has a modified copy of the data in his cache, he will assert "snoop retry." Upon receipt of the retry signal the BIU will abort the transaction and relinquish the bus. The snooping CPU will acquire the bus and update memory with his copy of the line. The BIU will then re-arbitrate for the bus and retry the aborted line fill. The line fill transfer begins with the aligned double-word containing the missed data (critical word first), followed by the remaining double-word(s) in the line, and then (if necessary) by the double-word(s) at the beginning of the line (wrap around). When the missed word is received from the bus it is written to the cache and forwarded to the load/store unit which stores the data in the register file and allows the processor to resume execution. As the remaining data in the missed line is received from the bus, it is written into the cache and forwarded to the load/store unit if it needs the data. The cache remains busy and not accessible to the processor until the line transfer is complete. If a bus error is encountered on the dirty line flush or on the bus access to the missed word, then a Data Access Exception is taken. If a bus error occurs on any other word in the line transfer then the fetched line is simply marked invalid. If no bus error is encountered, the line is marked valid and unmodified.

3.7.2.2.1. Line Replacement

On a cache miss, a line must be selected to hold the new data which will be fetched from memory. The address of the missed data is used to select two cache lines - one from each bank. If one of the lines is empty (invalid) then it is selected to receive the data. If both lines are full or empty, then a pseudorandom selection algorithm is used to select one of the two lines. The algorithm employs a one bit counter which toggles the selection bias from one bank to the other on each cache miss.

3.7.2.3. Data Cache Write

The cache operates in either write-though or write-back mode determined on a page-by-page basis by a bit in the address translation cache entry. If two logical pages map to the same physical page it is considered a programming error for them to specify different cache write policies.

3.7.2.3.1. Write-back Mode

In write-back mode, write operations do not necessarily update memory. For this reason write-back mode is the preferred mode of operation when it is necessary to minimize bus bandwidth utilization, e.g. multiprocessor systems without secondary caches. Using write-back on a 7/2 bus (7 clocks to first data, 2 clocks per data transfer thereafter) will reduce bus utilization by roughly 20%.

Write Hit to Modified Line: Data is simply written to the cache with no state transition.

Write Hit to Unmodified Line: Data is written into the cache and the line is marked modified. Also, if the ATC indicates that the address is "global", then an invalidation bus transaction is performed. The invalidation transaction notifies other caches on the bus that any local copy they may currently have is no longer valid. The invalidation cycle is similar to a write cycle but data is not actually written and therefore normal write cycle bus latency is avoided.

Write Miss: A cache line is first selected to receive data from memory. The selection algorithm gives first priority to invalid lines. If neither of the two candidate lines in the selected set are invalid, then one of the lines is selected at random for replacement. If the selected line is valid-modified, then the dirty line is sent to the Bus Interface Unit (BIU) to be written out (pushed) to memory. If the selected line is not modified or when the dirty line flush is complete, the physical address of the missed data is sent to the BIU along with a request to retrieve the missed cache line. The BIU arbitrates for the bus and initiates an 8-word read-with-intent-to-modify burst transfer. This special read-with-intent-to-modify cycle is like a normal read cycle but has the side effect of broadcasting to other caches on the bus that the line being fetched will be modified and they must invalidate any local copy of the line they may have. If another processor on the bus recognizes the address as global and has a modified copy of the data in his cache then he will assert "snoop retry." Upon receipt of the retry signal the BIU will abort the line fill transaction and relinquish the bus. The snooping CPU will acquire the bus and update memory with his copy of the line. The BIU will then re-arbitrate for the bus and retry the aborted line fill. The transfer begins with the aligned double-word containing the missed data, followed by the remaining double-word(s) in the line, then (if necessary) by the double-word(s) at the beginning of the line (wrap around). As data in the missed line is received from the bus, it is written directly into the cache. If a bus error is encountered on the dirty line flush or the line fill operation, a Data Access Exception is taken. Once the line fill is complete the new store data is written into the cache and the line is marked valid and modified. At this point, if the machine has stalled waiting for the store to complete, execution is allowed to resume.

3.7.2.3.2. Write-through Mode

In write-through mode, store operations always update memory. Write-through mode is used when external memory and internal cache images must always agree. This mode of operation is normally selected for systems employing an external secondary SRAM cache.

Write Hit: Data is written to both the cache and to memory. No cache state transition occurs. The write cycle signals "invalidate" so that any other cache on the bus which has a copy of the line containing the data will know to invalidate his copy of the line. If the write cycle experiences a bus error, the cache is still updated.

Write Miss: Data is written to memory only - not to the cache (write-no-allocate), and no state transition occurs. The write cycle performed also signals "invalidate".

3.7.2.4. Store-through Accesses

Store-through may be optionally specified on any triadic register form of store. Store-through operates in precisely the same manner as write-through:

Write Hit: Data is written to both the cache and to memory. If the line was already modified it remains modified, and if it was unmodified it is left unmodified. (The memory and cache are necessarily coherent.) If the memory write transaction experiences a bus error, the cache is still updated.

Write Miss: Data is written to memory only - no line is allocated in the cache and no data is written to the cache. The write cycle performed signals "invalidate".

3.7.2.5. Cache Inhibited Accesses

If the cache access is to a page which has the cache inhibit (CI) bit set in the ATC, the following action is taken[1]:

Read Hit to Modified Line: The modified line is flushed (copied-back) to memory and invalidated. New data is read directly from memory but not placed in the cache.

Read Hit to Unmodified Line: The line is invalidated. Data is read from memory but not placed in the cache.

Read Miss: Data is read from memory but not placed in the cache. The status of the line is not affected.

Write Hit to Modified Line: The modified line is flushed (copied-back) to memory and invalidated. New data is read directly from memory but not placed in the cache.

Write Hit to Unmodified Line: Data is written through to memory but not put into the cache. The line is invalidated.

---

1 The exact operation of cache-inhibited access may vary from one implementation to another. Several mechanisms are consistent with the functional intent of cache inhibited accesses and the choice is implementation dependent

Write Miss: Data is written through to memory but no data is placed in the cache. The status of the line is not affected.

3.7.2.6. xmem Accesses

xmem is a multiprocessor synchronization instruction which exchanges a general register with a memory location in an indivisible read-write transaction. It is normally used to implement semaphores or resource locks in multiprocessor systems and therefore its action must always be reflected through to memory, i.e., its memory cycles cannot be filtered by the cache.

Miss: Both read and write cycles bypass the cache an go straight to memory. No cache transactions occur.

Hit on Modified Line: The dirty line is flushed to memory and invalidated, then the **xmem** read and write cycles bypass the cache.

Hit on unmodified Line: The line is invalidated and once again the read and write cycles bypass the cache.

3.7.2.6.1. Spin Locks

xmem is often used to implement a resource lock. The lock must be maintained in global memory so that it is visible to all processors. Processors may repeatedly or even continuously poll the lock for a chance to get access to the resource. To reduce bus traffic in this situation, it is recommended that processors poll the lock by performing regular load operations which will operate out of the cache. When the previous owner of the lock releases it in memory, snooping will ensure that the cached copy of the lock is invalidated and the new value of the lock will be recognized by the polling processor. When the lock becomes available it can then go for ownership using the atomic reference provided by **xmem**.

3.7.2.7. Snooping

The 88110 uses a retry protocol to insure that, at all times, only one cache in the system has a modified copy of a given cache line. The protocol allows other caches on the bus to have local copies which are all consistent. Whenever a processor writes data, the protocol notifies other processors that their copy of the line containing the data is now stale and must be invalidated.

Snooping is performed by watching externally initiated bus transactions and comparing all global addresses to the internal cache tags. If a global address is seen which matches one of the cache tags, it is a snoop hit. Two separate independently accessible copies of the tags are maintained to allow snooping to occur in parallel with processor cache accesses. Processor access to the cache is interrupted only in the event of a snoop hit as described below.

Snoop Hit on Read to Modified Line: The snooping processor asserts "snoop retry" to the processor which initiated the transfer. The initiating processor will

abort the transaction and release the bus. The snooping CPU arbitrates for the bus and writes his modified copy of the line to memory and marks it unmodified. The initiating CPU will then reaquire the bus and retry the aborted transaction.

Snoop Hit on Read to Unmodified Line: No action taken.

Snoop Hit on Read-with-intent-to-modify to Modified Line: The same retry sequence is performed as in the case of a snoop hit on read to modified line, but the snooping processors copy of the line is marked invalid.

Snoop Hit on Read-with-intent-to-modify to Unmodified Line: The hit entry is invalidated.

Snoop Hit on Write to Unmodified Line: The hit entry is invalidated. Even if the hit is on a partial line write (byte, half-word, word, or double-word store), the entire line in the cache is invalidated.

Snoop Hit on Write to Modified Line: Same as snoop hit on read-with-intent-to-modify.

Snoop Miss: No action taken.

3.7.2.8. Data Cache Flushing and Invalidation

The data cache is a physically addressed cache and is fully hardware snooped to maintain coherency with other caches in the system and main memory. Therefore, software does not normally have to enforce data cache coherency. But in some circumstances, such as rearranging the virtual memory map, it will be necessary to flush and possibly invalidate the data cache.

A supervisor mode controlled mechanism is provided to either invalidate or flush the data cache. Both operations are triggered by writing the appropriate command to the Data MMU/Cache Command Register (DCMD). The invalidation operation requires 2 clocks plus the time required to serialize the machine imposed by the stcr to write to the command to the DCMD. The flush operation causes all dirty lines in the cache to be transferred out to memory. The flush can take the number of clocks required to serialize the machine plus 130 clocks plus up to 256 cache line burst-write memory transactions. The cache is considered busy during the entire flush operation.

A more selective mechanism is also provided which allows any particular line in the cache to be flushed or invalidated. This mechanism is invoked by first writing the physical address of the line to be invalidated into an address control register and then writing a "flush DCache line" or "invalidate DCache line" command to the DCMD. Invalidation requires two clocks plus serialization time (no copyback occurs on invalidation). Flushing will require serialization time plus 2 clocks plus possibly a burst-write memory transaction to copyback the cache line if it is dirty.

3.7.3. TARGET INSTRUCTION CACHE

The 88110 implements a branch target instruction cache (TIC) for eliminating bubbles in the instruction pipeline on flow control changes. The cache is a fully-associative 32-entry logically addressed cache. Each entry in the cache holds the first two instructions of a branch target instruction stream and a 30-bit logical address tag. Whenever a branch instruction is encountered, the branch cache is accessed using the address of the branch itself in parallel with decode of the instruction. This allows the branch cache to be accessed before the target address of the branch has been computed - a full clock before the target stream could be fetched from the instruction cache. The cache is accessed by comparing the logical address of the branch to the tag in all 32 entries simultaneously. If a match is found, it is a TIC hit and the two target instructions are sent to the instruction unit. If the branch eventually turns out not to be taken, the instructions are discarded; the entry in the TIC is not invalidated. If no address match is found, it is a TIC miss. In this case, if the branch turns out to be taken, a new entry is allocated in the TIC. When the first two instructions of the target stream are finally fetched from the instruction cache they are placed in the newly allocated TIC entry and the entry is marked "valid." If there were no empty (invalid) entries in the TIC to accept the new entry then one of the valid entries is chosen for replacement using a FIFO replacement policy. If the branch turns out to not be taken no entry is allocated in the TIC.

Unlike the instruction and data caches, the branch target cache is logically addressed. Therefore, it must be invalidated on all context switches. A mechanism is provided to flash invalidate the TIC. It is activated by writing a command to the Instruction MMU/Cache/TIC Command Register (ICMD). The invalidation operation requires two clocks plus the time required to serialize the machine imposed by the stcr command to load the ICMD.

3.8 MEMORY MANAGEMENT UNIT

The 88110 implements separate instruction and data memory management units. Each unit provides two 4G Byte (User/Supervisor) logical address spaces and enforces access privileges on block and page levels for both of these spaces. Used and modified status is maintained for each page to assist implementation of a demand-paged virtual memory system. The 88110 memory mapping tables are upward compatible with the 88200. A block diagram of the address translation is shown below:

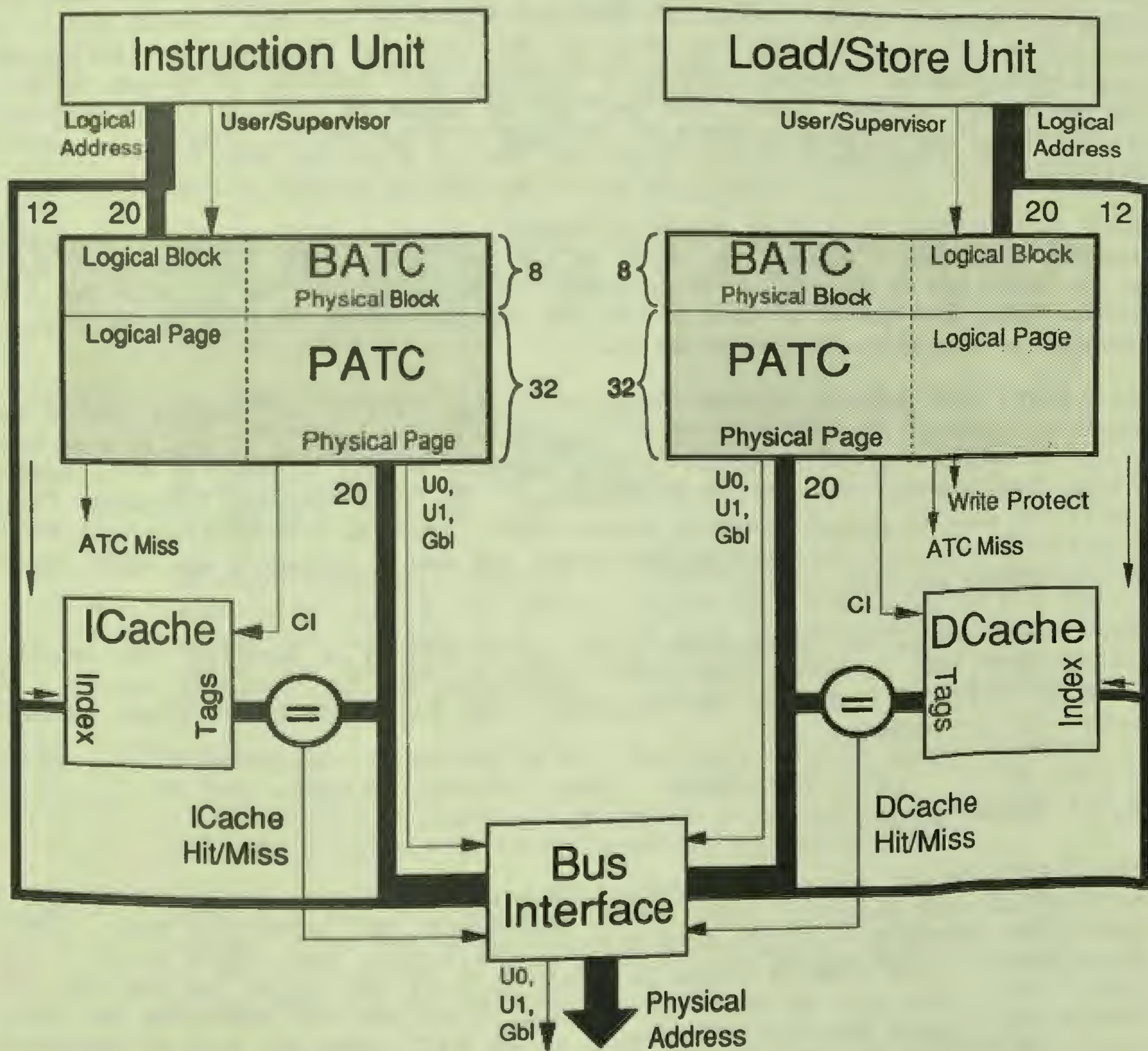


Figure 3.8.1 - Address Translation Block Diagram

3.8.1 ADDRESS TRANSLATION CACHES

A set of memory mapping tables, normally maintained by the operating system, are kept in main memory. The tables describe the mapping of logical program addresses (virtual addresses) to physical memory addresses. These tables are almost always a multilevel hierarchy to minimize their size. If every logical to physical address translation required resolving by walking memory based tables, the performance of the machine would be unacceptable. To avoid this overhead, the 88110 implements on-chip address translation caches which hold the results of recent traversals of the memory based tables. On each memory reference, the logical address is looked up in the address translation caches in parallel with the data caches being checked for the data. If the data (or instruction) itself is not found in the data caches then the address translation cache delivers the physical address to the bus interface unit. In the infrequent event of data not being in the cache and the ATC not having the requisite logical to physical mapping, a hardware state machine is invoked to walk the appropriate memory based tables and get the logical to physical address mapping. It then stores the mapping in the ATC and sends it to the BIU to complete the memory reference.

Each ATC implements a fully associative lookup. On every memory access, the logical address is compared against all entries in the appropriate ATC (instruction or data). If the upper bits of the logical address match the logical address tag in one of the ATC entries (hit), that entry is used to map the logical program address to the physical memory address which is sent to the bus.

The 88110 uses separate Address Translation Caches (ATC's) for mapping Blocks and Pages of memory. The Block Address Translation Cache (BATC) is used to map large areas of system and user memory without usurping a large number of ATC entries. Block size can vary from 512K to 64Mbytes. The Page Address Translation Cache (PATC) is used for mapping pages of 4Kbytes each. Separate sets of ATC's (both BATC and PATC) are used for mapping instructions and data to eliminate any delay caused by contention or arbitration.

The PATC entries are automatically filled on a miss by a hardware state machine which walks the memory page tables maintained by the operating system. A software PATC fill option is also supported. The BATC entries are always managed completely by software.

3.8.1.1 Block ATC

The BATC contains eight 32-bit entries. Each entry contains a logical address tag, a physical address mapping, some control information, and protection data. These entries are loaded by setting up the BATC Index Register and writing to the BATC Write Port. A size mask is used to determine the block size (512k, 1M, 2M, 4M, 8M, 16M, 32M, 64M) for all BATC entries and is set via the Instruction (or Data) MMU/Cache Control Register. The format of the BATC entry for both the Instruction BATC and the Data BATC is shown in the figure below.

Block ATC Entry

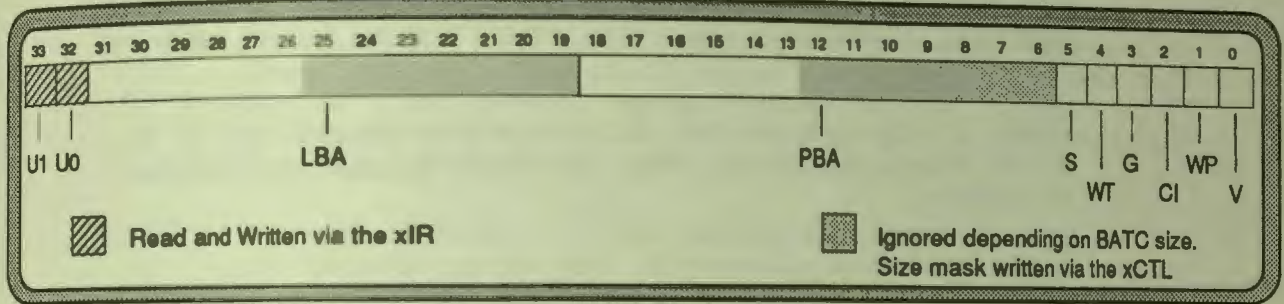


Figure 3.8.1.1 - Block ATC Entry

LBA Logical Block Address - This field contains the logical address (tag) that is matched against the logical address of a memory access. If an address match is found (including the supervisor mode bit S), and the valid (V) bit is set, then the logical address is mapped to the physical address as specified in the PBA field.

PBA Physical Block Address - If the logical address hits on this entry, the 6 - 13 (depending on block size) most significant bits of the logical address are replaced by the bits in this field to form the translated physical address.

U0,U1 User Page Attributes 0,1 - These bits are not used by the MMU but are user definable from software. They are presented on external pins during bus transactions mapped by this entry. They are loaded into the BATC entry via the xIR (Instruction or Data Index Register) at the time of a Block ATC write.

S Supervisor Mode Bit - This bit is compared to the supervisor mode which is in effect for the memory access being translated. If this bit matches the supervisor mode, and the LBA matches the logical address, and this entry is valid (V=1), then this entry is used to map the logical address to the physical address specified in the PBA field.

WT Writethrough - If this bit is set, then cache memory updates mapped by this entry are performed using a writethrough policy. If this bit is clear then cache memory updates are performed using a write-back policy. (note: if the CI bit is set, this bit has no effect.)

G Global - If this bit is set, then memory mapped by this entry is global memory. The state of this bit is reflected on an external pin during the bus cycle and can be used by other CPU's on the bus to enable or disable snooping on this address.

CI Cache Inhibit - If this bit is set, then data in the block mapped by this entry is not cached. All accesses to this block will go through to memory and no data is read from, written to, or allocated in, the cache. If this bit is clear then data mapped by this entry is cached normally.

WP Write Protect - If this bit is set, then memory mapped by this entry is write protected. Any write access to this page will cause a data access exception. If this bit is clear then memory mapped by this entry can be written.

V Valid - If this bit is set, then this entry is currently valid and if the logical address matches the LBA, this entry will be used to translate the address.

3.8.1.2 Page ATC

The PATC contains 32, 64-bit entries that provide address translation, control, and protection information for logical-to-physical page translation. The format of the PATC entry for both the Instruction PATC and the Data PATC is shown in the figure below.

Page ATC Entry

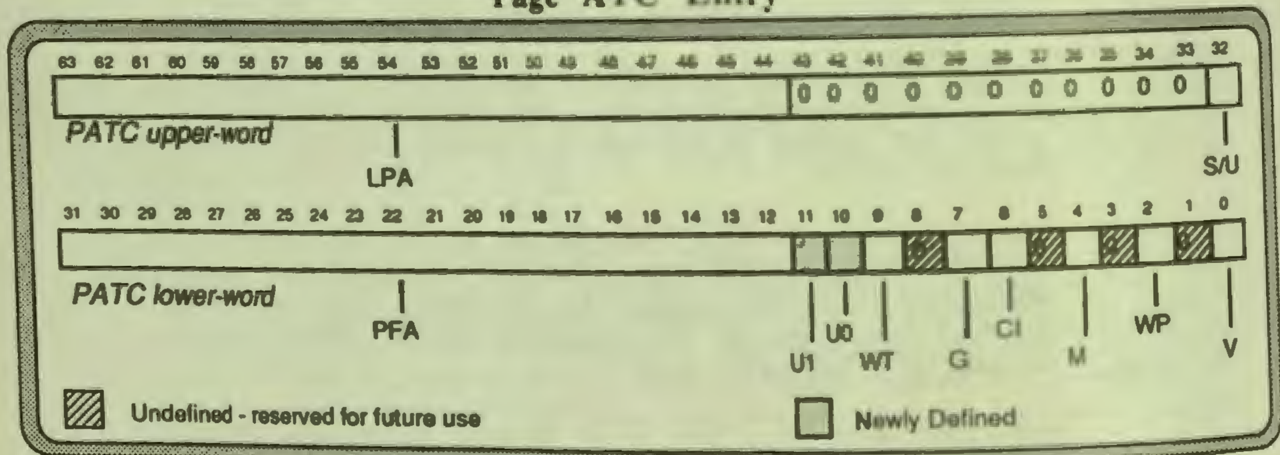


Figure 3.8.1.2 - Page ATC Entry

LPA Logical Page Address - This field contains the logical address that is to be mapped to a physical address by this ATC entry. The LPA is used as a tag which is matched against the logical address of subsequent memory references. If an address match is found (hit) then this entry is used to translate the logical address to a physical address as specified in the PFA. The 12 least significant bits of the logical address are not translated but are instead passed directly to the caches and bus interface unit. This results in a page size of 4K bytes.

S/U Supervisor/User Bit - The supervisor mode in effect for the memory address being translated acts like a 33rd bit of logical address. If the supervisor mode matches the S/U, and the LPA matches the logical address, and this entry is valid, then this entry is used to translate the logical address to a physical address as specified in the PFA. Thus if the S/U bit is a one, the entry will map only supervisor mode accesses. If the bit is a zero, the entry will map only user mode accesses.

PFA Page Frame Address - This field contains the upper 20 bits of the physical address that the logical address is being mapped to. On a memory access, if the upper 20 bits of the logical address match the LPA and access privileges are not violated, then the 20 most significant bits of the logical address are replaced by the PFA before being used by the caches and bus interface unit.

U0,U1 User Page Attribute 0,1 - These bits are not used by the MMU but are user definable via the page descriptors or by a PATC Control register write (using the xIR). They are presented on external pins during the bus transaction.

WT Writethrough - If this bit is set then cache memory updates are performed using a writethrough policy. If this bit is clear then cache memory updates are performed using a copyback policy. (note: if the CI bit is set, this bit has no effect.)

G Global - If this bit is set, then memory mapped by this entry is global memory. The state of this bit is reflected on an external pin during the bus cycle and can be used by other CPU's on the bus to enable or disable snooping on this address.

CI Cache Inhibit - If this bit is set, then data in the page mapped by this entry is not cached. All accesses to this page will go through to memory and no data is read from, written to, or allocated in, the cache. If this bit is clear then data mapped by this entry is cached normally.

M Modified - If this bit is set then a page mapped by this entry has been modified. If this bit is clear then a page mapped by this entry has not been modified. (NOTE: If this bit was clear previous to an access which causes it to be set, then a tablewalk is initiated which performs an indivisible read-modify-write cycle to update the page descriptor in memory.)

WP Write Protect - If this bit is set, then memory mapped by this entry is write protected. Any write access to this page will cause a data access exception. If this bit is clear then memory mapped by this entry can be written.

V Valid - If this bit is set, then this entry is currently valid and if the logical address matches the LPA, this entry it will be used to translate the address.

3.8.2 MMU CONTROL

The implementation of the MMU's allow the O.S. designer maximum flexibility for arranging a suitable memory management scheme for their particular system. At any given time the MMU can be in one of four modes: identity translation, BATC only translation, PATC only translation, and full BATC/PATC translation. These modes are described in the table and paragraphs below.

Translation Mode	To Activate: MMU Enable bit in ICTL or DCTL	To Activate: Translation Enable (TE) in User or Supr. Area Pointer	To Activate: Valid BATC Entry with Address Match	Source of Access Protection Information	Mapping
Identity	0	Don't Care	Don't Care	User or Supervisor Area Pointer	1:1
BATC Only	1	0	yes	BATC	BATC
	1	0	no	Area Pointer	1:1
PATC Only	1	1	no	PATC (or Tablewalk Descriptors)	PATC (or Tablewalk Descriptors
Full PATC and BATC	1	1	yes or no	BATC & PATC (or Tablewalk Descriptors	BATC & PATC (or Tablewalk Descriptors

Table 3.8.2 - MMU Address Translation Modes

3.8.2.1 Identity Translations

In the Identity Translation mode the MMU does not perform logical address translation, i.e., the logical program address is mapped directly (1:1) to the physical memory address. The access and protection information is retrieved from the appropriate User or Supervisor Area Pointer Register and applied to the memory access. To enable the identity translation mode, the MMU enable bit in the Instruction MMU/Cache Control Register (ICTL) (or Data MMU/Cache Control Register (DCTL)) must be cleared.

3.8.2.2 BATC Exclusive Translations

In the BATC Translation mode, the MMU maps logical address through the Block Address Translation Cache exclusively. This mode allows the operating system to explicitly manage system memory by specifying block level mappings and access privileges. BATC Translation mode is activated by clearing the translation enable bit

in the xSAP or xUAP (x=Instr. or Data), and setting the MMU enable bit in the ICTL (or DCTL). When activated, any logical address matching a valid BATC entry is mapped to the physical address specified in the entry and the corresponding access protection bits are applied to the memory reference.

3.8.2.3 PATC/Tablewalk Exclusive Translations

The PATC/Tablewalk Translation mode provides logical address translation and access protection through the PATC (on a hit) or through a descriptor fetched from memory based mapping tables by a tablewalk (on a PATC miss). To enable this mode, all entries in the BATC must be invalid, the translation enable bit in the xSAP or xUAP (x=Instr. or Data) must be set, and the MMU enable bit in the ICTL (or DCTL) must be set.

3.8.2.4 Full (BATC/PATC/Tablewalk) Address Translations

In the Full Address Translation mode, logical address translation and access protection is provided through the BATC, PATC, or a memory based descriptor fetched by a tablewalk. To enable this mode, the translation enable bit in the xSAP or xUAP (x=Instr. or Data) must be set, and the MMU enable bit in the ICTL (or DCTL) must be set.

If a match occurs in the PATC *and* BATC, the BATC entry is used to map the access. Multiple matching entries within the PATC or within the BATC are considered a programming error and unexpected results may occur.

3.8.3 TRANSLATION DESCRIPTORS

The memory based address translation tables are defined and accessed via address translation descriptors. A descriptor has three basic components: 1) an address which is either a pointer to the next lower level in the table hierarchy or, at the very end of the chain, the physical memory address of the page being mapped, 2) a set of access privilege rights to all pages below this level of the table hierarchy, and 3) possibly some status bits for maintaining state information for the page.

The 88110 defines four types of descriptors for use with the hardware tablewalk mechanism which are compatible with those used by the 88200. At the very top of the table hierarchy are area descriptors. These descriptors are the root of all memory based mapping tables and are kept in registers on chip. Four descriptors, one each for user code, user data, supervisor code, and supervisor data are maintained. The area descriptors point to segment tables in memory. Each segment table contains 1024 segment descriptors which each point to page tables. Each page table contains 1024 page descriptors. Each page descriptor may then describe the actual mapping and state for a page or point (indirection) to another page descriptor.

3.8.3.1 Area Descriptors

The area descriptors in the ISAP, DSAP, IUAP and DUAP registers contain the base address of the segment tables and the area control bits. They also enable

PATC/Tablewalk address translation via the TE bit. The U1,U0 user attribute bits define the state of the U1,U0 pins during all tablewalk accesses. The area descriptor format is shown below.

Area Descriptor Format

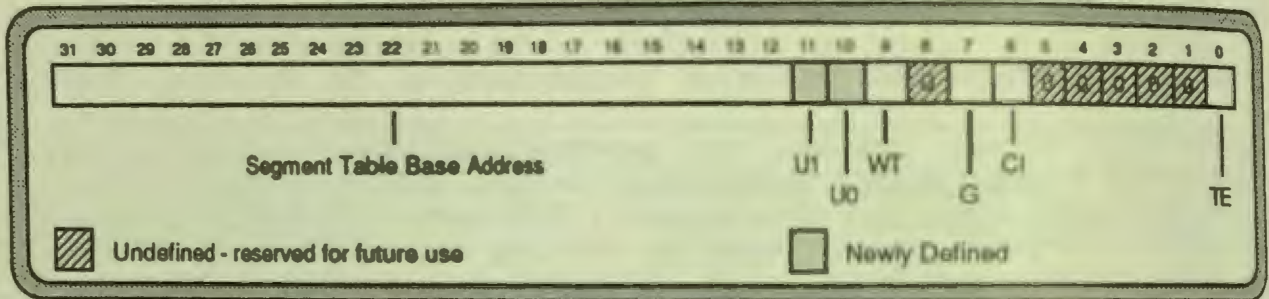


Figure 3.8.3.1 - Area Descriptor Format

U1,U0:	Address Translation User Attribute bits
WT:	Writethrough
G:	Global Access
CI:	Cache Inhibit
TE:	PATC/Tablewalk Translation Enable

3.8.3.2 Segment Descriptors

Each segment descriptor contains the base address of a page table. In addition, each segment descriptor contains segment-level protection and control information, which is logically ORed with the information from other descriptors. The segment descriptor format is shown below.

Segment Descriptor Format

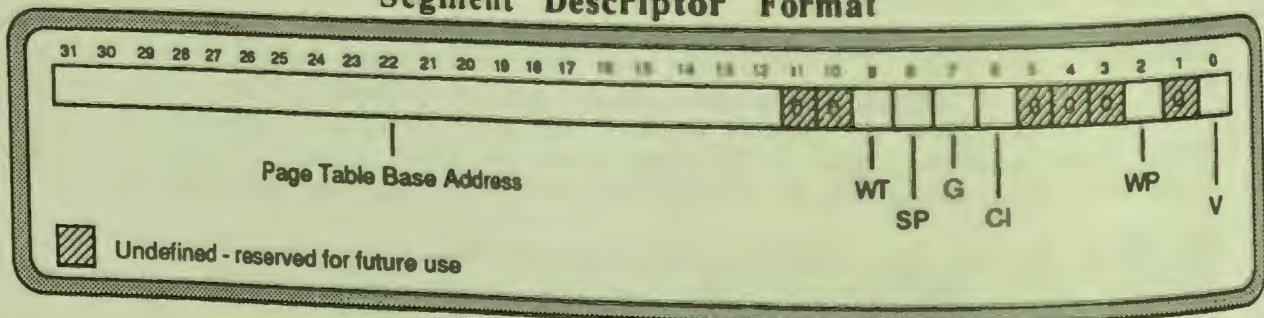


Figure 3.8.3.2 - Segment Descriptor Format

WT:	Writethrough
SP:	Supervisor Protection
G:	Global Access
CI:	Cache Inhibit
WP:	Write Protect
V:	Valid Bit

3.8.3.3 Page Descriptors

Each Page Descriptor contains the address of a physical page frame into which the logical page address is mapped. In addition, each descriptor contains page-level protection and control information, which is logically ORed with the area and segment information to form the protection and control for a particular page. The page descriptor format is shown below.

Page Descriptor Format

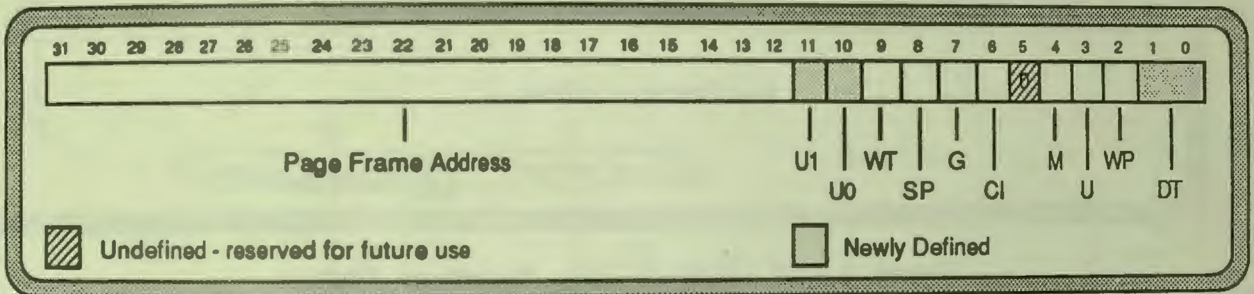


Figure 3.8.3.3 - Page Descriptor Format

U1:	User Page Attribute
U0:	User Page Attribute
WT:	Writethrough
SP:	Supervisor Protection
G:	Global Access
CI:	Cache Inhibit
M:	Modified
U:	Used
WP:	Write Protect
DT:	Descriptor Type
	00 - Invalid descriptor
	01 - Valid Page descriptor
	10 - Masked Protection Indirection descriptor
	11 - Indirection descriptor

3.8.3.4 Indirect Page Descriptors

If the DT field in a page descriptor is equal to 11 or 10, then the descriptor is actually a pointer to the actual page descriptor. Such a pointer is called an Indirect Page Descriptor and its format is shown in the following figure.

Indirect Page Descriptor Format

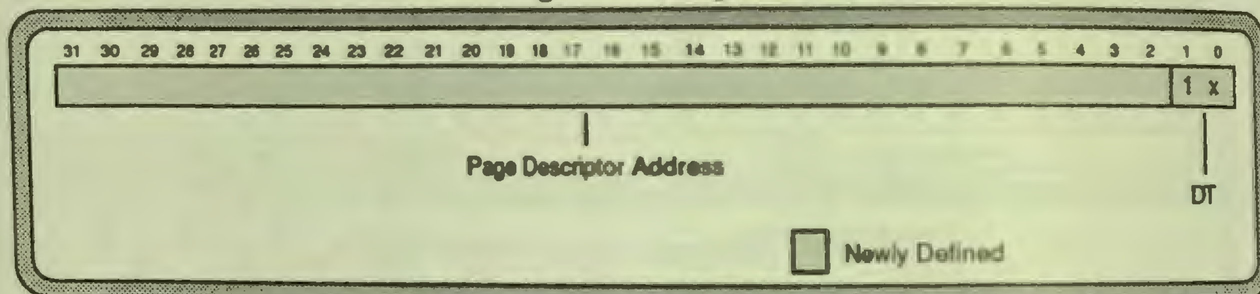


Figure 3.8.3.4 - Indirect Page Descriptor Format

PDA: Final Page Descriptor Address
DT: Descriptor Type
00 - Invalid descriptor
01 - Valid Page descriptor
10 - Masked Protection Indirection descriptor
11 - Indirection descriptor

3.8.4 PATC MISS AND REFILL

3.8.4.1 Hardware ATC Fill (hardware tablewalk)

When an PATC miss occurs, the 88110 selects a PATC entry for replacement using a FIFO algorithm and then performs a two level tablewalk to get the page descriptor for the referenced address. The figure below illustrates the hardware tablewalk sequence.

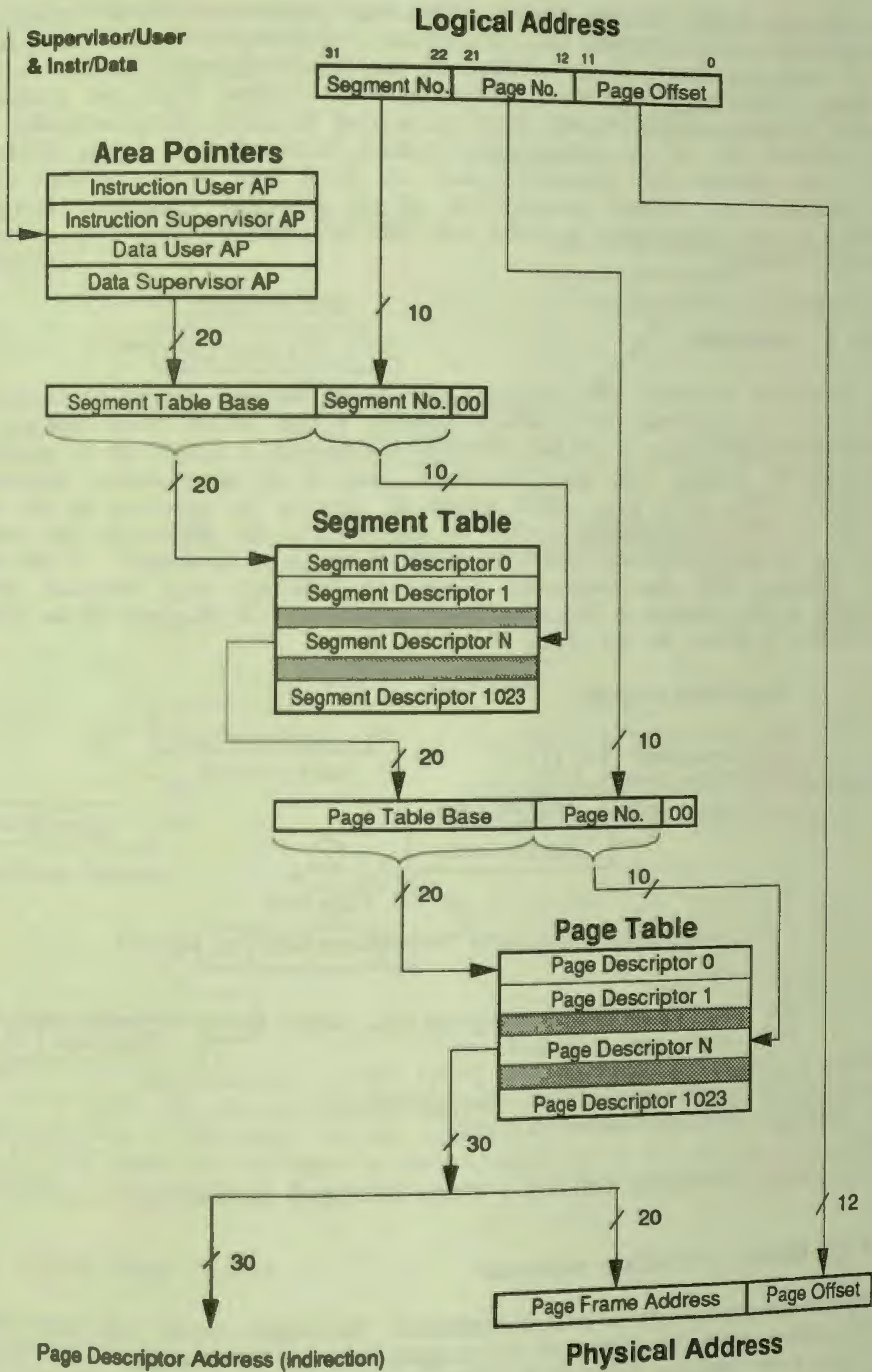


Figure 3.8.4.1 - Hardware Tablewalk

The tablewalk begins at the segment table base address found in the appropriate instruction/data/user/supervisor Area Descriptor. The segment table is indexed by the 10 most significant bits, <31:22> of the logical address to get the segment descriptor. The Page Table Base Address from the segment descriptor is indexed by the next 10 lesser significant bits, <21:12>, to find the final page descriptor. The 20 most significant bits of the missed logical address is loaded into the tag (LPA) of the PATC entry selected for replacement, and the 20-bit Page Frame Address from the page descriptor and logical inclusive OR of all protection bits found in the area, segment, and page descriptors is loaded into the lower half of the PATC entry (format described earlier).

3.8.4.1.1 Indirection

It is sometimes desirable for multiple pages to be mapped through a common page descriptor. For example, in a virtual memory system which maps multiple logical address to a single page, it is more efficient to maintain a single set of used/modified state bits by sharing one page descriptor than it is to maintain multiple page descriptors to the same page which would all have to be searched by the page-out software to control replacement. This is supported in the 88110 by two indirection modes set in the Descriptor Type (DT) field of the page descriptor. If the accessed page descriptor has the indirection mode set (DT=11), then hardware uses this descriptor as the address to the actual page descriptor. A diagram of the Indirection functionality is given in the figure below.

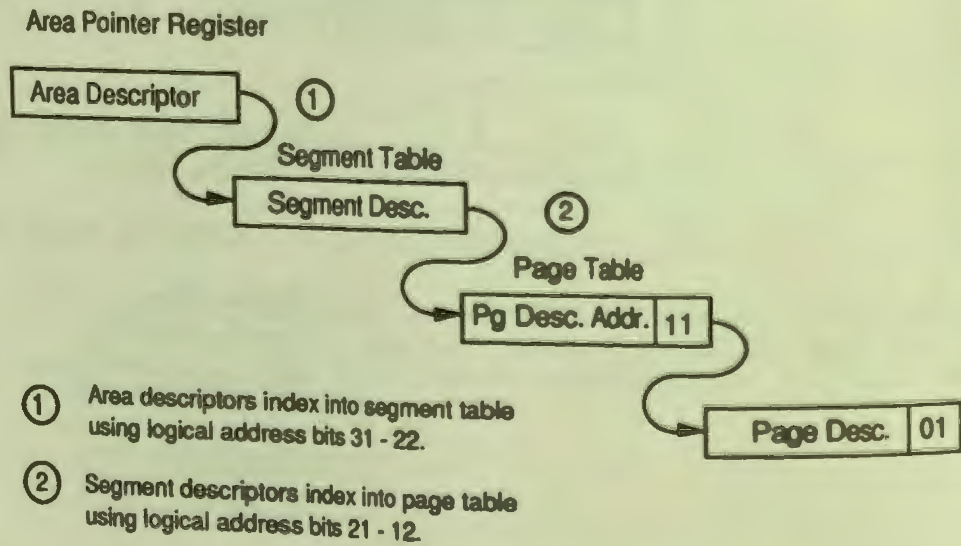


Figure 3.8.4.1.1 - Page Descriptor Indirection

3.8.4.1.2 Masked Protection Indirection

The 88110 also provides a masked protection indirection mode. In this mode, the 88110 allows multiple pages to be mapped through a common page descriptor, however, under variable protections. With this capability, a single page could be marked as write protected (Read Only) to one access and not write protected (Read/Write) to another.

In this case, if the accessed page descriptor has the masked protection indirection mode set (DT=10), then hardware uses the descriptor as the address to the actual page descriptor. However, the protection for the page is the combined protection of the corresponding area and segment descriptors only. The figure below gives an example of the masked protection indirection mode (where access A has Read/Write permission and accesses B & C have Read Only permission.).

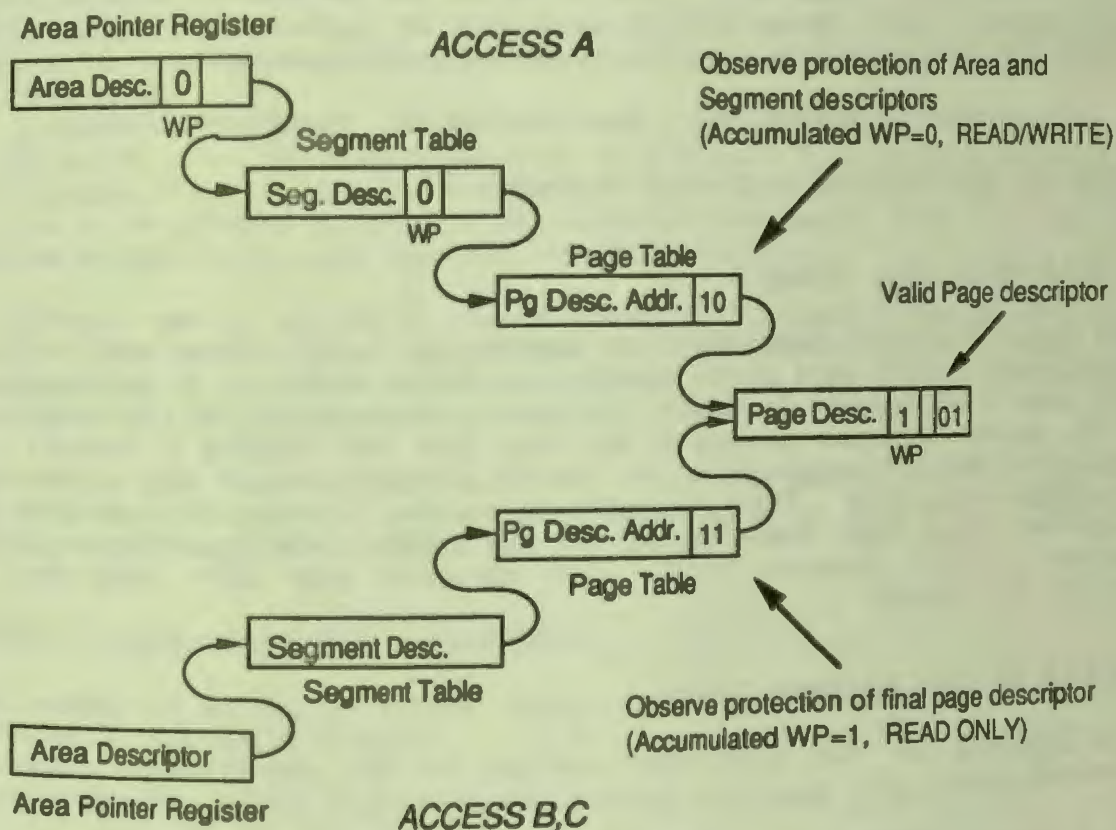


Figure 3.8.4.1.2 - Masked Protection Indirection

3.8.4.1.3 Page Used/Modified Status

If a memory write operation hits in the PATC and the entry is marked unmodified, then the PATC entry and the associated page descriptor in memory must be updated to reflect the new state of the page. In this situation the 88110 initiates a tablewalk, like that for a PATC miss, is performed to update the modified bit in the corresponding page descriptor. This operation is referred to as a UM (Used/Modified) update.

3.8.4.1.4 Sharing Page Tables

It is not uncommon in multiprocessor systems for processors in the same physical address space to share a common set of page tables. In this case it is necessary that the page descriptors be kept in a consistent state so that all processors see pages in the same state. Inconsistent page descriptors can arise in the case of UM update. If two processors were to read an unused/unmodified page descriptor simultaneously, one could immediately set the state to used/modified followed by a update by the

second processor marking it used/unmodified. Thus the modified status of the page would be lost. To prevent this situation, the 88110 performs the update to the used and modified bits in the page descriptor as an indivisible bus transaction.

UM update first initiates a normal tablewalk. If an update to the used bit in the Page descriptor is required, then tablewalk is followed by an indivisible (bus locked) read-modify-write operation which sets the used bit on and sets the modified bit on if it is a write operation OR if the modified bit is already set in the page descriptor. The 88110's UM update operation, then, has the following bus sequence:

<SegDescRead> <PageDescRead> [<PageDescReadLock , PageDescWriteUnlock>]

where the bus transaction in braces [] is conditional.

3.8.4.1.5 Paged Page Tables

The entire set of page tables needed for mapping the logical address space onto the physical address space need not be coresident in physical memory. To indicate that a page table is not currently in memory, the system software would clear the "valid" bit in the segment descriptor pointing to that page table thus marking it "invalid." A subsequent tablewalk encountering the invalid descriptor would take a data (or instruction) access fault. The system software can then determine from the Data (or Instruction) Access Fault Status Register that this exception was caused by an invalid segment descriptor (Segment Fault). The appropriate page table could then be brought into memory.

3.8.4.1.6 Hardware Tablewalk Timing

The following are the approximate timings for the non-exception Hardware Tablewalk cases:

Case 1 - without Indirection; no U,M update - 9 clocks
Case 2 - without Indirection; with U,M update - 16 clocks
Case 3 - with Indirection; no U,M update - 13 clocks
Case 4 - with Indirection; with U,M update - 20 clocks

3.8.4.2 Software ATC Fill (software tablewalk)

While the hardware tablewalk mechanism provided by the 88110 is the fastest ATC refill mechanism available, the defined mapping tables may by unsuitable for some operating systems. The 88110 therefore provides an efficient mechanism for software ATC fill to give complete flexibility to the system designer.

Software tablewalks are supported in the following ways:

1. An efficient mechanism for trapping to software on an ATC miss.
2. The missed virtual address is supplied in a register for quick access.
3. The ATC entry number to be replaced is provided in the ATC Index Register.
4. The upper word in the PATC entry is preloaded (Logical Page Address & S/U).
5. A mechanism is provided for software to load an ATC entry.

1) Software tablewalk is enabled by setting a bit in the Instruction MMU/Cache or Data MMU/Cache control registers (ICTL or DCTL). When this bit is set, an ATC miss will trap through either the Instruction MMU or Data MMU ATC Miss vectors which are provided solely for software tablewalk. Since these vectors are reserved exclusively for software tablewalk, it will not be necessary to search through status registers to determine the cause of the exception.

2) On an Instruction or Data ATC Miss exception, the virtual address of the faulting reference is stored in the Instruction Access or Data Access Logical Address register (ILAR or DLAR).

3) On an ATC miss, the index of the entry to be replaced is automatically deposited in the Instruction MMU or Data MMU ATC Index register (IIR or DIR). Thus, all that is required is for software to write to the appropriate write port - there is no need to compute a replacement entry and setup the Index Register.

4) The upper-word in the PATC entry is automatically preloaded when an ATC Miss occurs. That is, the Logical Page Address and Supervisor/User Status is loaded into the upper-word of the PATC entry to be replaced. Thus, all that is required is for software to write to the lower word in the PATC.

5) Software may load any ATC entry by writing the desired ATC data into the appropriate ATC Write Port register (IPPU, IPPL, DPPU, or DPPL). (See section on Loading and Storing a PATC Entry.)

3.8.4.2.1 Loading and Storing a PATC Entry

The loading and storing of a PATC Entry is a ordered, three-step (less for updates following an ATC Miss) sequence.

To write (or stcr) a new PATC entry the following steps <u>must</u> occur in order:

1. Place number of next PATC entry to be replaced into PATC index field in the IIR or DIR. *(optional if following an ATC Miss)*
2. Perform a write (stcr) into the upper-word of the PATC entry using the xPPU control register.*(optional if following an ATC Miss)*
3. Perform a write (stcr) into the lower-word of the PATC entry using the xPPL control register.

To read (or ldcr) a PATC entry the following steps <u>must</u> occur in order:

1. Place number of PATC entry to be read into PATC index field in the IIR or DIR.
2. Perform a load (ldcr) from the lower-word of the PATC entry using the xPPL control register.
3. Perform a load (ldcr) from the upper-word of the PATC entry using the xPPU control register.

To support software tablewalk, when an ATC miss occurs, the logical address and supervisor/user bit is preloaded into the PATC upper-word buffer. It will reside in this buffer until a write lower instruction is executed to actually store it into the PATC. However, upon detecting an ATC Miss, software can read the contents of this buffer (i.e. the missed logical address and supervisor/user status) by simply

executing a ldcr from the xPPU control register. The contents of this buffer are guaranteed until a ldcr from the xPPL control register is performed or another miss occurs.)

3.8.5 CACHE/MMU FAULT CONDITIONS

The following sections describe the faults or error conditions that may arise during a Code or Data Cache/MMU access.

3.8.5.1 Bus Error

If a bus error occurs during any code or data access, the (BE) bit is set in the ISR (or DSR). The physical address where the bus error occurred is placed in the Instruction Access Physical Address Register (IPAR) (or Data Access Physical Address Register - DPAR). (Note: This bit is not set if a bus error occurs during a tablewalk or a probe generated tablewalk.) A Code or Data Access Exception is then generated.

3.8.5.2 MMU Faults

If during the course of a normal address translation (non-probe), an invalid segment or page descriptor, tablewalk bus error, supervisor violation, or write protect violation is detected the MMU Fault (F) bit is set in the ISR (or DSR). A Code or Data Access Exception is then generated. The following sections describe each of these in more detail.

3.8.5.2.1 Invalid Segment and Page Descriptors

If an invalid segment or page descriptor is encountered during a normal (non-probe) tablewalk, the search is aborted and the physical address of the invalid segment or page descriptor is placed in the Instruction Access Physical Address Register (IPAR) (or DPAR for Data MMU accesses). To distinguish between an invalid segment or page descriptor fault and the other MMU faults, system software would need to perform an MMU probe of the faulting logical address (See section 4.6.6.2 Probe Error Conditions).

3.8.5.2.2 Tablewalk Bus Error

If a bus error occurs during a normal (non-probe) tablewalk, the search is aborted and the physical address where the bus error occurred is placed in the Instruction Access Physical Address Register (IPAR) (or DPAR for Data MMU accesses). To distinguish between the other MMU faults and the tablewalk bus error, system software would need to perform an MMU probe of the faulting logical address (See section 4.6.6.2 Probe Error Conditions).

3.8.5.2.3 Supervisor and Write Protection Violations

Unlike the above MMU faults, if a supervisor protection violation or a write protection violation (FOR DATA MMU ACCESSES ONLY) occurs during a normal (non-probe) tablewalk, the table search is NOT aborted. Instead the tablewalk is allowed to continue until the end of the tablewalk at which time the MMU fault bit (F) is set and a code (or data) access exception is generated. The physical address of the page descriptor is placed in the Instruction Access Physical Address Register (IPAR) (or DPAR for Data MMU accesses). To distinguish supervisor violation errors and write violation errors, system software would also need to perform an MMU probe of the faulting logical address. It would then compare the resulting status information (SP and WP bits) with the environment conditions of the access (See section on Probe Error Conditions).

3.8.5.3 Cache Copyback Errors (Data Cache Only)

If a copyback error occurs during a data copyback initiated by a cache miss transaction, then the (CP) bit is set in the DSR. The physical address where the copyback error occurred is placed in the DPAR. A Data Access Exception is then generated.

3.8.5.4 Cache Write Allocate Bus Error (Data Cache Only)

If a bus error occurs during a line read operation of a write cache miss implementing the write allocation policy, then the (WA) bit is set in the DSR. The physical address where the allocate bus error occurred is placed in the DPAR. A Data Access Exception is then generated.

3.8.6 PROBE TRANSACTIONS

Probe transactions are provided by the 88110 to load descriptors into the PATC, reload current PATC entries with more recent status, and to accumulate status information for a logical-to-physical address translation. Probes are useful in determining if a local (exclusive) page has been modified so that any cached data may be written back to memory when a task terminates. Probes are also useful to obtain the physical address that is mapped from a certain logical address.

3.8.6.1 Probe Implementation

The 88110 is capable of performing both Instruction MMU Probes and Data MMU Probes. The following outlines only the process for performing an Instruction MMU probe, however, a Data MMU probe can be achieved in exactly the same manner except using Data MMU Control Registers.

Before performing a probe transaction, the operating system initializes the Instruction System Address Register (ISAR) with the logical address it wishes to probe. It then writes a probe command to the Instruction Command Register (ICMD). There are two types of probe commands: probe user address and probe supervisor address. Both commands search the Instruction MMU BATC and PATC for the logical

address in the ISAR, however, the actions taken by the probe vary depending upon the control settings of the MMU. The different probe cases are described below.

Case 1: If the MMU is disabled (via the MEN bit in the xCTL):

Both BATC and PATC entries are searched. If a match is found in either, then translation status is written into the Instruction Access Status Register (ISR) and the corresponding physical address is written into the Instruction Access Physical Address Register (IPAR). If a match is not found in either, it is indicated via the PH and BH bits of the xSR. The value in the xPAR as well as the other status bits in the xSR are undefined. The table walk is ***not*** performed in either case.

Case 2: If the MMU is enabled and a matching BATC entry is present:

Upon completing the search, translation status is written into the Instruction Access Status Register (ISR) and the corresponding physical address is written into the Instruction Access Physical Address Register (IPAR). (Note: for more information on probe status and how it is presented in the ISR see section on Cache and MMU Control Registers).

Case 3: If the MMU is enabled, a matching BATC entry is not present, and hardware tablewalks are enabled (via TEN bit in the xCTL):

Upon completing the search, a new PATC entry is created (or a matching PATC entry is reloaded) and a tablewalk is performed through the Instruction Supervisor Area Pointer Register (ISAP) or the Instruction User Area Pointer Register (IUAP). This is done even if there is a matching PATC entry for the purpose of obtaining the most recent translation status for that entry. During the tablewalk, access and protection information is accumulated from the area, segment, and page tables and loaded (or reloaded) into the PATC entry. The accumulated translation status is also written into the ISR and the corresponding physical address is written into the IPAR. If a probe error is encountered while tablewalking for a matching PATC entry, that entry will be invalidated. If a probe error occurs while tablewalking for a non-matching address then a new PATC entry is not created.

Case 4: If the MMU is enabled, a matching BATC entry is not present, and hardware tablewalks are disabled (software tablewalk case):

If a match is found in the PATC, then translation status is written into the Instruction Access Status Register (ISR) and the corresponding physical address is written into the Instruction Access Physical Address Register (IPAR). If a match is not found in either, it is indicated via the PH and BH bits of the xSR. The value in the xPAR as well as the other status bits in the xSR are undefined. The table walk is ***not*** performed in either case.

3.8.6.2 Probe Error Conditions

The major difference between probe errors and normal (non-probe) address translation errors is that probe errors never cause a Code (or Data) Access Exception. The following describes each of the possible probe errors in more detail.

3.8.6.2.1 Invalid Segment or Page Descriptors

If an invalid segment or page descriptor is encountered during a probe generated tablewalk, the search is aborted and the ISR (or DSR for Data MMU probes) is updated to reflect a segment descriptor invalid (SI=1) or a page descriptor invalid (PI=1). The physical address of the invalid segment or page descriptor is placed in the Instruction Access Physical Address Register (IPAR) (or DPAR for Data MMU probes).

3.8.6.2.2 Tablewalk Bus Error

If a bus error occurs during a probe generated tablewalk, the search is aborted and the ISR (or DSR) is updated to reflect a tablewalk bus error (TBE=1). The physical address where the tablewalk bus error occurred is placed in the IPAR (or DPAR).

3.8.7 BREAKPOINTS ON LOGICAL ADDRESSES

The 88110 offers a new and significant debugging feature for the 88000 family. The 88110 Instruction and Data MMU will each contain two logical breakpoint registers which are available for system debugging support.

Breakpoints are enabled by setting the BPEN (breakpoint enable) bit in the xCTL. The breakpoint registers are loaded in the same manner as the PATC entries, except in this case, breakpoint registers 0,1 are mapped as PATC entries 32,33 (via the Index Register). As with the PATC, to write (or **stcr**) a new breakpoint register entry the following steps ***must*** occur in order:

1. Place the number of the Breakpoint entry to be replaced (32,33 for breakpoint registers 0,1, respectively) into the PATC/Breakpoint index field of the IIR or DIR.
2. Perform a write (**stcr**) into the upper-word of the Breakpoint entry using the xPPU control register.
3. Perform a write (**stcr**) into the lower-word of the Breakpoint entry using the xPPL control register.

To read (or **ldcr**) a breakpoint register entry the following steps <u>must</u> occur in order:

1. Place the number of the Breakpoint entry to be read (32,33 for breakpoint registers 0,1, respectively) into the PATC/Breakpoint index field of the IIR or DIR.
2. Perform a load (**ldcr**) from the lower-word of the Breakpoint entry using the xPPL control register.
3. Perform a load (**ldcr**) from the upper-word of the Breakpoint entry using the xPPU control register.

The breakpoint register entry (upper and lower words) is defined differently than the PATC entry and is presented in the figure below.

Breakpoint Register Entry

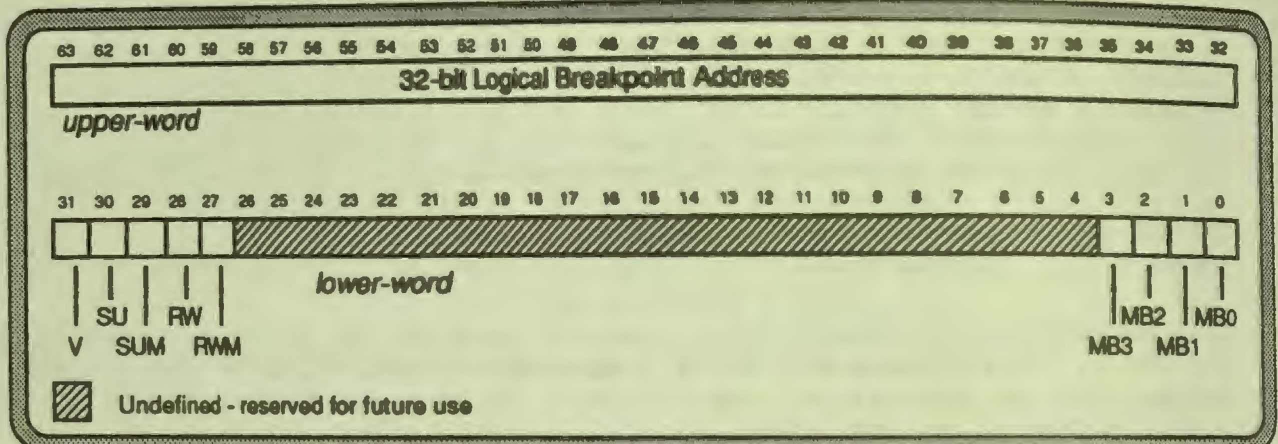


Figure 3.8.7 - Breakpoint Register Entry

The 32-bit logical breakpoint address is accessed as the upper-word of the breakpoint register.

The supervisor/user(SU), read/write(RW) criteria, and valid bit(V) (or enable) for each breakpoint register is set via the lower word of the breakpoint entry. Therefore, either breakpoint register can be set to break upon a read or write access to a supervisor or user space. In addition, the supervisor/user and read/write field for each breakpoint register can also be masked (break on any combination). These masks (SUM and RWM) are also set via the lower word of the breakpoint register.

In addition to the supervisor/user and read/write masks, four address masks exist for breaking on variable address ranges. These four address masks are also set via the lower word of the breakpoint register and can be defined to qualify a break as follows:

if (MB0,MB1,MB2,MB3 = 0) - Break on byte addresses (bits 31-0)
if (MB0=1)&&(MB1,MB2,MB3 = 0) - Break on half-word addresses (bits 31-1)
if (MB0,MB1=1)&&(MB2,MB3 = 0) - Break on word addresses (bits 31-2)
if (MB0,MB1,MB2=1)&&(MB3 = 0) - Break on double-word addresses (bits 31-3)
if (MB0,MB1,MB2,MB3=1) - Break on quad-word addresses (bits 31-4)

All 32 bits of each breakpoint register can be written, however, breakpoint matches are a function of the granularity of each access compared. That is, a word access will only be compared with bits 31-2 of the breakpoint register to determine a match, where as, a byte access will compare all 32 bits. This scheme holds true for byte, half-word, word, double word, and quad-word accesses.

Finally, when the logical address of the access (load or store on the data side or fetch on the instruction side) matches the address in the breakpoint register, breakpoints are enabled (via the BPEN bit in the xCTL), and all other conditions are met (such as address range, access granularity, read/write, supervisor/user, register valid) the Breakpoint Exception bit (BPE) is set in the Instruction (or Data) Access Status Register. A Code or Data Access Exception is then generated.

3.8.8 CACHE AND MMU CONTROL REGISTERS

The following Control Registers are dedicated for use by the on-chip caches and memory management units. As with the other control registers in the 88110, they are accessible via the **stcr** and **ldcr** instructions. However, unlike the other control registers in the 88110, the Cache and MMU Control registers do not support the **xcr** (exchange control register) operation.

Instruction MMU/Cache/TIC Command Register (ICMD)

The Instruction MMU/Cache/TIC Command register controls Cache flushing, Cache and ATC invalidation, and MMU probing. A **stcr** (write) to the Command Register initiates the specified action. The 5 least significant bits (4-0) of the data written to the ICMD register contain the command code. Reading the ICMD will return all zero's. The following commands are defined:

Instruction MMU/Cache/TIC Command Register (ICMD)

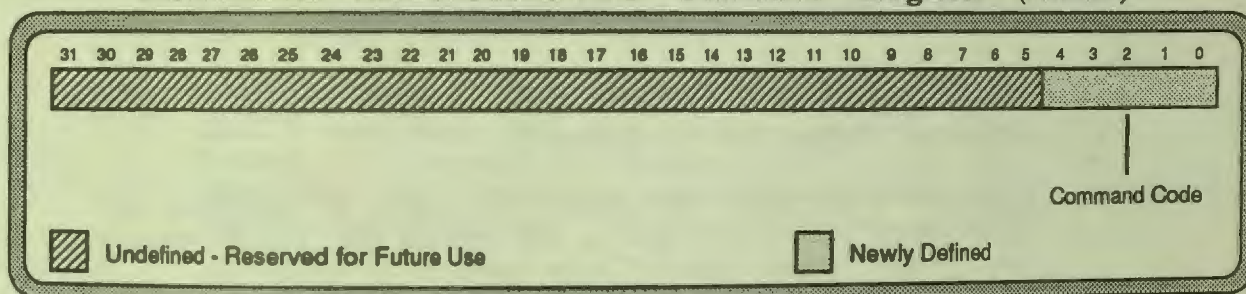


Figure 3.8.8.a - Instruction MMU/Cache/TIC Command Register

Code	Command
00000 -	No Operation
00001 -	Reserved
00010 -	Invalidate TIC Cache
00011 -	Invalidate ICache and TIC Cache
00100 -	Invalidate ICache Line
00101 -	Reserved
00110 -	Reserved
00111 -	Reserved
01000 -	MMU Probe Supervisor
01001 -	MMU Probe User
01010 -	Invalidate All Supervisor ATC Entries
01011 -	Invalidate All User ATC Entries
011xx -	Reserved
1xxxx -	Reserved

Instruction MMU/Cache Control Register (ICTL)

This register controls the operating modes of the Instruction Cache, Branch Target Cache, and the Instruction MMU, including the mask bits to specify the BATC Block Size (512k to 64M).

Instruction MMU/Cache Control Register (ICTL)

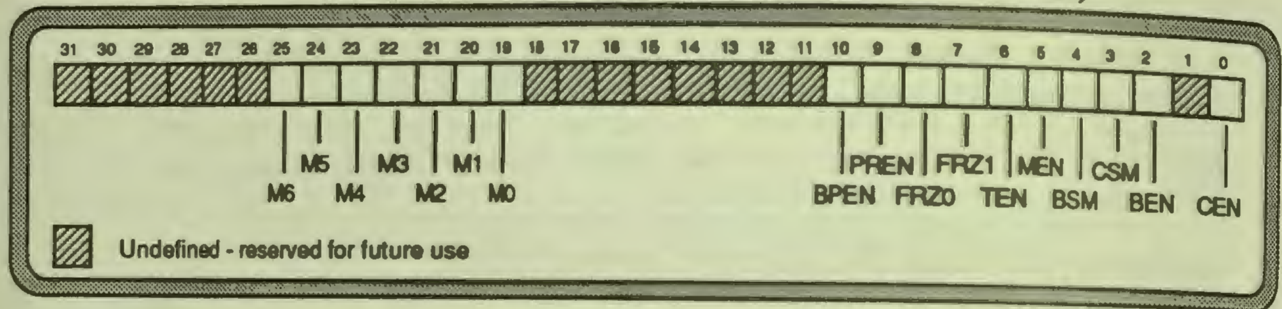


Figure 3.8.8.b - IMMU/ICache Control Register

Field	Description
CEN	Instruction Cache Enable/Disable. When disabled, instruction fetches pass directly to the bus interface unit and the ICache is not accessed or updated. On reset the Instruction Cache is disabled.
BEN	TIC cache Enable/Disable. When disabled, no instructions are fetched from the TIC cache and it is not accessed or updated. On reset the Branch Target Cache is disabled.
CSM	Instruction Cache SRAM test mode. This feature is specifically for rapid testing of the cache and is not intended for general program use. When enabled, normal ICache operation is suspended. Instruction fetches bypass the ICache and go directly to the bus interface. The ICache data, tag, and status are now readable and writeable directly as memory locations beginning at the base address specified in the Instruction System Address Register (ISAR). On reset the ICache is NOT in SRAM test mode. The ISAR should be explicitly set up prior to putting the ICache in SRAM mode.
BSM	TIC cache SRAM test mode. When enabled, normal TIC operation is suspended. This feature is specifically for rapid testing of the TIC and is not intended for general program use. Instruction fetches bypass the TIC and go directly to the ICache. The TIC data, tag, and status are now readable and writeable directly as memory locations beginning at the base address specified in the Instruction System Address Register (ISAR). SRAM mode control is encoded so that either the ICache or TIC may be in SRAM mode but not both. On reset the TIC is NOT in SRAM mode. The ISAR should be explicitly set up prior to putting the TIC in SRAM mode.
MEN	Instruction MMU Enable/Disable. When this bit is enabled Address translations can occur via the BATC/PATC or Tablewalks. If this bit is disabled then all logical addresses are presented as the physical address (Identity Translation) and the access/protection information is taken from the ISAP or IUAP. On reset the Instruction MMU is disabled.

TEN Instruction MMU Hardware Tablewalk Enable/Disable. When enabled (the default case), a hardware tablewalk is performed when a ATC miss occurs. When disabled, a trap to the Instruction MMU ATC Miss vector occurs on an ATC miss.

FRZ0 Instruction Cache Freeze Bank 0 Enable/Disable. When enabled all of the first lines (line 0) of each bank in the Instruction cache are locked.

FRZ1 Instruction Cache Freeze Bank 1 Enable/Disable. When enabled all of the second lines (line 1) of each bank in the Instruction cache are locked.

PREN Branch Prediction Enable. When disabled, the branch reservation station is disabled and a branch with a data dependency will stall instruction issue. When enabled, branches with data dependencies issue to the branch reservation station and conditional instruction issue occurs in the direction specified by the branch opcode. On reset, branch prediction is disabled.

BPEN Breakpoint Enable. When disabled, the breakpoint registers will not cause a code access fault upon detecting a matching logical address. disabled. When enabled, the breakpoint registers will cause a code access fault upon detecting a matching logical address provided the matching register is valid. On reset, breakpoints are disabled.

M6:0 Instruction MMU BATC block size selection bits. The block sizes mapped by the BATC are programmable according to the following table:

Instruction BATC Size Mask Bits							Block Size
M_6	M_5	M_4	M_3	M_2	M_1	M_0	
1	1	1	1	1	1	1	64MB
0	1	1	1	1	1	1	32MB
0	0	1	1	1	1	1	16MB
0	0	0	1	1	1	1	8MB
0	0	0	0	1	1	1	4MB
0	0	0	0	0	1	1	2MB
0	0	0	0	0	0	1	1MB
0	0	0	0	0	0	0	512KB
Any Other Combination							undefined

Table 3.8.8.a - Instruction BATC Block Size Selection

Instruction System Address Register (ISAR)

The instruction cache or branch target cache can be placed in an SRAM test mode under control of the MMU/Cache Control Register. When in this mode it is mapped as a contiguous block of memory into the logical address space. The Instruction System Address Register (ISAR) contains the base address of the cache

while it is in SRAM mode. This register can be loaded to relocate the cache in memory. When the ICache is mapped as an SRAM, word 0 of set 0 of line 0 starts at the base address, followed by the 128 sets of line 1, followed by the tags, in the direction of increasing memory addresses. When the TIC is mapped as SRAM, TIC target instruction 0 of entry 0 is located at the base address, also located in the ISAR, followed by instruction 0 of the remaining entries, followed by instruction 1 of all entries, followed by the tags, in the direction of increasing memory addresses.

This register is also written by software to pass logical addresses for ICache Invalidation on Line Granularity and Instruction MMU probes.

Instruction Supervisor Area Pointer Register (ISAP)

The Instruction Supervisor Area Pointer Register is a read/write MMU control register. It is written with the area pointer segment table address and control/protection information for instruction supervisor address space. The format of this register is the same as previously shown for the area descriptor.

Instruction User Area Pointer Register (IUAP)

The Instruction User Area Pointer Register is a read/write MMU control register. It is written with the area pointer segment table address and control/protection information for instruction user address space. The format of this register is the same as previously shown for the area descriptor.

Instruction MMU ATC Index Register (IIR)

The Instruction MMU ATC Index Register is a read/write MMU control register. It specifies which BATC entry is to be loaded with the contents of the Block ATC Write Port as well as which PATC entry is to be loaded with the contents of the Page ATC Write Ports. It can also specify which instruction breakpoint register is to be loaded (where Breakpoint Register 0 = index of 32 and Breakpoint Register 1 = index of 33). The IIR also contains the User Attribute bits. A field description of this register is given below.

Instruction MMU ATC Index Register (IIR)

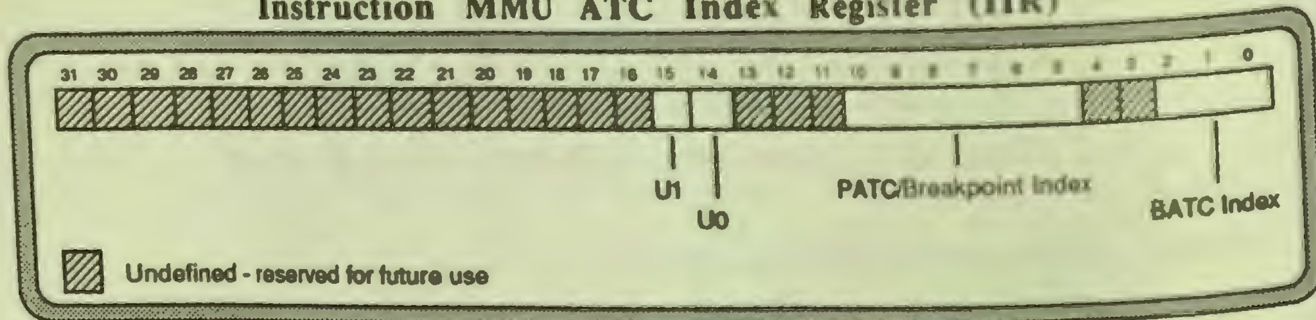


Figure 3.8.8.c - Instruction MMU ATC Index Register

Field	Description
BATC Index	Next BATC Entry to be replaced (0-7).
PATC/Bkpt Index	Next PATC Entry to be replaced (0-31).
	(Note: Entry 32 specifies Breakpoint Register 0 Entry 33 specifies Breakpoint Register 1)
	U1,U0 User Block Attribute bits.

Instruction MMU Block ATC Read/Write Port (IBP)

When a write (stcr) is performed with the Instruction MMU Block ATC Read/Write Port, the data is loaded into a Instruction BATC entry. Likewise, when a read (ldcr) is performed with the Instruction MMU Block ATC Read/Write Port, the data is read from a Instruction BATC entry. The specific BATC entry to be loaded is indicated by the Instruction MMU ATC Index Register (IIR). The Block size mask bits (variable from 512K to 64M) are fixed for all eight BATC entries and set via the ICTL. The User Block Attribute bits specified by the IIR are also used to load the BATC entry. The User Block Attribute bits are only accessible via the IIR. A description of the IBP register is shown in the figure below.

Instruction MMU Block ATC Read/Write Port (IBP)

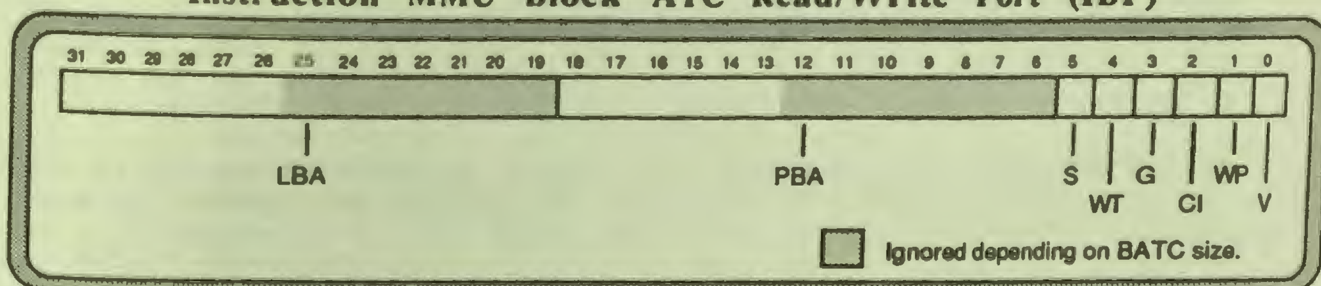


Figure 3.8.8.d - IMMU BATC Read/Write Port

Field	Description
V	Valid Bit
WP	Write Protection
CI	Cache Inhibit
G	Global
WT	Writethrough memory update policy
S	Supervisor/User Address Space
PBA	Physical Block Address. Use: a) bits (18-6) for 512K Blocks b) bits (18-7) for 1M Blocks c) bits (18-8) for 2M Blocks d) bits (18-9) for 4M Blocks e) bits (18-10) for 8M Blocks f) bits (18-11) for 16M Blocks g) bits (18-12) for 32M Blocks h) bits (18-13) for 64M Blocks
LBA	Logical Block Address. Use: a) bits (31-19) for 512K Blocks

LBA Logical Block Address. Use:

a) bits (31-19) for 512K Blocks
b) bits (31-20) for 1M Blocks
c) bits (31-21) for 2M Blocks
d) bits (31-22) for 4M Blocks
e) bits (31-23) for 8M Blocks
f) bits (31-24) for 16M Blocks
g) bits (31-25) for 32M Blocks
h) bits (31-26) for 64M Blocks

Instruction MMU Page ATC Read/Write Port - Upper (IPPU)

When a write (**stcr**) is performed with the Instruction MMU Page ATC Read/Write Port - Upper, the data is temporarily buffered in preparation for loading into the most significant word of a Instruction PATC entry (or Breakpoint entry). The data is NOT loaded into the the most significant word until a **stcr** is performed for the corresponding PATC read/write port - lower (IPPL). (See section on Loading and Storing a PATC Entry.)

When a read (**ldcr**) is performed with the Instruction MMU Page ATC Read/Write Port - Upper (IPPU), the data is read from the most significant word of an Instruction PATC entry (or Breakpoint entry) and transferred into the general purpose register. However in this case, a read (**ldcr**) from the Instruction MMU Page ATC Read/Write Port - Lower (IPPL) must PRECEDE this action. (See section on Loading and Storing a PATC Entry.)

The specific PATC entry (or Breakpoint entry) to be loaded is indicated by the Instruction MMU ATC Index Register (IIR). The format of this register is the same as the most significant (upper) word of the PATC entry (or Breakpoint entry) as previously described .

Instruction MMU Page ATC Read/Write Port - Lower (IPPL)

When a write (**stcr**) is performed with the Instruction MMU Page ATC Read/Write Port - Lower, the data is loaded into the least significant word of a Instruction PATC entry (or Breakpoint entry). (Note: at this time the data for the upper PATC word, which should have been set up previously, is now written). (See section on Loading and Storing a PATC Entry.)

When a read (**ldcr**) is performed with the Instruction MMU Page ATC Read/Write Port - Lower (IPPL), the data is read from the least significant word of the Instruction PATC entry (or Breakpoint entry). This action should precede the read (**ldcr**) of the Instruction MMU Page ATC Read/Write Port - Upper (IPPU). (See section on Loading and Storing a PATC Entry.)

The specific PATC entry (or Breakpoint entry) to be loaded is indicated by the Instruction MMU ATC Index Register (IIR). The format of this register is the same as the least significant (lower) word of the PATC entry (or Breakpoint entry) as previously described .

Instruction Access Status Register (ISR)

The Instruction Access Status Register is a read/write control register. This register provides Instruction MMU probe results (address translation status) and fault information for Code Access Exceptions including Instruction Breakpoint Exceptions. This register must be cleared by software after any exceptions.

Instruction Access Status Register (ISR)

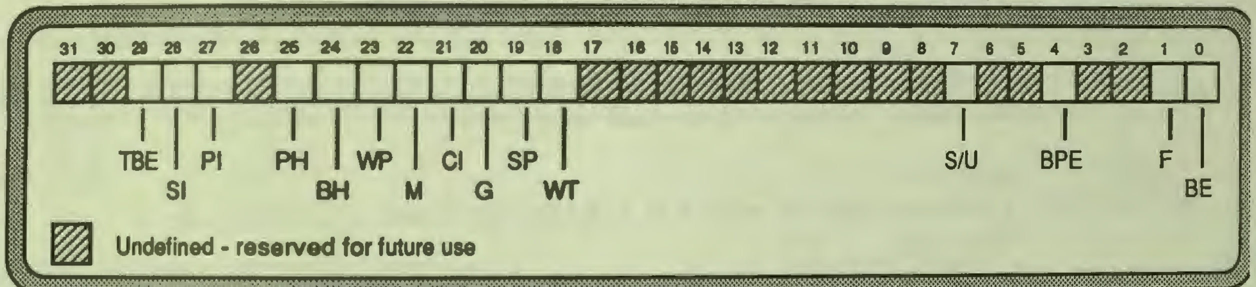


Figure 3.8.8.e - Instruction Access Status Register

Field	Description
BE	Bus Error - Indicates that a bus error occurred.
F	MMU Fault - Indicates that a fault occurred during the course of a normal (non-probe) address translation. MMU faults include: invalid segment descriptor, invalid page descriptor, tablewalk bus error, supervisor violation, and write protect violation.
BPE	Breakpoint Exception occurred - The Logical Address where the break occurred is located in the Instruction Access Logical Address Register (ILAR).
S/U	Supervisor/User Status - Indicates the supervisor/user status of the instruction access in error.
WT	Writethrough - (NOTE: The WT bit is not used for Instruction Accesses but is provided as an indicator to reflect content of descriptors.) 1 = Data at the probed address is cached with the writethrough memory update policy. 0 = Data at the probed address is cached with the copyback memory update policy.
SP	Supervisor Privilege - 1 = Probed address can only be accessed in the supervisor mode. 0 = Probed address can be accessed in the user or supervisor mode.
G	Global - 1 = One or more of the descriptors for the probed address are marked global. 0 = None of the descriptors for the probed address are marked global.

CI — Cache Inhibit -
1 = Data at the probed address cannot be cached.
0 = Data at the probed address can be cached.

M — Modified - (NOTE: The M bit is not used for Instruction Accesses but is provided as an indicator to reflect content of descriptors.)
1 = Data at the probed address has been modified in memory or with respect to memory (cached data).
0 = Data at the probed address has not been modified.

WP — Write Protection -
1 = Probed address is write protected.
0 = Probed address can be read or written.

BH — BATC Hit -
1 = Probed address resulted in a BATC hit instead of a PATC hit or tablewalk.
0 = Probed address was found in the PATC or was generated by a tablewalk.

PH — PATC Hit -
1 = Probed address resulted in a PATC hit.
0 = Probed address was not found in the PATC.

PI — Page Descriptor Invalid - Indicates that an invalid page descriptor was encountered during a Probe generated tablewalk.

SI — Segment Descriptor Invalid - Indicates that an invalid segment descriptor was encountered during a Probe generated tablewalk.

TBE — Probe Tablewalk Bus Error - Indicates that an bus error was encountered during a Probe generated tablewalk.

Instruction Access Logical Address Register (ILAR)

The Instruction Access Logical Address Register is a read/write control register. In the case where an instruction ATC miss occurs and hardware tablewalks are disabled (via the ICTL), this register contains the logical address. It also contains the logical address of Instruction (code) access exceptions.

Instruction Access Physical Address Register (IPAR)

The Instruction Access Physical Address Register is a read/write control register. This register contains the Physical address where the Instruction (code) access exception occurred or the physical address of the tablewalk descriptor (segment or page) in error. The MC88110 updates this register whenever an exception occurs on an instruction access.

Data MMU/Cache Command Register (DCMD)

The Data MMU/Cache Command register controls cache flushing, cache and ATC invalidation, and MMU probing. A stcr (write) to the Command Register initiates the specified action. The 5 least significant bits (4-0) of the value written to the DCMD register contain the command code. Reading the DCMD will return all zero's. The following commands are defined:

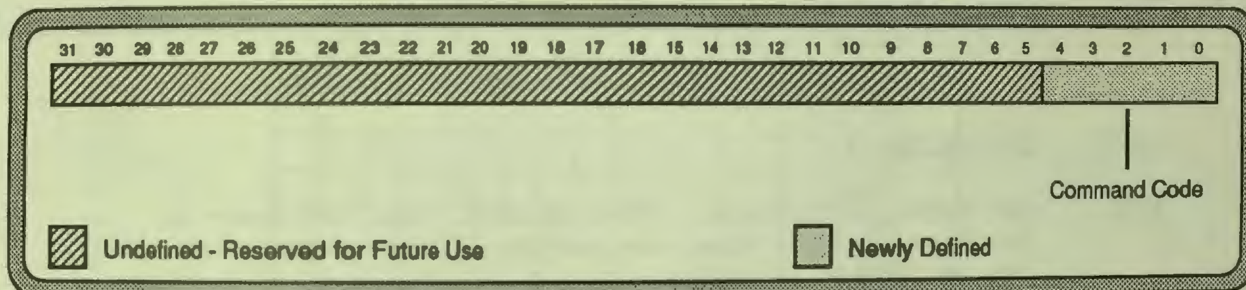


Figure 3.8.8.f - Data MMU/Cache Command Register

Code	Command
00000 -	No Operation
00001 -	Reserved
00010 -	Flush DCache (copyback)
00011 -	Invalidate DCache
00100 -	Flush DCache Line (copyback)
00101 -	Invalidate DCache Line
00110 -	Reserved
00111 -	Reserved
01000 -	MMU Probe Supervisor
01001 -	MMU Probe User
01010 -	Invalidate All Supervisor ATC Entries
01011 -	Invalidate All User ATC Entries
011xx -	Reserved
1xxxx -	Reserved

Data MMU/Data Cache Control Register (DCTL)

This register controls the operating modes of the Data Cache. It also controls the operating modes of the Data MMU, including the mask bits to specify the BATC Block Size (512k to 64M)

Data MMU/Cache Control Register (DCTL)

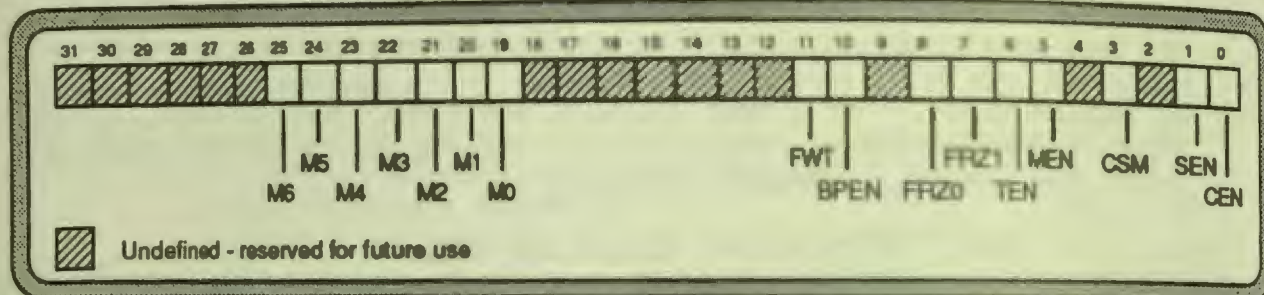


Figure 3.8.8.g - DMMU/DCache Control Register

Field	Description
CEN	Data Cache Enable/Disable. When disabled, loads and stores go directly to the bus interface unit and the DCache is not updated. On reset the DCache is disabled.
SEN	Data Cache Snooping Enable/Disable.
CSM	Data Cache SRAM test mode. This feature is specifically for rapid testing of the DCache and is not intended for general program use. When enabled, normal DCache operation is suspended. Loads and Stores bypass the DCache and go directly to the bus interface. The DCache data, tag, and status are now readable and writeable directly as memory locations beginning at the base address specified in the Data System Address Register (DSAR). On reset the DCache is NOT in SRAM mode. The DSAR should be explicitly setup prior to putting the DCache in SRAM mode.
MEN	Data MMU Enable/Disable. When this bit is enabled Address translations can occur via the BATC/PATC or Tablewalks. If this bit is disabled then all logical addresses are presented as the physical address (Identity Translation) and the access/protection information is taken from the DSAP or DUAP. On reset the Data MMU is disabled.
TEN	Data MMU Hardware Tablewalk Enable/Disable. When enabled (the default case), a hardware tablewalk is performed when a ATC miss occurs. When disabled, a trap to the Data MMU ATC Miss vector occurs on an ATC miss.
FRZ0	Data Cache Freeze Bank 0 Enable/Disable. When enabled all of the first lines (line 0) of each bank in the Data cache are locked.
FRZ1	Data Cache Freeze Bank 1 Enable/Disable. When enabled all of the second lines (line 1) of each bank in the Data cache are locked.
BPEN	Breakpoint Enable. When disabled, the breakpoint registers will not cause a code access fault upon detecting a matching logical address. When enabled, the breakpoint registers will cause a code access fault upon detecting a matching logical address provided the matching register is valid. On reset, breakpoints are disabled.

FWT Force Write-through. Forces all stores to write-through the cache independent of the page status or store-through instruction option, but does not affect normal operation of the WT* pin.

M6:0 Data MMU BATC block size selection bits. The block sizes mapped by the BATC are programmable according to the following table:

Data BATC Size Mask Bits							Block Size
M_6	M_5	M_4	M_3	M_2	M_1	M_0	
1	1	1	1	1	1	1	64MB
0	1	1	1	1	1	1	32MB
0	0	1	1	1	1	1	16MB
0	0	0	1	1	1	1	8MB
0	0	0	0	1	1	1	4MB
0	0	0	0	0	1	1	2MB
0	0	0	0	0	0	1	1MB
0	0	0	0	0	0	0	512KB
Any Other Combination							undefined

Table 3.8.8.b - Data BATC Block Size Selection

Data System Address Register (DSAR)

The data cache can also be placed in an SRAM mode under control of the MMU/Cache Control Register. The function of this register and the mapping of the data cache into logical address space is the same as for the instruction cache.

This register is also written by software to pass logical addresses for DCache Flush (w/ Copyback) on Line Granularity, DCache Invalidation on Line Granularity, and Data MMU probes.

Data Supervisor Area Pointer Register (DSAP)

The Data Supervisor Area Pointer Register is a read/write MMU control register. It is written with the area pointer segment table address and control/protection information for data supervisor address space. The format of this register is the same as previously described for the area descriptor.

Data User Area Pointer Register (DUAP)

The Data User Area Pointer Register is a read/write MMU control register. It is written with the area pointer segment table address and control/protection information for data user address space. The format of this register is the same as previously described for the area descriptor.

Data MMU ATC Index Register (DIR)

The Data MMU ATC Index Register is a read/write MMU control register. It specifies which Data BATC entry is to be loaded with the contents of the Block ATC Write Port as well as which Data PATC entry is to be loaded with the contents of the Page ATC Write Port. It can also specify which data breakpoint register is to be loaded (where Breakpoint Register 0 = index of 32 and Breakpoint Register 1 = index of 33). The DIR also contains the User Attribute bits. A field description of this register is given below.

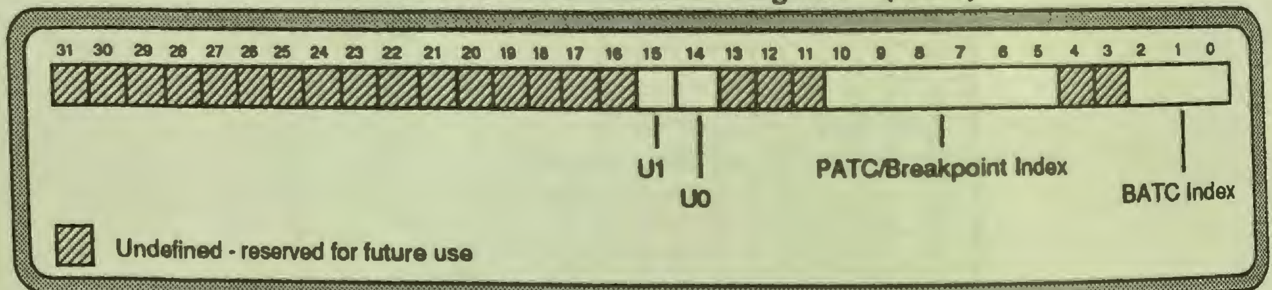


Figure 3.8.8.h - DMMU ATC Index Register

Field	Description
BATC Index	Next BATC Entry to be replaced.
PATC/Bkpt Index	Next PATC Entry to be replaced (0-31). (Note: Entry 32 specifies Breakpoint Register 0 Entry 33 specifies Breakpoint Register 1)
U1,U0	User Block Attribute bits.

Data MMU Block ATC Read/Write Port (DBP)

When a write (stcr) is performed with the Data MMU Block ATC Read/Write Port, the data is loaded into a Data BATC entry. Likewise, when a read (ldcr) is performed with the Data MMU Block ATC Read/Write Port, the data is read from a Data BATC entry. The specific BATC entry to be loaded is indicated by the Data MMU ATC Index Register (DIR). The Block size mask bits (variable from 512K to 64M) are fixed for all eight BATC entries and set via the DCTL. The User Block Attribute bits specified by the DIR are also used to load the BATC entry. The User Block Attribute bits are only accessible via the DIR. A description of the DBP register is shown in the figure below.

Data MMU Block ATC Read/Write Port (DBP)

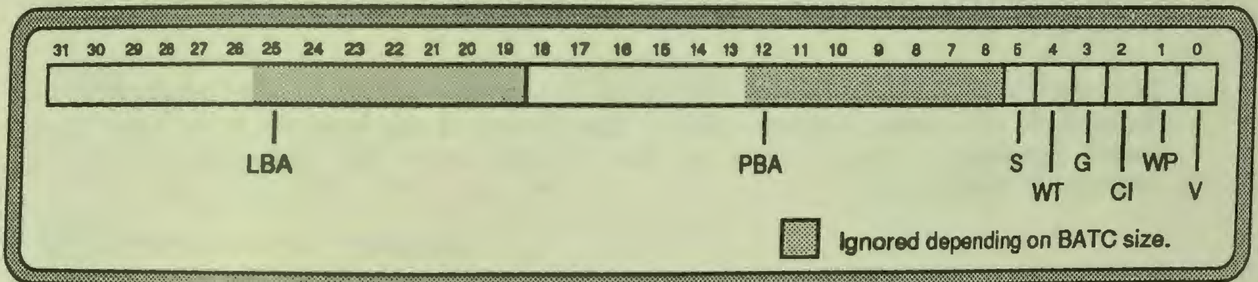


Figure 3.8.8.I - DMMU BATC Read/Write Port

Field	Description
V	Valid Bit
WP	Write Protection
CI	Cache Inhibit
G	Global
WT	Writethrough memory update policy
S	Supervisor/User Address Space
PBA	Physical Block Address. Use: a) bits (18-6) for 512K Blocks b) bits (18-7) for 1M Blocks c) bits (18-8) for 2M Blocks d) bits (18-9) for 4M Blocks e) bits (18-10) for 8M Blocks f) bits (18-11) for 16M Blocks g) bits (18-12) for 32M Blocks h) bits (18-13) for 64M Blocks
LBA	Logical Block Address. Use: a) bits (31-19) for 512K Blocks b) bits (31-20) for 1M Blocks c) bits (31-21) for 2M Blocks d) bits (31-22) for 4M Blocks e) bits (31-23) for 8M Blocks f) bits (31-24) for 16M Blocks g) bits (31-25) for 32M Blocks h) bits (31-26) for 64M Blocks

Data MMU Page ATC Read/Write Port - Upper (DPPU)

When a write (**stcr**) is performed with the Data MMU Page ATC Read/Write Port - Upper (DPPU), the data is temporarily buffered in preparation for loading into the most significant word of a Data PATC entry (or Breakpoint entry). The data is NOT loaded into the the most significant word until a **stcr** is performed for the corresponding PATC read/write port - lower (DPPL). (See section on Loading and Storing a PATC Entry.)

When a read (**ldcr**) is performed with the Data MMU Page ATC Read/Write Port - Upper (DPPU), the data is read from the most significant word of a Data PATC entry (or Breakpoint entry) and transferred into the general purpose register.

However in this case, a read (ldcr) from the Data MMU Page ATC Read/Write Port - Lower (DPPL) must *precede* this action. (See section on Loading and Storing a PATC Entry.)

The specific PATC entry (or Breakpoint entry) to be loaded is also indicated by the Data MMU ATC Index Register (DIR). The format of this register is the same as the most significant (upper) word of the PATC entry (or Breakpoint entry) as previously described.

Data MMU Page ATC Read/Write Port - Lower (DPPL)

When a write (stcr) is performed with the Data MMU Page ATC Read/Write Port - Lower (DPPL), the data is loaded into the least significant word of a Data PATC entry (or Breakpoint entry). (Note: at this time the data for the upper PATC word, which should have been set up previously, is now written). (See section on Loading and Storing a PATC Entry.)

When a read (ldcr) is performed with the Data MMU Page ATC Read/Write Port - Lower (DPPL), the data is read from the least significant word of the Data PATC entry (or Breakpoint entry). This action should precede the read (ldcr) of the Data MMU Page ATC Read/Write Port - Upper (DPPU). (See section on Loading and Storing a PATC Entry.)

The specific PATC entry (or Breakpoint entry) to be loaded is indicated by the Data MMU ATC Index Register (IIR). The format of this register is the same as the least significant (lower) word of the PATC entry (or Breakpoint entry) as previously described.

Data Access Status Register (DSR)

The Data Access Status Register is a read/write control register. This register provides Data MMU probe results (address translation status) and fault information for Data Access Exceptions including Data Breakpoint Exceptions. This register must also be cleared by software after any exceptions.

Data Access Status Register (DSR)

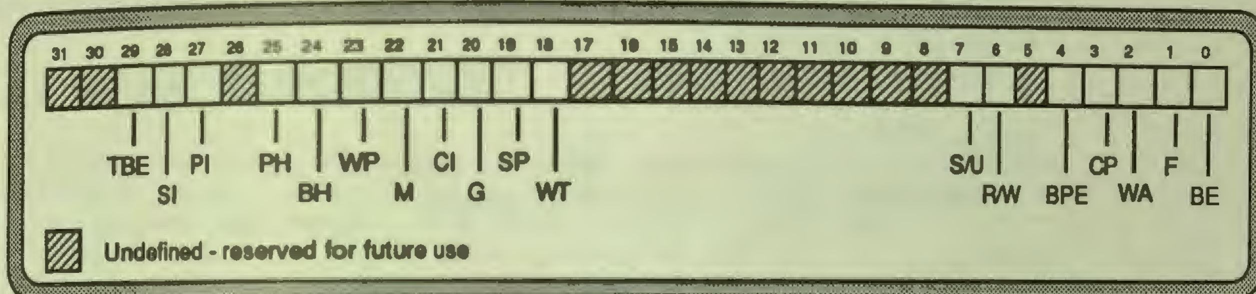


Figure 3.8.8.j - Data Access Status Register

Field	Description
BE	Bus Error - Indicates that a bus error occurred or that a cache flush was unsuccessful.
F	MMU Fault - Indicates that a fault occurred during the course of a normal (non-probe) address translation. MMU faults include: invalid segment descriptor, invalid page descriptor, tablewalk bus error, supervisor violation, and write protect violation.
WA	Write-Allocate Bus Error - Indicates that a bus error occurred during the line read operation of a write cache miss implementing the Write Allocation policy.
CP	Copyback Error - Indicates that an error occurred during a cache copyback initiated by a normal replacement of a dirty cache entry.
BPE	Breakpoint Exception occurred - The Logical Address where the break occurred is located in the Data Access Logical Address Register (DLAR).
R/W	Read/Write Status - Indicates the read/write status of the data access in error.
S/U	Supervisor/User Status - Indicates the supervisor/user status of the data access in error.
WT	Writethrough - 1 = Data at the probed address is cached with the writethrough memory update policy. 0 = Data at the probed address is cached with the copyback memory update policy.
SP	Supervisor Privilege - 1 = Probed address can only be accessed in the supervisor mode. 0 = Probed address can be accessed in the user or supervisor mode.

G	Global - 1 = One or more of the descriptors for the probed address are marked global. 0 = None of the descriptors for the probed address are marked global.
CI	Cache Inhibit - 1 = Data at the probed address cannot be cached. 0 = Data at the probed address can be cached.
M	Modified - 1 = Data at the probed address has been modified in memory or with respect to memory (cached data). 0 = Data at the probed address has not been modified.
WP	Write Protection - 1 = Probed address is write protected. 0 = Probed address can be read or written.
BH	BATC Hit - 1 = Probed address resulted in a BATC hit instead of a PATC hit or tablewalk. 0 = Probed address was found in the PATC or was generated by a tablewalk.
PH	PATC Hit - 1 = Probed address resulted in a PATC hit. 0 = Probed address was not found in the PATC.
PI	Page Descriptor Invalid - Indicates that an invalid page descriptor was encountered during a Probe generated tablewalk.
SI	Segment Descriptor Invalid - Indicates that an invalid segment descriptor was encountered during a Probe generated tablewalk.
TBE	Probe Tablewalk Bus Error - Indicates that an bus error was encountered during a Probe generated tablewalk.

Data Access Logical Address Register (DLAR)

The Data Access Logical Address Register is a read/write control register. In the case where a data ATC miss occurs and hardware tablewalks are disabled (via the DCTL), this register contains the logical address. It also contains the logical address of Data access exceptions.

Data Access Physical Address Register (DPAR)

The Data Access Physical Address Register is a read/write control register. This register contains the Physical address where the Data access exception occurred or the physical address of the tablewalk descriptor (segment or page) in error. The MC88110 updates this register whenever a bus exception occurs on a data access.

3.9 BUS INTERFACE

3.9.1 SIGNAL DESCRIPTION

The following section describes the input and output signals of the 88110 in their functional groups. Each signal is explained in a brief paragraph with reference to other sections that contain more detail about the signal and the related operations. The 88110 signals are shown below grouped by function.

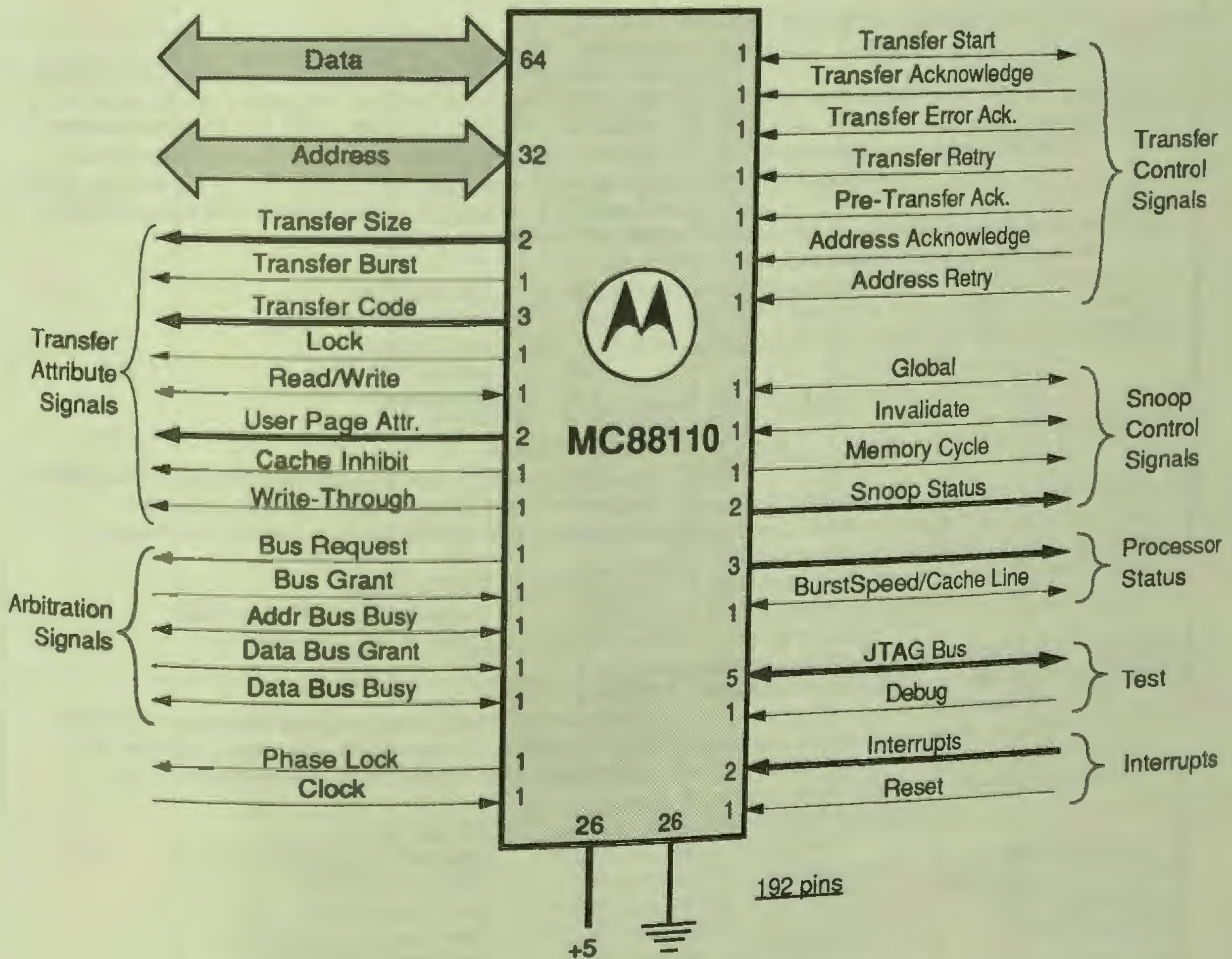


Figure 3.9.1 - 88110 Pinout

Motorola Confidential Proprietary - DO NOT COPY

Mnemonic	Pins	Active	Direction	Name and Description
Data Transfer Signals				
D_{63} - D_0	64	High	I/O	**Data Bus:** These bi-directional signals provide the data path for the current transaction. They are driven by the data bus master on writes and are inputs during reads.
A_{31} - A_0	32	High	I/O	**Address Bus:** These signals provide the address for the current bus transaction. They are driven when the master has address bus ownership and are inputs when snooping the bus for cache coherency purposes.
Transfer Attribute Signals				
$TSIZ_{1-0}$	2	High	Output	**Transfer Size:** The TSIZ pins indicate the size of the requested data transfer. Only transfers aligned on their respective size boundanes are supported. TSIZ may be used along with A_2-A_0 to determine which portion of the data bus contains write data or should contain valid read data. TSIZ is ignored by the memory system when TBST* is asserted. The timing for this signal coincides with addresses. $TSIZ_{1-0}$ — Transfer Size 0 0 — double-word (64 bits) 0 1 — word (32 bits) 1 0 — half-word (16 bits) 1 1 — byte (8 bits)
TBST*	1	Low	Output	**Transfer Burst:** The 88110 uses this signal to indicate the size of the initiated transaction. The timing for this signal coincides with addresses. The Transfer Burst is encoded as follows: TBST* — Transfer Burst 1 — Double-word or less (see TSIZ) 0 — 8 Word Burst (32 bytes)
TC_{2-0}	3	High	Output	**Transfer Code:** These signals provide supplemental information about the current address. The timing for this signal coincides with addresses. The Transfer Code is encoded as follows. TC_{2-0} — Transfer Code 0 0 0 — Data MMU Tablewalk 0 0 1 — User Data Access 0 1 0 — User Code Access 0 1 1 — Data Cache Copyback 1 0 0 — Code MMU Tablewalk 1 0 1 — Supervisor Data Access 1 1 0 — Supervisor Code Access 1 1 1 — Snoop Copyback

Mnemonic	Pins	Active	Direction	Name and Description
UPA*$_{1-0}$	2	Low	Output	**User Page Attributes:** These pins reflect the contents of the UPA bits in the ATC entry used to map the current address. (Note sense of UPA bits in ATC is opposite that of the pins.) During tablewalks and identity translations the signals reflect the values in the appropriate Area Descriptor. On copybacks these signals are driven high. The timing of these signals coincides with addresses.
R/W*	1	High	Output	**Read/Write*:** This signal is driven to indicate the direction of the data transfer. When asserted it indicates that a data "read" is in progress. The timing for this signal coincides with addresses.
LK*	1	Low	Output	**Lock*:** The 88110 drives this signal to indicate that an access is part of an atomic data access sequence. It is asserted for XMEM operations and during the Used/Modified bit update sequence of tablewalks. The timing for this signal coincides with addresses.
CI*	1	Low	Output	**Cache Inhibit*:** This pin reflects the content of the CI bit in the ATC entry being used to map the current address. CI* is asserted to indicate that the address/data will not be cached. The timing for this signal coincides with addresses.
WT*	1	Low	Output	**Write-through*:** This signal indicates that the current transaction is "writing though" the on-chip data cache (if the access is cache enabled). This pin is asserted if the MMU page is marked "write through" or if a bus write is the result of a store-through operation. The timing for this signal coincides with addresses.
Transfer Control Signals				
TS*	1	Low	I/O	**Transfer Start*:** The 88110 asserts this signal to indicate that a transaction has begun and the driven address is valid. This signal is asserted for one cycle and then negated. This signal is an output while the 88110 is address bus master. Snooping 88110's use this signal as an input to qualify valid addresses.
PTA*	1	Low	Input	**Pre-Transfer Acknowledge*:** The memory system asserts PTA* to indicate that the initial (or only) TA* assertion of transaction <u>may</u> follow on the next rising clock edge. The window between TS* asserted and PTA* asserted allows loads/stores (data cache hits only) to continue to access the data cache even though a bus transaction is in progress. Simple memory implementations may ground this input and disable cache/bus decoupling.
TA*	1	Low	Input	**Transfer Acknowledge*:** When the 88110 is the bus master, this signal is asserted by the current slave on every clock that new data is valid during a bus read operation, or accepted during a bus write operation. Reference section on transfer termination for a more detailed description.
TEA*	1	Low	Input	**Transfer Error Acknowledge*:** TEA* is asserted by the slave to indicate a bus error has occurred.
TRTRY*	1	Low	Input	**Transfer Retry*:** TRTRY* is asserted by the slave to request that the transaction be terminated and re-initiated.

Motorola Confidential Proprietary - DO NOT COPY

Mnemonic	Pins	Active	Direction	Name and Description
AACK*	1	Low	Input	**Address Acknowledge*:** The 88110 monitors AACK* while it is the bus master generating a valid address. When AACK* is asserted, the 88110 stops driving the address bus and negates Address Bus Busy (ABB*). AACK* is sampled beginning with the rising clock edge following the address bus master's assertion of TS*. Upon receiving a legitimate AACK*, the 88110 may immediately re-arbitrate (or be parked) to regain the address bus, and then forward a new address onto the bus.
Snoop Control Signals				
INV*	1	Low	I/O	**Invalidate*:** When asserted, the INV* pin indicates that the current transaction should be treated as a write for coherency purposes. The timing for this signal coincides with addresses.
MC*	1	Low	Output	**Memory Cycle*:** When asserted, the MC* indicates that an actual data transfer is in progress (as opposed to an Invalidate transaction). The timing for this signal coincides with the address timing. R/W* — INV* MC* — Transfer Write — N N — reserved Write — N A — reserved Write — A N — Invalidate (no data transferred) Write — A A — write Read — N N — reserved Read — N A — Read Read — A N — reserved Read — A A — Read with intent to modify A = Asserted, N = Negated
GBL*	1	Low	I/O	**Global*:** Address bus masters assert this pin to indicate that the transaction in progress is marked as "global" by the MMU. When the cpu is not address bus master, the global pin is an input. When the global pin is asserted, a cpu will enable snooping on the current address. The timing of this signal coincides with addresses.
ARTRY*	1	Low	Input	**Address Retry*:** When ARTRY* is asserted, the address bus master terminates the transaction and will re-initiate the transaction at a later time. The address bus master qualifies ARTRY* with AACK*, or the rising clock edge immediately following the assertion of AACK* (AACK* must remain asserted), or TA* or TRTRY*, before terminating the transaction. When ARTRY* is asserted, a bystander 88110 removes its bus request, ignores bus grant, and suppresses any subsequent bus request. All potential masters (as well as the retried master) will be blocked from requesting the bus until ARTRY* has been negated.

Mnemonic	Pins	Active	Direction	Name and Description
$SSTAT^*_{1-0}$	1	Low	Output	**Snoop Status*** Indicates status of snooping CPU. $SSTAT^*_1$ may be tied directly to ARTRY* to generate snoop and collision retries. $\underline{SSTAT^*_{1-0}}$ — <u>Status</u> Z Z — No Collision, No Snoop Hit Z A — Snoop Hit Shared A Z — Pipeline Collision A A — Snoop Hit Modified Z = High Impedance , A = asserted
Arbitration Signals				
BR*	1	Low	Output	**Bus Request*:** The 88110 asserts this signal when it requests bus ownership and continues to assert it until it has received a qualified bus grant (BG* asserted AND ABB* negated).
BG*	1	Low	Input	**Bus Grant*:** This signal is used by the external bus arbiter to grant bus ownership to the 88110 in response to a bus request. The 88110 (and other bus masters) must qualify this bus grant with a non-busy Address bus (ABB* negated) before assuming address bus ownership. The external arbiter may "park" the '110 on the bus by asserting Bus Grant after Bus Request has been negated.
ABB*	1	Low	I/O	**Address Bus Busy*:** This signal is asserted by the current Address master. Potential address masters use this input to qualify Bus Grant.
DBG*	1	Low	Input	**Data Bus Grant*:** The memory system asserts DBG* to grant data bus ownership to the 88110. DBG* asserted is qualified with DBB* negated before data bus ownership is assumed.
DBB*	1	Low	I/O	**Data Bus Busy*:** This signal is an output when the 88110 is the data bus master and as an input when the 88110 is asserting or has asserted TS*. DBB* is sampled by the CPU to qualify DBG* before assuming data bus ownership. When the qualified data bus grant is received, DBB* is asserted for the duration of the data transfer.

Mnemonic	Pins	Active	Direction	Name and Description
Processor Status Pins				
$PSTAT_{2-0}$	3	High	Output	**Processor Status Pins:** These signals are not yet defined. $PSTAT_{2-0}$ — Status 000 — ? 001 — ? 010 — ? 011 — ? 100 — ? 101 — ? 110 — ? 111 — ?
CL	1	High	I/O	**Cache Line / Burst Speed:** Burst Speed is sampled when RESET* is negated. If the pin is sensed low, burst transfers will operate in the half-speed mode (the 88110 internally forces one WAIT state on the clock immediately following TA* asserted). If the pin is sensed high burst transfers will handshake at the full CPU clock rate. The pin is internally pulled-up so that if half-speed operation is not desired the pin may be left unconnected. Outside of reset, this signal is an output. The signal itself indicates which line in the cache is selected. It can be used with other signals (R/W*, INV*, MC*, LK*, TBST*, TC*, A11-A5) to determine the next state of a particular Instruction or Data cache line. The timing for this signal coincides with addresses. CL — Cache Line 0 — Line 0 1 — Line 1
Interrupt Signals				
NMI*	1	Low	Input	**Non-Maskable Interrupt*:** Non-Maskable Interrupt. Interrupt is sampled by the CPU at the rising edge of each bus clock. The interrupt signal can be completely asynchronous but will only be detected on a given clock if setup and hold times are satisfied. When a valid interrupt is detected the 88110 will unconditionally trap through the non-maskable interrupt vector.
INT*	1	Low	Input	**Interrupt*:** Maskable Interrupt. Interrupt is sampled by the CPU at the rising edge of each bus clock. The interrupt signal can be completely asynchronous but will only be detected on a given clock if setup and hold times are satisfied. When a valid interrupt is detected and the interrupt is enabled by the Interrupt Enable Bit in the PSR, the 88110 will trap through the maskable interrupt vector.
RST*	1	Low	Input	**Reset*:** Asserting RST* forces the 88110 to cease current operations. Negating RST* begins execution at the RESET vector location with all the caches, MMU's and breakpoints disabled.

Mnemonic	Pins	Active	Direction	Name and Description
Clock Signals				
CLK	1	↑	Input	**Clock:** The leading edge of CLK is used by the 88110 as the internal and external timing reference.
PLOCK*	1	Low	Output	**Phase Lock*:** When asserted, this signal indicates that the internal PLL circuitry is locked to the incoming CLK.
Test Pins				
TRST*	1	Low	Input	**JTAG Test Reset*:** Assertion of this signal causes asynchronous initialization of the internal JTAG TAP controller. This signal conforms to the "IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture."
TMS	1	High	Input	**JTAG Test Mode Select:** TMS is decoded by the internal JTAG TAP controller to distinguish the primary operations of the test support circuitry. This signal conforms to the "IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture."
TCK	1	↑↓	Input	**JTAG Test Clock:** This signal clocks the internal boundary scan test support circuitry. This signal conforms to the "IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture."
TDI	1	High	Input	**JTAG Test Data Input:** The value present on this pin is clocked into the selected JTAG test instruction or data register on the rising edge of TCK. This signal conforms to the "IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture."
TDO	1	High	Output	**JTAG Test Data Output:** The contents of the selected internal instruction or data test register are shifted out onto this pin on the falling edge of TCK. This signal conforms to the "IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture."
DBUG*	1	Low	Input	**Debug*:** When asserted this signal disables all caches, MMU's and breakpoints.
Power Supply Pins				
V_{dd}	26			**+5V Power**
GND	26			**Ground**
	192			Total Pins

Table 3.9.1 - Pin Descriptions

3.9.2 PACKAGING

The 88110 is a high performance part requiring a high performance package. The 88110 will be packaged using Tape Automated Bonding (TAB). TAB packaging allows high pin count devices to be delivered in a small footprint with extremely low lead inductances. Production parts will be in a 240 pin TAB package with 15 mil lead spacing. The tape used will be a 35mm three layer (polyimid/adhesive/copper) laminate gang bonded to bumps on the chip's I/O pads. Chips are passivated and glob topped with silicon epoxy to protect them during assembly in the system. Packaged parts may be assembled on the PCB by a pulsed thermode solder process.

Cooling could be accomplished by mounting the chip back-side-down onto a stud which goes through the substrate to a heat sink on the reverse side of the board. The chip would be attached to the stud using a thin insulating adhesive. If the maximum ambient operating temperature is to be 55°C, a thermal resistance ($\O_{JA}$) of 9°C/Watt between the chip and the air should be provided to guarantee high reliability at full operating frequency.

The 88110 will also be offered in a 240 pin cavity-down ceramic pin grid array (PGA) package with epoxy attached heat sink.

3.9.3 BUS OPERATION

The 88110 bus interface uses separate address and data bus ports. The address bus is 32-bits wide and the data bus is 64-bits wide. The interface is synchronous with all operations referenced to the rising clock edge. Bus signals all operate at TTL logic levels using controlled slew rate drivers. The bus protocol is a multiple master burst transfer protocol with centralized arbitration. Bus parking and fairness may be supported by the external arbiter to minimize bus arbitration time. The address bus can be disconnected from the data bus for implementation of pipelined or split-response interfaces. A data transfer can handshake at either full or one-half (or less) the CPU clock speed.

3.9.3.1 Arbitration

The 88110's Arbitration signals consists of address arbitration signals (BR*, BG*, ABB*) and data arbitration signals (DBG*, DBB*). Potential bus masters arbitrate directly for address bus ownership using BR*, BG*, and ABB*. Data bus request is implicit based on Transfer Start (TS*). Unless abnormally terminated (see ARTRY*), every data tenure requires the memory system to complete the transaction by granting data bus ownership.

Arbitration takes place when a potential master (or masters) asserts Bus Request (BR*). Arbitration for bus ownership is performed by external arbitration logic. The arbitration for each tenure can occur within a single bus clock. The arbitration logic can be simple combinational logic which provides a single prioritized output (BG^*_{0-N}) when one or more of its inputs (BR^*_{0-N}) asserted. The arbitration logic must guarantee only one BG* is asserted during the specified timing window.

When the 88110 needs to access memory, it asserts its Bus Request pin (BR*) and waits for the arbitration logic to return a Bus Grant (BG* asserted). The 88110 qualifies BG* asserted with a non-busy address bus (ABB* negated). The 88110 then asserts ABB* and becomes the address bus master. The 88110 removes its Bus Request (BR*) after receiving a qualified Bus Grant (BG*). Once address bus mastership is established the 88110 drives the address onto the address bus and asserts TS* indicating that the address is valid.

The arbiter may choose to park the 88110 on the bus by continuing to assert BG*. A subsequent internal bus request will bypass the arbitration sequence and will cause the CPU to become master-elect. The 88110 will park when it receives a qualified bus grant, even if a bus request was not asserted. DBB* will always negate after the transaction is complete. The data bus cannot be parked, even if DBG* is continually asserted.

The 88110 arbitrates uniquely for each transaction. The 88110 does not automatically hold bus ownership in order to perform certain transactions within a single bus tenure. However, the arbiter may selectively park the 88110 in order to force multiple transactions within a tenure. For example, the arbiter can use the lock pin (LK*) to group the read and write portions of an XMEM operation into a single indivisible bus tenure. In addtion, the 88110 continues to assert BR* between the read and write portions of a locked seqeuence.

Note that the arbitration cycle provides a "dead" cycle which eases the bus master turnaround on the address bus. Also note that no turnaround is necessary when parked on the bus, thus saving one clock for parked transactions.

3.9.3.2 Handshaking

After becoming the address bus master, the CPU initiates a transaction by asserting the Transfer Start (TS*) signal. The CPU is then ready to transfer the data corresponding to the address, size, and direction indicated by the Address (A31-A0), the size (TBST, TSIZ1-0) and the Read/Write (R/W*, INV*, MC*) pins. The CPU will initiate either a read, write, or invalidate transaction.

Reads and writes are either single beat transfers (64 bits or less) or 4-beat burst-transfers of 2 words per beat (8 words total). Burst reads begin with the address of the critical double-word. Burst writes for copybacks due to cache line replacement also begin with the address of the critical double-word. Burst writes for snoop hit copybacks begin with the address of the double-word containing the hit address. Burst writes for cache line flushes begin with the first double-word in the line.

Invalidates are address-only transfers with no data exchange, although write data is driven by the CPU. Invalidates are treated as single beat writes for handshaking purposes.

Data transfer begins when the CPU becomes data bus master. While asserting TS*, the CPU begins sampling Data Bus Grant (DBG*) and Data Bus Busy (DBB*). When the CPU receives a qualified data bus grant (DBG* asserted, DBB* negated) the CPU will assert DBB* and data transfer may begin. The CPU will not drive write data onto the bus until it receives the qualified DBG*, nor will it recognize TA*, TEA*, or TRTRY*. Non-pipelined memory systems can simply ground DBG*'s for all CPU's as both address and data bus hand-off will be controlled by ABB*.

The slave responds with an acknowledgement as it completes the transfer of the first datum. The slave asserts Transfer Acknowledge (TA*) to indicate that data is valid on a read or that data has been accepted on a write.

For burst transfers, the slave must continue to sink or supply data with TA* until the burst is complete. The slave must provide the data for burst reads starting with the double-word address as driven by the master. The 88110 will increment (and wrap-around to the beginning of the cache line if necessary) the double-word address (A4-A3) after each assertion of TA* (non-pipelined operation only). The slave may temporarily interrupt the stream by negating TA* on any double-word of the burst (WAIT state).

3.9.3.3 Burst Speed

Some systems will want to take advantage of the 88110's high level of integration but cannot afford the expense of a very high speed memory system. Therefore, an operating mode is provided which allows the data transfer and handshake to operate at one-half the CPU clock rate. In this mode timing requirements are eased considerably thus simplifying the design and allowing a less expensive interface.

The operating mode is determined statically when the 88110 is coming out of reset. The CacheLine/BurstSpeed pin is used to select the burst transfer rate (full CPU clock or half CPU clock) during reset.

If the BurstSpeed pin was initialized in the half-speed mode, the CPU effectively inserts a WAIT state after each TA* asserted during a burst transfer. This forces a WAIT state following each data beat and relieves the memory system from a full clock speed handshake. Immediately following each WAIT cycle, the sampling of TA* and data resumes with one clock resolution. Burst speed has no effect on single beat transfers.

3.9.3.4 Data Cache/Bus Decoupling

In some cases the data cache can continue to service load/store requests (ATC hits/data cache hits only) while bus operations are taking place. The memory system can create a decoupling window by giving the CPU early notice of the first TA* via the PTA* signal. When TS* is asserted, decoupled load/stores may proceed on any clock following the negation of PTA*. PTA* must be asserted at least one clock before the first TA* of a transaction to insure proper operation. Once asserted, this signal must remain asserted for the duration of the transaction. Memory systems not taking advantage of decoupling may simply hold this signal asserted. Tablewalks, copybacks, flushes, and XMEM transactions are not eligible for decoupling and the PTA* signal is ignored during these transactions.

3.9.3.5 Pipelined Addresses

The minimum address cycle time for any given transaction is 2 clocks. Whenever the number of data transfer clocks exceeds the minimum number of address clocks, the potential for pipelining exists. By asserting the Address Acknowledge (AACK*) pin, the slave can signal the CPU that it no longer needs the current address. AACK* is recognized on the rising edge clock following the assertion of TS*.

After receiving an AACK*, the address bus is freed up and another master may initiate a transaction. (A single 88110 will not pipeline out a second address until the data transfer for the previous one is complete.) It is possible to run an out-of-order protocol on a multi-master bus by using DBG*, however, data must always be returned to a given 88110 in the order it was requested.

If AACK* is received by a parked address master (BG* asserted), the CPU negates ABB* and releases the address bus, unless a new transaction will start on the next clock, or a locked read is in progress. A subsequent internal bus request will bypass the arbitration sequence and will cause the CPU to become master-elect.

If AACK* is received by an unparked address master (BG* negated), the CPU will release the address bus (negate ABB*) and allow another master to obtain address bus ownership.

When implementing a multi-master, 1-level pipelined bus, the memory system can rely on 88110's to control data bus hand-off via DBB*. The memory system must ensure that no more than two transactions are outstanding. The memory system may do this by grounding all DBG*'s and never asserting AACK* until DBB* is negated.

The pipelined master asserts TS* to indicate that address is valid and then waits for the current data transfer to complete before it assumes full bus ownership and transfers its own data.

Multi-level pipelined busses or split response busses must take responsibility for data hand-off via DBG*.

3.9.3.6 Snoops/Collisions and Address Retry

When a snooping CPU hits on a modified entry, the CPU will assert the SSTAT*$_1$ signal. SSTAT*$_1$ output may be directly or indirectly coupled to each CPU's ARTRY* input. A qualified ARTRY* is one which is asserted coincident with, or 1 clock immediately following initial assertion of AACK* (AACK* must remain asserted). If AACK* is not asserted ARTRY is qualified with the initial assertion of TA* or TRTRY*. Upon receiving a qualified ARTRY*, the current address bus master terminates the transaction, releases the address bus, and re-initiates the transaction. The memory system must ensure that for global accesses the initial TA* is coincident with or is canceled after ARTRY* has been asserted and qualified. Transferring the first double-word to the cpu before the arrival of a qualified ARTRY* may result in stale data being read or good data being overwritten. The snoop response time for the 88110 is fixed. SSTAT*$_{1-0}$ is valid the second rising clock edge after the external address has been sampled (TS* asserted and clock rise).

The snooping CPU (with a snoop hit on a modified entry) will assert its bus request coincidentally with SSTAT*$_1$. SSTAT*$_1$ (and consequently the ARTRY* pin) is negated after the snooping CPU sees ABB* negated.

When ARTRY* is asserted, other potential masters will remove their bus requests and ignore any bus grants. Their bus requests (including the bus request of the original master) will remain blocked until ARTRY* is negated. This blocking protocol allows the snooper who asserted the ARTRY* a window of opportunity to receive uncontested ownership of the bus. The size of this window can be controlled by the memory system via the AACK* signal.

The SSTAT*$_1$ signal is also asserted by the CPU to prevent pipelined address collisions. The possibility of collision exists after a master's address has been pipelined (AACK* asserted) but before the data transfer has completed. Accesses to the same line address by other masters could cause coherency failures since the tag is not loaded until the data transfer is complete.

To prevent coherency failure, each CPU maintains a pipelined address latch for collision detection. This latch is loaded by the CPU when it receives an AACK* in response to a global data address and is cleared when the data transfer is complete (code addresses are never loaded). If another master initiates an access to the same global line address, a pipeline collision occurs. The CPU with the pending data transfer will assert ARTRY* via SSTAT*$_1$, deferring the offending master until the CPU has loaded its tags. Since snoop copybacks are not global, the re-initiated transaction which might immediately follow a snoop copyback will not be flagged as a collision.

The rules for assertion and negation of collision ARTRY* apply as described above in the case of a snoop hit. The collision retry can be thought of as a superset of the

snoop retry protocol. The collision latch is really another snoop tag that forces an address retry on all hits, clean or dirty.

3.9.3.7 Termination

This section explains the bus transfer termination for the 88110. The possible termination cases are defined in the following table:

DBB*	TA*	TEA*	TRTRY*	ARTRY*	AACK*	Termination
A	A	N	N	N	x	Normal
A	x	A	x	x	x	Error
A	x	N	A	N	x	Transfer Retry
A	x	N	x	A	A, A_1	Address Retry

A = Asserted
N = Negated
x = Don't Care
A_1 = One clock (rising edge) following assertion

Table 3.9.3.7 - Termination Cases

TA*, TEA*, and TRTRY* are ignored by the processor until the clock edge following a qualified data bus grant. After a transfer has been terminated, TA*, TEA*, and TRTRY* are again ignored. For both the pipelined and non-pipelined cases, the master elect as well as all other competitors for the bus will ignore bus error and retry terminations. For the pipelined case, where the master elect has its address on the address bus, the master elect will acquire control of the bus after the abnormal termination is completed.

For termination purposes, ARTRY* is sampled starting with the rising clock edge following TS* asserted and ending with the rising clock edge following AACK* asserted (AACK* must be held asserted for 2 clocks if ARTRY* is 1 clock late w.r.t. AACK*.) All ARTRY* assertions outside this window not terminate a transaction but will invoke the bus request blocking protocol. ARTRY* asserted after the first data beat will not result in transaction retry.

3.9.3.7.1 Normal Transfer Acknowledge

TA* is asserted by the responding slave to complete the data transfer. TEA* and TRTRY* remain negated during the transfer.

3.9.3.7.2 Terminate with Bus Error

TEA* is asserted by the responding slave to terminate the transaction with an error. At this point the master will release ownership of the data bus. Address bus ownership is released only if the CPU is not master-elect (the driven address is in step with the returning data). If the error is signalled before or coincident with the first data beat of a burst, the cache line will be loaded as invalid and the cpu will take an exception. For cache line fills, fetches/loads maybe satisfied as the cache line is

being filled. If a bus error occurs on a forwarded double word the cache line is loaded as invalid and the cpu will take an access exception.

3.9.3.7.3 Terminate with Transfer Retry

TRTRY* is asserted by the responding slave to terminate the data with an indication to retry. A retry termination can only occur on the first double word of a burst transfer. If TRTRY* is asserted anytime after the first double word of a burst, the transaction and tenure will be terminated, the cache line will be loaded as invalid, and the bus interface will *not* re-initiate the transaction.

3.9.3.7.4 Terminate with Address Retry

A qualified ARTRY* will terminate the transaction independent of data bus ownership. If ARTRY* occurs before the data bus is acquired, the data tenure is cancelled and DBB* will not be asserted by the CPU. If a qualified ARTRY* occurs during data bus ownership, the data transfer will be terminated. The address master will re-initiate the transaction after ARTRY* is negated.

3.9.4 BUS TIMING EXAMPLES

3.9.4.1 Single Beat Transfers

3.9.4.1.1 Fastest Single Beat Reads

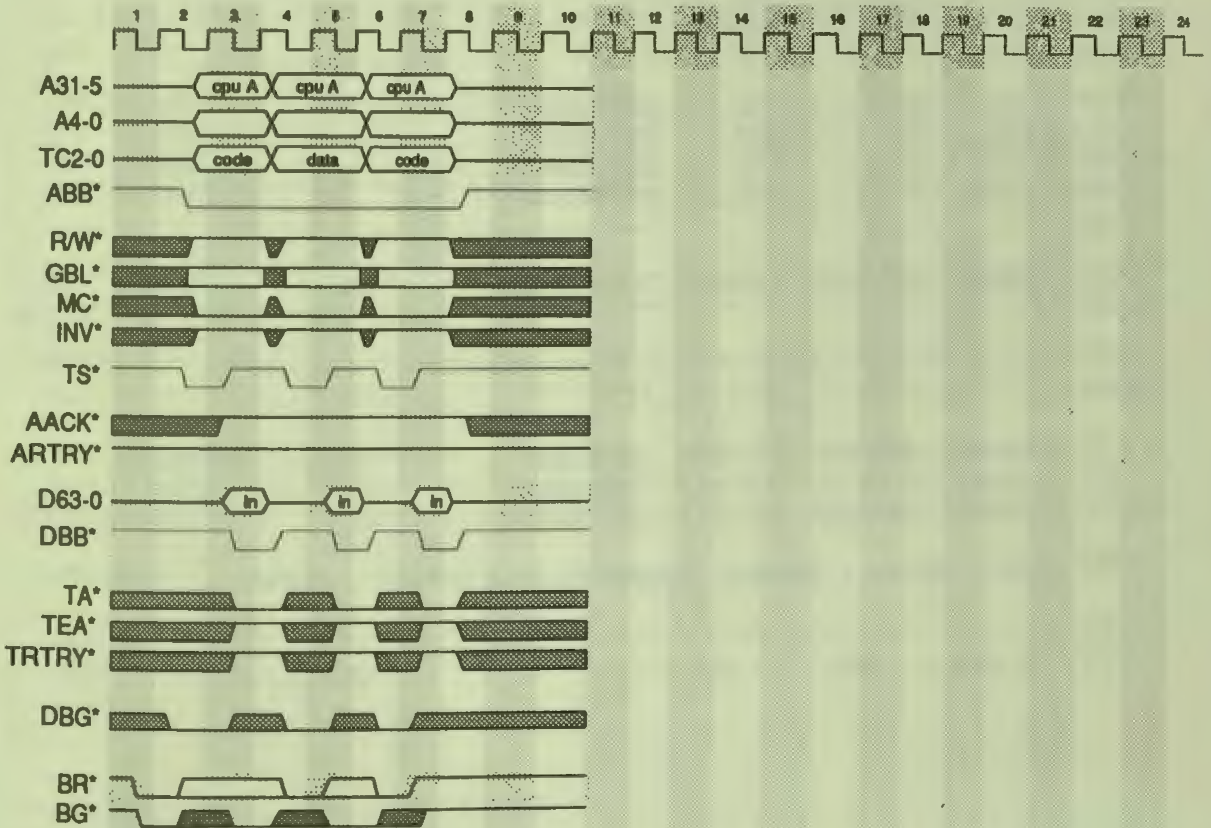


Figure 3.9.4.1.1 - Fast Single Beat Reads

3.9.4.1.2 Fastest Single Beat Writes

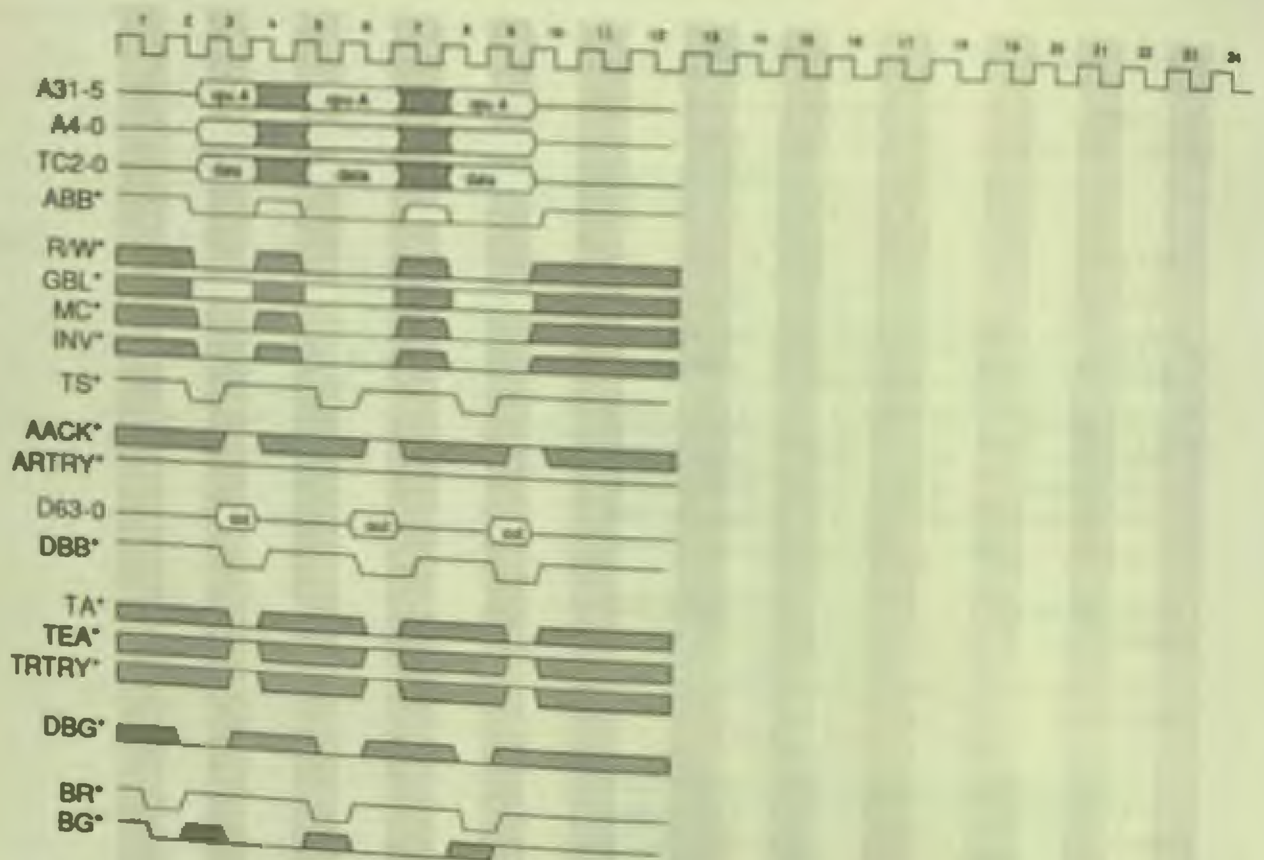


Figure 3.9.4.1.2 - Fast Single Beat Writes

3.9.4.1.3 Single Beat Reads with Waits

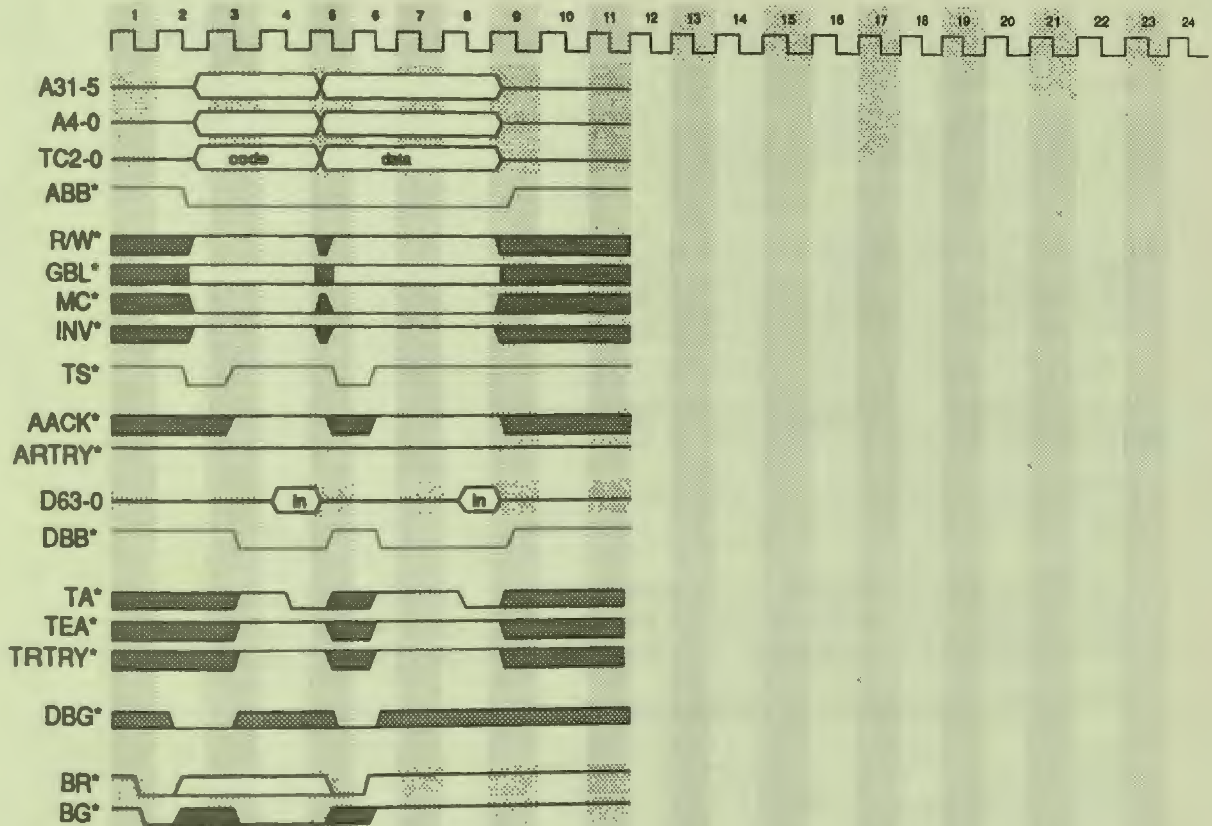


Figure 3.9.4.1.3 - Single Beat Reads with Waits

3.9.4.1.4 Single Beat Writes with Waits

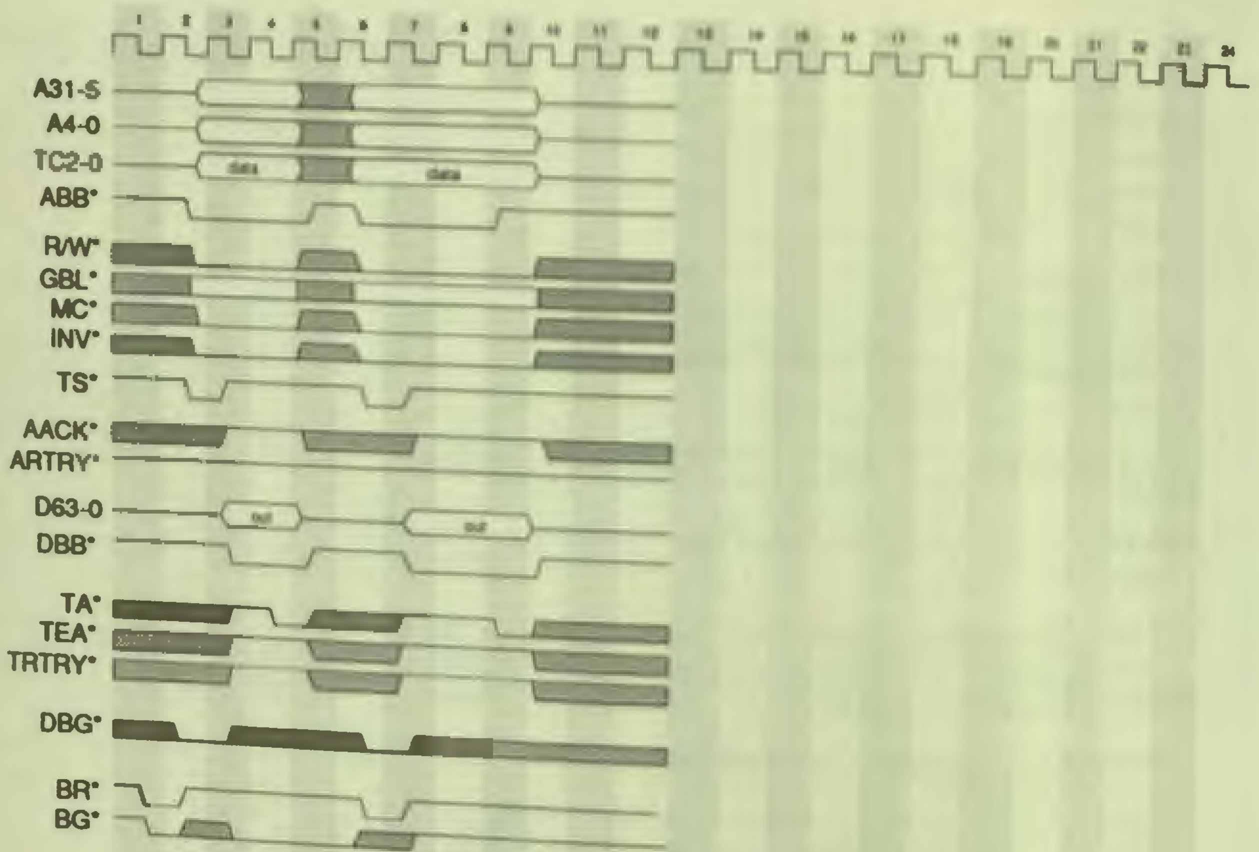


Figure 3.9.4.1.4 - Single Beat Writes with Waits

3.9.4.1.5 Single Beat Reads with Data Bus Grant

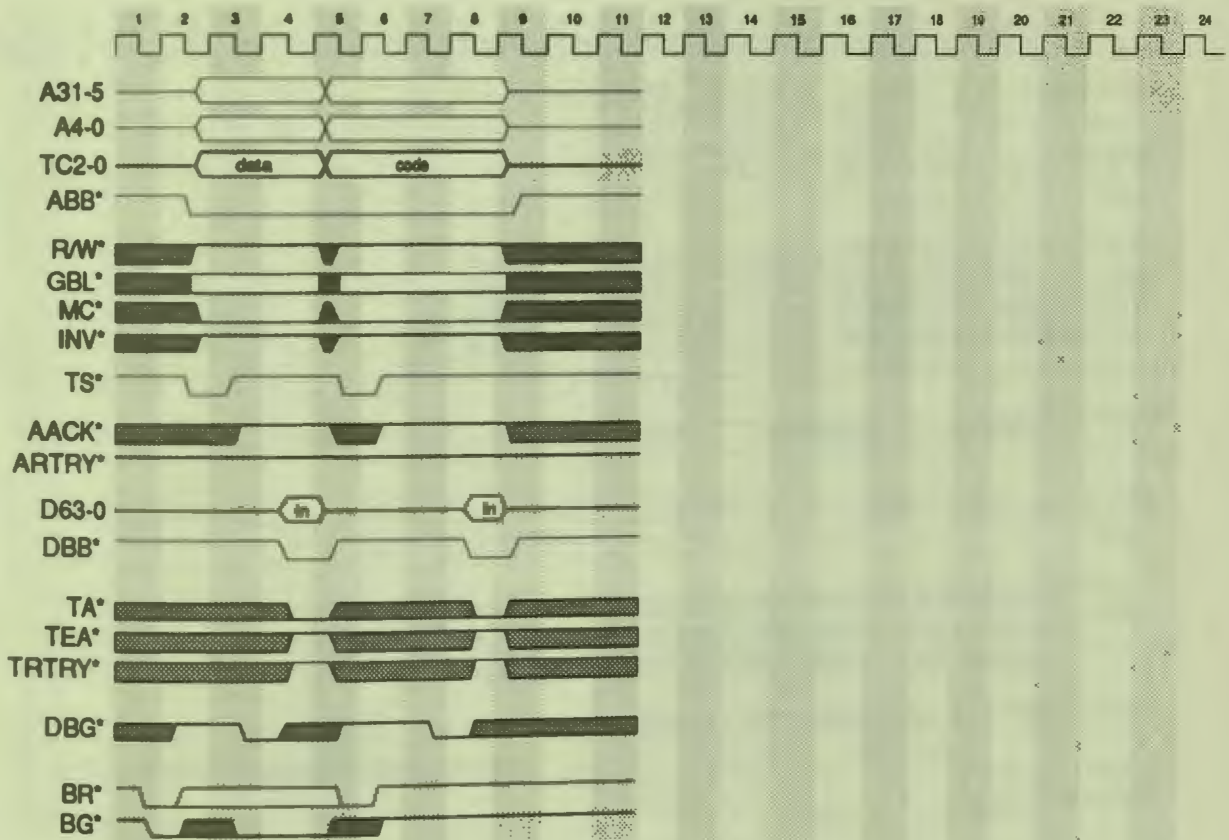


Figure 3.9.4.1.5 - Single Beat Reads with Data Bus Grant

3.9.4.1.6 Single Beat Writes with Data Bus Grant

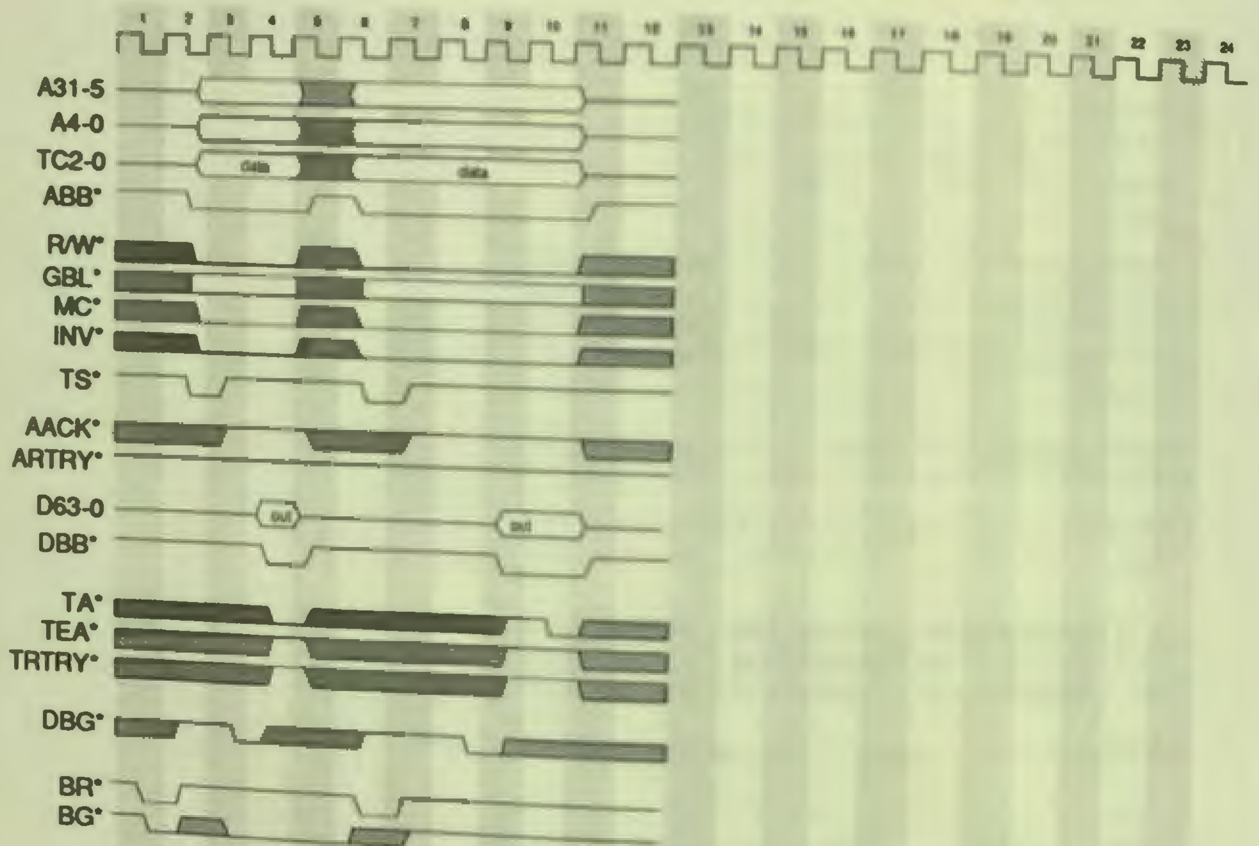


Figure 3.9.4.1.6 - Single Beat Writes with Data Bus Grant

3.9.4.1.7 Single Beat Read/Write/Read Sequences

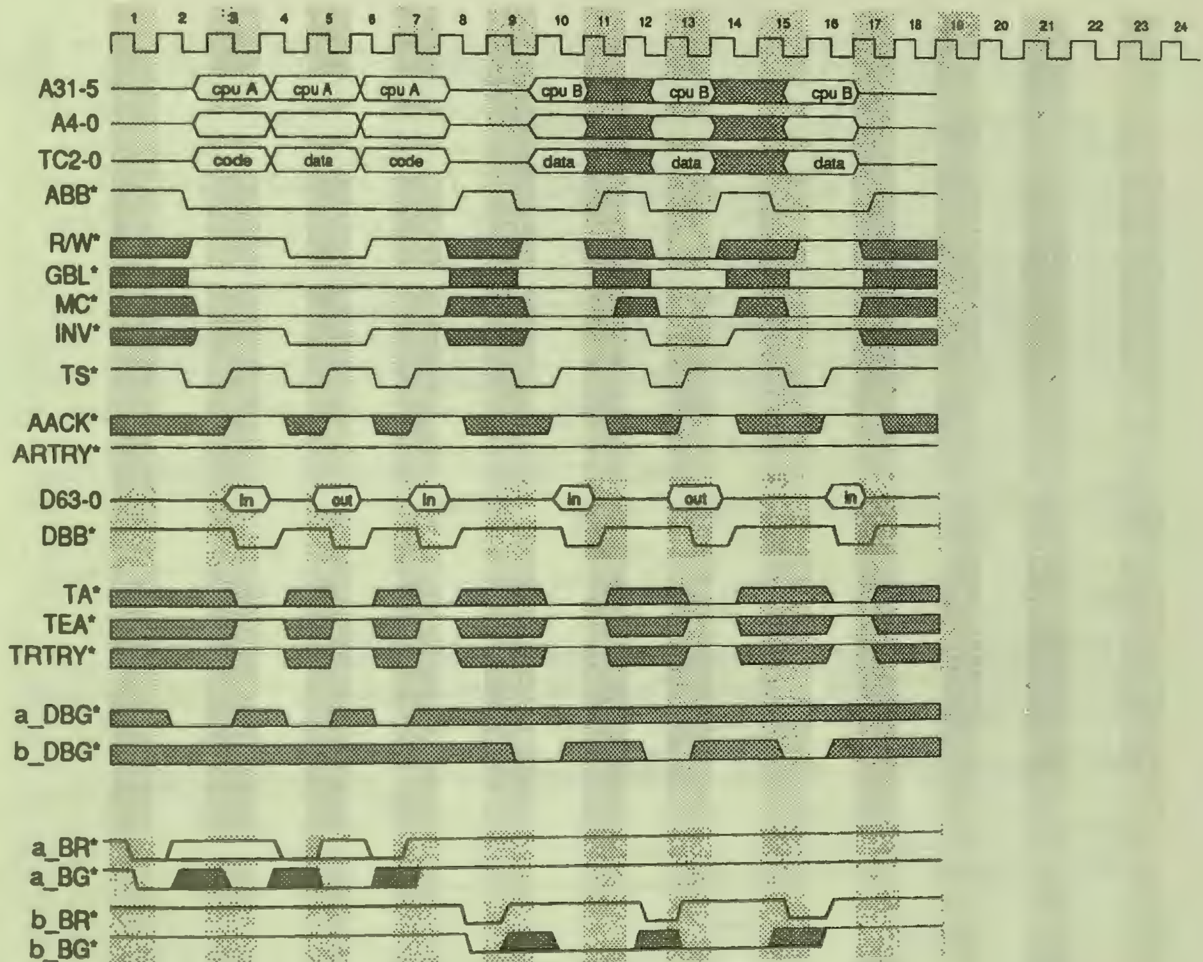


Figure 3.9.4.1.7 - Single Beat Read/Write/Read Sequences

3.9.4.1.8 Non-pipelined xmem

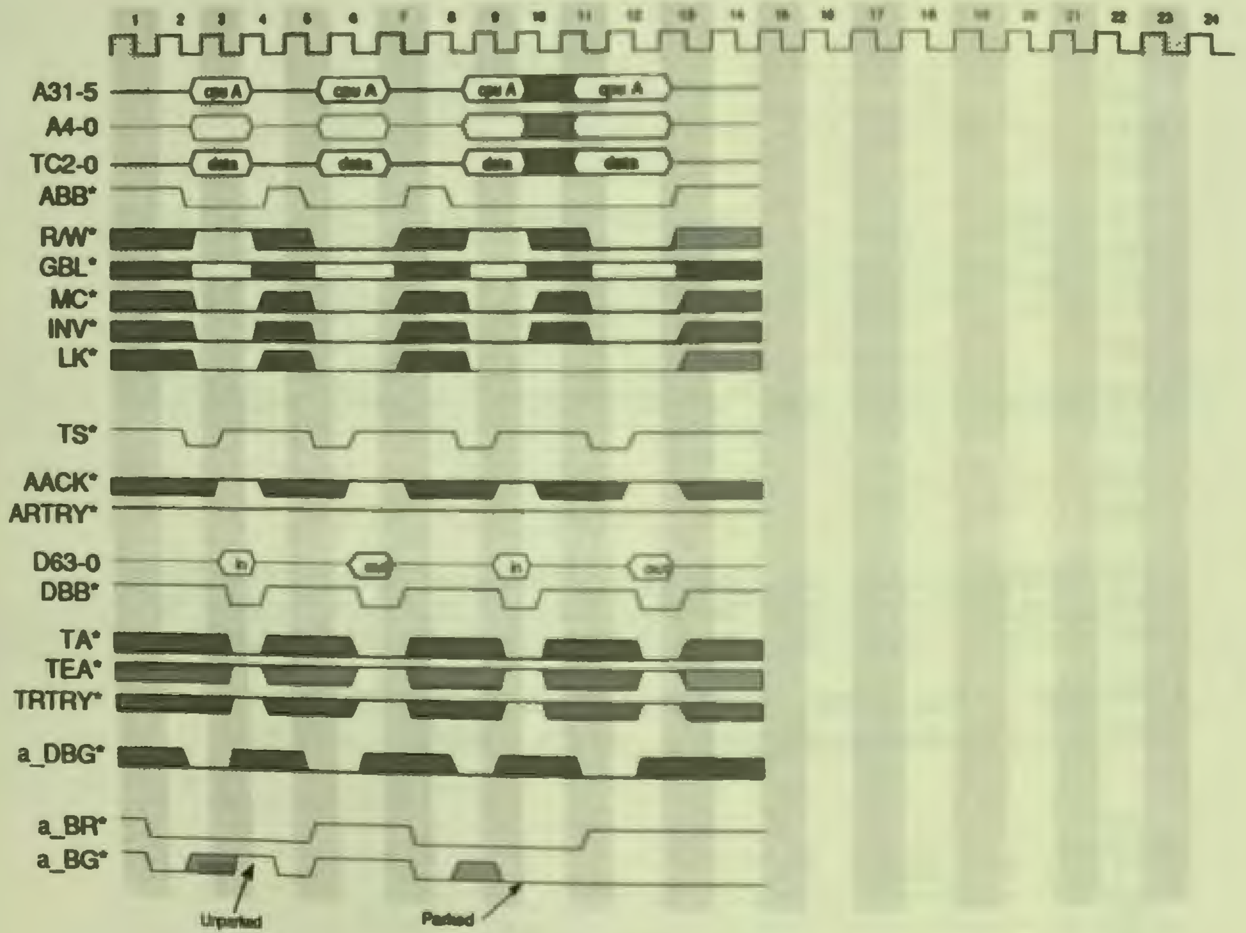


Figure 3.9.4.1.8 - Non-pipelined xmem

3.9.4.1.9 Invalidation vs. Read vs. Write

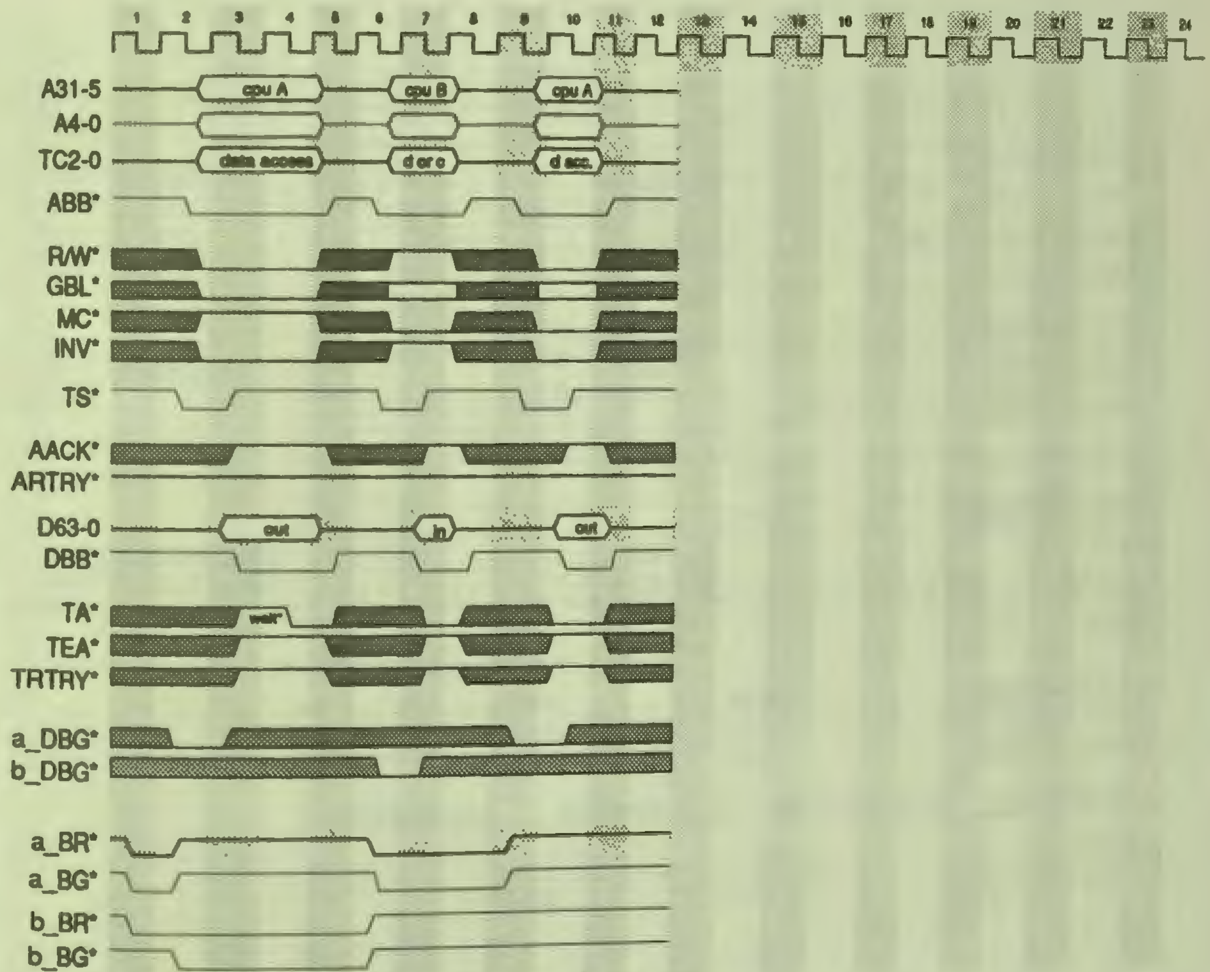


Figure 3.9.4.1.9 - Invalidation vs. Read vs. Write

3.9.4.2 Burst Mode Transfers

3.9.4.2.1 Fastest Burst Transfers

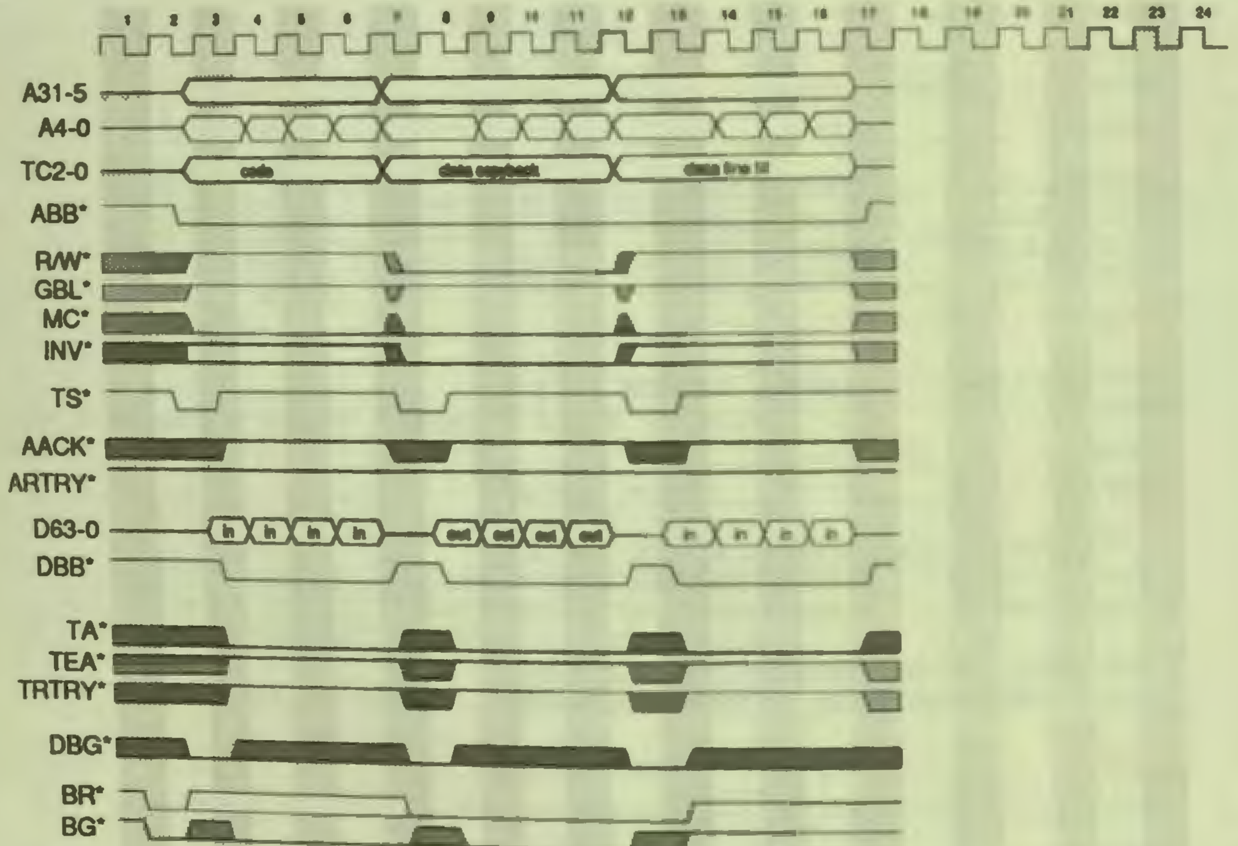


Figure 3.9.4.2.1 - Fastest Burst Transfers

3.9.4.2.2 Burst Read with Waits/DBG*

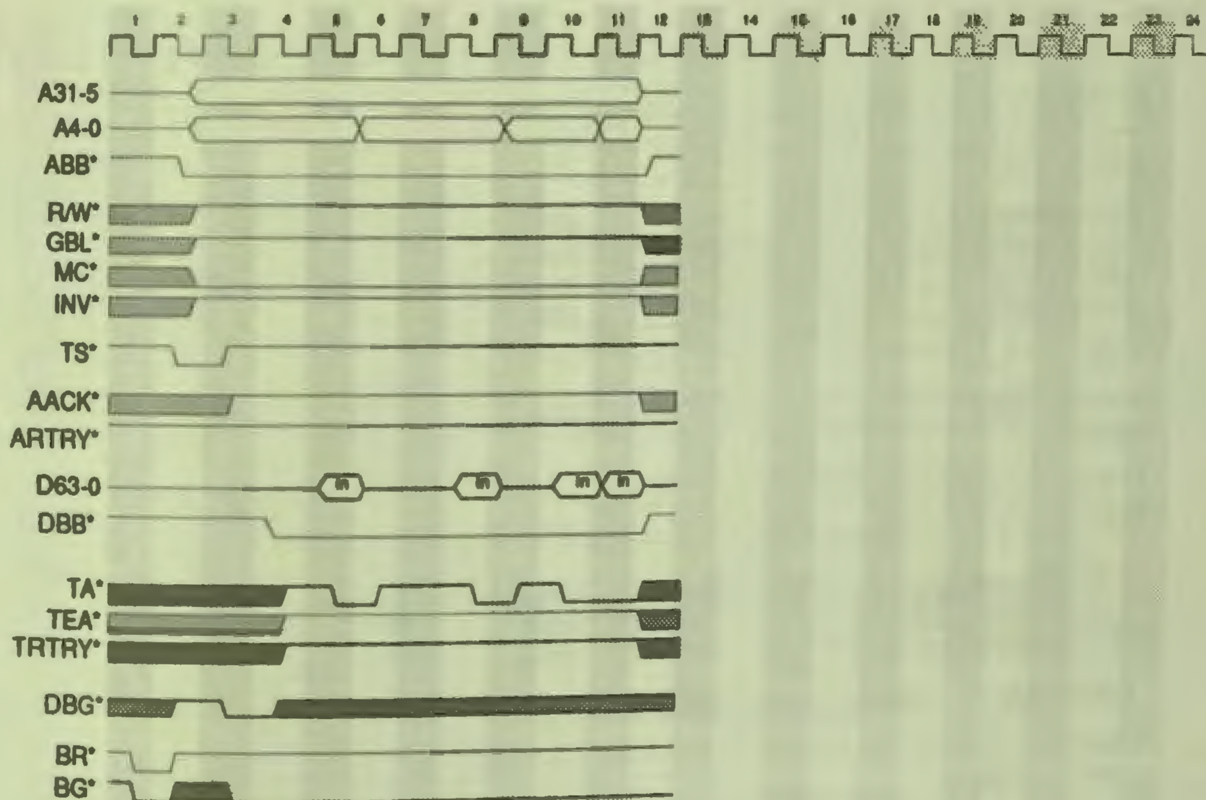


Figure 3.9.4.2.2 - Burst Read with Waits/DBG*

3.9.4.2.3 Burst Write with Waits/DBG*

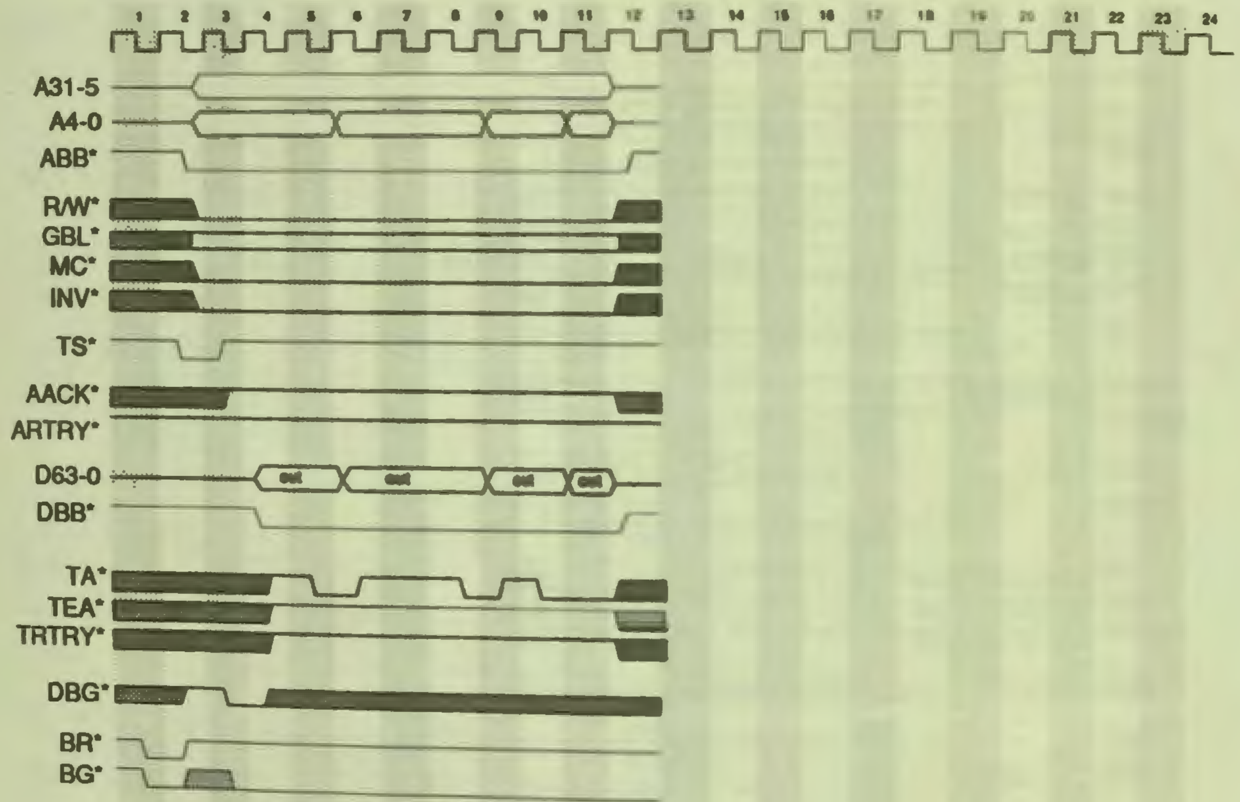


Figure 3.9.4.2.3 - Burst Write with Waits/DBG*

3.9.4.2.4 Burst Read In Half-Speed Mode

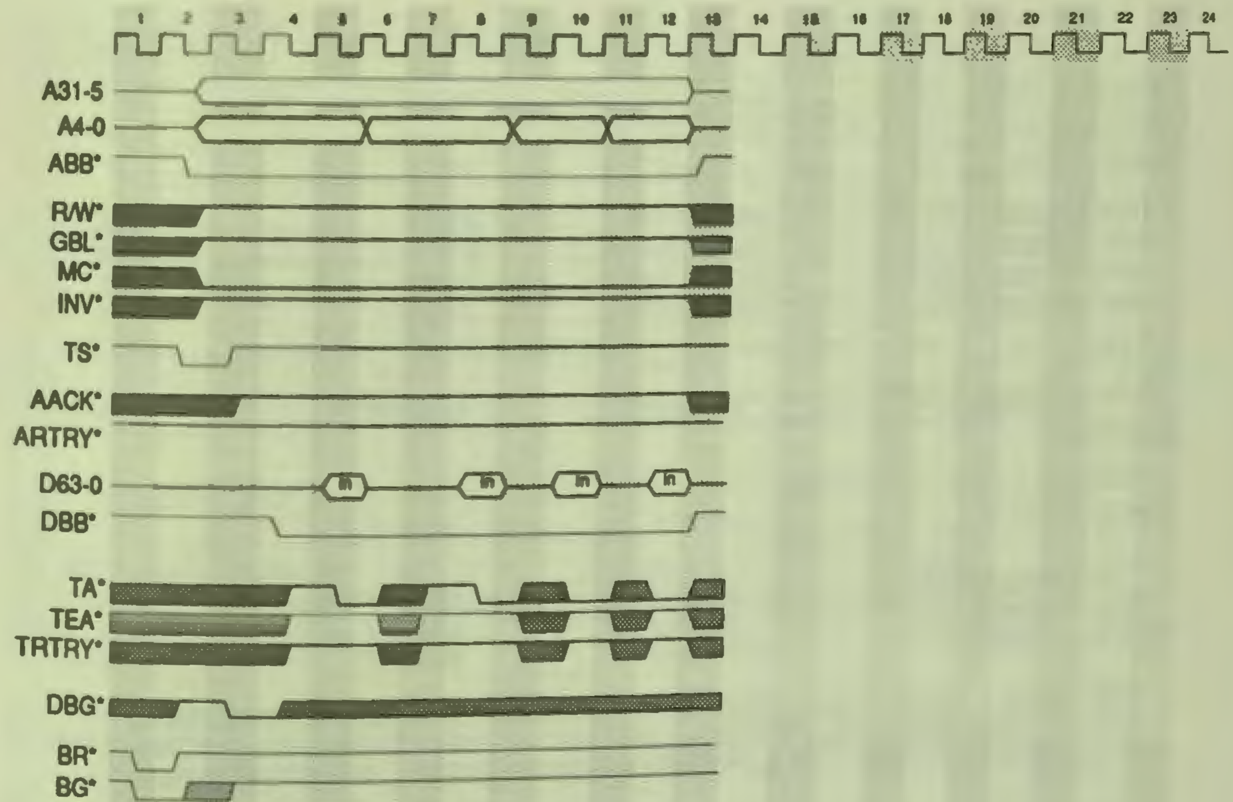
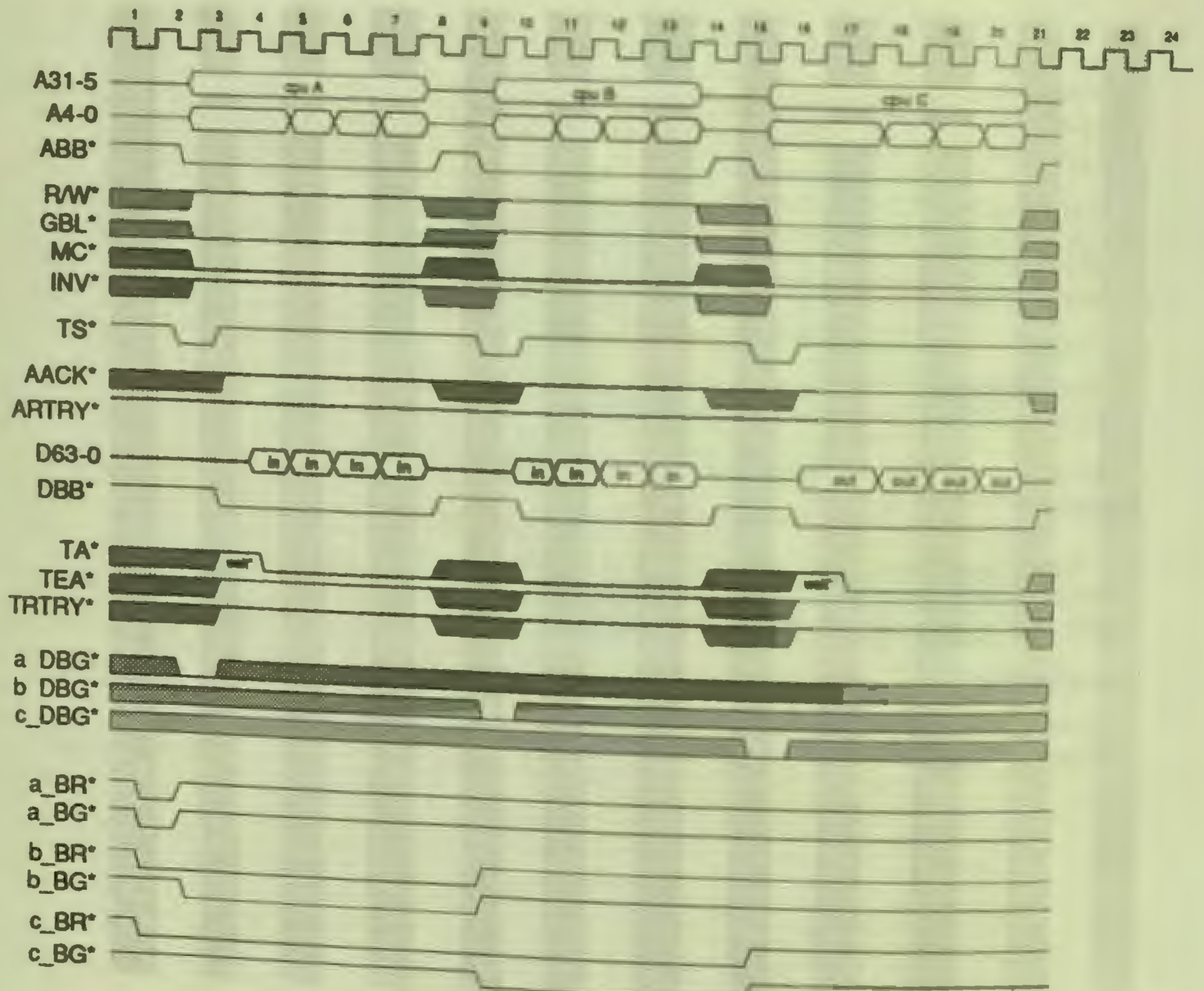


Figure 3.9.4.2.4 - Burst Read In Half-Speed Mode

3.9.4.2.5 Non-pipelined Burst Transfers



wait* : Required if access is global and snooping is desired.

Figure 3.9.4.2.5 - Non-pipelined Burst Transfers

3.9.4.2.6 Pre-Transfer Acknowledge

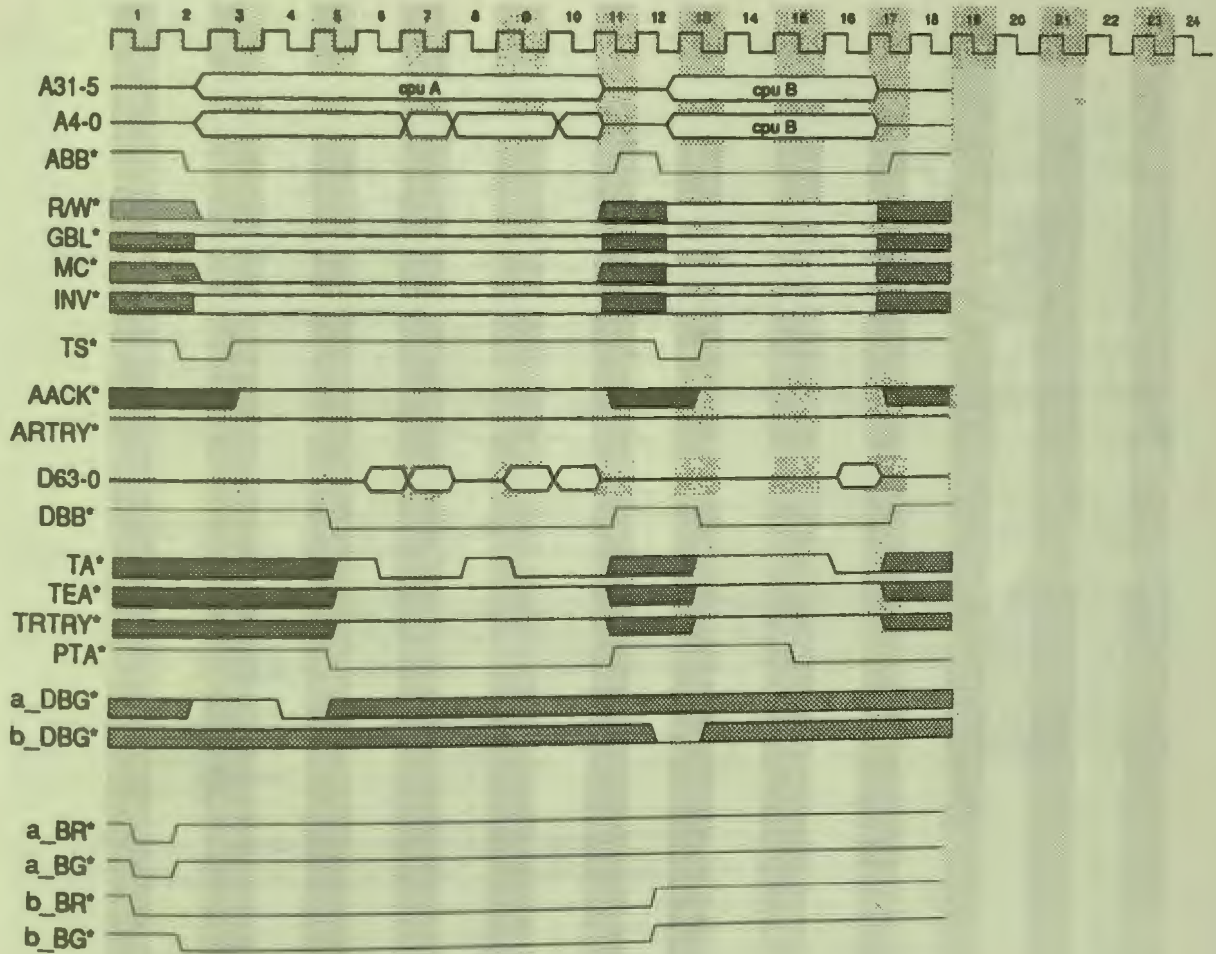


Figure 3.9.4.2.6 - PTA* Examples

3.9.4.2.7 Pipelined using DBB* Data Tenure Hand-off

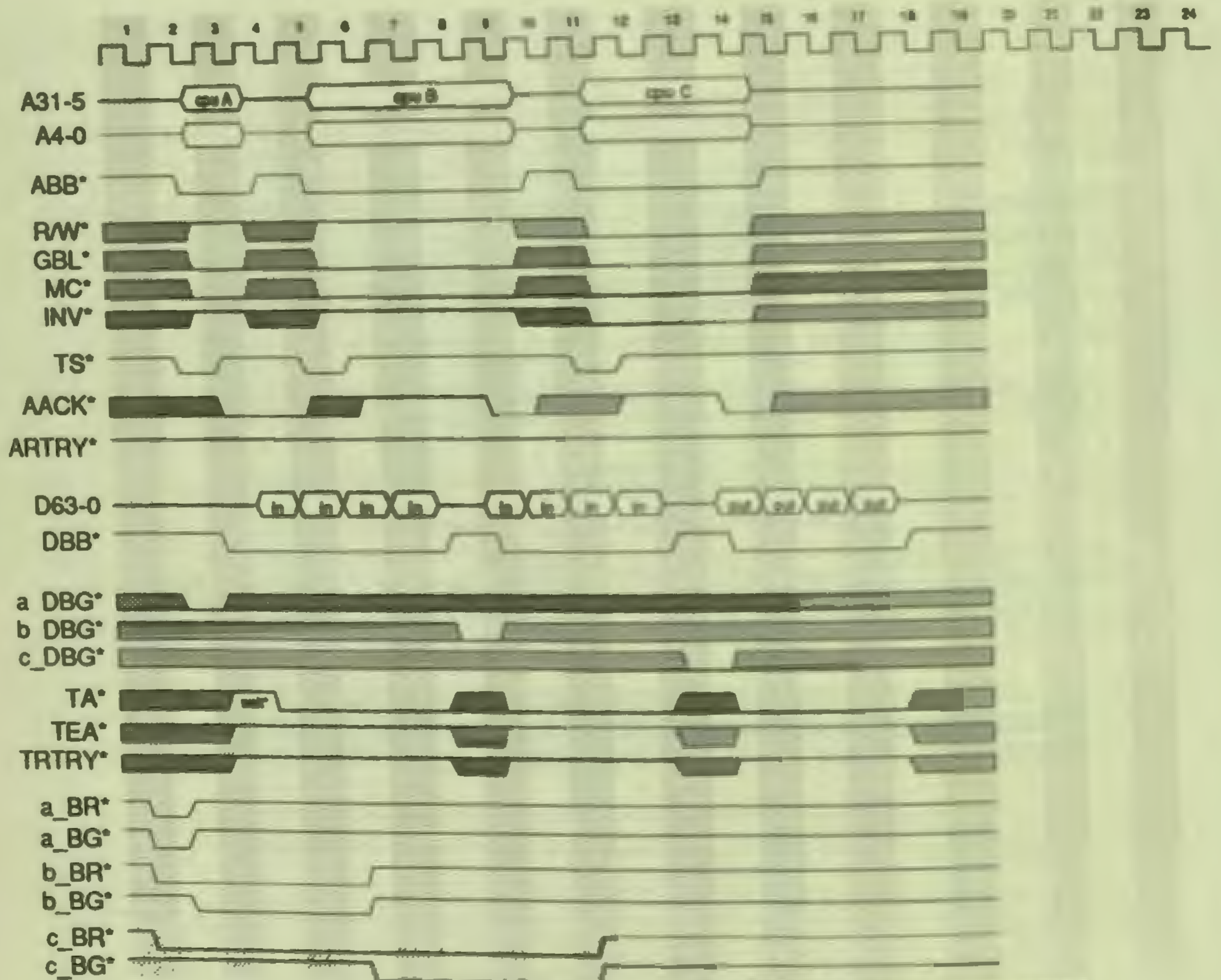


Figure 3.9.4.2.7 - Pipelined using DBB* Data Tenure Hand-off

3.9.4.3 Termination Cycles

3.9.4.3.1 Non-Pipelined Error Termination

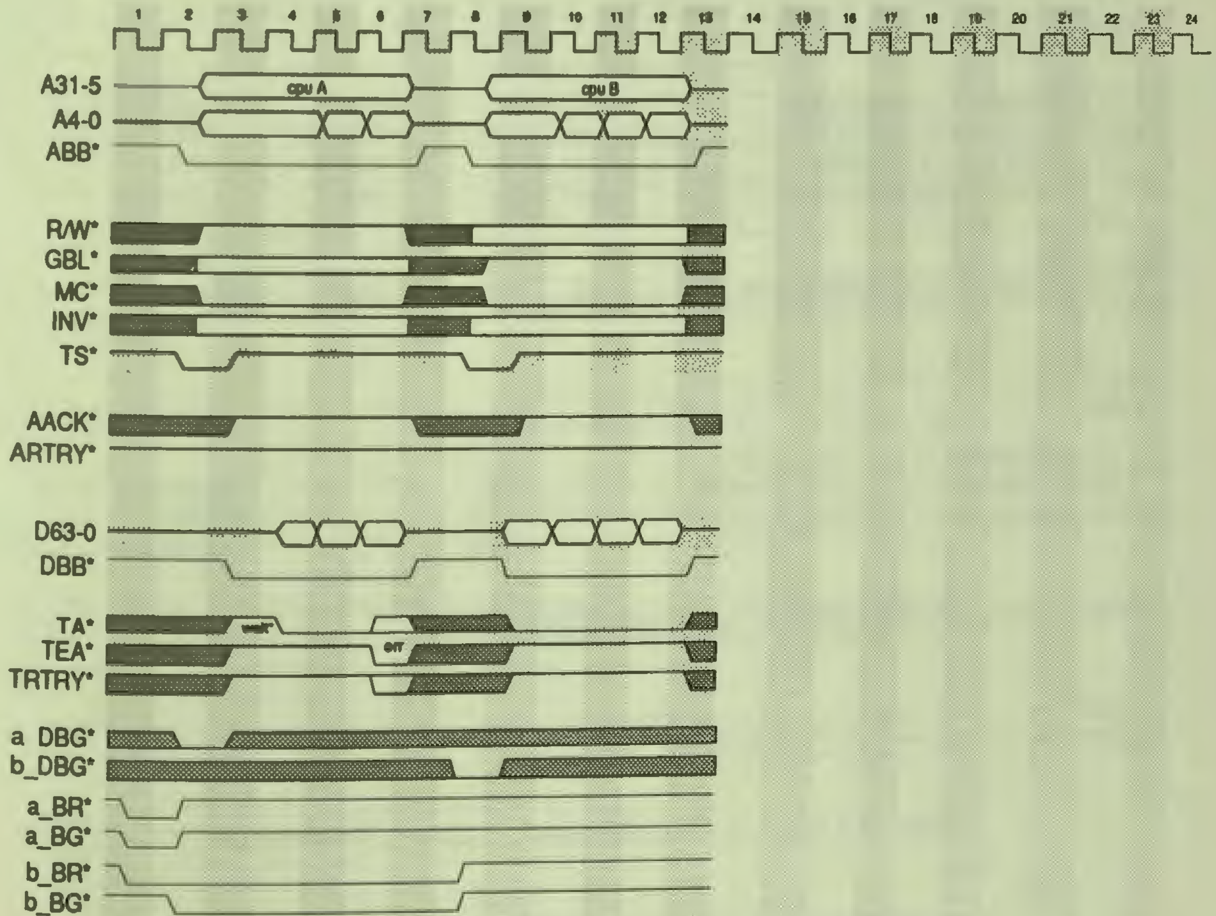


Figure 3.9.4.3.1 - Non-Pipelined Error Termination

3.9.4.3.2 Non-Pipelined TRTRY* Termination

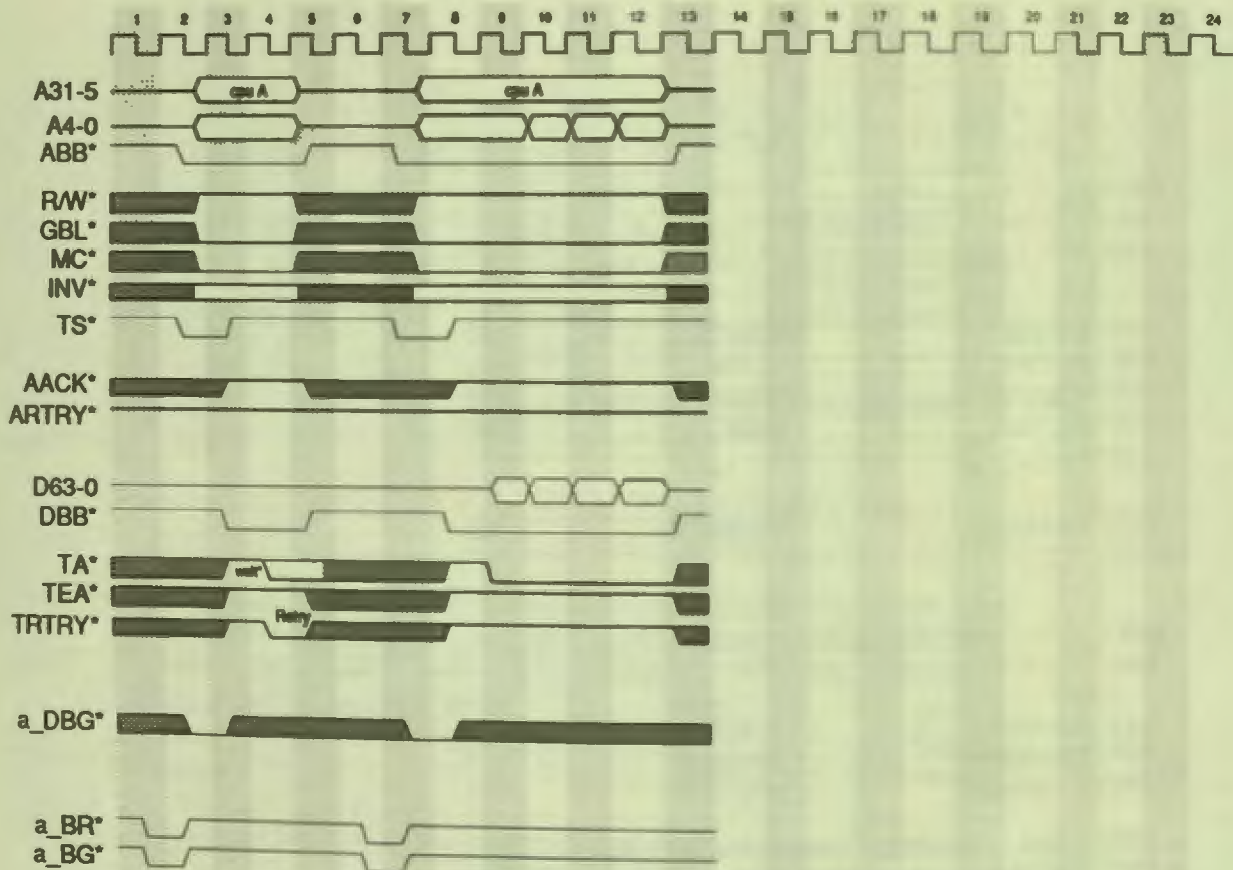


Figure 3.9.4.3.2 - Non-Pipelined TRTRY* Termination

3.9.4.3.3 Pipelined TRTRY* Termination

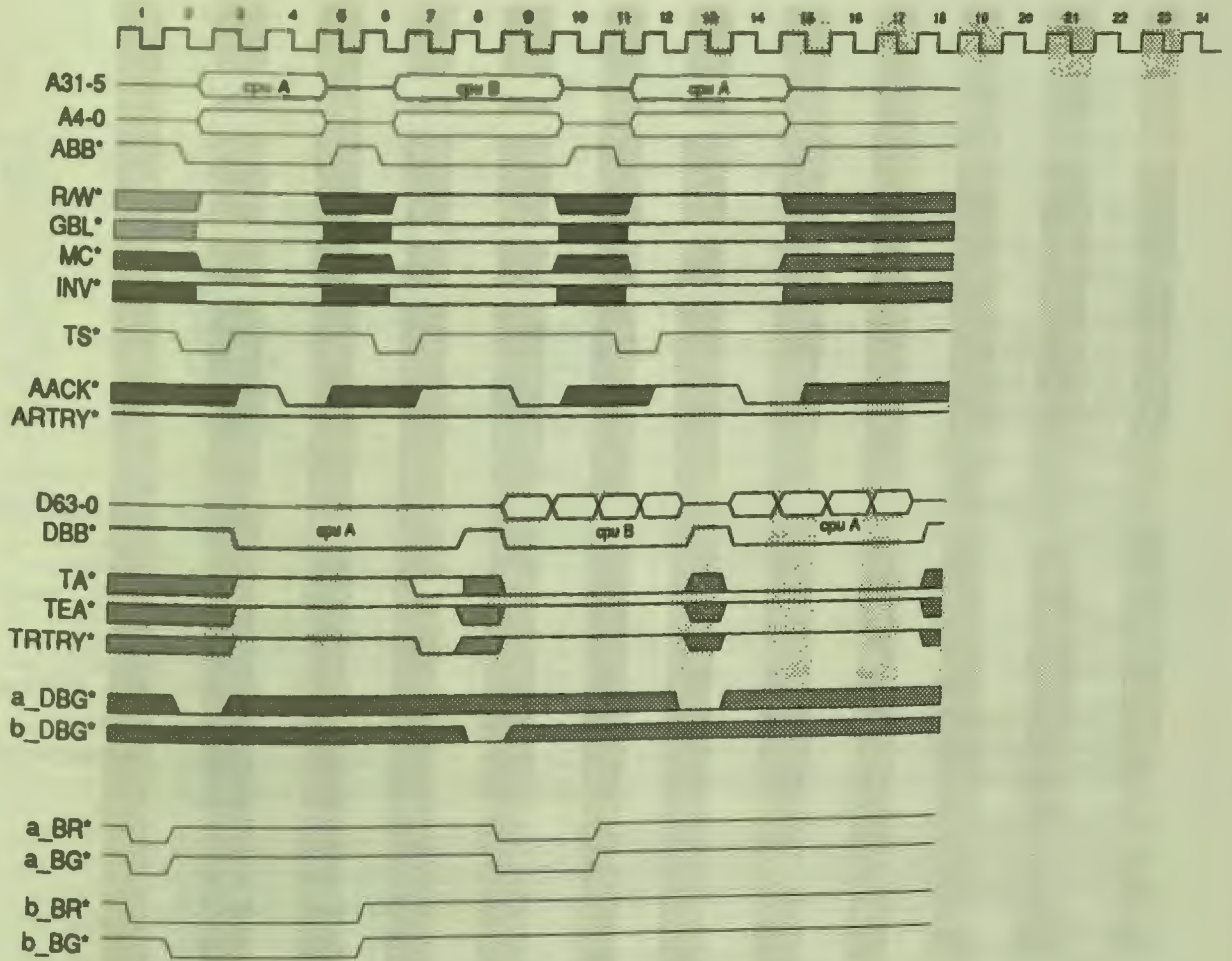


Figure 3.9.4.3.3 - Pipelined TRTRY* Termination

3.9.4.4 Snoop Hit Transfers

3.9.4.4.1 Non-Pipelined Snoop Hit ARTRY*

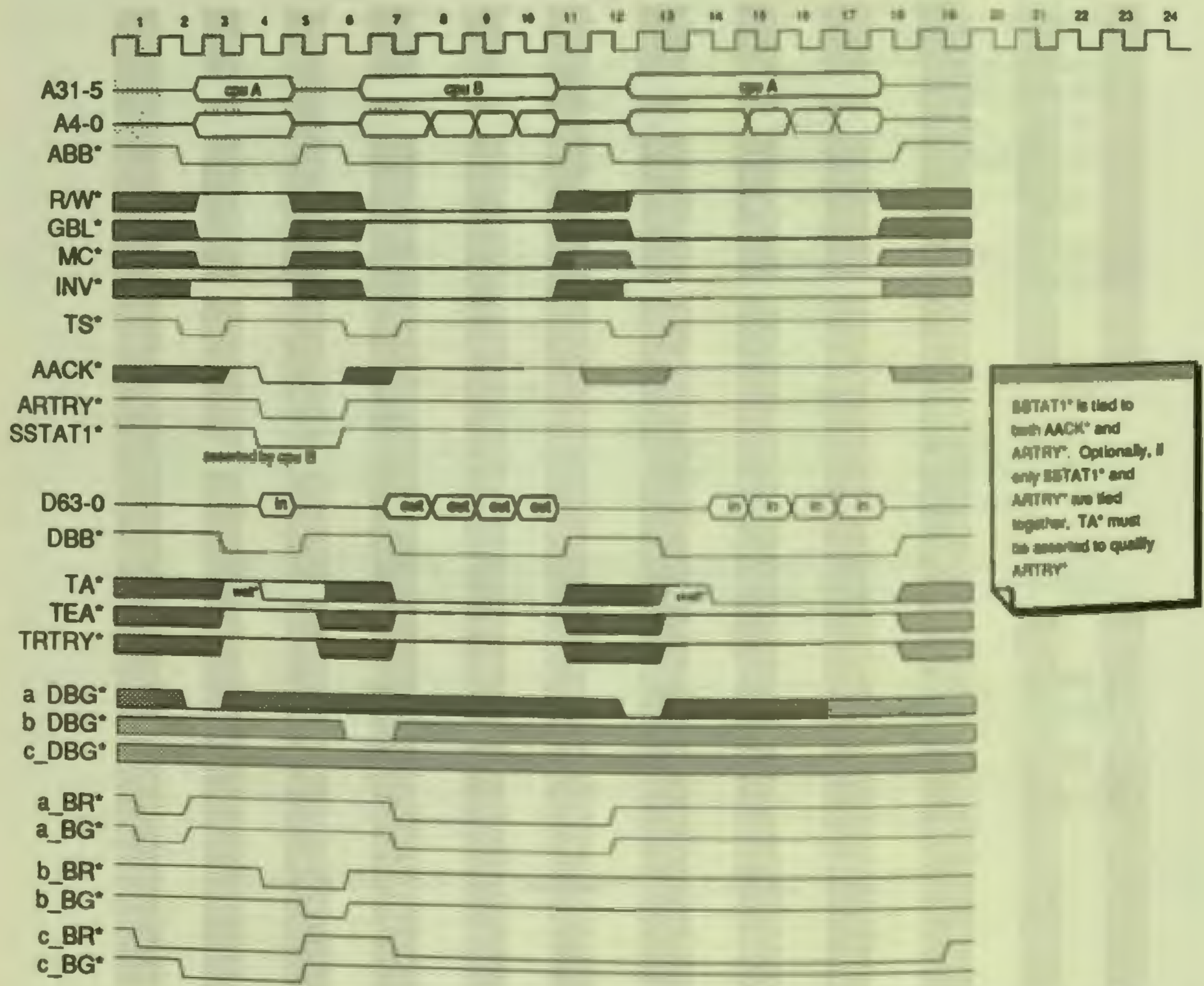


Figure 3.9.4.4.1 - Non-Pipelined Snoop Hit ARTRY*

3.9.4.4.2 Pipelined Snoop Hit ARTRY*

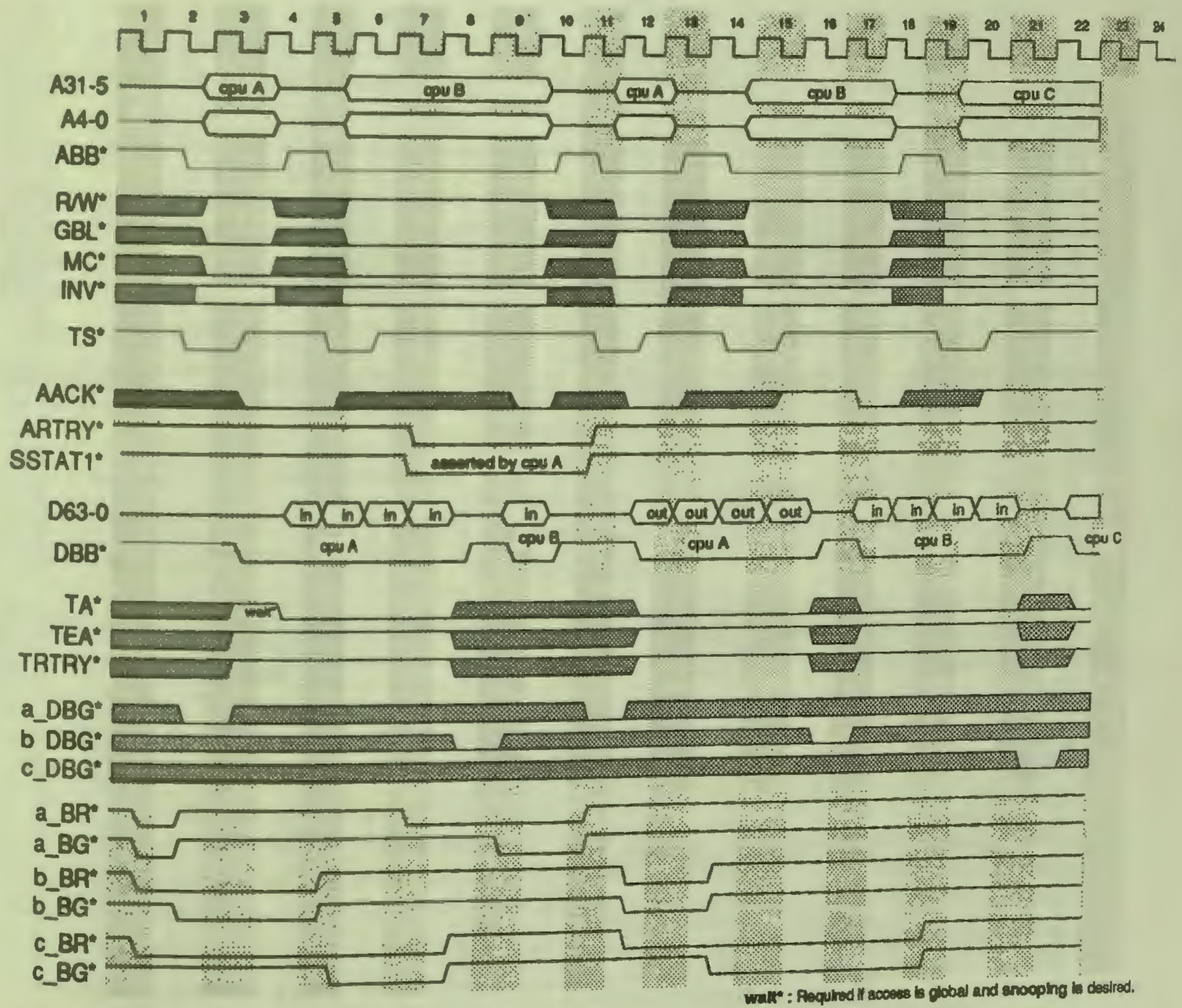


Figure 3.9.4.4.2 - Pipelined Snoop Hit ARTRY*

3.9.4.4.3 Open Pipeline Snoop Hit and Collision ARTRY*

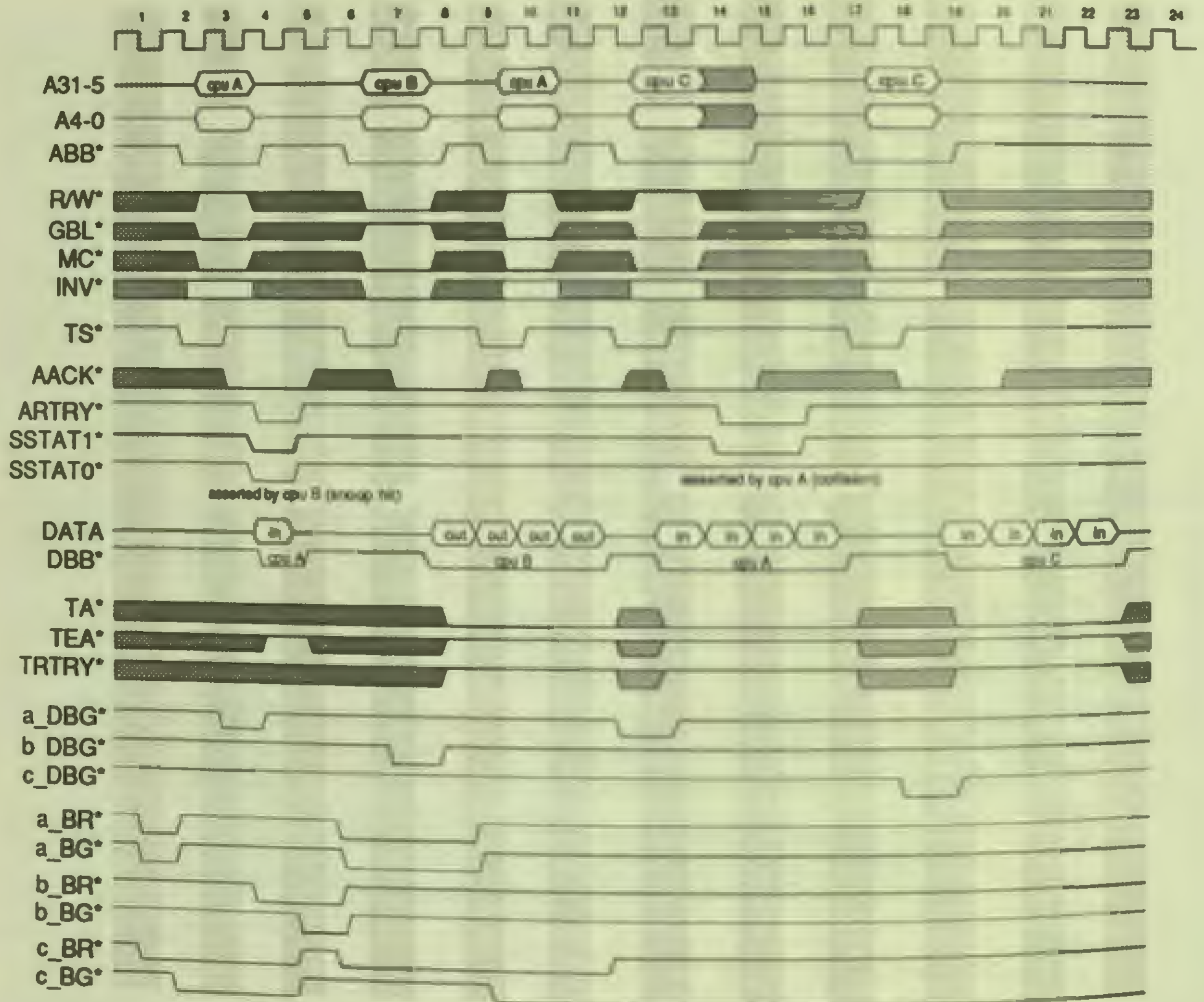


Figure 3.9.4.4.3 - Open Pipe Snoop Hit and Collision ARTRY*

A. - APPENDIX

A.1. - EXCEPTION VECTOR TABLE

Number	Address Vector Base +	Exception Definition
0	$00	Reset
1	$08	Maskable Interrupt
2	$10	Instruction Access
3	$18	Data Access
4	$20	Misaligned Address
5	$28	Unimplemented Opcode
6	$30	Privilege Violation
7	$38	Bounds Check Violation
8	$40	Integer Divide-by-zero
9	$48	Integer Overflow
10	$50	Unrecoverable Error
11	$58	Nonmaskable Interrupt *
12	$60	Data MMU ATC Miss *
13	$68	Instruction MMU ATC Miss *
14-113		Reserved
114	$390	SFU1 - Floating Point Exception
115	$398	Reserved
116	$3A0	SFU2 - Graphics Unimplemented Opcode
117	$3A8	Reserved
118	$3B0	SFU3 - Unimplemented Opcode
119	$3B8	Reserved
120	$3C0	SFU4 - Unimplemented Opcode
121	$3C8	Reserved
122	$3D0	SFU5 - Unimplemented Opcode
123	$3D8	Reserved
124	$3E0	SFU6 - Unimplemented Opcode
125	$3E8	Reserved
126	$3F0	SFU7 - Unimplemented Opcode
127	$3F8	Reserved
128-511		Reserved - User Trap Vectors

* Not present in 88100

Table A.1 - Exception Vector Table

A.2. - CONTROL REGISTERS

Number	Acronym	Register Name	Figure
cr0	PID	Processor Identification	
cr1	PSR	Processor Status Register	3.2.4
cr2	EPSR	Exception Processor Status Register	
cr3		*Unimplemented*	
cr4	EIP	Exception Instruction Pointer	
cr5-cr6		*Unimplemented*	
cr7	VBR	Vector Base Register	
c8-cr15		*Unimplemented*	
cr16	SR0	Storage Register 0	
cr17	SR1	Storage Register 1	
cr18	SR2	Storage Register 2	
cr19	SR3	Storage Register 3	
cr20	SR4	Storage Register 4	
cr21-cr24		*Unimplemented*	
cr25	ICMD	Instruction MMU/Cache/TIC Command	3.8.8.a
cr26	ICTL	Instruction MMU/Cache Control	3.8.8.b
cr27	ISAR	Instruction System Address	
cr28	ISAP	Instruction MMU Supervisor Area Pointer	3.8.3.1
cr29	IUAP	Instruction MMU User Area Pointer	3.8.3.1
cr30	IIR	Instruction MMU ATC Index Register	3.8.8.c
cr31	IBP	Instruction MMU BATC R/W Port	3.8.8.d
cr32	IPPU	Instruction MMU PATC R/W Port - Upper	3.8.1.2
cr33	IPPL	Instruction MMU PATC R/W Port - Lower	3.8.1.2
cr34	ISR	Instruction Access Status Register	3.8.8.e
cr35	ILAR	Instruction Access Logical Address	
cr36	IPAR	Instruction Access Physical Address	
cr37-cr39		*Unimplemented*	
cr40	DCMD	Data MMU/Cache Command	3.8.8.f
cr41	DCTL	Data MMU/Cache Control	3.8.8.g
cr42	DSAR	Data System Address	
cr43	DSAP	Data MMU Supervisor Area Pointer	3.8.3.1
cr44	DUAP	Data MMU User Area Pointer	3.8.3.1
cr45	DIR	Data MMU ATC Index Register	3.8.8.h
cr46	DBP	Data MMU BATC R/W Port	3.8.8.i
cr47	DPPU	Data MMU PATC R/W Port - Upper	3.8.1.2
cr48	DPPL	Data MMU PATC R/W Port - Lower	3.8.1.2
cr49	DSR	Data Access Status Register	3.8.8.j
cr50	DLAR	Data Access Logical Address	
cr51	DPAR	Data Access Physical Address	
cr52-cr63		*Unimplemented*	

Table A.2.1 - SFU0 Control Registers

Number	Acronym	Register Name	Figure
fcr0	FPECR	Floating Point Exception Cause Register	3.3.2.1
fcr1-fcr61		*Unimplemented*	
fcr62	FPSR	Floating Point Status Register	2.2.5.2
fcr63	FPCR	Floating Point Control Register	2.2.5.2

Table A.2.2 - SFU1 Control Registers

A.3. - OPCODE ASSIGNMENTS

Highlighted in gray are differences from the 88100.

	31 28 27 26	25 21	20 16	15 0
ld.d (x)	0 0 0 0 0 0	D	S1	SIMM16
ld (x)	0 0 0 0 0 1	D	S1	SIMM16
ld.u (g)	0 0 0 0 1 b	D	S1	SIMM16
ld (g)	0 0 0 1 TY	D	S1	SIMM16
st (g)	0 0 1 0 TY	D	S1	SIMM16
st.d (x)	0 0 1 1 0 0	D	S1	SIMM16
st (x)	0 0 1 1 0 1	D	S1	SIMM16
st.x (x)	0 0 1 1 1 0	D	S1	SIMM16
ld.x (x)	0 0 1 1 1 1	D	S1	SIMM16
and	0 1 0 0 0 U	D	S1	IMM16
mask	0 1 0 0 1 U	D	S1	IMM16
xor	0 1 0 1 0 U	D	S1	IMM16
or	0 1 0 1 1 U	D	S1	IMM16
addu	0 1 1 0 0 0	D	S1	IMM16
subu	0 1 1 0 0 1	D	S1	IMM16
divu	0 1 1 0 1 0	D	S1	IMM16
mulu	0 1 1 0 1 1	D	S1	IMM16
add	0 1 1 1 0 0	D	S1	SIMM16
sub	0 1 1 1 0 1	D	S1	SIMM16
div	0 1 1 1 1 0	D	S1	SIMM16
cmp	0 1 1 1 1 1	D	S1	SIMM16

.u:	unsigned
.s:	single-word
.d:	double-word
.x:	quad-word
(x):	extended register file
(g):	general register file
b:	0 - half-word, 1 - byte
U:	lower, upper
TY:	double-word, single-word, half-word, byte
SIMM16:	16-bit unsigned or signed value depending on immediate mode
IMM16:	16-bit unsigned value

Figure A.3.a - Immediate Opcodes

	31–26	25–21	20–16	15–14	13–11	10–5	4–0
ldcr	1 0 0 0 0 0	D	0 0 0 0 0	0 1	0 0 0	CRS	0 0 0 0 0
stcr	1 0 0 0 0 0	0 0 0 0 0	S1	1 0	0 0 0	CRD	S2
xcr	1 0 0 0 0 0	D	S1	1 1	0 0 0	CRS/CRD	S2
fldcr	1 0 0 0 0 0	D	0 0 0 0 0	0 1	0 0 1	CRS	0 0 0 0 0
fstcr	1 0 0 0 0 0	0 0 0 0 0	S1	1 0	0 0 1	CRD	S2
fxcr	1 0 0 0 0 0	D	S1	1 1	0 0 1	CRS/CRD	S2

	31–26	25–21	20–16	15	14–11	10–9	8–7	6–5	4–0
fmul	1 0 0 0 0 1	D	S1	R	0 0 0 0	T1	T2	TD	S2
fcvt	1 0 0 0 0 1	D	0 0 0 0 0	R	0 0 0 1	0 0	T2	TD	S2
flt (→g)	1 0 0 0 0 1	D	0 0 0 0 0	0 0 1 0 0		0 0	0 0	TD	S2
flt (→x)	1 0 0 0 0 1	D	0 0 0 0 0	0 0 1 0 0		0 1	0 0	TD	S2
fadd	1 0 0 0 0 1	D	S1	R	0 1 0 1	T1	T2	TD	S2
fsub	1 0 0 0 0 1	D	S1	R	0 1 1 0	T1	T2	TD	S2
fcmp	1 0 0 0 0 1	D	S1	R	0 1 1 1	T1	T2	0 0	S2
fcmpu	1 0 0 0 0 1	D	S1	R	0 1 1 1	T1	T2	0 1	S2
mov→g	1 0 0 0 0 1	D	0 0 0 0 0	1	1 0 0 0	0 0	T2*	0 0	S2
mov→x	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 0 0	0 1	T2†	0 0	S2
int	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 0 1	0 0	T2	0 0	S2
nint	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 1 0	0 0	T2	0 0	S2
trnc	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 1 1	0 0	T2	0 0	S2
fdiv	1 0 0 0 0 1	D	S1	R	1 1 1 0	T1	T2	TD	S2
fsqrt	1 0 0 0 0 1	D	0 0 0 0 0	R	1 1 1 1	0 0	T2	TD	S2

CRS: Source Control Register
CRD: Destination Control Register
R: 0 - source operands in GRF, 1 - source operands in XRF
T1: Source 1 size: 00 - single, 01 - double, 10 - extended, 11 - unused
T2: Source 2 size: 00 - single, 01 - double, 10 - extended, 11 - unused
T2*: Source 2 size: 00 - single, 01 - double, 10 - unused, 11 - unused
T2†: if R=1 then T2† =10 else T2† =T2*.
TD: Destination size: 00 - single, 01 - double, 10 - extended, 11 - unused
→g: destination operand in GRF
→x: destination operand in XRF

Figure A.3.b - Control Register Access and SFU1 Opcodes

	31–26	25–21	20–16	15–11	10–9	8–7	6–5	4–0
pmul	1 0 0 0 1 0	D	S1	0 0 0 0 0	0 0	0 0	0 0	S2
padd	1 0 0 0 1 0	D	S1	0 0 1 0 0	0 0	0 0	T	S2
padds	1 0 0 0 1 0	D	S1	0 0 1 0 0	0 0	S	T	S2
psub	1 0 0 0 1 0	D	S1	0 0 1 1 0	0 0	0 0	T	S2
psubs	1 0 0 0 1 0	D	S1	0 0 1 1 0	0 0	S	T	S2
pcmp	1 0 0 0 1 0	D	S1	0 0 1 1 1	0 0	0 0	1 1	S2
ppack	1 0 0 0 1 0	D	S1	0 1 1 0 0	R		T	S2
punpk	1 0 0 0 1 0	D	S1	0 1 1 0 1	0 0	0 0	T	0 0 0 0 0
prot	1 0 0 0 1 0	D	S1	0 1 1 1 0	R		0 0	0 0 0 0 0
prot	1 0 0 0 1 0	D	S1	0 1 1 1 1	0 0	0 0	0 0	S2*

T:
- 00 = 4-bit (valid only for **punpk**)
- 01 = 8-bit
- 10 = 16-bit
- 11 = 32-bit
- Note: Not all types make sense on all instructions. Types other than those specified in the text cause an SFU2 exception.

R:
- 0000= rotate 64-bit register pair left 0 (or 64) bits.
- 0001= rotate left 4 bits.
- 0010= rotate left 8 bits.
- rrrr= rotate left (rrrr × 4) bits.
- Note: only rotations of 8,16, and 32 are meaningful for **ppack** and only in limited combinations with T.

S2*: The nibble-wise (4-bit) rotate count is specified in bits <5:2> of rS_2. Other bits (<31:6>,<1:0>) are ignored but should be set to zero to assure future compatibility.

S:
- 00: padd, non-saturating
- 01: unsigned ± unsigned = unsigned saturation
- 10: unsigned ± signed = unsigned saturation
- 11: signed ± signed = signed saturation

Figure A.3.c - SFU2 Graphics Opcodes

	31–27	26	25–0
br	1 1 0 0 0	N	D26
bsr	1 1 0 0 1	N	D26

	31–27	26	25–21	20–16	15–0
bb0	1 1 0 1 0	N	B5	S1	D16
bb1	1 1 0 1 1	N	B5	S1	D16
bcnd	1 1 1 0 1	N	M5	S1	D16

	31–26	25–21	20–16	15–10	9–5	4–0
clr	1 1 1 1 0 0	D	S1	1 0 0 0 0 0	W5	O5
set	1 1 1 1 0 0	D	S1	1 0 0 0 1 0	W5	O5
ext	1 1 1 1 0 0	D	S1	1 0 0 1 0 0	W5	O5
extu	1 1 1 1 0 0	D	S1	1 0 0 1 1 0	W5	O5
mak	1 1 1 1 0 0	D	S1	1 0 1 0 0 0	W5	O5
rot	1 1 1 1 0 0	D	S1	1 0 1 0 1 0	0 0 0 0 0	O5

	31–26	25–21	20–16	15–9	8–0
tb0	1 1 1 1 0 0	B5	S1	1 1 0 1 0 0 0	VEC9
tb1	1 1 1 1 0 0	B5	S1	1 1 0 1 1 0 0	VEC9
tcnd	1 1 1 1 0 0	M5	S1	1 1 1 0 1 0 0	VEC9

	31–27	26	25–21	20–16	15–12	11	10	9	8	7	6–5	4–0
xmem	1 1 1 1 0	1	D	S1	0 0 0 0	0	w	S	U	0	0 0	S2
ld.u	1 1 1 1 0	1	D	S1	0 0 0 0	1	b	S	U	0	0 0	S2
ld	1 1 1 1 0	R	D	S1	0 0 0 1	TY		S	U	0	0 0	S2
st	1 1 1 1 0	R	D	S1	0 0 1 0	TY		S	U	T	0 0	S2
lda [ ]	1 1 1 1 0	F	D	S1	0 0 1 1	TY		1	0	0	0 0	S2

N: execute next
D26: 26-bit sign extended displacement
D16: 16-bit sign extended displacement
B5: bit number
W5: bit field width
O5: bit field offset
M5: condition match field
VEC9: vector number
U: access user space
T: 0 - normal store,
1 - store-through the cache

.u: unsigned
[], S: scaled
R: 0 - XRF destination, 1 GRF dest.
$TY_{R=1}$: double, word, half-word, byte
$TY_{R=0}$: double, single, quad
F: scale factor select
$TY_{F=1}$: 8x, 4x, 2x, 1x scale factor
$TY_{F=0}$: 8x, 4x, 16x, scale factor
b: 0 - half-word, 1 - byte
w: 0 - byte, 1 - word

Figure A.3.d - Branch, Bit Field, and Memory Opcodes

	31–26	25–21	20–16	15–11	10	9–5	4–0
and	1 1 1 1 0 1	D	S1	0 1 0 0 0	C	0 0 0 0 0	S2
xor	1 1 1 1 0 1	D	S1	0 1 0 1 0	C	0 0 0 0 0	S2
or	1 1 1 1 0 1	D	S1	0 1 0 1 1	C	0 0 0 0 0	S2

	31–26	25–21	20–16	15–10	9	8	7–5	4–0
addu	1 1 1 1 0 1	D	S1	0 1 1 0 0 0	i	o	0 0 0	S2
subu	1 1 1 1 0 1	D	S1	0 1 1 0 0 1	i	o	0 0 0	S2
divu	1 1 1 1 0 1	D	S1	0 1 1 0 1 0	0	d	0 0 0	S2
mulu	1 1 1 1 0 1	D	S1	0 1 1 0 1 1	0	d	0 0 0	S2
muls	1 1 1 1 0 1	D	S1	0 1 1 0 1 1	1	0	0 0 0	S2
add	1 1 1 1 0 1	D	S1	0 1 1 1 0 0	i	o	0 0 0	S2
sub	1 1 1 1 0 1	D	S1	0 1 1 1 0 1	i	o	0 0 0	S2
divs	1 1 1 1 0 1	D	S1	0 1 1 1 1 1	0	0	0 0 0	S2
cmp	1 1 1 1 0 1	D	S1	0 1 1 1 1 1	0	0	0 0 0	S2

	31–26	25–21	20–16	15–5	4–0
clr	1 1 1 1 0 1	D	S1	1 0 0 0 0 0 0 0 0 0 0	S2
set	1 1 1 1 0 1	D	S1	1 0 0 0 1 0 0 0 0 0 0	S2
ext	1 1 1 1 0 1	D	S1	1 0 0 1 0 0 0 0 0 0 0	S2
extu	1 1 1 1 0 1	D	S1	1 0 0 1 1 0 0 0 0 0 0	S2
mak	1 1 1 1 0 1	D	S1	1 0 1 0 0 0 0 0 0 0 0	S2
rot	1 1 1 1 0 1	D	S1	1 0 1 0 1 0 0 0 0 0 0	S2

	31–26	25–16	15–11	10	9–5	4–0
jmp	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 0 0 0	N	0 0 0 0 0	S2
jsr	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 0 0 1	N	0 0 0 0 0	S2

	31–26	25–21	20–16	15–5	4–0
ff1	1 1 1 1 0 1	D	0 0 0 0 0	1 1 1 0 1 0 0 0 0 0 0	S2
ff0	1 1 1 1 0 1	D	0 0 0 0 0	1 1 1 0 1 1 0 0 0 0 0	S2
tbnd	1 1 1 1 0 1	0 0 0 0 0	S1	1 1 1 1 1 0 0 0 0 0 0	S2
rte	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0		1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0	
tbnd	1 1 1 1 1 0	0 0 0 0 0	S1	IMM16	

C: complement second operand

i: use carry-in

o: set carry-out

d: single, double

N: execute next

Figure A.3.e - Triadic Operations Opcodes