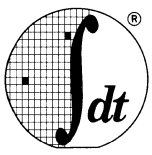
# IDT79R3051™ Family Hardware User's Manual


Integrated Device Technology, Inc.

# IDT79R3051™ Family Hardware User's Manual



Integrated Device Technology, Inc.

**LIFE SUPPORT POLICY**

**Integrated Device Technology's products are not authorized for use as critical components in life support devices or systems unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of IDT.**

1. **Life support devices or systems are devices or systems which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.**
2. **A critical component is any components of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.**

Integrated Device Technology, Inc.

# TABLE OF CONTENTS

**List of Figures**

**List of Tables**

# *ABOUT THIS MANUAL*

This manual provides a qualitative description of the operation of members of the IDT R3051 family of integrated RISControllers.

A quantitative description of the processor electrical interface is provided in the data sheets for these products. Also included in the data sheets are the mechanical descriptions of the part, including packaging and pin-out.

Additional information on development tools, complementary support chips, and the use of these products in various applications, are provided in separate data sheets and applications notes.

Any of this information is readily available from your local IDT sales representative.

Integrated Device Technology, Inc.

# FAMILY OVERVIEW

CHAPTER 1

## INTRODUCTION

The IDT R3051 family is a series of high-performance 32-bit microprocessors featuring a high-level of integration, and targeted to high-performance but cost sensitive embedded processing applications. The R3051 family is designed to bring the high-performance inherent in the MIPS RISC architecture into low-cost, simplified, power sensitive applications.

Thus, functional units have been integrated onto the CPU core in order to reduce the total system cost, rather than to increase the inherent performance of the integer engine. Nevertheless, the R3051 family is able to offer 35 MIPS of integer performance at 40 MHz without requiring external SRAM or caches.

Further, the R3051 family brings dramatic power reduction to these embedded applications, allowing the use of low-cost packaging for devices up to 25 MHz. Thus, the R3051 family allows customer applications to bring maximum performance at minimum cost.

## FEATURES

- Instruction set compatible with IDT 79R3000A and R3001A RISC CPUs
- High level of integration minimizes system cost
- 35 MIPS at 40 MHz
- Low cost 84-pin PLCC packaging
- Large on-chip instruction and data caches
- Flexible bus interface allows simple, low cost designs.
- Single, double-frequency clock input
- 20 through 40 MHz operation
- On-chip 4-deep write buffer eliminates memory write stalls
- On-chip 4-deep read buffer supports burst or simple block reads
- On-chip DMA arbiter

## DEVICE OVERVIEW

There are currently four members of the R3051 family. All differences relate to the size of the on-chip instruction cache, and the structure of the on-chip memory management unit. All four devices are pin compatible, and user software compatible. They all utilize the same execution engine and bus interface unit, and all contain 2kB of data cache. The four family members are:

- R3052"E" incorporates an 8kB instruction cache, and full function memory management unit (MMU) including 64-entry fully associative Translation Lookaside Buffer (TLB).
- R3052, which also incorporates an 8kB instruction cache but does not include the TLB.
- R3051"E", which incorporates 4kB of instruction cache along with the full function MMU/TLB set.
- R3051, which incorporates 4kB of instruction cache but omits the TLB.

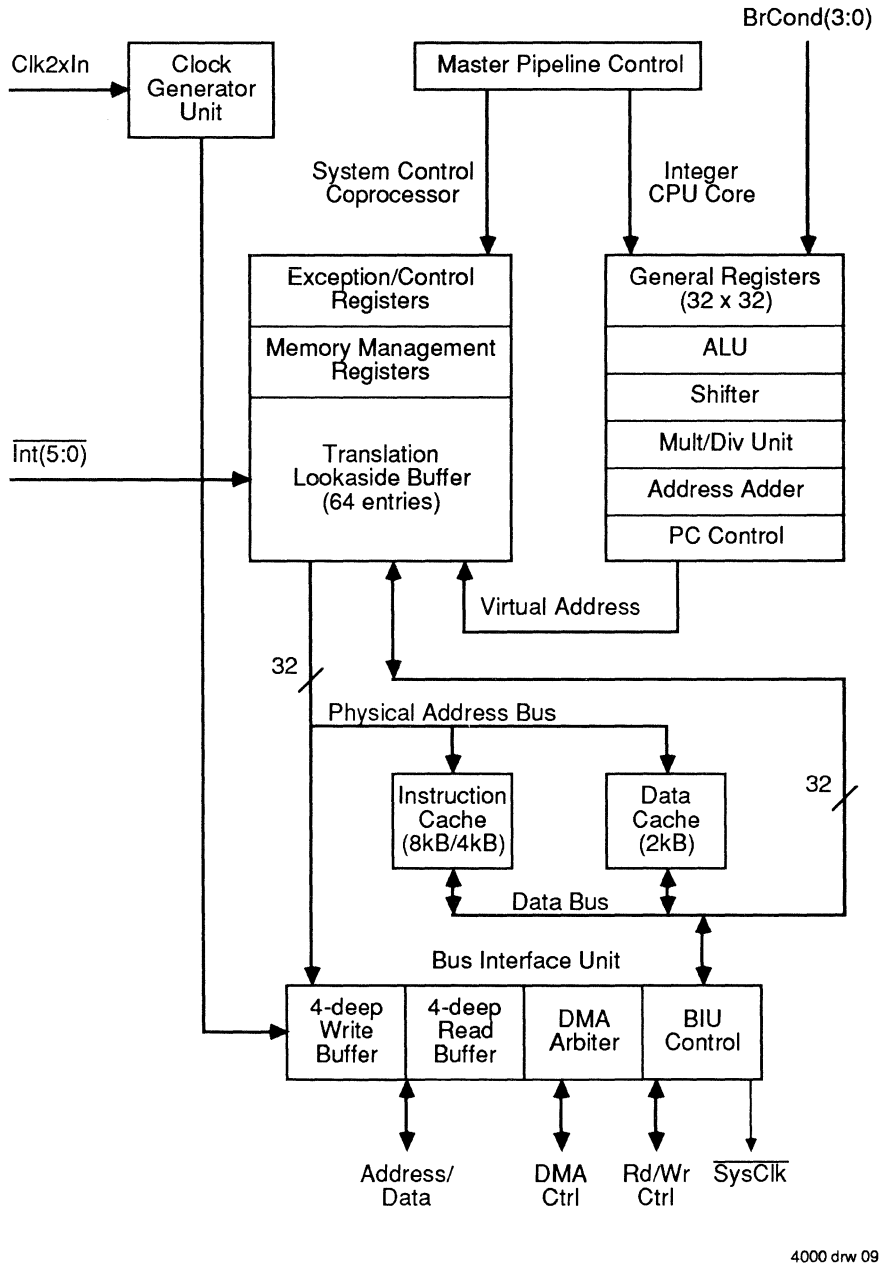



**Figure 1.1. Block Diagram**

Figure 1.1 shows a block level representation of the functional units within the R3051/52. The R3051/52 could be viewed as the embodiment of a discrete solution built around the R3000 or R3001. However, by integrating this functionality on a single chip, dramatic cost and power reductions are achieved.

An overview of these blocks is presented here, with detailed information on each block found in subsequent chapters.

### CPU Core

The CPU core is a full 32-bit RISC integer execution engine, capable of sustaining close to single cycle execution rate. The CPU core contains a five stage pipeline, and 32 orthogonal 32-bit registers. The R3051 family implements the MIPS-I ISA. In fact, the execution engine of the R3051 family is the same as the execution engine of the R3000 and R3001. Thus, the R3051 family is binary compatible with those CPU engines.

### System Control Co-Processor

The R3051 family also integrates on-chip the System Control Co-processor, CP0. CP0 manages both the exception handling capability of the R3051/52, as well as the virtual to physical mapping of the R3051/52. These topics are discussed in subsequent chapters.

There are two versions of the R3051/52 MMU: the extended architecture versions, the R3051E and R3052E, which incorporate the same MMU as the R3000 and R3001. These versions contain a fully associative 64-entry TLB which maps 4kB virtual pages into the physical address space. The virtual to physical mapping thus includes kernel segments which are hard-mapped to physical addresses, and kernel and user segments which are mapped page by page by the TLB into anywhere in the 4GB physical address space. In this TLB, 8 pages can be "locked" by the kernel to insure deterministic response in real-time applications.

The R3051 family base versions, the R3051 and R3052, remove the TLB and institute a fixed address mapping for the various segments of the virtual address space. These devices still support distinct kernel and user mode operation, but do not require page management software, leading to a simpler software model.

### Clock Generator Unit

The R3051/52 is driven from a single, double frequency input clock. On-chip, the clock generator unit is responsible for managing the interaction of the CPU core, caches, and bus interface. The clock generator unit replaces the external delay line required in R3000 and R3001 based applications.

### Instruction Cache

The current family includes two different instruction cache sizes: the R3051 and R3051E each contain 4kB of instruction cache, and the R3052/R3052E each contain 8kB of instruction cache. In all devices, the instruction cache is organized with a line size of 16 bytes (four entries). These relatively large caches achieve hit rates in excess of 95% in most applications, and substantially contributes to the performance inherent in the R3051 family. The cache is implemented as a direct mapped cache, and is capable of caching instructions from anywhere within the 4GB physical address space. The cache is implemented using physical addresses (rather than virtual addresses), and thus does not require flushing on context switch.

### Data Cache

The R3051 family incorporates an on-chip data cache of 2kB, organized as a line size of 4 bytes (one word). This relatively large data cache achieves hit rates in excess of 90% in most applications, and contributes substantially to the performance inherent in the R3051 family. As with the instruction cache, the data cache is implemented as a direct mapped physical address cache. The cache is capable of mapping any word within the 4GB physical address space.

The data cache is implemented as a write through cache, to insure that main memory is always consistent with the internal cache. In order to minimize processor stalls due to data write operations, the bus interface unit incorporates a 4-deep write buffer which captures address and data at the processor execution rate, allowing it to be retired to main memory at a much slower rate without impacting system performance.

### Bus Interface Unit

The R3051 family uses its large internal caches to provide the majority of the bandwidth requirements of the execution engine, and thus can utilize a simple bus interface connected to slow memory devices.

The R3051 family bus interface utilizes a 32-bit address and data bus multiplexed onto a single set of pins. The bus interface unit also provides an ALE (Address Latch Enable) output signal to de-multiplex the AD bus, and simple handshake signals to process CPU read and write requests. In addition to the read and write interface, the R3051 family incorporates a DMA arbiter, to allow an external master to control the external bus.

The R3051 family incorporates a 4-deep write buffer to decouple the speed of the execution engine from the speed of the memory system. The write buffers capture and FIFO processor address and data information in store operations, and present it to the bus interface as write transactions at the rate the memory system can accommodate.

The R3051 family read interface performs both single word reads and quad word reads. Single word reads work with a simple handshake, and quad word reads can either utilize the simple handshake (in lower performance, simple systems) or utilize a tighter timing mode when the memory system can burst data at the processor clock rate. Thus, the system designer can choose to utilize page or nibble mode DRAMs (and possibly use interleaving, if desired, in high-performance systems), or use simpler techniques to reduce complexity.

In order to accommodate slower quad word reads, the R3051 family incorporates a 4-deep read buffer FIFO, so that the external interface can queue up data within the processor before releasing it to perform a burst fill of the internal caches.

## SYSTEM USAGE

The IDT R3051 family has been specifically designed to easily connect to low-cost memory systems. Typical low-cost memory systems utilize slow EPROMs, DRAMs, and application specific peripherals. These systems may also typically contain large, slow static RAMs, although the IDT R3051 family has been designed to not specifically require the use of external SRAMs.

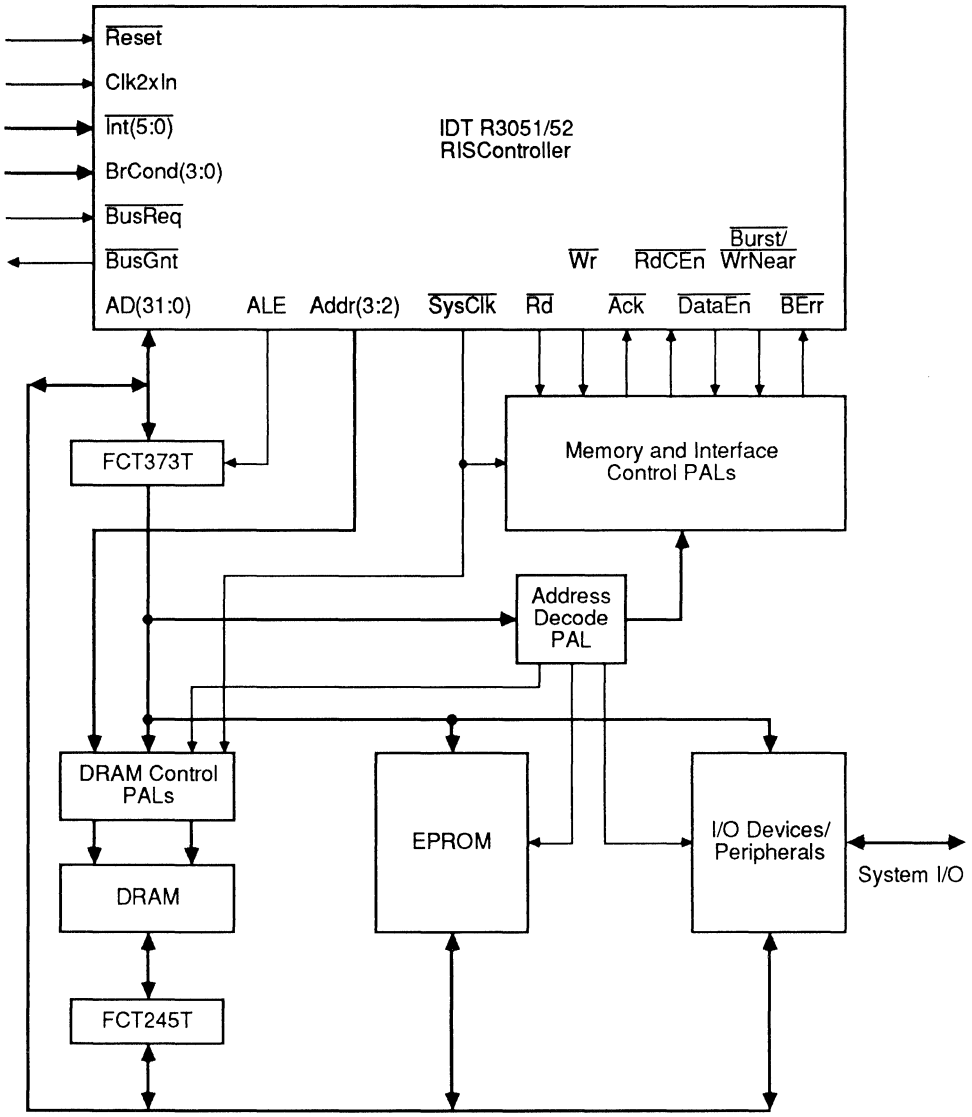



**Figure 1.2. System Diagram**

Figure 1.2 shows a typical system implementation using off-the-shelf logic devices. Transparent latches are used to de-multiplex the R3051/52 address and data busses from the AD bus. The data paths between the memory system elements and the R3051/52 AD bus is managed by simple octal devices. A small set of simple PALs is used to control the various data path elements, and to control the handshake between the memory devices and the R3051/52. IDT has also implemented the R3720/21/22 support chip set specifically tailored to R3051/52-based systems. This chip set directly interfaces the processor to DRAM, ROM, and I/O devices, eliminating the requirement for discrete logic chips and PAL devices, as illustrated in Figure 1.3.

Depending on the cost vs. performance tradeoffs appropriate to a given application, the system design engineer could include true burst support from the DRAM to provide for high-performance cache miss processing, or utilize a simpler, lower performance memory system to reduce cost and simplify the design.

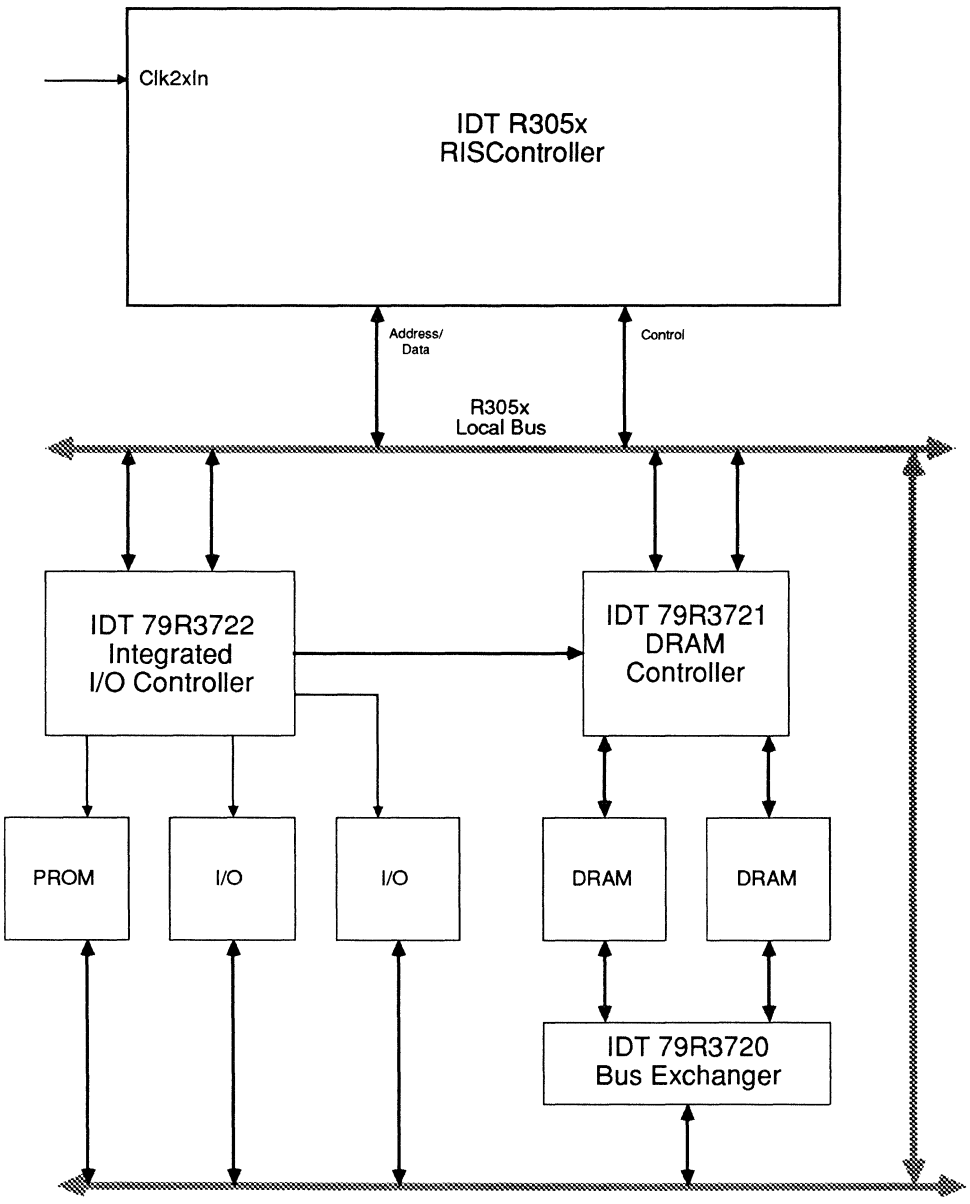



**Figure 1.3. R3720/21/22 Support Chip Set System**

## DEVELOPMENT SUPPORT

The IDT R3051 family is supported by a rich set of development tools, ranging from system simulation tools through PROM monitor and debug support, applications software and utility libraries, logic analysis tools, and sub-system modules.

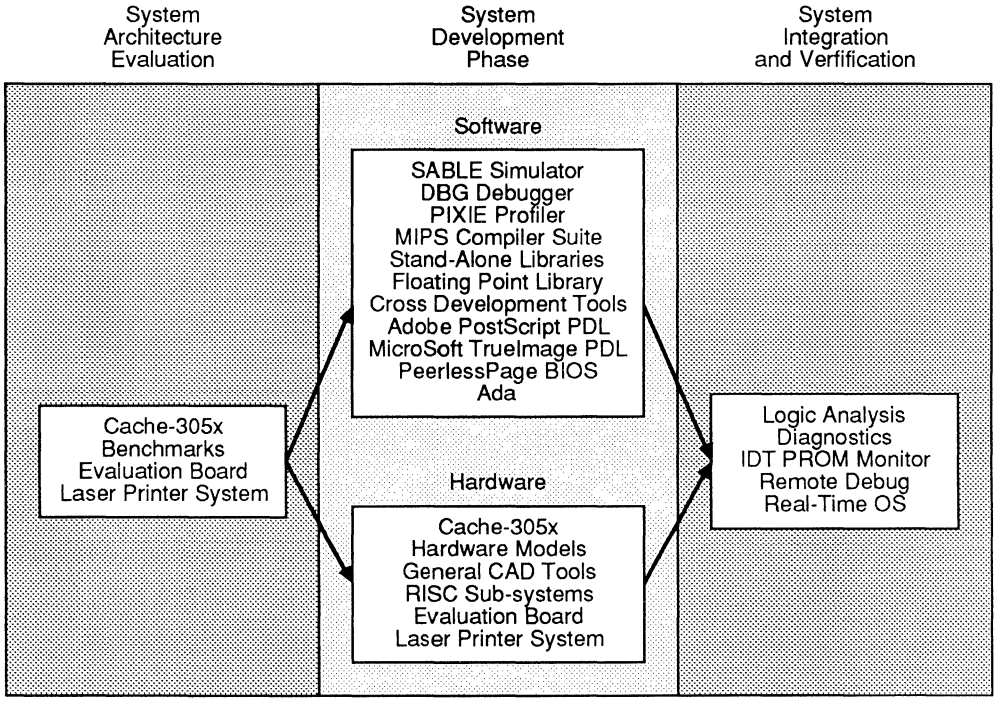



**Figure 1.4. Development Support**

Figure 1.4 is an overview of the system development process typically used when developing R3051 family applications. The R3051 family is supported in all phases of project development. These tools allow timely, parallel development of hardware and software for R3051 family based applications, and include tools such as:

- A program, Cache-R305x, which allows the performance of an R3051/52 based system to be modeled and understood without requiring actual hardware.
- Sable, an instruction set simulator.
- Optimizing compilers from MIPS, the acknowledged leader in optimizing compiler technology.
- Cross development tools, available in a variety of development environments.
- The high-performance IDT floating point library software.
- The IDT Evaluation Board, which includes RAM, EPROM, I/O, and the IDT PROM Monitor.
- The IDT Laser Printer System board, which directly drives a low-cost print engine, and runs Microsoft TrueImage™ Page Description Language on top of PeerlessPage™ Advanced Printer Controller BIOS.
- Adobe PostScript™ Page Description Language, ported to the R3000 instruction set, runs on the IDT R3051 family.
- The IDT Prom Monitor, which implements a full prom monitor (diagnostics, remote debug support, peek/poke, etc.).

## PERFORMANCE OVERVIEW

The R3051 family achieves a very high-level of performance. This performance is based on:

- An efficient execution engine. The CPU performs ALU operations and store operations in single cycle, and has an effective load time of 1.3 cycles, and branch execution rate of 1.5 cycles (based on the ability of the compilers to avoid software interlocks). Thus, the execution engine achieves over 35 MIPS performance when operating out of cache.
- Large on-chip caches. The R3051 family contains caches which are substantially larger than those on the majority of today's microprocessors. These large caches minimize the number of bus transactions required, and allow the R3051 family to achieve actual sustained performance very close to its peak execution rate.
- Autonomous multiply and divide operations. The R3051 family features an on-chip integer multiplier/divide unit which is separate from the other ALU. This allows the R3051/52 to perform multiply or divide operations in parallel with other integer operations, using a single multiply or divide instruction rather than "step" operations.
- Integrated write buffer. The R3051 family features a four deep write buffer, which captures store target addresses and data at the processor execution rate and retires it to main memory at the slower main memory access rate. Use of on-chip write buffers eliminates the need for the processor to stall when performing store operations.
- Burst read support. The R3051 family enables the system designer to utilize page mode or nibble mode RAMs when performing read operations to minimize the main memory read penalty and increase the effective cache hit rates.

These techniques combine to allow the processor to achieve over 28 MIPS integer performance, and 64,000 dhrystones without the use of external caches or zero wait-state memory devices.

The performance difference between the R3051 and R3052 depends on the application software and the design of the memory system. Performance differences in the range of 5 to 25% have been seen in various applications. Since both devices are pin and software compatible, the system designer has maximum freedom in trading between performance and cost. The development tool Cache-305x allows the system designer to analyze and understand the performance difference between these devices in his application.

## INTRODUCTION

The IDT R3051 family contains the same basic execution core as the IDT MIPS R3000 and the IDT R3001. In addition to being able to run software written for either of these processors, this enables the R3051 family to achieve dramatic levels of performance, based on the efficiency of the execution engine.

This chapter gives an overview of the MIPS-I architecture implemented in the R3051 family, and discusses the programmers model for this device. Further detail is available in the book "mips RISC Architecture", available from IDT.

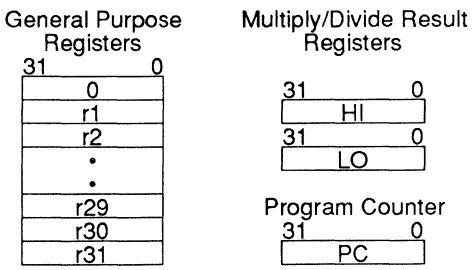
## R3051 FAMILY PROCESSOR FEATURES OVERVIEW

The R3051 family has many of the same attributes of the IDT R3000/R3001, at a higher level of integration geared to lower system cost. These features include:

- **Full 32-bit Operation.** The R3051 family contains thirty-two 32-bit registers, and all instructions and addresses are 32 bits.
- **Efficient Pipelining.** The CPU utilizes a 5-stage pipeline design to achieve an execution rate approaching one instruction per cycle. Pipeline stalls, hazards, and exceptional events are handled precisely and efficiently.
- **Large On-Chip Instruction and Data Caches.** The R3051 family utilizes large on-chip caches to provide high-bandwidth to the execution engine. The large size of the caches insures high hit rates, minimizing stalls due to cache miss processing and dramatically contributing to overall performance. Both the instruction and data cache can be accessed during a single CPU cycle.
- **On-chip Memory Management.** The IDT R3051/52"E" utilizes the same memory management scheme as the R3000/R3001, providing a 64 fully-associative TLB to provide fast virtual to physical address translation of the 4GB address space. The base IDT R3051/52 does not utilize the TLB, but performs fixed segment-based mapping of the virtual space to physical addresses.

## R3051 FAMILY CPU REGISTERS OVERVIEW

The IDT R3051 family provides 32 general purpose 32-bit registers, a 32-bit Program Counter, and two dedicated 32-bit registers which hold the result of an integer multiply or divide operation. The CPU registers, illustrated in Figure 2.1, are discussed later in this chapter.




4000 drw 01

**Figure 2.1. CPU Registers**

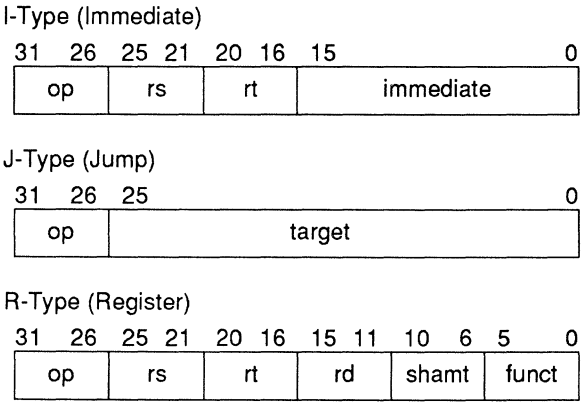
Note that the MIPS-I architecture does not use a traditional Program Status Word (PSW) register. The functions normally provided by such a register are instead provided through the use of "Set" instructions and conditional branches. By avoiding the use of traditional condition codes, the architecture can be more finely pipelined. This, coupled with the fine granularity of the instruction set, allows the compilers to achieve dramatically higher levels of optimizations than for traditional architectures.

Overflow and exceptional conditions are then handled through the use of the on-chip *Status* and *Cause* registers, which reside on-chip as part of the System Control Co-Processor (Co-Processor 0). These registers contain information about the run-time state of the machine, and any exceptional conditions it has encountered.

## INSTRUCTION SET OVERVIEW

All R3051 family instructions are 32-bits long, and there are only three basic instruction formats. This approach dramatically simplifies instruction decoding, permitting higher frequency operation. More complicated (but less frequently used) operations and addressing modes are synthesized by the assembler, using sequences of the basic instruction set. This approach enables object code optimizations at a finer level of resolution than achievable in micro-coded CPU architectures.

Figure 2.2 shows the instruction set encoding used by the MIPS architecture. This approach simplifies instruction decoding in the CPU.




where:

| | |
|---|---|
| op | is a 6-bit operation code |
| rs | is a five bit source register specifier |
| rt | is a 5-bit target register or branch condition |
| immediate | is a 16-bit immediate, or branch or address displacement |
| target | is a 26-bit jump target address |
| rd | is a 5-bit destination register specifier |
| shamt | is a 5-bit shift amount |
| funct | is a 6-bit function field |

4000 drw 02

**Figure 2.2. Instruction Encoding**

The R3051 family instruction set can be divided into the following basic groups:

- **Load/Store** instructions move data between memory and the general registers. They are all encoded as "I-Type" instructions, and the only addressing mode implemented is base register plus signed, immediate offset. This directly enables the use of three distinct addressing modes: register plus offset; register direct; and immediate.

- **Computational** instructions perform arithmetic, logical, and shift operations on values in registers. They are encoded as either "R-Type" instructions, when both source operands as well as the result are general registers, and "I-Type", when one of the source operands is a 16-bit immediate value. Computational instructions use a three address format, so that operations don't needlessly interfere with the contents of source registers.

- **Jump and Branch** instructions change the control flow of a program. A Jump instruction can be encoded as a "J-Type" instruction, in which case the Jump target address is a paged absolute address formed by combining the 26-bit immediate value with four bits of the Program Counter. This form is used for subroutine calls.

  Alternately, Jumps can be encoded using the "R-Type" format, in which case the target address is a 32-bit value contained in one of the general registers. This form is typically used for returns and dispatches.

  Branch operations are encoded as "I-Type" instructions. The target address is formed from a 16-bit displacement relative to the Program Counter.

  The Jump and Link instructions save a return address in Register r31. These are typically used as subroutine calls, where the subroutine return address is stored into r31 during the call operation.

- **Co-Processor** instructions perform operations on the co-processor set. Co-Processor Loads and Stores are always encoded as "I-Type" instructions; co-processor operational instructions have co-processor dependent formats.

  In the R3051 family, the System Control Co-Processor (CP0) contains registers which are used in memory management and exception handling.

  Additionally, the R3051 family implements four BrCond inputs. Software can use the Branch on Co-Processor Condition instructions to test the state of these external inputs, and thus they may be used like general purpose input ports.

- **Special** instructions perform a variety of tasks, including movement of data between special and general registers, system calls, and breakpoint operations. They are always encoded as "R-Type" instructions.

Table 2.1 lists the instruction set mnemonics of the R3051 family. More detail on these operations is presented later in this chapter. For further detail, consult "mips RISC Architecture", or one of the language programming guides, available from IDT.

| OP | Description | OP | Description |
|---|---|---|---|
| | **Load/Store Instructions** | | **Multiply/Divide Instructions** |
| LB | Load Byte | MULT | Multiply |
| LBU | Load Byte Unsigned | MULTU | Multiply Unsigned |
| LH | Load Halfword | DIV | Divide |
| LHU | Load Halfword Unsigned | DIVU | Divide Unsigned |
| LW | Load Word | | |
| LWL | Load Word Left | MFHI | Move From HI |
| LWR | Load Word Right | MTHI | Move To HI |
| SB | Store Byte | MFLO | Move From LO |
| SH | Store Halfword | MTLO | Move To LO |
| SW | Store Word | | |
| SWL | Store Word Left | | **Jump and Branch Instructions** |
| SWR | Store Word Right | J | Jump |
| | | JAL | Jump and Link |
| | **Arithmetic Instructions** | JR | Jump to Register |
| | **(ALU Immediate)** | JALR | Jump and Link Register |
| ADDI | Add Immediate | BEQ | Branch on Equal |
| ADDIU | Add Immediate Unsigned | BNE | Branch on Not Equal |
| SLTI | Set on Less Than Immediate | BLEZ | Branch on Less than or Equal to Zero |
| SLTIU | Set on Less Than Immediate Unsigned | BGTZ | Branch on Greater Than Zero |
| ANDI | AND Immediate | BLTZ | Branch on Less Than Zero |
| ORI | OR Immediate | BGEZ | Branch on Greater Than or Equal to Zero |
| XORI | Exclusive OR Immediate | | |
| LUI | Load Upper Immediate | BLTZAL | Branch on Less Than Zero and Link |
| | | BGEZAL | Branch on Greater Than or Equal to Zero and Link |
| | **Arithmetic Instructions** | | |
| | **(3-operand, register-type)** | | **Special Instructions** |
| ADD | Add | | |
| ADDU | Add Unsigned | SYSCALL | System Call |
| SUB | Subtract | BREAK | Break |
| SUBU | Subtract Unsigned | | |
| SLT | Set on Less Than | | **Coprocessor Instructions** |
| SLTU | Set on Less Than Unsigned | LWCz | Load Wrod from Coprocessor |
| AND | AND | SWCz | Store Word to Coprocessor |
| OR | OR | MTCz | Move To Coprocessor |
| XOR | Exclusive OR | MFCz | Move From Coprocessor |
| NOR | NOR | CTCz | Move Control To Coprocessor |
| | | CFCz | Move Control From Coprocessor |
| | **Shift Instructions** | COPz | Coprocessor Operation |
| SLL | Shift Left Logical | BCzT | Branch on Coprocessor z True |
| SRL | Shift Right Logical | BCzF | Branch on Coprocessor z False |
| SRA | Shift Right Arithmetic | | |
| SLLV | Shift Left Logical Variable | | **System Control Coprocessor** |
| SRLV | Shift Right Logical Variable | | **(CP0) Instructions** |
| SRAV | Shift Right Arithmetic Variable | MTC0 | Move To CP0 |
| | | MFC0 | Move From CP0 |
| | | TLBR | Read indexed TLB entry |
| | | TLBWI | Write indexed TLB entry |
| | | TLBWR | Write Random TLB entry |
| | | TLBP | Probe TLB for matching entry |
| | | RFE | Restore From Exception |

4000 tbl 01

**Table 2.1. R3051 Family Instruction Set Mnemonics**

## R3051 FAMILY PROGRAMMING MODEL

This section describes the organization of data in the general registers and in memory, and discusses the set of general registers available. A summary description of all of the CPU registers is presented.

### Data Formats and Addressing

The R3051 family defines a word as 32-bits, a half-word as 16-bits, and a byte as 8-bits. The byte ordering convention is configurable during hardware reset (Chapter 9) into either a *big-endian* or *little-endian* convention.

When configured as a big-endian system, byte 0 is always the most significant (leftmost) byte in a word. This is the order used in MC680x0® microprocessors, and systems from MIPS.

When configured as a little-endian system, byte 0 is always the least significant (rightmost) byte in a word. This is compatible with the iAPX® x86 microprocessors and systems from Digital Equipment Corporation.

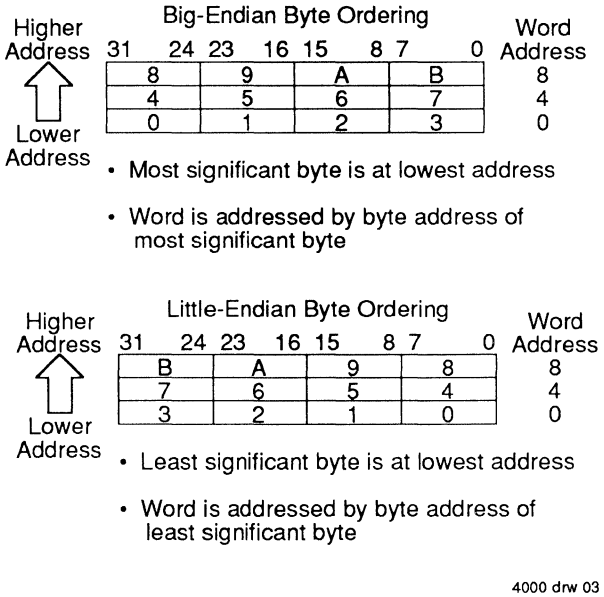



**Figure 2.3. Byte Ordering Conventions**

Figure 2.3 shows the ordering of bytes within words and the ordering of words within multiple word structures for the big-endian and little-endian conventions.

The R3051 family uses byte addressing for all accesses, including half-word and word. The R3051 family has alignment constraints that require half-word access to be aligned on an even byte boundary, and word access to be aligned on a modulo-4 byte boundary. Thus, in big-endian systems, the address of a multiple-byte data item is the address of the most-significant byte, while in little-endian systems it is the address of the least-significant byte of the structure.

For compatibility with older programs written for 8- or 16-bit machines, the MIPS instruction set provides special instructions for addressing 32-bit words which are not aligned on 4-byte boundaries. These instructions, which are Load/Store Left/Right, are used in pairs to provide addressing of misaligned words. This effectively means that these types of data movements require only one-additional instruction cycle over that required for properly aligned words, and provides a much more efficient way of dealing with this case than is possible using sequences of loads/stores and shift operations. Figure 2.4 shows the bytes accessed when addressing a mis-aligned word with a byte address of 3, for each of the two byte ordering conventions.

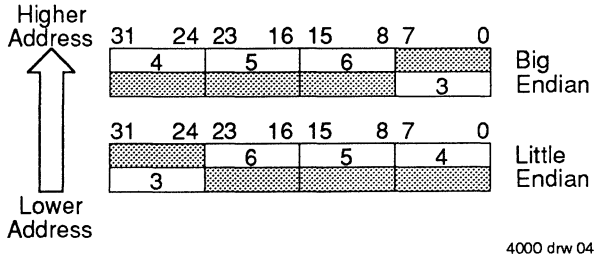



**Figure 2.4. Unaligned Words**

### R3051 Family CPU General Registers

The R3051 family contains 32-general registers, each containing a single 32-bit word. The 32 general registers are treated symmetrically (orthogonally), with two notable exceptions: general register r0 is hardwired to a zero value, and r31 is used as the link register in Jump and Link instructions

Register r0 maintains the value zero under all conditions when used as a source register, and discards data written to it. Thus, instructions which attempt to write to it may be used as No-Op Instructions. The use of a register wired to the zero value allows the simple synthesis of different addressing modes, no-ops, register or memory clear operations, etc., without requiring expansion of the basic instruction set.

Register r31 is used is the link register in jump and link instructions. These instructions are used in subroutine calls, and the subroutine return address is placed in register r31. This register can be written to or read as a normal register in other operations.

In addition to the general registers, the CPU contains two registers (HI and LO) which store the double-word, 64-bit result of integer multiply operations, and the quotient and remainder of integer divide operations.

### R3051 Family CP0 Special Registers

In addition to the general CPU registers, the R3051 family contains a number of special registers on-chip. These registers logically reside in the on-chip System Control Co-processor CP0, and are used in memory management and exception handling.

Table 2.2 shows the logical CP0 address of each of the registers. The format of each of these registers, and their use, is discussed in Chapter 4 (Memory Management), and Chapter 5 (Exception Handling).

| Number | Mnemonic | Description |
|---|---|---|
| 0 | Index | Programmable pointer into on-chip TLB array |
| 1 | Random | Pseudo-random pointer into on-chip TLB array (read only) |
| 2 | EntryLo | Low-half of TLB entry |
| 3 | Reserved | |
| 4 | Context | Pointer to kernel virtual Page Table Entry Table |
| 5-7 | Reserved | |
| 8 | BadVAddr | Bad virtual address |
| 9 | Reserved | |
| 10 | EntryHi | High-half of TLB entry |
| 11 | Reserved | |
| 12 | SR | Status Register |
| 13 | Cause | Cause of last exception |
| 14 | EPC | Exception Program Counter |
| 15 | PRId | Processor Revision Identifier |
| 16-31 | Reserved | |

4000 tbl 02

**Table 2.2. R3051 Family CP0 Registers**

### R3051 Family Operating Modes

The R3051 family supports two different operating modes: *User* and *Kernel* modes. The R3051/52 normally operates in User mode until an exception is detected, forcing it into kernel mode. It remains in Kernel mode until a Return From Exception (RFE) instruction is executed, returning it to its previous operation mode.

The processor supports these levels of protection by segmenting the 4GB virtual address space into 4 distinct segments. One segment is accessible from either the User state or the Kernel mode, and the other three segments are only accessible from kernel mode.

In addition to providing memory address protection, the kernel can protect the co-processors (in the case of the R3051 family, CP0) from access or modification by the user task.

Finally, the R3051 family supports the execution of user programs with the opposite byte ordering (Reverse Endianness) of the kernel, facilitating the exchange of programs and data between dissimilar machines.

Chapter 3 discusses the memory management facilities of the processor.

### R3051 Family Pipeline Architecture

The IDT R3051 family uses the same basic pipeline structure as that implemented in the R3000 and R3001. Thus, the execution of a single instruction is performed in five distinct steps.

- Instruction Fetch (IF). In this stage, the instruction virtual address is translated to a physical address and the instruction is read from the internal Instruction Cache.
- Read (RD). During this stage, the instruction is decoded and required operands are read from the on-chip register file.
- ALU. The required operation is performed on the instruction operands.
- Memory Access (MEM). If the instruction was a load or store, the Data Cache is accessed. Note that there is a skew between the instruction cycle which fetches the instruction and the one in which the required data transfer occurs. This skew is a result of the intervening pipestages.
- Write Back (WB). During the write back pipestage, the results of the ALU stage operation are updated into the on-chip register file.

Each of these pipestages requires approximately one CPU cycle, as shown in Figure 2.5. Parts of some operations lap into the next cycle, while other operations require only 1/2 cycle.

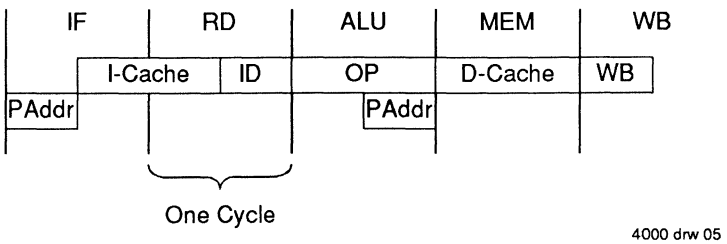



**Figure 2.5. 5-Stage Pipeline**

The net effect of the pipeline structure is that a new instruction can be initiated every clock cycle. Thus, the execution of five instructions at a time is overlapped, as shown in Figure 2.6.

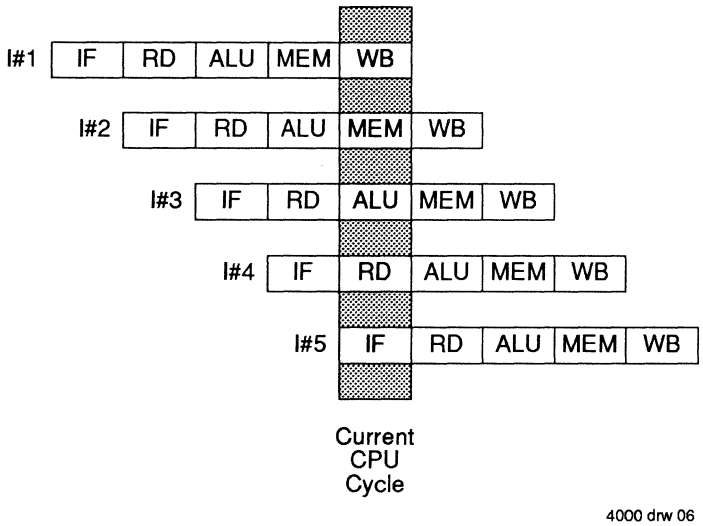



**Figure 2.6. 5-Instructions per Clock Cycle**

The pipeline operates efficiently, because different CPU resources such as address and data bus access, ALU operations, and the register file, are utilized on a non-interfering basis.

**Pipeline Hazards**

In a pipelined machine such as the R3051/52, there are certain instructions which, based on the pipeline structure, can potentially disrupt the smooth operation of the pipeline. The basic problem is that the current pipestage of an instruction may require the result of a previous instruction, still in the pipeline, whose result is not yet available. This class of problems is referred to as pipeline hazards.

An example of a potential pipeline hazard occurs when a computational instruction (instruction n+1) requires the result of the immediately prior instruction (instruction n). Instruction n+1 wants to access the register file during the RF pipestage. However, instruction n has not yet completed its register writeback operation, and thus the current value is not available directly from the register file. In this case, special logic within the execution engine forwards the result of instruction n's ALU operation to instruction n+1, prior to the true writeback operation. The pipeline is undisturbed, and no pipeline *stalls* need to occur.

Another example of a pipeline hazard handled in hardware is the integer multiply and divide operations. If an instruction attempts to access the HI or LO registers prior to the completion of the multiply or divide, that instruction will be *interlocked* (held off) until the multiply or divide operation completes. Thus, the programmer is isolated from the actual execution time of this operation. The optimizing compilers attempt to schedule as many instructions as possible between the start of the multiply/divide and the access of its result, to minimize stalls.

However, not all pipeline hazards are handled in hardware. There are two categories of instructions which require software intervention to insure logical operation. The optimizing compilers (and peephole scheduler of the assembler) are capable of insuring proper execution. These two instruction classes are:

- Load instructions have a delay, or latency, of one cycle before the data loaded from memory is available another instruction. This is because the ALU stage of the immediately subsequent instruction is processed simultaneously with the Data Cache access of the load operation. Figure 2.7 illustrates the cause of this delay slot.

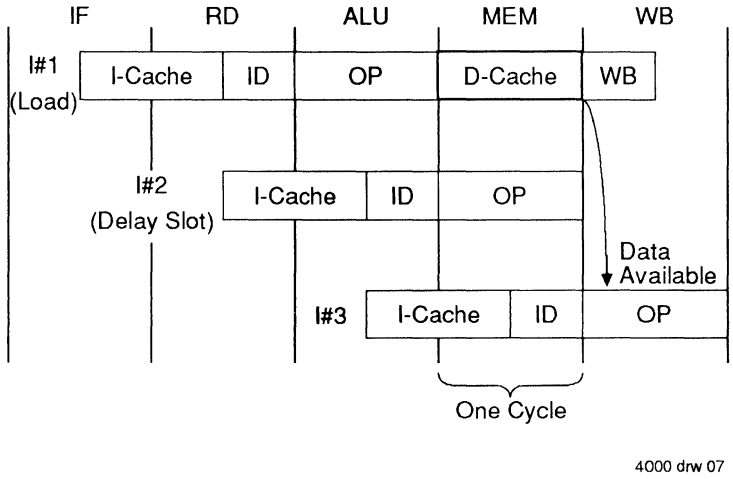



**Figure 2.7. Load Delay**

- Jump and Branch instructions have a delay of one cycle before the program flow change can occur. This is due to the fact that the next instruction is fetched prior to the decode and ALU stage of the jump/branch operation. Figure 2.8 illustrates the cause of this delay slot.

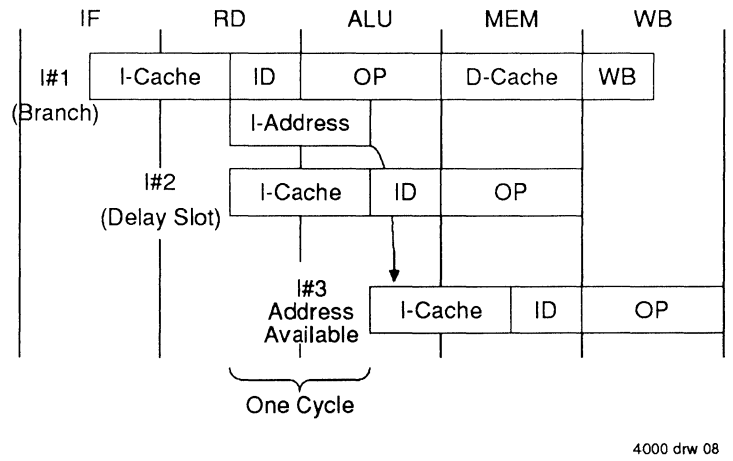



**Figure 2.8. Branch Delay**

The R3051/52 continues execution, despite the delay in the operation. Thus, loads, jumps and branches do not disrupt the pipeline flow of instructions, and the processor always executes the instruction immediately following one of these "delayed" instructions.

Rather than include extensive pipeline control logic, the MIPS-I instruction set gives responsibility for dealing with "delay slots" to software. Thus, the peephole optimizer (which can be performed as part of compilation or assembly) can re-order the code to insure that the instruction in the delay slot does not require the logical result of the "delayed" instruction. In the worst case, a NOP can be inserted to guarantee proper software execution.

Chapter 5 discusses the impact of pipelining on exception handling. In general, when an instruction causes an exception, it is desirable for all instructions initiated prior to that instruction to complete, and all subsequent instructions to abort. This insures that the machine state presented to the exception handler reflects the logical state that existed at the time the exception was detected. In addition, it is desirable to avoid requiring software to explicitly manage the pipeline when handling or returning from exceptions. The IDT R3051/52 pipeline is designed to properly manage exceptional events.

## R3051 FAMILY INSTRUCTION SET SUMMARY

This section provides an overview of the R3051 family instruction set by presenting each category of instructions in a tabular summary form. Refer to the "mips RISC Architecture" reference for a detailed description of each instruction.

### Instruction Formats

Every R3051 family instruction consists of a single word (32 bits) aligned on a word boundary. There are only three instruction formats as shown in Figure 2.2. This approach simplifies instruction decoding. More complicated (less frequently used) operations and addressing modes are synthesized by the compilers.

### Instruction Notational Conventions

In this manual, all variable sub-fields in an instruction format (such as rs, rt, immediate, and so on) are shown in lower-case names.

For the sake of clarity, an alias is sometimes used for a variable sub-field in the formats of specific instructions. For example, "base" rather than "rs" is used in the format for Load and Store instructions. Such an alias is always lower case, since it refers to a variable sub-field.

Instruction opcodes are shown in all upper case.

The actual bit encoding for all the mnemonics is specified at the end of this chapter.

### Load and Store Instructions

**Load/Store** instructions move data between memory and general registers. They are all I-type instructions. The only addressing mode directly supported is base register plus 16-bit signed immediate offset. This can be used to directly implement immediate addressing (using the r0 register) or register direct (using an immediate offset value of zero).

All load operations have a latency of one instruction. That is, the data being loaded from memory into a register is not available to the instruction that immediately follows the load instruction: the data is available to the second instruction after the load instruction. An exception is the target register for the "load word left" and "load word right" instructions, which may be specified as the same register used as the destination of a load instruction that immediately precedes it.

**Big-Endian**

| Size | AdrLo(1) | AdrLo(0) | BE(3) Data(31:24) | BE(2) Data(23:16) | BE(1) Data(15:8) | BE(0) Data(7:0) |
|---|---|---|---|---|---|---|
| Word | 0 | 0 | Yes | Yes | Yes | Yes |
| Tri-Byte | 0 | 0 | Yes | Yes | Yes | No |
| Tri-Byte | 0 | 1 | No | Yes | Yes | Yes |
| 16-Bit | 0 | 0 | Yes | Yes | No | No |
| 16-Bit | 1 | 0 | No | No | Yes | Yes |
| Byte | 0 | 0 | Yes | No | No | No |
| Byte | 0 | 1 | No | Yes | No | No |
| Byte | 1 | 0 | No | No | Yes | No |
| Byte | 1 | 1 | No | No | No | Yes |

**Little-Endian**

| Size | AdrLo(1) | AdrLo(0) | BE(3) Data(31:24) | BE(2) Data(23:16) | BE(1) Data(15:8) | BE(0) Data(7:0) |
|---|---|---|---|---|---|---|
| Word | 0 | 0 | Yes | Yes | Yes | Yes |
| Tri-Byte | 0 | 0 | No | Yes | Yes | Yes |
| Tri-Byte | 0 | 1 | Yes | Yes | Yes | No |
| 16-Bit | 0 | 0 | No | No | Yes | Yes |
| 16-Bit | 1 | 0 | Yes | Yes | No | No |
| Byte | 0 | 0 | No | No | No | Yes |
| Byte | 0 | 1 | No | No | Yes | No |
| Byte | 1 | 0 | No | Yes | No | No |
| Byte | 1 | 1 | Yes | No | No | No |

4000 tbl 03

**Table 2.3. Byte Addressing in Load/Store Operations**

The Load/Store instruction opcode determines the size of the data item to be loaded or stored as shown in Table 2.1. Regardless of access type or byte-numbering order (endian-ness), the address specifies the byte which has the smallest byte address of all bytes in the addressed field. For a big-endian access, this is the most significant byte; for a little-endian access, this is the least significant byte. Note that in an R3051/52 based system, the endianness of a given access is dynamic, in that the RE (Reverse Endianness) bit of the Status Register can be used to force user space accesses of the opposite byte convention of the kernel.

The bytes within the addressed word that are used can be determined directly from the access size and the two low-order bits of the address, as shown in Table 2.3. Note that certain combinations of access type and low-order address bits can never occur: only the combinations shown in Table 2.3 are permissible. The R3051 family indicates which bytes are being accessed by the byte-enable ($\overline{BE}$) bus.

Table 2.4 shows the load/store instructions supported by the MIPS ISA.

| Instruction | Format and Description |
|---|---|
| Load Byte | LB *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Sign-extend contents of addressed byte and load into *rt*. |
| Load Byte Unsigned | LBU *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Zero-extend contents of addressed byte and load into *rt*. |
| Load Halfword | LH *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Sign-extend contents of addressed byte and load into *rt*. |
| Load Halfword Unsigned | LHU *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Zero-extend contents of addressed byte and load into *rt*. |
| Load Word | LW rt, offset (base)<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Load contents of addressed word into register *rt*. |
| Load Word Left | LWL rt, offset (base)<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Shift addressed word left so that addressed byte is leftmost byte of a word.<br>Merge bytes from memory with contents of register rt and load result into register rt. |
| Load Word Right | LWR rt, offset (base)<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Shift addressed word right so that addressed byte is rightmost byte of a word.<br>Merge bytes from memory with contents of register rt and load result into register rt. |
| Store Byte | SB *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Store least significant byte of register *rt* at addressed location. |
| Store Halfword | SH *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Store least significant halfword of register *rt* at addressed location. |
| Store Word | SW *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Store least significant word of register *rt* at addressed location. |
| Store Word Left | SWL *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Shift contents of register *rt right* so that leftmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte. |
| Store Word Right | SWR *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to contents of register *base* to form address.<br>Shift contents of register *rt left* so that rightmost byte of the word is in position of addressed byte. Store bytes containing original data into corresponding bytes at addressed byte. |

4000 tbl 04

**Table 2.4. Load and Store Instructions in the R3051 Family**

### Computational Instructions

**Computational** instructions perform arithmetic, logical and shift operations on values in registers. They occur in both R-type (both operands are registers) and I-type (one operand is a 16-bit immediate) formats. There are four categories of computational instructions:

- **ALU Immediate** instructions are summarized in Table 2.5a.
- **3-Operand Register-Type** instructions are summarized in Table 2.5b.
- **Shift** instructions are summarized in Table 2.5c.
- **Multiply/Divide** instructions are summarized in Table 2.5d.

| Instruction | Format and Description |
|---|---|
| ADD Immediate | ADDI *rt, rs, immediate*<br>Add 16-bit sign-extended *immediate* to register *rs* and place 32-bit result in register *rt*. Trap on two's complement overflow. |
| ADD Immediate Unsigned | ADDIU *rt, rs, immediate*<br>Add 16-bit sign-extended *immediate* to register *rs* and place 32-bit result in register *rt*. Do not trap on overflow. |
| Set on Less Than Immediate | SLTI *rt, rs, immediate*<br>Compare 16-bit sign-extended *immediate* with register *rs* as signed 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0.<br>Place result in register *rt*. |
| Set on Less Than Unsigned Immediate | SLTIU *rt, rs, immediate*<br>Compare 16-bit sign-extended *immediate* with register *rs* as unsigned 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0. Place result in register *rt*. Do not trap on overflow. |
| AND Immediate | ANDI *rt, rs, immediate*<br>Zero-extend 16-bit *immediate*, AND with contents of register *rs* and place result in register *rt*. |
| OR Immediate | ORI *rt, rs, immediate*<br>Zero-extend 16-bit *immediate*, OR with contents of register *rs* and place result in register *rt*. |
| Exclusive OR Immediate | XORI *rt, rs, immediate*<br>Zero-extend 16-bit *immediate*, exclusive OR with contents of register *rs* and place result in register *rt*. |
| Load Upper Immediate | LUI *rt, immediate*<br>Shift 16-bit *immediate* left 16 bits. Set least significant 16 bits of word to zeroes. Store result in register *rt*. |

4000 tbl 05

**Table 2.5a. ALU Immediate Operations in the R3051 Family**

| Instruction | Format and Description |
|---|---|
| Add | ADD *rd, rs, rt*<br>Add contents of registers *rs* and *rt* and place 32-bit result in register *rd*. Trap on two's complement overflow. |
| ADD Unsigned | ADDU *rd, rs, rt*<br>Add contents of registers *rs* and *rt* and place 32-bit result in register *rd*. Do not trap on overflow. |
| Subtract | SUB *rd, rs, rt*<br>Subtract contents of registers *rt* and *rs* and place 32-bit result in register *rd*. Trap on two's complement overflow. |
| Subtract Unsigned | SUBU *rd, rs, rt*<br>Subtract contents of registers *rt* and *rs* and place 32-bit result in register *rd*. Do not trap on overflow. |
| Set on Less Than | SLT *rd, rs, rt*<br>Compare contents of register *rt* to register *rs* (as signed 32-bit integers).<br>If register *rs* is less than *rt*, result = 1; otherwise, result = 0. |
| Set on Less Than Unsigned | SLTU *rd, rs, rt*<br>Compare contents of register *rt* to register *rs* (as unsigned 32-bit integers). If register *rs* is less than *rt*, result = 1; otherwise, result = 0. |
| AND | AND *rd, rs, rt*<br>Bit-wise AND contents of registers *rs* and *rt* and place result in register *rd*. |
| OR | OR *rd, rs, rt*<br>Bit-wise OR contents of registers *rs* and *rt* and place result in register *rd*. |
| Exclusive OR | XOR *rd, rs, rt*<br>Bit-wise Exclusive OR contents of registers *rs* and *rt* and place result in register *rd*. |
| NOR | NOR *rd, rs, rt*<br>Bit-wise NOR contents of registers *rs* and *rt* and place result in register *rd*. |

4000 tbl 06

**Table 2.5b. Three Operand Register-Type Operations in the R3051 Family**

| Instruction | Format and Description |
|---|---|
| Shift Left Logical | SLL *rd, rt, shamt*<br>Shift contents of register *rt* left by *shamt* bits, inserting zeroes into low order bits. Place 32-bit result in register *rd*. |
| Shift Right Logical | SRL *rd, rt, shamt*<br>Shift contents of register *rt* right by *shamt* bits, inserting zeroes into high order bits. Place 32-bit result in register *rd*. |
| Shift Right Arithmetic | SRA *rd, rt, shamt*<br>Shift contents of register *rt* right by *shamt* bits, sign-extending the high order bits. Place 32-bit result in register *rd*. |
| Shift Left Logical Variable | SLLV *rd, rt, rs*<br>Shift contents of register *rt* left. Low-order 5 bits of register *rs* specify number of bits to shift. Insert zeroes into low order bits of *rt* and place 32-bit result in register *rd*. |
| Shift Right Logical Variable | SRLV *rd, rt, rs*<br>Shift contents of register *rt* right. Low-order 5 bits of register *rs* specify number of bits to shift. Insert zeroes into high order bits of *rt* and place 32-bit result in register *rd*. |
| Shift Right Arithmetic Variable | SRAV *rd, rt, rs*<br>Shift contents of register *rt* right. Low-order 5 bits of register *rs* specify number of bits to shift. Sign-extend the high order bits of *rt* and place 32-bit result in register *rd*. |

4000 tbl 07

**Table 2.5c. Shift Operations in the R3051 Family**

| Instruction | Format and Description |
|---|---|
| Multiply | MULT *rs, rt*<br>Multiply contents of registers *rs* and *rt* as twos complement values. Place 64-bit result in special registers HI/LO |
| Multiply Unsigned | MULTU *rs, rt*<br>Multiply contents of registers *rs* and *rt* as unsigned values. Place 64-bit result in special registers HI/LO |
| Divide | DIV *rs, rt*<br>Divide contents of register *rs* by *rt* treating operands as twos complements values. Place 32-bit quotient in special register LO, and 32-bit remainder in HI. |
| Divide Unsigned | DIVU *rs, rt*<br>Divide contents of register *rs* by *rt* treating operands as unsigned values. Place 32-bit quotient in special register LO, and 32-bit remainder in HI. |
| Move From HI | MFHI *rd*<br>Move contents of special register HI to register *rd.* |
| Move From LO | MFLO *rd*<br>Move contents of special register LO to register *rd.* |
| Move To HI | MTHI *rd*<br>Move contents of special register rd to special register HI. |
| Move To LO | MTLO *rd*<br>Move contents of register rd to special register LO. |

4000 tbl 08

**Table 2.5d. Multiply and Divide Operations in the R3051 Family**

## Jump and Branch Instructions

**Jump and Branch** instructions change the control flow of a program. All Jump and Branch instructions occur with a one instruction delay: that is, the instruction immediately following the jump or branch is always executed while the target instruction is being fetched from storage, regardless of whether the branch is to be taken.

An assembler has several possibilities for utilizing the branch delay slot productively:

- It can insert an instruction that logically precedes the branch instruction in the delay slot since the instruction immediately following the jump/branch effectively belongs to the block preceding the transfer instruction.
- It can replicate the instruction that is the target of the branch/jump into the delay slot provided that no side-effects occur if the branch falls through.
- It can move an instruction up from below the branch into the delay slot, provided that no side-effects occur if the branch is taken.
- If no other instruction is available, it can insert a NOP instruction in the delay slot.

The J-type instruction format is used for both jumps-and-links for subroutine calls. In this format, the 26-bit target address is shifted left two bits, and combined with high-order 4 bits of the current program counter to form a 32-bit absolute address.

The R-type instruction format which takes a 32-bit byte address contained in a register is used for returns, dispatches, and cross-page jumps.

Branches have 16-bit offsets relative to the program counter (I-type). Jump-and-Link and Branch-and-Link instructions save a return address in register 31.

Table 2.6a summarizes the R3051 family Jump instructions and Table 2.6b summarizes the Branch instructions.

| Instruction | Format and Description |
|---|---|
| Jump | J target<br>Shift 26-bit target address left two bits, combine with high-order 4 bits of PC and jump to address with a one instruction delay. |
| Jump and Link | JAL target<br>Shift 26-bit target address left two bits, combine with high-order 4 bits of PC and jump to address with a one instruction delay. Place address of instruction following delay slot in r31 (link register). |
| Jump Register | JR *rs*<br>Jump to address contained in register *rs* with a one instruction delay. |
| Jump and Link Register | JALR *rs, rd*<br>Jump to address contained in register *rs* with a one instruction delay. Place address of instruction following delay slot in *rd.* |

4000 tbl 09

**Table 2.6a. Jump Instructions in the R3051 Family**

| Instruction | Format and Description |
|---|---|
| | **Branch Target:** All Branch instruction target addresses are computed as follows: Add address of instruction in delay slot and the 16-bit offset (shifted left two bits and sign-extended to 32 bits). All branches occur with a delay of one instruction. |
| Branch on Equal | BEQ *rs, rt, offset*<br>Branch to target address if register *rs* equal to *rt* |
| Branch on Not Equal | BNE *rs, rt, offset*<br>Branch to target address if register *rs* not equal to *rt.* |
| Branch on Less than or Equal Zero | BLEZ *rs,offset*<br>Branch to target address if register *rs* less than or equal to 0. |
| Branch on Greater Than Zero | BGTZ *rs,offset*<br>Branch to target address if register *rs* greater than 0. |
| Branch on Less Than Zero | BLTZ *rs,offset*<br>Branch to target address if register *rs* less than 0. |
| Branch on Greater than or Equal Zero | BGEZ rs,*offset*<br>Branch to target address if register *rs* greater than or equal to 0. |
| Branch on Less Than Zero And Link | BLTZAL *rs, offset*<br>Place address of instruction following delay slot in register r31 (link register). Branch to target address if register *rs* less than 0. |
| Branch on greater than or Equal Zero And Link | BGEZAL *rs, offset*<br>Place address of instruction following delay slot in register r31 (link register). Branch to target address if register *rs* is greater than or equal to 0. |

4000 tbl 10

**Table 2.6b. Branch Instructions in the R3051 Family**

### Special Instructions

The two **Special** instructions let software initiate traps. They are always R-type. Table 2.7 summarizes the Special instructions.

| Instruction | Format and Description |
|---|---|
| System Call | SYSCALL<br>Initiates system call trap, immediately transferring control to exception handler. |
| Breakpoint | BREAK<br>Initiates breakpoint trap, immediately transferring control to exception handler. |

4000 tbl 11

**Table 2.7. Special Instructions in the R3051 Family**

### Co-processor Instructions

**Co-processor** instructions perform operations in the co-processors. Co-processor Loads and Stores are I-type. Co-processor computational instructions have co-processor-dependent formats (see co-processor manuals). For the R3051 family, the BCzT/F instructions are used to test the state of the BrCond inputs. Outside of these operations, the only co-processor operations of relevance are those targeted at the on-chip CP0.

Table 2.8 summarizes the Co-processor Instruction Set of the MIPS ISA.

| Instruction | Format and Description |
|---|---|
| Load Word to Co-processor | LWCz *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to *base* to form address. Load contents of addressed word into co-processor register *rt* of co-processor unit z. |
| Store Word from Co-processor | SWCz *rt, offset (base)*<br>Sign-extend 16-bit *offset* and add to *base* to form address. Store contents of co-processor register *rt* from co-processor unit z at addressed memory word. |
| Move To Co-processor | MTCz *rt, rd*<br>Move contents of CPU register *rt* into co-processor register *rd* of co-processor unit z. |
| Move from Co-processor | MFCz *rt,rd*<br>Move contents of co-processor register *rd* from co-processor unit z to CPU register *rt*. |
| Move Control To Co-processor | CTCz *rt,rd*<br>Move contents of CPU register *rt* into co-processor control register *rd* of co-processor unit z. |
| Move Control From Co-processor | CFCz *rt,rd*<br>Move contents of control register *rd* of co-processor unit z into CPU register *rt*. |
| Co-processor Operation | COPz *cofun*<br>Co-processor z performs an operation. The state of the R3051/52 is not modified by a co-processor operation. |
| Branch on Co-processor z True | BCzT *offset*<br>Compute a branch target address by adding address of instruction in the 16-bit *offset* (shifted left two bits and sign-extended to 32-bits). Branch to the target address (with a delay of one instruction) if co-processor z's condition line is true. |
| Branch on Co-processor z False | BCzF *offset*<br>Compute a branch target address by adding address of instruction in the 16-bit *offset* (shifted left two bits and sign-extended to 32-bits). Branch to the target address (with a delay of one instruction) if co-processor z's condition line is false. |

4000 tbl 12

**Table 2.8. Co-Processor Operations in the R3051 Family**

### System Control Co-processor (CP0) Instructions

**Co-processor 0** instructions perform operations on the System Control Co-processor (CP0) registers to manipulate the memory management and exception handling facilities of the processor. Memory Management is discussed in chapter 4; exception handling is covered in detail in chapter 5.

Table 2.9 summarizes the instructions available to work with CP0.

| Instruction | Format and Description |
|---|---|
| Move To CP0 | MTC0 *rt, rd*<br>Store contents of CPU register *rt* into register *rd* of CP0. This follows the convention of store operations. |
| Move From CP0 | MFC0 *rt, rd*<br>Load CPU register *rt* with contents of CP0 register *rd.* |
| Read Indexed TLB Entry | TLBR<br>Load *EntryHi* and *EntryLo* registers with TLB entry pointed at by *Index* register. |
| Write Indexed TLB Entry | TLBWI<br>Load TLB entry pointed at by *Index* register with contents of *EntryHi* and *EntryLo* registers. |
| Write Random TLB Entry | TLBWR<br>Load TLB entry pointed at by *Random* register with contents of *EntryHi* and *EntryLo* registers. |
| Probe TLB for Matching Entry | TLBP<br>Load *Index* register with address of TLB entry whose contents match *EntryHi* and *EntryLo.* If no TLB entry matches, set high-order bit of *Index* register. |
| Restore From Exception | RFE<br>Restore previous interrupt mask and mode bits of *status* register into current status bits. Restore old status bits into previous status bits. |

4000 tbl 13

**Table 2.9. System Control Co-Processor (CP0) Operations in the R3051 Family**

## R3051 FAMILY OPCODE ENCODING

Table 2.10 shows the opcode encoding for the R3051 family.

**OPCODE**

| 31..29 \ 28..26 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | SPECIAL | BCOND | J | JAL | BEQ | BNE | BLEZ | BGTZ |
| 1 | ADDI | ADDIU | SLTI | SLTIU | ANDI | ORI | XORI | LUI |
| 2 | COP0 | COP1 | COP2 | COP3 | † | † | † | † |
| 3 | † | † | † | † | † | † | † | † |
| 4 | LB | LH | LWL | LW | LBU | LHU | LWR | † |
| 5 | SB | SH | SWL | SW | † | † | SWR | † |
| 6 | LWC0 | LWC1 | LWC2 | LWC3 | † | † | † | † |
| 7 | SWC0 | SWC1 | SWC2 | SWC3 | † | † | † | † |

**SPECIAL**

| 5..3 \ 2..0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | SLL | † | SRL | SRA | SLLV | † | SRLV | SRAV |
| 1 | JR | JALR | † | † | SYSCALL | BREAK | † | † |
| 2 | MFHI | MTHI | MFLO | MTLO | † | † | † | † |
| 3 | MULT | MULTU | DIV | DIVU | † | † | † | † |
| 4 | ADD | ADDU | SUB | SUBU | AND | OR | XOR | NOR |
| 5 | † | † | SLT | SLTU | † | † | † | † |
| 6 | † | † | † | † | † | † | † | † |
| 7 | † | † | † | † | † | † | † | † |

**BCOND**

| 20..19 \ 18..16 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | BLTZ | BGEZ | | | | | | |
| 1 | | | | | | | | |
| 2 | BLTZAL | BGEZAL | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |

**COPz**

| 25..24 \ 23..21 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | MF | | CF | | MT | | CT | |
| 1 | BC | † | † | † | † | † | † | † |
| 2 | **Co-Processor Specific** | | | | | | | |
| 3 | **Operations** | | | | | | | |

| 20..19 \ 18..16 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | BCzF | BCzT | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |

**CP0**

| 4..3 \ 2..0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | TLBR | TLBWI | | | | TLBWR | |
| 1 | TLBP | | | | | | | |
| 2 | RFE | | | | | | | |
| 3 | | | | | | | | |

4000 tbl 14

**Table 2.10. Opcode Encoding for R3051 Family**

## INTRODUCTION

The R3051 family achieves its high standard of performance by combining a fast, efficient execution engine (that of the R3000A) with high-memory bandwidth, supplied from its large internal instruction and data caches. These caches insure that the majority of processor execution occurs at the rate of one instruction per clock cycle, and serve to decouple the high-speed execution engine from slower, external memory resources.

Portions of this chapter review the fundamentals of general cache operation, and may be skipped by readers already familiar with these concepts. This chapter also discusses the particular organization of the on-chip caches of the R3051 family. However, as these caches are managed by the R3051/52 CPU itself, the system designer does not typically need to be explicitly aware of this structure.

## FUNDAMENTALS OF CACHE OPERATION

High-performance microprocessor based systems frequently borrow from computer architecture principles long used in mini-computers and mainframes. These principles include instruction execution pipelining (discussed in Chapter 2) and instruction and data caching.

A cache is a high-speed memory store which contains the instructions and data most likely to be needed by the processor. That is, rather than implement the entire memory system with zero wait-state memory devices, a small zero wait-state memory is implemented. This memory, called a cache, then contains the instructions/data most likely to be referenced by the processor. If indeed the processor issues a reference to an item contained in the cache, then a zero wait-state access is made; if the reference is not contained in the cache, then the long latency associated with the true processor memory is incurred. The processor will achieve its maximum performance as long as its references "hit" (are resident) in the cache.

Caches rely on the principles of locality of software. These principles state that when a data/instruction element is used by a processor, it and its close neighbors are likely to be used again soon. The cache is then constructed to keep a copy of instructions and data referenced by the processor, so that subsequent references occur with zero wait-states.

Since the cache is typically many orders of magnitude smaller than main memory, each cache element must contain both the data (or instruction) required by the processor, as well as information which can be used to determine whether a cache "hit" occurs. This information, called the cache "TAG", is typically some or all of the address in main memory of the data item contained in that cache element as well as a "Valid" flag for that cache element. Thus, when the processor issues an address for a reference, the cache controller compares the TAG with the processor address to determine whether a hit occurs.

## R3051 FAMILY CACHE ORGANIZATION

There are a number of algorithms possible for managing a processor cache. This section describes the cache organization of the R3051 family.

### Basic Cache Operation

When the processor makes a reference, its 32-bit internal address bus contains the address it desires. The processor address bus is split into two parts; the low-order address bits specify a location in the cache to access, and

the remaining high-order address bits contain the value expected from the cache TAG. Thus, both the instruction/data element and the cache TAG are fetched simultaneously from the cache memory. If the value read from the TAG memories is the same as the high-order address bits, a cache hit occurs and the processor is allowed to operate on the instruction/data element retrieved. Otherwise, a cache miss is processed. This operation is illustrated in Figure 3.1.

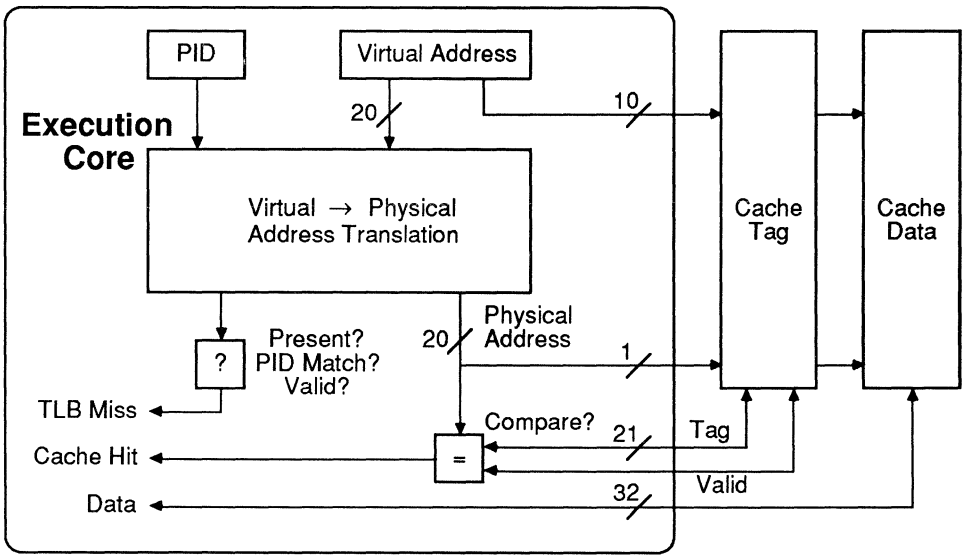



**Figure 3.1. Cache Line Selection**

To maximize performance, the R3051 family implements a Harvard Architecture caching strategy. That is, there are two separate caches: one contains instructions (operations), and the other contains data (operands). By separating the caches, higher overall bandwidth to the execution core is achieved, and thus higher performance is realized.

### Memory Address to Cache Location Mapping

The R3051 family caches are direct-mapped. That is, each main memory address can be mapped to (contained in) only one particular cache location. This is different from set-associative mappings, where each main memory location has multiple candidates for address mapping (although it should only be resident in one location at a time)

This organization, coupled with the large cache sizes resident on the R3051 family, achieve extremely high hit rates while maximizing speed and minimizing complexity and power consumption.

### Cache Addressing

The address presented to the cache and cache controller is that of the physical (main) memory element to be accessed. That is, the virtual address to physical address translation is performed by the memory management unit prior to the processor issuing its reference address.

Some microprocessors utilize virtual indexing in the cache, where the processor virtual address is used to specify the cache element to be retrieved. This type of cache structure complicates software and slows embedded applications:

- When the processor performs a context switch, a virtually indexed cache must be flushed. This is because two different tasks can use the same virtual address but mean totally different physical addresses. This cache flushing for a large cache dramatically slows context switch performance.

- Software must be aware of and specifically manage against "aliasing" problems. An alias occurs when two different virtual addresses correspond to the same physical address. If that occurs in a virtually indexed cache, then the same data element may be present in two different cache locations. If one virtual address is used to change the value of that memory location, and a different address used to read it later, then the second reference will not get the current value of that data item.

By providing for the memory management unit in the processor pipeline, physical cache addressing is used with no inherent speed penalty.

**Write Policy**

The R3051 family utilizes a write through cache. That is, whenever the processor performs a write operation to memory, then both the cache (data and TAG fields) and main memory are written. If the reference is uncacheable, then only main memory is written.

To minimize the delays associated with updating main memory, the R3051 family contains a 4 element write buffer. The write buffer captures the target address and data value in a single processor clock cycle, and subsequently performs the main memory write at its own, slower rate. The write buffer can FIFO up to 4 pending writes, as described in a later chapter.

**Partial Word Writes**

In the case of partial word writes, the R3051 family operates by performing a read-modify-write sequence in the cache: the store target address is used to perform a cache fetch; if the cache "hits", then the partial word data is merged with the cache and the cache is updated. Even if the cache read results in a hit, the memory interface will only see the partial word write, rather than the full word. This allows the designer to observe the actual activity of the execution core.

If the cache lookup of a partial word write "misses" in the cache, then only main memory is updated.

**Instruction Cache Line Size**

The "line size" of a cache refers to the number of cache elements mapped by a single TAG element. In the R3051 family, the instruction cache line size is 16 bytes, or four words.

This means that each cache line contains four adjacent words from main memory. In order to accommodate this, an instruction cache miss is processed by performing a quad word (block) read from the main memory, as discussed in a later chapter. This insures that a cache line contains four adjacent memory locations. Note that since the instruction cache is typically never written into directly by user software, the larger line size is permissible. If software does explicitly store into the instruction cache (perform store operations with the caches "swapped"), the programmer must insure that either the written lines are left invalidated, or that they contain four adjacent instructions.

Block refill uses the principle of locality of reference. Since instructions typically execute sequentially, there is a high probability that the instruction address immediately after the current instruction will be the next instruction. Block refill then brings into the cache those instructions immediately near the current instruction, resulting in a higher instruction cache hit rate.

Block refill also takes advantage of the difference between memory latency and memory bandwidth. Memory latency refers to the amount of time required to perform a processor request, while bandwidth refers to the rate at which subsequent transfers can occur. Factors that affect memory latency include address decoding, bus arbitration, and memory pre-charge requirements; factors which maximize bandwidth include the use of page mode or nibble mode accesses, memory interleaving, and burst memory devices.

The processing of a quad word read is discussed in a later chapter; however, it is worth noting that the R3051/52 can support either true burst accesses or can utilize a simpler, slower memory protocol for quad word reads.

### Data Cache Line Size

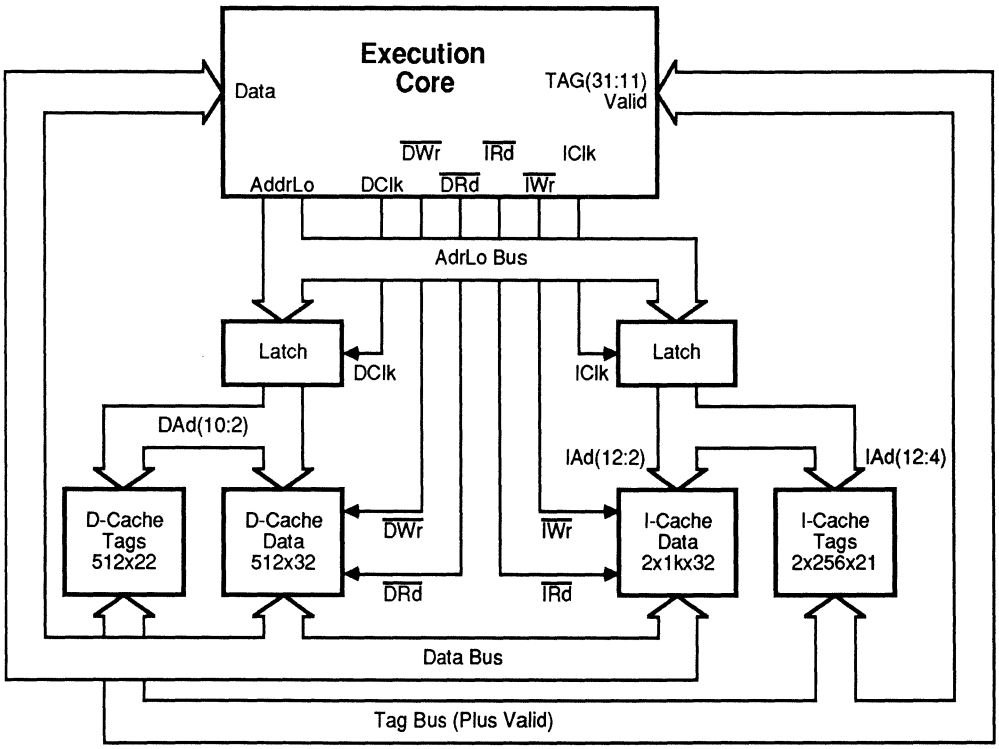
The data cache line size is different from that of the instruction cache, based on differences in their use. The data cache is organized as a line size of one word (four bytes).

This is optimal for the write policy of the data cache: since an individual cache word may be written by a software store instruction, the cache controller cannot guarantee that four adjacent words in the cache are from adjacent memory locations. Thus each word is individually tagged. The partial word writes (less than 4 bytes) are handled as a read-modify-write sequence, as described above.

Although the data cache line size is one word, the system may elect to perform data cache updates using quad word reads (block refill). The performance of the data cache update options can be simulated using Cache-305x; some systems may achieve higher performance through the use of data cache burst fill. No "streaming" occurs on data cache refills.

### Summary

The on-chip caches of the R3051 family can be thought of as constructed from discrete devices around the R3001A. The block diagram of the cache interface for the R3052/R3052E is shown in Figure 3.2; the interface for the R3051/R3051E is similarly constructed.




4000 drw 13

**Figure 3.2. R3051 Family Execution Core and Cache Interface**

## CACHE OPERATION

The operation of the on-chip caches is very straightforward, and is automatically handled by the processor.

### Basic Cache Fetch Operation

As with the R3000A/R3001A, the R3051 family can access both the instruction and data caches in a single clock cycle, resulting in 320 MB/sec bandwidth to the execution core. It does this by time multiplexing the cycle in the cache interface:

- During the first phase, a data cache address is presented, and a previous instruction cache read is completed.
- During the second phase, the data cache is read into the processor (or written by the processor). Also, the instruction cache is addressed with the next desired instruction.
- During the first phase of the next cycle, the instruction fetch begun in the previous phase is completed and a new data transaction is initiated.

This operation is illustrated in Figure 3.3. As long as the processor hits in the cache, and no internal stall conditions are encountered, it will continue to execute *run* cycles. A run cycle is defined to be a clock cycle in which forward progress in the processor pipeline occurs.

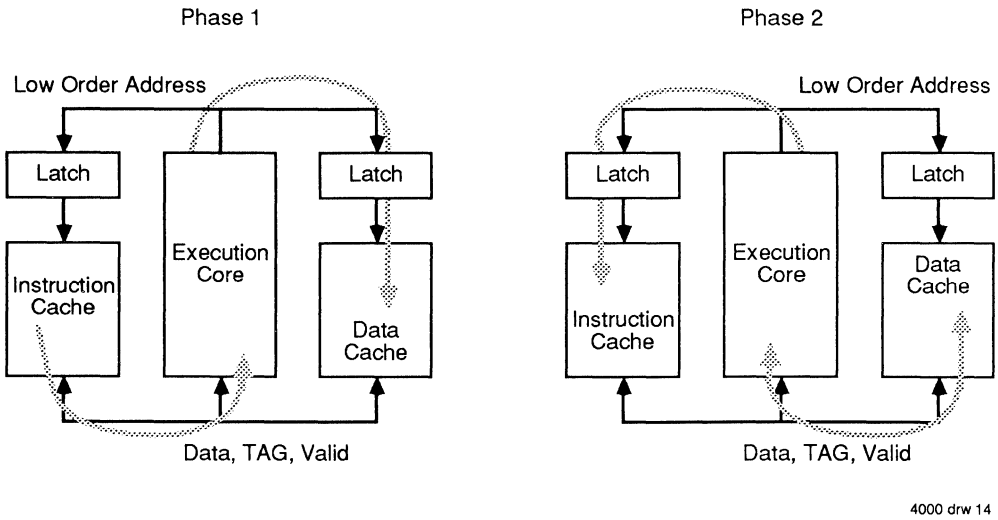



**Figure 3.3. Phased Access of Instruction and Data Caches**

### Cache Miss Processing

In the case of a cache miss (due to either a failed tag comparison or because the processor issued an uncacheable reference), the main memory interface (discussed in a later chapter) is invoked. If, during a given clock cycle, both the instruction and data cache miss, the data reference will be resolved before the instruction cache miss is processed.

While the processor is waiting for a cache miss to be processed, it will enter *stall* cycles until the bus interface unit indicates that it has obtained the necessary data.

When the bus interface unit returns the data from main memory, it is simultaneously brought to the execution unit and written into the on-chip caches. This is performed in a processor *fixup* cycle.

During a fixup cycle, the processor re-issues the cache access that failed; this occurs by having the processor re-address the instruction and data caches, so that the data may be written into the caches. If the cache miss was due to an uncacheable reference, the write is not performed, although a fixup cycle does occur.

### Instruction Streaming

A special feature of the R3051 family is utilized when performing block reads for instruction cache misses. This process is called *instruction streaming.* Instruction streaming is simultaneous instruction execution and cache refill.

As the block is brought in, the processor refills the instruction cache. Execution of the instructions within the block begins when the instruction corresponding to the cache miss is returned by the bus interface unit to the execution core. Execution continues until the end of the block is reached (in which case normal execution is resumed), or until some event forces the processor core to discontinue execution of that stream. These events include:

- Taken branches
- Data cache miss
- Internal stalls (TLB miss, multiply/divide interlock)
- Exceptions

When one of these events occur, the processor re-enters simple cache refill until the rest of the block has been written into the cache.

## CACHEABLE REFERENCES

Chapter 4 on memory management explains how the processor determines whether a particular reference (either instruction or data) is to a memory location that may reside in the cache. The fundamental mechanism is that certain virtual addresses are considered to be "cacheable". If the processor attempts to make a reference to a cacheable address, then it will employ its cache management protocol through that reference. Otherwise, the cache will be bypassed, and the execution engine core will directly communicate with the bus interface unit to process the reference.

Whether a given reference should be cacheable or not depends very much on the application, and on the target of the reference. Generally, I/O devices should be referenced as uncacheable data; for example, if software was polling a status register, and that register was cached, then it would never see the I/O device update the status (note that the compiler suite supports the "volatile" data type to insure that the status register in this case never gets allocated into an internal register).

There may be other instances where the uncacheable attribute is appropriate. For example, software which directly manipulates or flushes the caches can not be cached; similarly, boot software can not rely on the state of the caches, and thus must operate uncached at least until the caches are initialized.

## SOFTWARE DIRECTED CACHE OPERATIONS

In order to support certain system requirements, the R3051 family provides mechanisms for software to explicitly manipulate the caches. These mechanisms support diagnostics, cache and memory sizing, and cache flushing. In general, these mechanisms are enabled/disabled through the use of the Status Register in CP0.

The primary mechanisms for supporting these operations are cache swapping and cache isolation. Cache swapping forces the processor to use the data cache as an instruction cache, and vice versa. It is useful for allowing the processor to issue store instructions which cause the instruction cache to be written. Cache isolation causes the current data cache to be "isolated" from main memory; store operations do not cause main memory to be written, and all load operations "hit" in the data cache.

These mechanisms are enabled through the use of the "IsC" (Isolate Cache) and SwC (Swap Cache) bits of the status register, which resides in the on-chip System Control Co-Processor (CP0). Instructions which immediately precede and succeed these operations must not be cacheable, so that the actual swapping/isolation of the cache does not disrupt operation.

### Cache Sizing

It is possible for software to determine whether it is executing on an R3051 or an R3052 by determining the size of the instruction cache. Although typical software does not need to be aware of this difference, there may be some applications which desire to have software distinguish between the two processors.

Cache sizing in an R3051/52 is performed much like traditional memory sizing algorithms, but with the cache isolated. This avoids side-effects in memory from the sizing algorithm, and allows the software to use the "Cache Miss" bit of the status register in the sizing algorithm.

To determine the size of the instruction cache, software must:

1: Swap Caches
2: Isolate Caches
3: Write distinct values at location 00000FFC and then at FFFFFFFC
4: Read location 00000FFC.
5: Examine the CM (Cache Miss) bit of the status register; if it indicates a cache miss, then location FFFFFFFC overwrote location 00000FFC, and the cache size is 4kB. Otherwise, the cache size is 8kB.

Of course a more generalized algorithm could be developed to determine the cache size; this may be desirable for compatibility with discrete R3000A/R3001A systems or future R3051 family members. However, any algorithm will probably include the Swap and Isolate of the Instruction Cache, and the use of the Cache Miss bit. Sizing the data cache is done with a similar algorithm, although the caches need not be swapped.

Note that this software should operate as uncached. Once this is done, software should return the caches to their normal state by performing either a complete cache flush or an invalidate of those cache lines modified by the sizing algorithm.

### Cache Flushing

Cache flushing refers to the act of invalidating (indicating a line does not have valid contents) lines within either the instruction or data caches. Flushing must be performed before the caches are first used as real caches, and might also be performed during main memory page swapping or at certain context switches (note that the R3051 family implements physical caches, so that cache flushing at context switch time is not generally required).

The basic concept behind cache flushing is to have the "Valid" bit of each cache line set to indicate invalid. This is done in the R3051 family by having the cache isolated, and then writing a partial word quantity into the current data cache. Under these conditions, the R3051/52 will negate the "Valid" bit of the target cache line.

Again, this software should operate as uncached. To flush the data cache:

1: Isolate Caches
2: Perform a byte write every 4 bytes, starting at location 0, until 512 such writes have been performed (512 is based on the 2kB data cache).
3: Return the data cache to its normal state by clearing the IsC function.

To flush the instruction cache:

1: Swap Caches
2: Isolate Caches
3: Perform a byte write every 16 bytes (based on the instruction cache line size of 16 bytes). This should be done until each line (256 lines in the R3051, 512 in the R3052) have been invalidated. Note that treating the R3051 as an R3052 by flushing/invalidating 512 lines is acceptable though less efficient.
4: Return the caches to their normal state (unswapped and not isolated).

To minimize the execution time of the cache flush, this software should probably use an "unrolled" loop. That is, rather than have one iteration of the loop invalidate only one cache line, each iteration should invalidate multiple lines. This spreads the overhead of the loop flow control over more cache line invalidates, thus reducing execution time.

### Forcing Data into the Caches

Using these basic tools, it is possible to have software directly place values into the caches. When combined with appropriate memory management techniques, this could be used to "lock" values into the on-chip caches, by insuring that software does not issue other address references which may displace these locked values.

In order to force values into a cache, the cache should be Isolated. If software is trying to write instructions into the instruction cache, then the caches should also be swapped.

When forcing values into the instruction cache, software must take care with regards to the line size of the instruction cache. Specifically, a single TAG and Valid field describe four words in the instruction cache; software must then insure that any instruction cache line tagged as Valid actually contains valid data from all four words of the block.

## SUMMARY

The on-chip caches of the R3051 family are key to the inherent performance of the processor. The R3051 family design, however, does not require the system designer (either software or hardware) to explicitly manage this important resource, other than to correctly choose virtual addresses which may or may not be cached, and to flush the caches at system boot. This contributes to both the simplicity and performance of an R3051/52 based system.

## INTRODUCTION

The R3051 family provides two basic flavors of memory management. The base versions (the R3051 and R3052) provide segment-based virtual to physical address translation, and support the segregation of kernel and user tasks without requiring extensive virtual page management. The extended versions (the R3051E and R3052E) provide a full featured memory management unit (MMU) identical to the MMU structure of the R3000A and R3001A. The extended MMU uses an on-chip translation lookaside buffer (TLB) and dedicated registers in CP0 to provide for software management of page tables.

This chapter describes the operating states of the processor (kernel and user), and describes the virtual to physical address translation mechanisms provided in both versions of the architecture.

## VIRTUAL MEMORY IN THE R3051 FAMILY

There are two primary purposes of the memory management capabilities of the R3051 family.

- Various areas of main memory can have individual sets of attributes attributed to them. For example, some segments may be indicated as requiring kernel status to be accessed; others may have cacheable or uncacheable attributes. The virtual to physical address translation of the R3051/52 establishes the rules appropriate for a given virtual address.
- The virtual memory system can be used to logically expand the physical memory space of the processor, by translating addresses composed in a large virtual address space into the physical address space of the system. This is particularly important in applications where software may not be explicitly aware of the hardware resources of the processor system, and includes applications such as X-Window display systems. These types of applications are better served by the "E" (extended architecture) versions of the processor.

Figure 4.1 shows the form of an R3051 family virtual address. The most significant 20 bits of the 32-bit virtual address are called the virtual page number, or VPN. In the extended architecture versions, the VPN allows mapping of virtual addresses based on 4k Byte pages; in the base versions, only the three highest bits (segment number) are involved in the virtual to physical address translation.

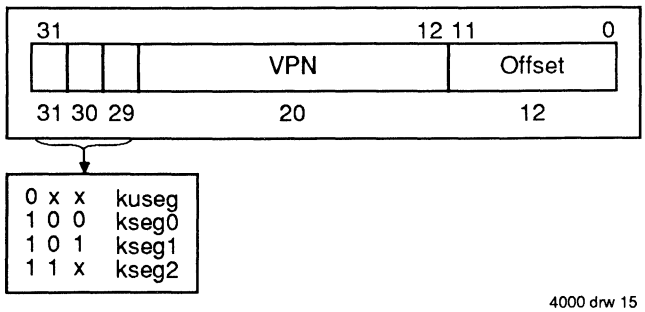



**Figure 4.1. Virtual Address Format**

In all versions, the three most significant bits of the virtual address identify which virtual address segment the processor is currently referencing; these segments have associated with them the mapping algorithm to be employed, and whether virtual addresses in that segment may reside in the cache. The

translation of the virtual address to an equivalent privilege level/segment is the same for the base and extended versions of the architecture.

## PRIVILEGE STATES

The R3051 family provides for two unique privilege states: the "Kernel" mode, which is analogous to the "supervisory" mode provided in many systems, and the "User" mode, where non-supervisory programs are executed. Kernel mode is entered whenever the processor detects an exception; when a Restore From Exception (RFE) instruction is executed, the processor will return either to its previous privilege mode or to User mode, depending on the state of the machine and when the exception was detected.

### User Mode Virtual Addressing

While the processor is operating in User mode, a single, uniform virtual address space **(kuseg)** of 2 GBytes is available for Users. All valid user-mode virtual addresses have the most significant bit of the virtual address cleared to 0. An attempt to reference a Kernel address (most significant bit of the virtual address set to 1) while in User mode will cause an Address Error Exception (see chapter 5). Kuseg begins at virtual address 0 and extends linearly for 2 GBytes. This segment is typically used to hold user code and data, and the current user processes. The virtual to physical address translation depends on whether the processor is a base or extended architecture version.

### Kernel Mode Virtual Addressing

When the processor is operating in Kernel mode, four distinct virtual address segments are simultaneously available. The segments are:

- **kuseg.** The kernel may assert the same virtual address as a user process, and have the same virtual to physical address translation performed for it as the translation for the user task. This facilitates the kernel having direct access to user memory regions. The virtual to physical address translation depends on whether the processor is a base or extended architecture version.
- **kseg0.** Kseg0 is a 512 MByte segment, beginning at virtual address 0x8000_0000. This segment is always translated to a linear 512 MByte region of the physical address space starting at physical address 0. All references through this segment are cacheable.
  When the most significant three bits of the virtual address are "100", the virtual address resides in kseg0. The physical address is constructed by replacing these three bits of the virtual address with the value "000". As these references are cacheable, kseg0 is typically used for kernel executable code and some kernel data.
- **kseg1.** Kseg1 is also a 512 MByte segment, beginning at virtual address 0xa000_0000. This segment is also translated directly to the 512 MByte physical address space starting at address 0. All references through this segment are uncacheable.
  When the most significant three bits of the virtual address are "101", the virtual address resides in kseg1. The physical address is constructed by replacing these three bits of the virtual address with the value "000". Unlike kseg0, references through kseg1 are not cacheable. This segment is typically used for I/O registers, boot ROM code, and operating system data areas such as disk buffers.
- **kseg2.** This segment is analogous to kuseg, but is accessible only from kernel mode. This segment contains 1 GByte of linear addresses, beginning at virtual address 0xc000_0000. As with kuseg, the virtual to physical address translation depends on whether the processor is a base or extended architecture version.

When the two most significant bits of the virtual address are "11", the virtual address resides in the 1024 MByte segment kseg2. The virtual to physical translation is done either through the TLB (extended versions of the processor) or through a direct segment mapping (base versions). An operating system would typically use this segment for stacks, per-process data that must be re-mapped at context switch, user page tables, and for some dynamically allocated data areas.

Thus, in both the base and extended versions of the processor, kseg0 and kseg1 are always mapped in the same fashion, to the lowest 512 MBytes of the physical address space. In both versions of the architecture, kseg0 references may reside in the on-chip cache, while kseg1 references may never reside in the on-chip caches.

The mapping of kuseg and kseg2 from virtual to physical addresses depends on whether the processor is a base or extended version of the architecture.

A base version is distinguishable from an extended version in software by examining the TS (TLB Shutdown) bit of the Status Register after reset, before the TLB is used. If the TS bit is set (1) immediately after reset, indicating that the TLB is non-functional, then the current processor is a base version of the architecture. If the TS bit is cleared after reset, then the software is executing on an extended architecture version of the processor.

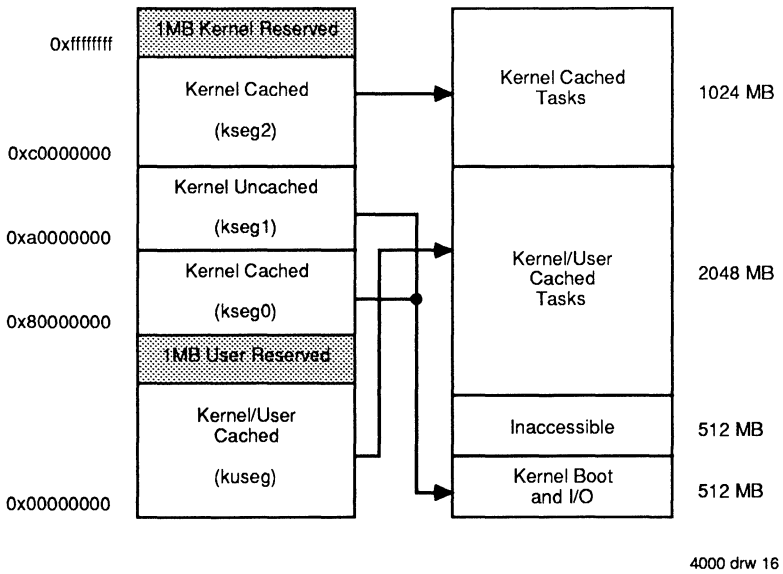



**Figure 4.2. Virtual to Physical Address Translation in Base Versions**

## BASE VERSIONS ADDRESS TRANSLATION

Processors which only implement the base versions of memory management perform direct segment mapping of virtual to physical addresses, as illustrated in Figure 4.2. Thus, the mapping of kuseg and kseg2 is performed as follows:

- Kuseg is always translated to a contiguous 2 GByte region of the physical address space, beginning at location 0x4000_0000. That is, the value "00" in the two highest order bits of the virtual address space are translated to the value "01", with the remaining 30 bits of the virtual address unchanged.
- Virtual addresses in kseg2 are directly output as physical addresses; that is, references to kseg2 occur with the physical address unchanged from the virtual address.
- The upper 1 MByte of each of Kuseg and Kseg2 should not be used. This region is being reserved for compatibility with future revisions of the chip, which may include on-chip resources which map to these virtual addresses.

The base versions of the architecture allow kernel software to be protected from user mode accesses, without requiring virtual page management software. User references to kernel virtual address will result in an address error exception.

Some systems may elect to protect external physical memory as well. That is, the system may include distinct memory devices which can only be accessed from kernel mode. The physical address output determines whether the reference occured from kernel or user mode, according to Table 4.1.

| Physical Address (31:29) | Virtual Address Segment |
|---|---|
| '000' | Kseg0 or Kseg1 |
| '001' | Inaccessible |
| '01x' | Kuseg |
| '10x' | Kuseg |
| '11x' | Kseg2 |

4000 tbl 15

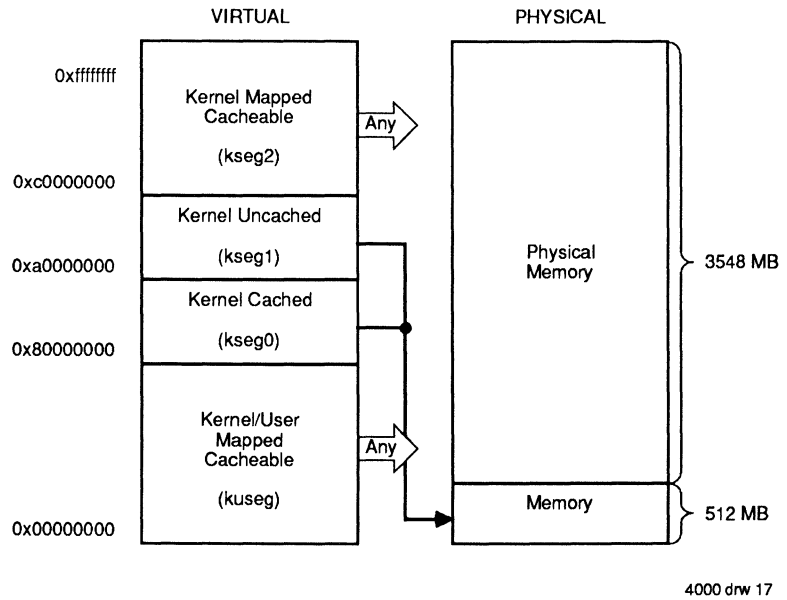
**Table 4.1. Virtual and Physical Address Relationships in Base Versions**

Thus, some systems may wish to limit accesses to some memory or I/O devices to those physical address bits which correspond to kernel mode virtual addresses.

Alternately, some systems may wish to have the kernel and user tasks share common areas of memory. Those systems could choose to have their address decoder ignore the high-order physical address bits, and compress all of memory into the lower region of physical memory. The high-order physical address bits may be useful as privilege mode status outputs in these systems.

## EXTENDED VERSIONS ADDRESS TRANSLATION

The extended versions of the architecture use a full featured MMU, like that found in the R3000A and R3001A, to manage the virtual to physical address translation of kuseg and kseg2. This MMU maps 4kByte virtual pages to 4kByte physical pages, and controls the attribute of these pages on a page by page basis. The extended versions of the architecture map the virtual address space as illustrated in Figure 4.3.




4000 drw 17

**Figure 4.3. Virtual to Physical Address Mapping of Extended Architecture**

Note that kuseg and kseg2 may be mapped anywhere in the 4GByte physical address space. Thus, the external memory system may not be able to examine the physical address outputs from the processor to determine the virtual segment origin of the reference. Software in such a system will be much more responsible for managing the separation of kernel and user resources.

Pages are mapped by substituting a 20-bit physical frame number (PFN) for the 20-bit virtual page number field of the virtual address. This substitution is performed through the use of the on-chip Translation Lookaside Buffer (TLB). The TLB is a fully associative memory that holds 64 entries to provide a mapping of 64 4kByte pages. When a virtual reference to kuseg or kseg2 occurs, each TLB entry is probed to see if it maps the corresponding VPN.

The mapping function is provided as part of the on-chip System Control Co-Processor, CP0. CP0 supports address translation, exception handling, and other privileged transactions. CP0 contains the TLB and the other registers shown in Figure 4.4.

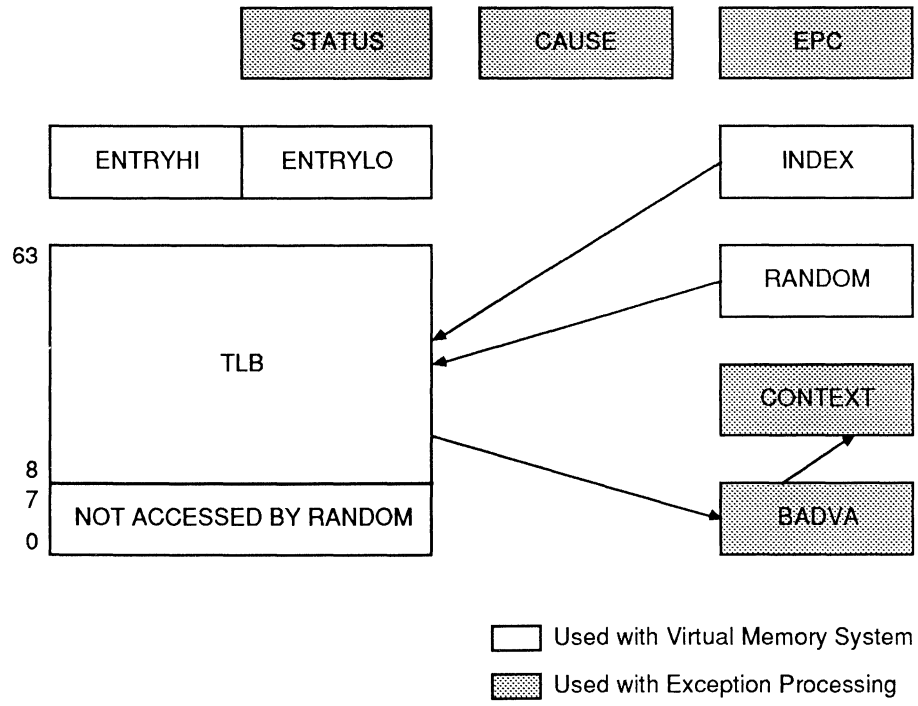



**Figure 4.4. The System Coprocessor Registers**

The sections that follow describes the virtual to physical address mapping performed by the TLB.

### TLB Entries

Each TLB entry is 64 bits wide, and its format is illustrated in Figure 4.5. Each field of a TLB entry has a corresponding field in the EntryHi/EntryLo register pair (described next). Figure 4.6 describes each of the fields of a TLB entry.

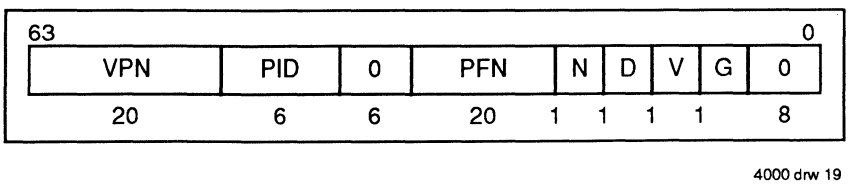
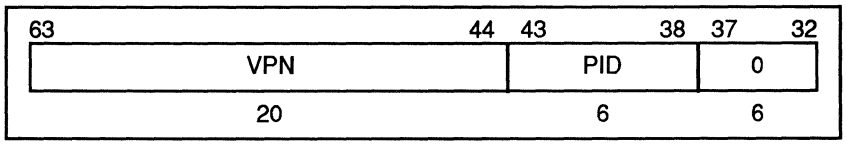



**Figure 4.5. Format of a TLB Entry**

### EntryHi and EntryLo Registers

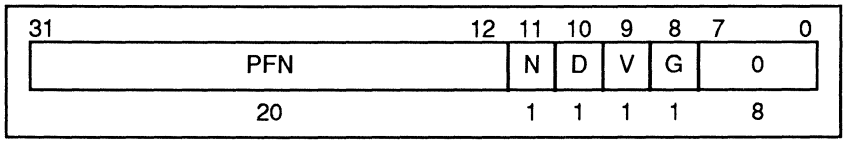
These two registers provide the data path for operations which read, write, or probe the TLB file. The format of these registers is the same as the format of a TLB entry, and is illustrated in Figure 4.6.

TLB EntryHi Register




| | |
|---|---|
| *VPN* | Virtual Page Number. Bits 31..12 of virtual address. |
| *PID* | Process ID field. A 6-bit field which lets multiple processes share the TLB while each process has a distinct mapping of otherwise identical virtual page numbers. |
| *0* | Reserved. Currently ignores writes, returns zero when read. |

TLB EntryLo Register




| | |
|---|---|
| *PFN* | Page Frame Number. Bits 31..12 of the physical address. The R3051/52"E" maps a virtual page to the PFN. |
| *N* | Non-cacheable. If this bit is set, the page is marked as non-cacheable and the R3051/52"E" directly accesses main memory instead of first accessing the cache. |
| *D* | Dirty. If this bit is set, the page is marked as "dirty" and therefore writable. This bit is actually a "write-protect" bit that software can use to prevent alteration of data. If an entry is accessed for a write operation when the D bit is cleared, the R3051/52"E" causes a TLB Mod trap. The TLB entry is not modified on such a trap. |
| *V* | Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS Miss occurs. |
| *G* | Global. If this bit is set, the R3051/52"E" ignores the PID match requirement for valid translation. In kseg2, the Global bit lets the kernel access all mapped data without requiring it to save or restore PID (Process ID) values. |
| *0* | Reserved. Must be written as '0', returns '0' when read. |

4000 drw 20

**Figure 4.6. The TLB EntryLo and EntryHi Registers**

For maximum software efficiency, operating system software could use the format of EntryLo to describe a Page Table Entry in the operating system Page Table; however, since PTE's are managed through software algorithms, rather than hardware, an operating system could choose a different format than that of EntryLo.

### Virtual Address Translation

During a virtual to physical address translation in kuseg or kseg2, the R3051/52"E" compares the PID and the highest 20 bits of the virtual address (the VPN) to the contents of each TLB entry. A generalized algorithm for this mapping is illustrated in Figure 4.7.

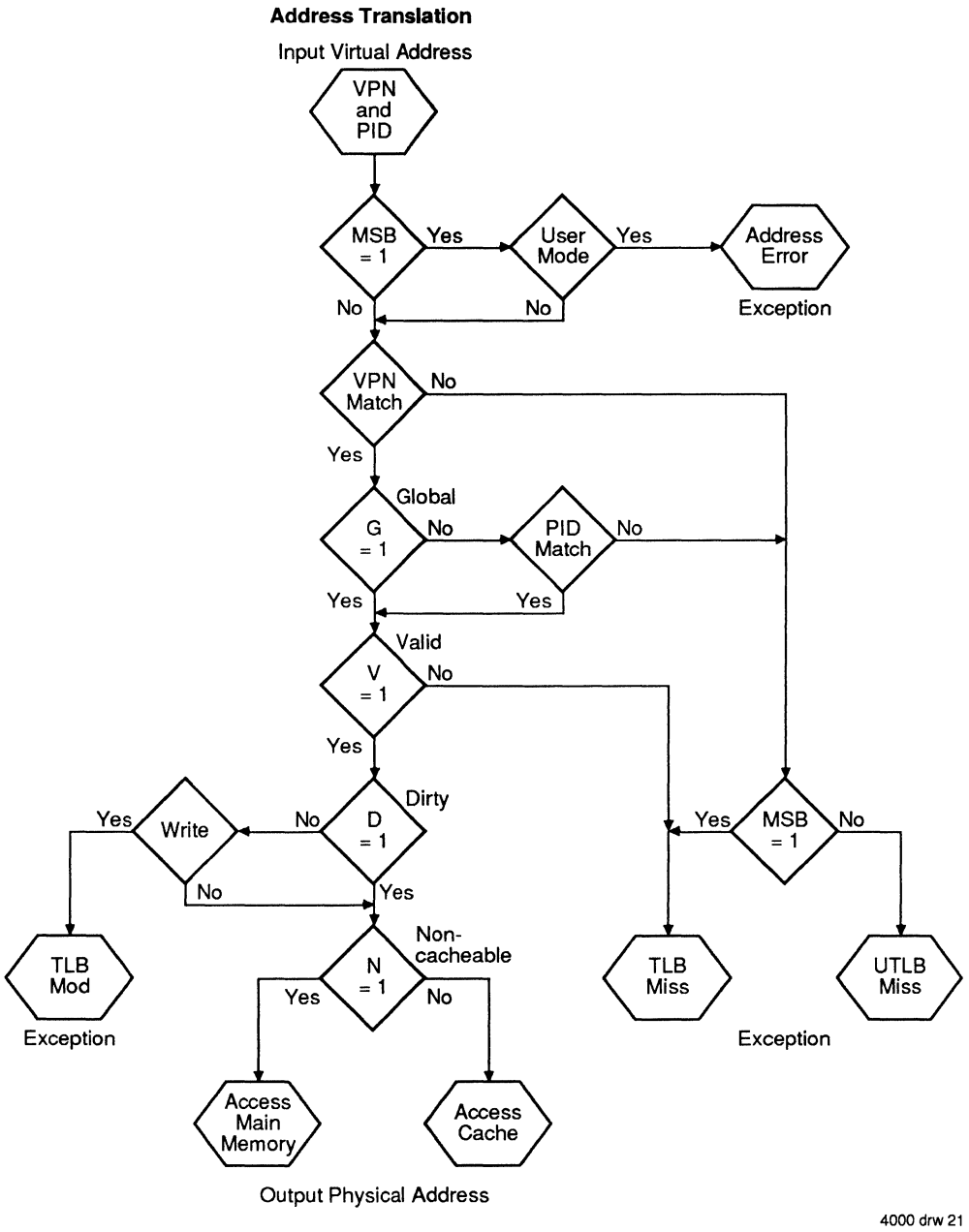



**Figure 4.7. TLB Address Translation**

A virtual address matches (is mapped by) a TLB entry if:

- the VPN of the virtual address matches the VPN field of aTLB entry
- either the "G" (global) bit of the TLB entry is set, or the PID field of the virtual address (stored in the EntryHi register) matches the PID field of the TLB entry.

If a match is found, then the corresponding physical address (PFN) field of the TLB entry is retrieved from the matching entry, along with the access control bits (N, D, and V). If no match is found, then either a TLB or UTLB miss exception will occur. Figure 4.8 shows the generation of a physical address from a specific virtual address mapped by the TLB.

If the access control bits (D and V) indicate that the access is not valid (either the TLB entry is not valid, or the page is write protected or not yet dirty), then a TLB modification or TLB miss exception will occur. If the N (Non-cacheable) bit is set, then the processor will not look in its caches for the data, but rather will directly use the bus interface unit to retrieve the word from main memory.

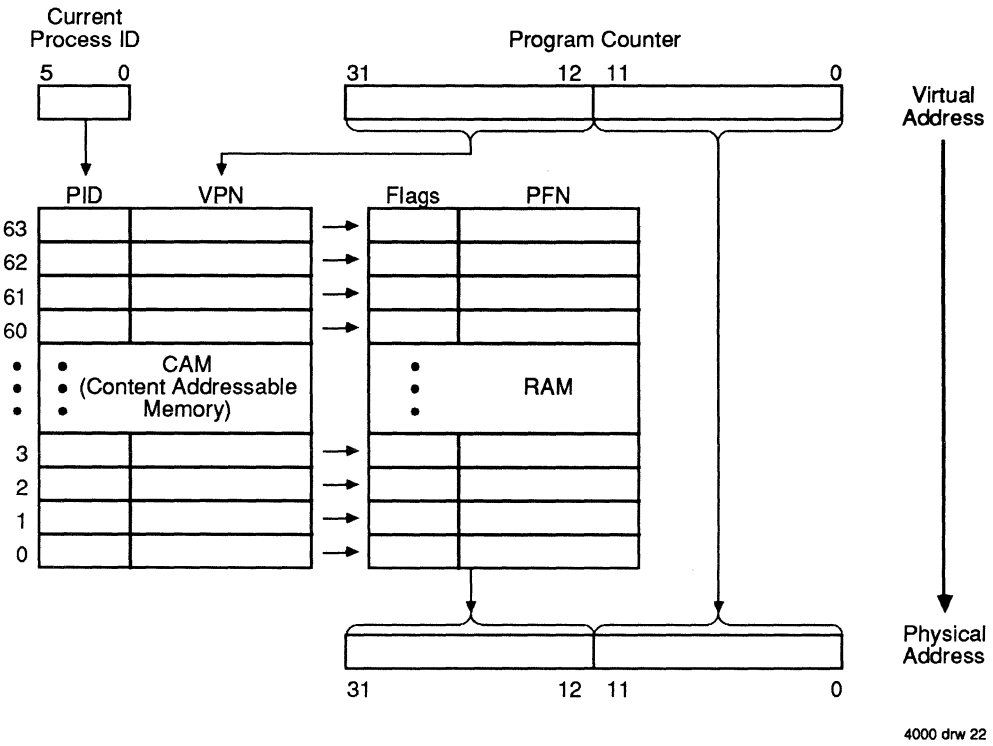
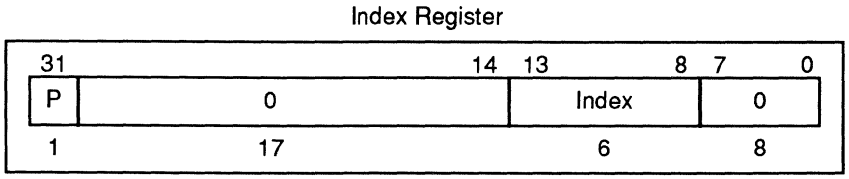



**Figure 4.8. Virtual to Physical TLB Translation**

## The Index Register

The *Index* register is a 32-bit, read-write register, which has a 6-bit field used to index to a specific entry in the 64-entry TLB file. The high-order bit of the register is a status bit which reflects the success or failure of a **TLB Probe (tlbp)** instruction, described later in this chapter.

The Index register also specifies the TLB entry that will be affected by the **TLB Read (tlbr)** and **TLB Write Index (tlbwi)** instructions. Figure 4.9 shows the format of the Index register.




| | |
|---|---|
| *P* | Probe failure. Set to 1 when the last TLBProbe (tlbp) instruction was unsuccessful. |
| *Index* | Index to the TLB entry that will be affected by the TLBRead and TLBWrite instructions. |
| *0* | Reserved. Must be written as zero, returns zero when read. |

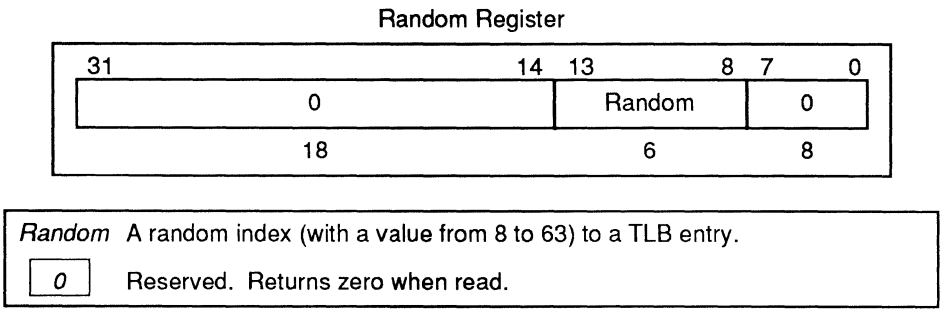
4000 drw 23

**Figure 4.9. The Index Register**

## The Random Register

The *Random* register is a 32-bit read-only register. The format of the Random register is shown in figure 4.10.

The six-bit Random field indexes a Random entry in the TLB. It is basically a counter which decrements on every clock cycle, but which is constrained to count in the range of 63 to 8. That is, software is guaranteed that the Random register will never index into the first 8 TLB entries. These entries can be




4000 drw 24

**Figure 4.10. The Random Register**

"locked" by software into the TLB file, guaranteeing that no TLB miss exceptions will occur in operations which use those virtual address. This is useful for particularly critical areas of the operating system.

The Random register is typically used in the processing of a TLB miss exception. The Random register provides software with a "suggested" TLB entry to be written with the correct translation; although slightly less efficient than a Least Recently Used (LRU) algorithm, Random replacement offers substantially similar performance while allowing dramatically simpler hardware and software management. To perform a TLB replacement, the **TLB Write Random (tlbwr)** instruction is used to write the TLB entry indexed by this register.

At reset, this counter is preset to the value '63'. Thus, it is possible for two processors to operate in "lock-step", even when using the Random TLB replacement algorithm. Also, software may directly read this register, although this feature probably has little utility outside of device testing and diagnostics.

### TLB Instructions

The R3051/52"E" provides instructions for working with the TLB, as listed in Table 4.2. These instructions are described briefly below. Their operation in base versions of the R3051 family architecture is undefined.

| Op Code | Description |
|---|---|
| tlbp | Translation Lookaside Buffer Probe |
| tlbr | Translation Lookaside Buffer Read |
| tlbwi | Translation Lookaside Buffer Write at Index |
| tlbwr | Translation Lookaside Buffer Write at Random |

4000 tbl 16

**Table 4.2. TLB Instructions**

**Translation Lookaside Buffer Probe (tlbp).** This instruction "probes" the TLB to see if an entry matches the EntryHi register contents. If a match occurs, the R3051/52E loads the Index register with the index of the entry that matched. If no match exists, The R3051/52E will set the high order bit (the *P* bit) of the Index Register.

**Translation Lookaside Buffer Read (tlbr).** This instruction loads the EntryHi and EntryLo registers with the contents of the TLB entry pointed to by the Index register.

**Translation Lookaside Buffer Write at Index (tlbwi).** This instruction loads the TLB entry pointed to by the Index register with the current values of the EntryHi and EntryLo register.

**Translation Lookaside Buffer Write at Random (tlbwr).** This instruction loads the TLB entry pointed to by the Random register with the current values of the EntryHi and EntryLo register.

### TLB Shutdown

The status register contains a single bit which indicates whether the TLB is operating properly. This bit, once set, may only be cleared by a device reset.

There are two reasons this bit might be set:

- If this bit is set at device reset, prior to the actual use of the TLB for address mapping, then this is not an "Extended" version of the R3051 family architecture, and thus no TLB is present.
- If this bit is cleared at reset, but set subsequently, then the TLB detected multiple virtual to physical mappings for the same VPN. This is either the result of improper software, or of improper operation of the TLB. If this condition is detected, the TLB will be shutdown, prohibiting further virtual to physical address mappings through the TLB. The virtual to physical translation of kuseg and kseg2 is undefined under these conditions.

## SUMMARY

The R3051 family provides two models of memory management: a very simple, segment based mapping, found in the base versions of the architecture, and a more sophisticated, TLB-based page mapping scheme, present in the extended versions of the architecture. Each scheme has advantages to different applications.

For example, many stand-alone applications have no need for paging, as the memory requirements of the application are absolutely determined when the system is designed. Examples of these types of systems include data communications applications, navigation, and process control.

Applications may have unpredictable memory requirements, since the target system can not predict the resource requirements of the various tasks which operate on it. This is the classic model for virtual memory management in general purpose computers. However, this model is increasingly appropriate in a number of embedded applications, such as X-Window Terminals. Applications such as these may be connected on a network to numerous hosts, each of which presents tasks to the system without explicit awareness of the resource utilization of other hosts. Virtual memory management in such applications may then be appropriate, with the unmapped segments (kseg0 and kseg1) used for the application operating system and I/O channels.

## INTRODUCTION

Processors in general execute code in a highly-directed fashion. The instruction immediately subsequent to the current instruction is fetched and then executed; if that instruction is a branch instruction, the program execution is diverted to the specified location. Thus, program execution is relatively straightforward and predictable.

Exceptions are a mechanism used to break into this execution stream and to force the processor to begin handling another task, typically related to either the system state or to the erroneous or undesirable execution of the program stream. Thus, exceptions typically are viewed by programmers as asynchronous interruptions of their program. (Note that exceptions are not necessarily unpredictable or asynchronous, in that the events which cause the exception may be exactly repeatable by the same software executing on the same data; however, the programmer does not typically "expect" an exception to occur when and where it does, and thus will view exceptions as asynchronous events).

The R3051 family architecture provides for extremely fast, flexible interrupt and exception handling. The processor makes no assumptions about interrupt causes or handling techniques, and allows the system designer to build his own model of the best response to exception conditions. However, the processor provides enough information and resources to minimize both the amount of time required to begin handling the specific cause of the exception, and to minimize the amount of software required to preserve processor state information so that the normal instruction stream may be resumed.

This chapter discusses exception handling issues in R3051/52-based systems. The topics examined are: the exception model, the machine state to be saved on an exception, and nested exceptions. Representative software examples of exception handlers are also provided, as are techniques and issues appropriate to specific classes of exceptions.

## R3051 FAMILY EXCEPTION MODEL

The exception processing capability of the R3051 family is provided to assure an orderly transfer of control from an executing program to the kernel. Exceptions may be broadly divided into two categories: they can be caused by an instruction or instruction sequence, including an unusual condition arising during its execution; or can be caused by external events such as interrupts. When an R3051/52 detects an exception, the normal sequence of instruction flow is suspended; the processor is forced to kernel mode where it can respond to the abnormal or asynchronous event. Table 5.1 lists the exceptions recognized by the R3051 family.

| Exception | Mnemonic | Cause |
|---|---|---|
| **Reset** | Reset | Assertion of the Reset signal causes an exception that transfers control to the special vector at virtual address 0xbfc0_0000. |
| **UTLB Miss** | UTLB | User TLB Miss. A reference is made (in either kernel or user mode) to a page in *kuseg* that has no matching TLB entry. This can occur only in extended architecture versions of the processor. |
| **TLB Miss** | TLBL (Load)<br>TLBS (Store) | A referenced TLB entry's Valid bit isn't set, or there is a reference to a *kseg2* page that has no matching TLB entry. This can occur only in extended architecture versions of the processor. |
| **TLB Modified** | Mod | During a store instruction, the Valid bit is set but the dirty bit is not set in a matching TLB entry. This can occur only in extended architecture versions of the processor. |
| **Bus Error** | IBE (Instruction)<br>DBE (Data) | Assertion of the Bus Error input during a read operation, due to such external events as bus timeout, backplane memory errors, invalid physical address, or invalid access types. |
| **Address Error** | AdEL (Load)<br>AdES (Store) | Attempt to load, fetch, or store an unaligned word; that is, a word or halfword at an address not evenly divisible by four or two, respectively. Also caused by reference to a virtual address with most significant bit set while in User Mode. |
| **Overflow** | Ovf | Twos complement overflow during add or subtract. |
| **System Call** | Sys | Execution of the SYSCALL Trap Instruction |
| **Breakpoint** | Bp | Execution of the break instruction |
| **Reserved Instruction** | RI | Execution of an instruction with an undefined or reserved major operation code (bits 31:26), or a special instruction whose minor opcode (bits 5:0) is undefined. |
| **Co-processor Unusable** | CpU | Execution of a co-processor instruction when the CU (Co-processor Usable) bit is not set for the target co-processor. |
| **Interrupt** | Int | Assertion of one of the six hardware interrupt inputs or setting of one of the two software interrupt bits in the Cause register. |

4000 tbl 17

**Table 5.1. R3051 Family Exceptions**

### Precise vs. Imprecise Exceptions

One classification of exceptions refers to the precision with which the exception cause and processor context can be determined. That is, some exceptions are precise in their nature, while others are "imprecise."

In a **precise exception**, much is known about the system state at the exact instance the exception is caused. Specifically, the exact processor context and the exact cause of the exception are known. The processor thus maintains its exact state before the exception was generated, and can accurately handle the exception, allowing the instruction stream to resume when the situation is corrected. Additionally, in a precise exception model, the processor can not advance state; that is, subsequent instructions, which may already be in the

processor pipeline, are not allowed to change the state of the machine.

Many real-time applications (especially ADA-generated applications) greatly benefit from a processor model which guarantees precise exception context and cause information. The MIPS architecture, including the R3051 family, implements a precise exception model for all exceptional events.

## EXCEPTION PROCESSING

The R3051/52's exception handling system efficiently handles machine exceptions, including Translation Lookaside Buffer (TLB) misses, arithmetic overflows, I/O interrupts, system calls, breakpoints, reset, and co-processor unusable conditions. Any of these events interrupt the normal execution flow; the R3051/52 aborts the instruction causing the exception and also aborts all those following in the exception pipeline which have already begun, thus not modifying processor context. The R3051/52 then performs a direct jump into a designated exception handler routine. This insures that the R3051/52 is always consistent with the precise exception model.

## EXCEPTION HANDLING REGISTERS

The system co-processor (CP0) registers contain information pertinent to exception processing. Software can examine these registers during exception processing to determine the cause of the exception and the state of the processor when it occurred There are five registers handling exception processing, shown in shaded boxes in Figure 5.1. These are the *Cause* register, the *EPC* register, the *Status* register, the *BadVAddr* register, and the *Context* register. A brief description of each follows.

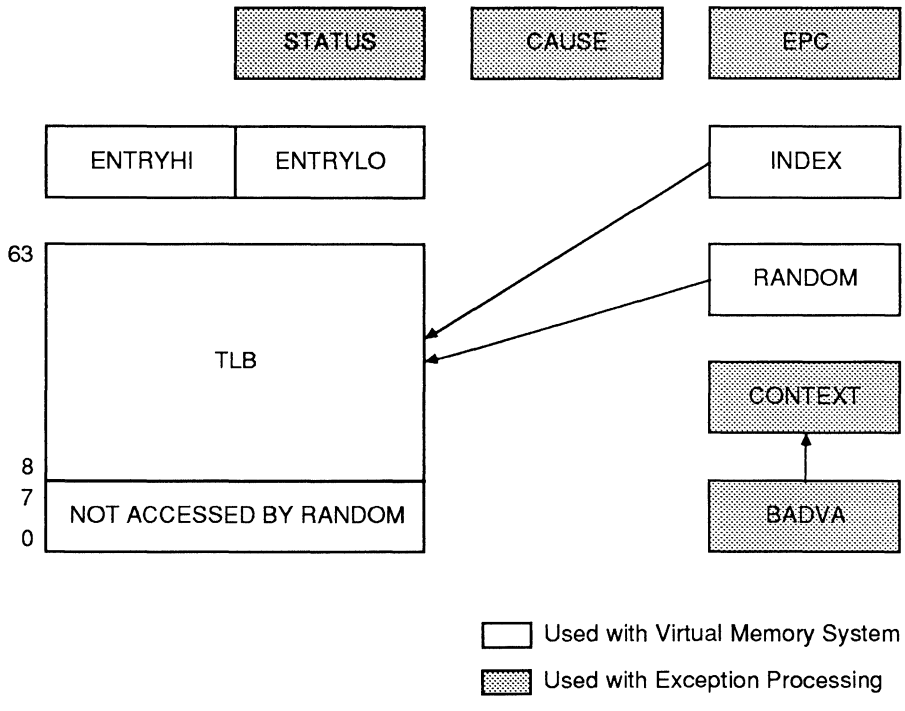



**Figure 5.1. The CP0 Execution Handling Registers**

Table 5.2 lists the register address of each of the CP0 registers (as used in CP0 operations); the register number is used by software when issuing co-processor load and store instructions.

| Register Name | Register Number (Dec.) |
|---|---|
| Status | $12 |
| Cause | $13 |
| Exception PC | $14 |
| TLB Entry Hi | $10 |
| TLB Entry Lo | $2 |
| Index | $0 |
| Random | $1 |
| Context | $4 |
| Bad Virtual Address | $8 |
| PrId | $15 |
| Reserved | $3, $5-$7, $9, $11, $16-$31 |

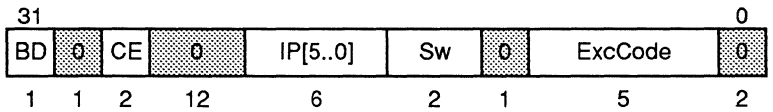
4000 tbl 18

**Table 5.2. Co-processor 0 Register Addressing**

## The Cause Register

The contents of the Cause register describe the last exception. A 5-bit exception code indicates the cause of the current exception; the remaining fields contain detailed information specific to certain exceptions.

All bits in this register, with the exception of the SW bits, are read-only. The SW bits can be written to set or reset software interrupts. Figure 5.2 illustrates the format of the Cause register. Table 5.3 details the meaning of the various exception codes.




BD: BRANCH DELAY
CE: COPROCESSOR ERROR
IP: INTERRUPTS PENDING
Sw: SOFTWARE INTERRUPTS*

ExcCode: EXCEPTION CODE FIELD

0 : RESERVED
Must Be Written as 0
Returns 0 when Read

*READ AND WRITE. THE REST ARE READ-ONLY.

4000 drw 26

**Figure 5.2. The Cause Register**

| Number | Mnemonic | Description |
|---|---|---|
| 0 | Int | External Interrupt |
| 1 | MOD | TLB Modification Exception |
| 2 | TLBL | TLB miss Exception (Load or instruction fetch) |
| 3 | TLBS | TLB miss exception (Store) |
| 4 | AdEL | Address Error Exception (Load or instruction fetch) |
| 5 | AdES | Address Error Exception (Store) |
| 6 | IBE | Bus Error Exception (for Instruction Fetch) |
| 7 | DBE | Bus Error Exception (for data Load or Store) |
| 8 | Sys | SYSCALL Exception |
| 9 | Bp | Breakpoint Exception |
| 10 | RI | Reserved Instruction Exception |
| 11 | CpU | Co-Processor Unusable Exception |
| 12 | Ovf | Arithmetic Overflow Exception |
| 13-31 | - | Reserved |

4000 tbl 19

**Table 5.3. Cause Register Exception Codes**

The meaning of the other bits of the cause register is as follows:

**BD** The Branch Delay bit is set (1) if the last exception was taken while the processor was executing in the branch delay slot. If so, then the EPC will be rolled back to point to the branch instruction, so that it can be re-executed and the branch direction re-determined.

**CE** The Co-processor Error field captures the co-processor unit number referenced when a Co-processor Unusable exception is detected.

**IP** The Interrupt Pending field indicates which interrupts are pending. Regardless of which interrupts are masked, the IP field can be used to determine which interrupts are pending.

**SW** The Software interrupt bits can be thought of as the logical extension of the IP field. The SW interrupts can be written to to force an interrupt to be pending to the processor, and are useful in the prioritization of exceptions. To set a software interrupt, a "1" is written to the appropriate SW bit, and a "0" will clear the pending interrupt. There are corresponding interrupt mask bits in the status register for these interrupts.

### The EPC (Exception Program Counter) Register

The 32-bit EPC register contains the virtual address of the instruction which took the exception, from which point processing resumes after the exception has been serviced. When the virtual address of the instruction resides in a branch delay slot, the EPC contains the virtual address of the instruction immediately preceding the exception (that is, the EPC points to the Branch or Jump instruction).

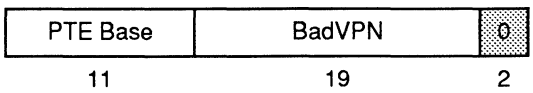
### Bad VAddr Register

The Bad VAddr register saves the entire bad virtual address for any addressing exception.

### Context Register

The Context register duplicates some of the information in the BadVAddr register, but provides this information in a form that may be more useful for a software TLB exception handler.

Figure 5.3 illustrates the layout of the Context register. The Context register is used to allow software to quickly determine the main memory address of the page table entry corresponding to the bad virtual address, and allows the TLB to be updated by software very quickly (using a nine-instruction code sequence).




0: RESERVED: READ AS 0, MUST BE WRITTEN AS 0

BadVPN: FAILING VIRTUAL PAGE NUMBER (SET BY HARDWARE; READ ONLY FIELD DERIVED FROM BADVADDR REGISTER)

PTE Base: BASE ADDRESS OF PAGE TABLE ENTRY; SET BY KERNEL SOFTWARE

4000 drw 27

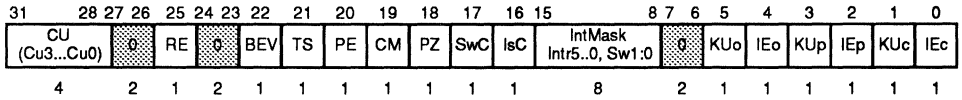
**Figure 5.3. Context Register**

### The Status Register

The Status register contains all the major status bits; any exception puts the system in Kernel mode. All bits in the status register, with the exception of the TS (TLB Shutdown) bit, are readable and writable; the TS bit is read-only. Figure 5.4 shows the functionality of the various bits in the status register.

The status register contains a three level stack (current, previous, and old) of the kernel/user mode bit (KU) and the interrupt enable (IE) bit. The stack is pushed when each exception is taken, and popped by the Restore From Exception instruction. These bits may also be directly read or written.

At reset, the SWc, KUc, and IEc bits are set to zero; BEV is set to one; and the value of the TS bit depends on whether the device is an Extended Architecture version (TS = 0) or base version (TS = 0). The rest of the bit fields are undefined after reset.




CU: COPROCESSOR USABILITY
BEV: BOOTSTRAP EXCEPTION VECTOR
TS: TLB SHUTDOWN
PE: PARITY ERROR
CM: CACHE MISS
PZ: PARITY ZERO
SwC: SWAP CACHES
IsC: ISOLATE CACHE
RE: REVERSE ENDIANNESS

IntMASK: INTERRUPT MASK
KUo: KERNEL/USER MODE, OLD
IEo: INTERRUPT ENABLE, OLD
KUp: KERNEL/USER MODE, PREVIOUS
IEp: INTERRUPT ENABLE, PREVIOUS
KUc: KERNEL/USER MODE, CURRENT
IEc: INTERRUPT ENABLE, CURRENT
0: RESERVED: READ AS ZERO
MUST BE WRITTEN AS ZERO

4000 drw 28

**Figure 5.4. The Status Register**

The various bits of the status register are defined as follows:

**CU** Co-processor Useability. These bits individually control user level access to co-processor operations, including the polling of the BrCond input port and the manipulation of the System Control Co-processor (CP0).

**RE** Reverse Endianness. The R3051 family allows the system to determine the byte ordering convention for the Kernel mode, and the default setting for user mode, at reset time. If this bit is cleared, the endianness defined at reset is used for the current user task. If this bit is set, then the user task will operate with the opposite byte ordering convention from that determined at reset.

**BEV** Bootstrap Exception Vector. The value of this bit determines the locations of the exception vectors of the processor. If BEV = 1, then the processor is in "Bootstrap" mode, and the exception vectors reside in uncacheable space. If BEV = 0, then the processor is in normal mode, and the exception vectors reside in cacheable space.

**TS** TLB Shutdown. This bit reflects whether the TLB is functioning. At reset, this bit can be used to determine whether the current processor is a base or extended architecture version. In extended architecture versions, this bit will also reflect whether the TLB is operating normally, as described in Chapter 4.

**CM** Cache Miss. This bit is set if a cache miss occurred while the cache was isolated. It is useful in determining the size and operation of the internal cache subsystem.

**SwC** Swap Caches. Setting this bit causes the execution core to use the on-chip instruction cache as a data cache and vice-versa. Resetting the bit to zero unswaps the caches. This is useful for certain operations such as instruction cache flushing.

**IsC** Isolate Cache. If this bit is set, the data cache is "isolated" from main memory; that is, store operations modify the data cache but do not cause a main memory write to occur, and load operations return the data value from the cache whether or not a cache hit occurred. This bit is also useful in various operations such as flushing, as described in Chapter 3.

**IM** Interrupt Mask. This 8-bit field can be used to mask the hardware and software interrupts to the execution engine (that is, not allow them to cause an exception). IM(1:0) are used to mask the software interrupts, and IM (7:2) mask the 6 external interrupts. A value of '0' disables a particular interrupt, and a '1' enables it. Note that the IE bit is a global interrupt enable; that is, if the IE is used to disable interrupts, the value of particular mask bits is irrelevant; if IE enables interrupts, then a particular interrupt is selectively masked by this field.

**KUo** Kernel/User old. This is the privilege state two exceptions previously. A '0' indicates kernel mode.

**IEo** Interrupt Enable old. This is the global interrupt enable state two exceptions previously. A '1' indicates that interrupts were enabled, subject to the IM mask.

**KUp** Kernel/User previous. This is the privilege state prior to the current exception A '0' indicates kernel mode.

**IEp** Interrupt Enable old. This is the global interrupt enable state prior to the current exception. A '1' indicates that interrupts were enabled, subject to the IM mask.

**KUc** Kernel/User current. This is the current privilege state. A '0' indicates kernel mode.

**IEc** Interrupt Enable current. This is the current global interrupt enable state. A '1' indicates that interrupts are enabled, subject to the IM mask.

**'0'** Fields indicated as '0' are reserved; they must be written as '0', and will return '0' when read.

### PrId Register

This register is useful to software in determining which revision of the processor is executing the code. The format of this register is illustrated in Figure 5.5; the value currently returned is 0x0000_0230, which is the same as the R3000A.


0: READ AS 0, MUST BE WRITTEN AS 0

Implementation: EXECUTION ENGINE IMPLEMENTATION CODE

Revision: REVISION LEVEL FOR THIS IMPLEMENTATION

4000 drw 29

**Figure 5.5. Format of Prid Register**

## EXCEPTION VECTOR LOCATIONS

The R3051 family separates exceptions into three vector spaces. The value of each vector depends on the BEV (Boot Exception Vector) bit of the status register, which allows two alternate sets of vectors (and thus two different pieces of code) to be used. Typically, this is used to allow diagnostic tests to occur before the functionality of the cache is validated; processor reset forces the value of the BEV bit to a 1. Tables 5.4 and 5.5 list the exception vectors for the R3051 family for the two different modes.

| Exception | Virtual Address | Physical Address |
|---|---|---|
| **Reset** | 0xbfc0_0000 | 0x1fc0_0000 |
| **UTLB Miss** | 0x8000_0000 | 0x0000_0000 |
| **General** | 0x8000_0080 | 0x0000_0080 |

4000 tbl 20

**Table 5.4. Exception Vectors When BEV = 0**

| Exception | Virtual Address | Physical Address |
|---|---|---|
| **Reset** | 0xbfc0_0000 | 0x1fc0_0000 |
| **UTLB Miss** | 0xbfc0_0100 | 0x1fc0_0100 |
| **General** | 0xbfc0_0180 | 0x1fc0_0180 |

4000 tbl 21

**Table 5.5. Exception Vectors When BEV = 1**

## EXCEPTION PRIORITIZATION

It is important to understand the structure of the R3051 family instruction execution unit in order to understand the exception priority model of the processor. The R3051 family runs instructions through a five stage pipeline, illustrated in Figure 5.6. The pipeline stages are:

- IF: Instruction Fetch. This cycle contains two parts: the IVA (Instruction Virtual Address) phase, which generates the virtual instruction address of the next instruction to be fetched, and the ITLB phase, which performs the virtual to physical translation of the address.

- RD: Read and Decode. This phase obtains the required data from the internal registers and also decodes the instruction.

- ALU: This phase either performs the desired arithmetic or logical operation, or generates the address for the upcoming data operation. For data operations, this phase contains both the data virtual address stage, which generates the desired virtual address, and the data TLB stage, which performs the virtual to physical translation.

- MEM: Memory. This phase performs the data load or store transaction.

- WB: Write Back. This stage updates the registers with the result data.

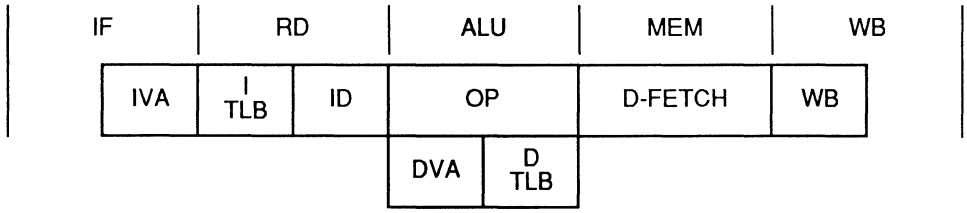



**Figure 5.6. Pipelining in the R3051 Family**

High performance is achieved because five instructions are operating concurrently, each in a different stage of the pipeline. However, since multiple instructions are operating concurrently, it is possible that multiple exceptions are generated concurrently. If so, the processor must decide which exception to process, basing this decision on the stage of the pipeline that detected the exception. The processor will then flush all preceding pipeline stages to avoid altering processor context, thus implementing precise exceptions. This determines the relative priority of the exceptions.

For example, an illegal instruction exception can only be detected in the instruction decode stage of the R3051/52; an Instruction Bus Error can only be determined in the I-Fetch pipe stage. Since the illegal instruction was fetched before the instruction which generated the bus error was fetched, and since it is conceivable that handling this exception might have avoided the second exception, it is important that the processor handle the illegal instruction before the bus error. Therefore the exception detected in the latest pipeline stage has priority over exceptions detected in earlier pipeline stages. All instructions fetched subsequent to this (all preceding pipeline stages) are flushed to avoid altering state information, maintaining the precise exception model.

Table 5.6 lists the priority of exceptions from highest first to lowest.

| Mnemonic | Pipestage |
|---|---|
| Reset | Any |
| AdEL | Memory (Load instruction) |
| AdES | Memory (Store instruction) |
| DBE | Memory (Load or store) |
| MOD | ALU (Data TLB) |
| TLBL | ALU (DTLB Miss) |
| TLBS | ALU (DTLB Miss) |
| Int | ALU |
| Sys | RD (Instruction Decode) |
| Bp | RD (Instruction Decode) |
| RI | RD (Instruction Decode) |
| CpU | RD (Instruction Decode) |
| Ovf | RD (Instruction Decode) |
| TLBL | I-Fetch (ITLB Miss) |
| AdEL | IVA (Instruction Virtual Address |
| IBE | RD (end of I-Fetch) |

4000 tbl 22

**Table 5.6. R3051 Family Exception Priority**

## EXCEPTION LATENCY

A critical measurement of a processor's throughput in interrupt driven systems is the interrupt "latency" of the system. Interrupt latency is a measurement of the amount time from the assertion of an interrupt until software begins handling that interrupt. Often included when discussing latency is the amount of overhead associated with restoring context once the exception is handled, although this is typically less critical than the initial latency.

In systems where the processor is responsible for managing a number of time-critical operations in real time, it is important that the processor minimize interrupt latency. That is, it is more important that *every* interrupt be handled at a rate above some given value, rather than *occasionally* handle an interrupt at very high speed.

Factors which affect the interrupt latency of a system include the types of operations it performs (that is, systems which have long sequences of operations during which interrupts can not be accepted have long latency), how much information must be stored and restored to preserve and restore processor context, and the priority scheme of the system.

Table 5.6 illustrates which pipestage recognizes which exceptions. As mentioned above, all instructions less advanced in the pipeline are flushed from the pipeline to avoid altering state execution. Those instructions will be restarted when the exception handler completes.

Once the exception is recognized, the address of the appropriate exception vector will be the next instruction to be fetched. In general, the latency to the exception handler is one instruction cycle, and at worst the longest stall cycle in that system.

## INTERRUPTS IN THE R3051 FAMILY

The R3051 family features two types of interrupt inputs: synchronized internally and non-synchronized, or direct.

The $\overline{\text{SInt}}$(2:0) bus (Synchronized Interrupts) allow the system designer to connect unsynchronized interrupt sources to the processor. The processor includes special logic on these inputs to avoid meta-stable states associated with switching inputs right at the processor sampling point. Because of this logic, these interrupt sources have slightly longer latency from the $\overline{\text{SInt}}$(n) pin to the exception vector than the non-synchronized inputs. The operation of the synchronized interrupts is illustrated in Figure 5.7.

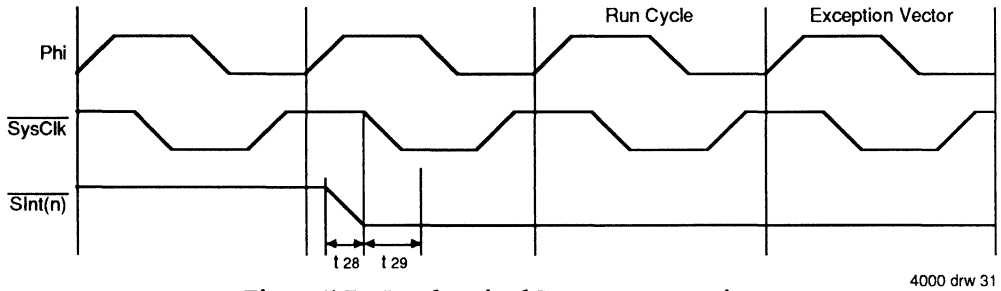



**Figure 5.7. Synchronized Interrupt Operation**

The other interrupts, $\overline{\text{Int}}$(5:3), do not contain this synchronization logic, and thus have slightly better latency to the exception vector. However, the interrupting agent must guarantee that it always meets the interrupt input set-up and hold time requirements of the processor. These inputs are useful for interrupting agents which operate off of the $\overline{\text{SysClk}}$ output of the R3051/52. The operation of these interrupts is illustrated in Figure 5.8.

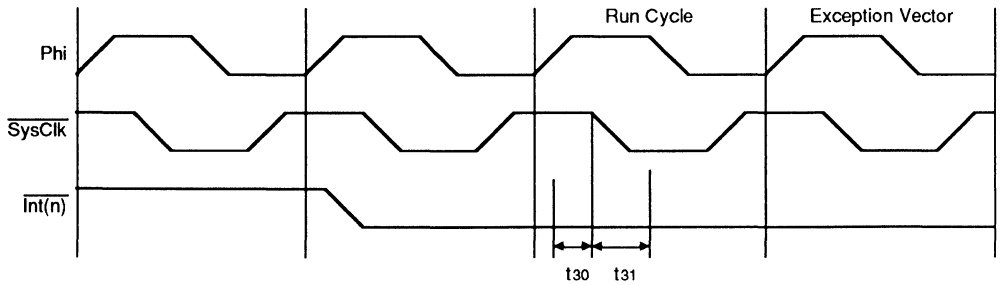



**Figure 5.8. Direct Interrupt Operation**

Since the interrupt exception is detected during the ALU stage of the instruction currently in the processor pipeline, at least one run cycle must occur between (or at) the assertion of the external interrupt input and the fetch of the exception vector. Thus, if the processor is in a stall cycle when an external agent sends an interrupt, it will execute at least one run cycle before beginning exception processing. In this instance, there would be no difference in the latency of synchronized and direct interrupt inputs.

All of the interrupts are level-sensitive and active low. They continue to be sampled after an interrupt exception has occurred, and are not latched within the processor when an interrupt exception occurs. It is important that the external interrupting agent maintain the interrupt line until software acknowledges the interrupt.

Note that future family members will incorporate the floating point on-chip. The MIPS architecture recommends that Int(3) be used to handle the floating point interrupt.

Each of the eight interrupts (6 hardware and 2 software) can be individually masked by clearing the corresponding bit in the Interrupt Mask field of the Status Register. All eight interrupts can be masked at once by clearing the IEc bit in the Status Register.

On the synchronized interrupts, care should be taken to allow at least two clock cycles between the negation of the interrupt input and the re-enabling of the interrupt mask for that bit.

The value shown in the interrupt pending bits of the Cause register reflects the current state of the interrupt pins of the processor. These bits are not latched (except for sampling from the data bus to guarantee that they are stable when examined), and the masking of specific interrupt inputs does not mask the bits from being read.

## USING THE BrCond INPUTS

In addition to the interrupt pins themselves, many systems can use the BrCond input port pins in their exception model. These pins can be directly tested by software, and can be used for polling or fast interrupt decoding.

As with the interrupt bus, there are two versions of the BrCond pins. BrCond(1:0) are direct inputs, and thus the set-up and hold requirements of the processor must be met. BrCond(3:2) are synchronized inputs, and thus may be driven by asynchronous sources. The timing requirements of the BrCond inputs are illustrated in Figure 5.9 and Figure 5.10.

Note that the future versions of the R3051 family which incorporate the floating point unit on-chip will utilize BrCond(1) to communicate with that execution unit, and thus it will be unavailable to external agents.

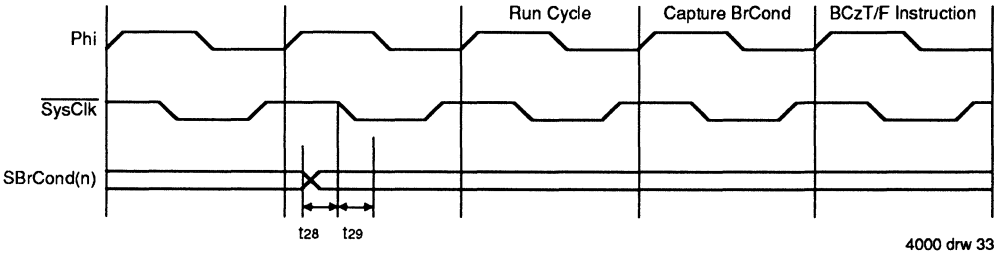



**Figure 5.9. Synchronized BrCond Inputs**

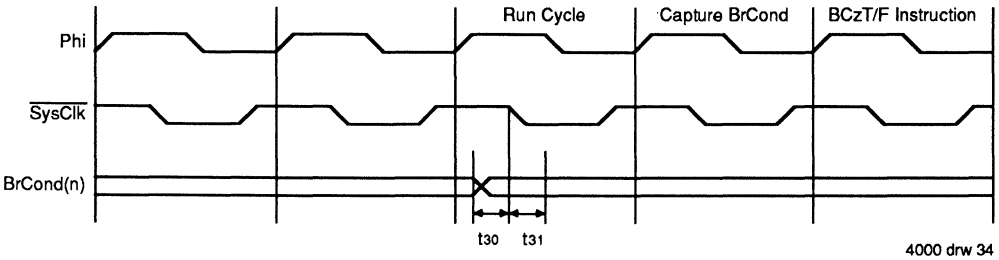



**Figure 5.10. Direct BrCond Inputs**

Similar to the interrupt inputs, at least one instruction must be executed (in the ALU stage) of the instruction pipeline prior to software being able to detect a change in one of these inputs. This is because the processor actually captures the value of these flags one instruction prior to the branch on co-processor instruction. Thus, if the processor is in a stall when the flag changes, there will be no difference in the time required for the processor to recognize synchronized or direct BrCond inputs.

## INTERRUPT HANDLING

The assertion of an unmasked interrupt input causes the R3051 family to branch to the general exception vector at virtual address 0x8000_0080, and write the 'Int' code in the Cause register. The IP field of the Cause register shows which of the six hardware interrupts are pending and the SW field in the Cause register show which of the two software interrupts are pending. Multiple interrupts can be pending at the same time, with no priority assumed by the processor.

When an interrupt occurs, the KUp, IEp, KUc and IEc bits of the Status register are saved in the KUo, IEo, KUp, IEp bit fields in the Status register, respectively, as illustrated in Figure 5.11. The current kernel status bit KUc and the interrupt bit IEc are cleared. This masks all the interrupts and places the processor in kernel mode. This sequence will be reversed by the execution of an rfe (restore from exception) instruction.

## INTERRUPT SERVICING

In case of an hardware interrupt, the interrupt must be cleared by de-asserting the interrupt line, which has to be done by alleviating the external conditions that caused the interrupt. Software interrupts have to be cleared by clearing the corresponding bits, SW(1:0), in the Cause register to zero.

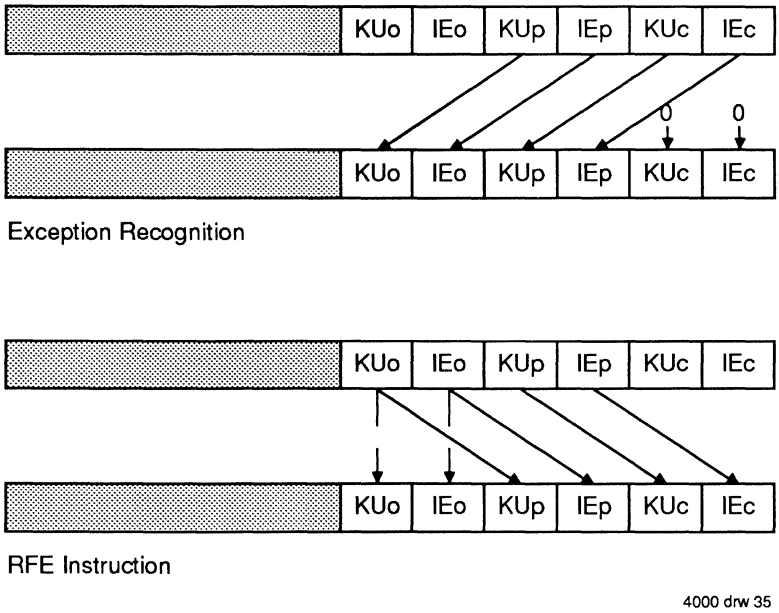



**Figure 5.11. Kernel and Interrupt Status Being Saved on Interrupts**

## BASIC SOFTWARE TECHNIQUES FOR HANDLING INTERRUPTS

Once an exception is detected the processor suspends the current task, enters kernel mode, disables interrupts, and begins processing at the exception vector location. The EPC is loaded with the address the processor will return to once the exception event is handled.

The specific actions of the processor depend on the cause of the exception being handled. The R3051 family classifies exceptions into three distinct classes: RESET, UTLB Miss, and General.

Coming out of reset, the processor initializes the state of the machine. In addition to initializing system peripherals, page tables, the TLB, and the caches, software clears both STATUS and CAUSE registers, and initializes the exception vectors.

The code located at the exception vector may be just a branch to the actual exception code; however, in more time critical systems the instructions located at the exception vector may perform the actual exception processing. In order to cause the exception vector location to branch to the appropriate exception handler (presuming that such a jump is appropriate), a short code sequence such as that illustrated in Figure 5.12 may be used.

```
        .set    noreorder               # tells the assembler not to reorder the code
/*
**      code sequence copied to UTLB exception vector
*/
        la      k0,excep_utlb           #address of utlb excp. handler
        j       k0                      # jump via reg k0
        nop

/*
**      code sequence copied to general exception vector
*/
        la      k0,excep_general        #address of general excp. handler
        j       k0                      # jump via reg k0
        nop
```

4000 drw 36

**Figure 5.12. Code Sequence to Initialize Exception Vectors**

It should be noted the the contents of register k0 are not preserved. This is not a problem for software, since MIPS compiler and assembler conventions reserve k0 for kernel processes, and do not use it for user programs. For the system developer it is advised that the use of k0 be reserved for use by the exception handling code exclusively. This will make debugging and development much easier.

## PRESERVING CONTEXT

The R3051 family has the following five registers related to exception processing:

1. The *Cause* register
2. The *EPC* (exception program counter) register
3. The *Status* register
4. The *BadVAddr* (bad virtual address) register
5. The *Context* register

Typical exception handlers preserve the status, cause, and EPC registers in general registers (or on the system stack). If the exception cause is due to a TLB miss, software may also preserve the bad virtual address and context registers for later processing.

Note that not all systems need to preserve this information. Since the R3051 family disables subsequent interrupts, it is possible for software to directly process the exception while leaving the processor context in the CP0 registers. Care must be taken to insure that the execution of the exception handler does not generate subsequent exceptions.

Preserving the context in general registers (and on the stack) does have the advantage that interrupts can be re-enabled while the original exception is handled, thus allowing a priority interrupt model to be built.

A typical code sequence to preserve processor context is shown in Figure 5.13. This code sequence preserves the context into an area of memory pointed to by the k0 kernel register. This register points to a block of memory capable of storing processor context. Constants identified by name (such as *R_EPC*) are used to indicate the offset of a particular register from the start of that memory area.

```
        la      k0,except_regs      # fetch address of reg save array
        sw      AT,R_AT*4(k0)       # save register AT
        sw      v0,R_V0*4(k0)       # save register v0
        sw      v1,R_V1*4(k0)       # save register v1
        mfc0    v0,C0_EPC           # fetch the epc register
        mfc0    v1,C0_SR            # fetch the status register
        sw      v0,R_EPC*4(k0)      # save the epc
        mfc0    v0,C0_CAUSE         # fetch the cause register
        sw      v1,R_SR*4(k0)       # save status register

/*      The above code is about the minimum required
**      The user specific code would follow
*/
```

4000 drw 37

**Figure 5.13. Preserving Processor Context**

It should be noted that this sequence for fetching the co-processor zero registers is required because there is a one clock delay in the register value actually being loaded into the general registers after the execution of the mfc0 instruction.

## DETERMINING THE CAUSE OF THE EXCEPTION

The cause register indicates the reason the exception handler was invoked. Thus, to invoke the appropriate exception service routine, software merely needs to examine the cause register, and use its contents to direct a branch to the appropriate handler.

One method of decoding the jump to an appropriate software routine to handle the exception and cause is shown in Figure 5.14. Register v0 contains the cause register, and register k0 still points to the register save array.

```
        .set    noreorder
        sw      a0,R_A0*4(k0)       # save register a0
        and     v1,v0,EXCMASK       # isolate exception code
        lw      a0,cause_table(v1)  # get address of interrupt routine.
        sw      a1,R_A1*4(k0)       # use delay slot to save register a1
        j       a0
        sw      k1,R_K1*4(sp)       # save k1 register
        .set    reorder             # re-enable pipeline scheduling
```

4000 drw 38

**Figure 5.14. Exception Cause Decoding**

The above sequence of instructions extracts the exception code from the cause register and uses that code to index into the table of pointers to functions (the cause_table). The *cause_table* data structure is shown in Figure 5.15.

```
int (*cause_table[16])() = {
        int_extern,             /* External interrupts                  */
        int_tlbmod,             /* TLB modification error               */
        int_tlbmiss,            /* load or instruction fetch            */
        int_tlbmiss,            /* write miss                           */
        int_addrerr,            /* load or instruction fetch            */
        int_addrerr,            /* write address error                  */
        int_ibe,                /* Bus error - Instruction fetch        */
        int_dbe,                /* Bus error - load or store data       */
        int_syscall,            /* SYSCALL exception                    */
        int_breakpoint,         /* breakpoint instruction               */
        int_trap,               /* Reserved instruction                 */
        int_cpunuse,            /* coprocessor unusable                 */
        int_trap,               /* Arithmetic overflow                  */
        int_unexp,              /* Reserved                             */
        int_unexp,              /* Reserved                             */
        int_unexp               /* Reserved                             */
        };
```

4000 drw 39

**Figure 5.15. Exception Service Branch Table**

Each of the entries in this table point to a function for processing the particular type of interrupt detected. The specifics of the code contained in each of these functions is unique for a given application; all registers used in these functions must be saved and restored.

## RETURNING FROM EXCEPTIONS

Returning from the exception routine is made through the *rfe* instruction. When the exception first occurs the R3051/52 automatically saves some of the processor context, the current value of the interrupt enable bit is saved into the field for the previous interrupt enable bit, and the kernel/user mode context is preserved.

The *IE* interrupt enable bit must be asserted (a one) for external interrupts to be recognized. The *KU* kernel mode bit must be a zero in kernel mode. When an exception occurs, external interrupts are disabled and the processor is forced into kernel mode. When the *rfe* instruction is executed at completion of exception handling, the state of the mode bits is restored to what it was when the exception was recognized (presuming the programmer restored the status register to its value when the exception occurred). This is done by "popping" the old/previous/current KU and IE bits of the status register.

The code sequence in Figure 5.16 is an example of exiting an interrupt handler. The assumption is that registers and context were saved as outlined above.

```
gen_excp_exit:
        .set    noreorder
                                # by the time we have gotten here
                                # all general registers have been
                                # restored (except of k0 and v0)
                                # reg. AT points to the reg save array
        lw      k0,C0_SR*4(AT)  # fetch status reg. contents
        lw      v0,R_V0*4(AT)   # restore reg. v0
        mtc0    k0,C0_SR        # restore the status reg. contents
        lw      k0,R_EPC*4(AT)  # Get the return address
        lw      AT,R_AT*4(AT)   # restore AT in load delay
        j       k0              # return from int. via jump reg.
        rfe                     # the rfe instr. is executed in the
                                # branch delay slot
        .set    reorder
```

4000 drw 40

**Figure 5.16. Returning from Exception**

This code sequence must either be replicated in each of the cause handling functions, or each of them must branch to this code sequence to properly exit from exception handling.

Note that this code sequence must be executed with interrupts disabled. If the exception handler routine re-enables interrupts they must be disabled when the CP0 registers are being restored.

## SPECIAL TECHNIQUES FOR INTERRUPT HANDLING

There are a number of techniques which take advantage of the R3051 family architecture to minimize exception latency and maximize throughput in interrupt driven systems. This section discusses a number of those techniques.

### Interrupt Masking

Only the six external and two software interrupts are maskable. The mask for these interrupts are in the status register.

To enable a given external interrupt, the corresponding bit in the status register must be set. The IEc bit in the status register must also be set. It follows that by setting and clearing these bits within the interrupt handler that interrupt priorities can be established. The general mechanism for doing this is performed within the external interrupt-handler portion of the exception handler.

The interrupt handler preserves the current mask value when the status register is preserved. The interrupt handler then calculates which (if any) external interrupts have priority, and sets the interrupt mask bit field of the status register accordingly. Once this is done, the IEc bit is changed to allow higher priority interrupts. Note that all interrupts must again be disabled when the return from exception is processed.

### Using BrCond For Fast Response

The R3051 family instruction set contains mechanisms to allow external or internal co-processors to operate as an extension of the main CPU. Some of these features may also be used in an interrupt-driven system to provide the highest levels of response.

Specifically, the R3051 family has external input port signals, the BrCond(0:3) signals. These signals are used by external agents to report status back to the processor. The instruction set contains instructions which allow the external bits to be tested, and branches to be executed depending on the value of BrCond.

An interrupt-driven system can use these BrCond signals, and the corresponding instructions, to implement an input port for time-critical interrupts. Rather than mapping an input port in memory (which requires external logic), the BrCond signals can be examined by software to control interrupt handling.

There are actually two methods of advantageously using this. One method uses these signals to perform interrupt polling; in this method, the processor continually examines these signals, waiting for an appropriate value before handling the interrupt. A sample code sequence is shown in Figure 5.17.

```
        .set        noreorder           # prevents the assembler from
                                        # reordering the code below

polling_loop:                           # branch to yourself until
        bc2f        polling_loop        # BrCond(2) is asserted
        nop
                                        # Once BrCond(2) is asserted, fall through
                                        # and begin processing the external event
fast_response_cp2:
                                        # code sequence that would do the
                                        # event processing


        b           polling_loop        # return to polling
```

4000 drw 41

**Figure 5.17. Polling System Using BrCond**

The software in this system is very compact, and easily resides in the on-chip cache of the processor. Thus, the latency to the interrupt service routine in this system is minimized, allowing the fastest interrupt service capabilities.

A second method utilizes external interrupts combined with the BrCond signals. In this method, both the BrCond signal and one of the external interrupt lines are asserted when an external event occurs. This configuration allows the CPU to perform normal tasks while waiting for the external event.

For example, assume that that a valve must be closed and then normal processing continued when BrCond(2) is asserted TRUE. The valve is controlled by a register that is memory-mapped to address 0xaffe_0020 and writing a one to this location closes the valve. The software in Figure 5.18 accomplishes this, using BrCond(2) to aid in cause decoding.

The number of cycles for a deterministic system is five cycles between the time the interrupt occurred and it was serviced. Interrupts were re-enabled in four additional cycles. Note that none of the processor context needs to be preserved and restored for this routine.

```
        .set    noreorder               # prevents the assembler from reordering
                                        # the code sequences below

/* This section of code is placed at the general exception
** vector location 0x8000_0080. When an external interrupt is
** asserted execution begins here.
*/

        bc2t    close_valve             # test for emergency condition and
        li      k0,1                    # jump to close valve if TRUE
        la      k0,gen_exp_hand         # otherwise,
        j       k0                      # jump to general exc. handler
        nop                             # and process less critical excepts.


/* This is the close valve routine - its sole purpose is to close the
** valve as quickly as possible. The registers 'k0' and 'k1' are reserved
** for kernel use and therefore need not be saved when a client or
** user program is interrupted. It should be noted that the value to
** write  to the valve close register was put in reg 'k0' in the
** branch delay slot above - so by the time we get here it is
** ready to output to the close register.
*/
close_valve:
        la      k1,0xaffe0020           # the address of the close register
        sw      k0,0(k1)                # write the value to the close register
        mfc0    k0,C0_EPC               # get the return address to cont  processing
        nop
        j       k0                      # return to normal processing
        rfe                             # restore previous interrupt mask
                                        # and kernel/user mode bits of the
                                        # status register.
        .set    reorder
```

4000 drw 42

**Figure 5.18. Using BrCond for Fast Interrupt Decoding**

### Nested Interrupts

Note that the processor does not automatically stack processor context when an exception occurs; thus, to allow nested exceptions it is important that software perform this stacking.

Most of the software illustrated above also applies to a nested exception system. However, rather than using just one register (pointed to by k0) as a save area, a stacking area must be implemented and managed by software. Also, since interrupts are automatically disabled once an exception is detected, the interrupt handling routine must mask the interrupt it is currently servicing, re-enable other interrupts (once context is preserved) through the IEc bit.

The use of Interrupt Mask bits of the status register to implement an interrupt prioritization scheme was discussed earlier. An analogous technique can be performed by using an external interrupt encoder to allow more interrupt sources to be presented to the processor.

Software interrupts can also be used as part of the prioritization of interrupts. If the interrupt service routine desires to service the interrupting agent, but not completely perform the interrupt service, it can cause the external agent to negate the interrupt input but leave interrupt service pending through the use of the SW bits of the Cause register.

### Catastrophic Exceptions

There are certain types of exceptions that indicate fundamental problems with the system. Although there is little the software can do to handle such events, they are worth discussing. Exceptions such as these are typically associated with faulty systems, such as in the initial debugging or development of the system.

Potential problems can arise because the processor does not automatically stack context information when an exception is detected. If the processor context has not been preserved when another exception is recognized, the value of the status, cause, and EPC registers are lost and thus the original task can not be resumed.

An example of this occurring is an exception handler performing a memory reference that results in a bus error (for example, when attempting to preserve context). The bus error forces execution to the exception vector location, overwriting the status, cause, and context registers. Proper operation cannot be resumed.

## HANDLING SPECIFIC EXCEPTIONS

This section documents some specific issues and techniques for handling particular R3051 family exceptions.

### Address Error Exception

#### Cause

This exception occurs when an attempt is made to load, fetch, or store a word that is not aligned on a word boundary. Attempting to load or store a half-word that is not aligned on a half-word boundary will also cause this exception. The exception also occurs in User mode if a reference is made to a virtual address whose most significant bit is set (a kernel address). This exception is not maskable.

#### Handling

The R3051 family branches to the General Exception vector for this exception. When the exception occurs, the R3051/52 sets the ADEL or ADES code in the Cause register ExcCode field to indicate whether the address error occurred during an instruction fetch or a load operation (ADEL) or a store operation (ADES).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively and clears the KUc and IEc bits.

When this exception occurs, the BadVAddr register contains the virtual address that was not properly aligned or that improperly addressed kernel data while in User mode. The contents of the VPN field of the Context and EntryHi registers are undefined.

#### Servicing

A kernel should hand the executing process a segmentation violation signal. Such an error is usually fatal although an alignment error might be handled by simulating the instruction that caused the error.

## Breakpoint Exception

### Cause

This exception occurs when the R3051/52 executes the BREAK instruction. This exception is not maskable.

### Handling

The R3051/52 branches to the General Exception vector for the exception and sets the BP code in the CAUSE register ExcCode field.

The R3051/52 saves the *KUp*, *IEp*, *KUc*, and *IEc* bits of the Status register in the *KUo*, *KUp*, and *IEp* bits, respectively, and clears the *KUc* and *IEc* bits.

The *EPC* register points at the BREAK instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the *EPC* register points at the BRANCH instruction that preceded the BREAK instruction and sets the *BD* bit of the Cause register.

### Service

The breakpoint exception is typically handled by a dedicated system routine. Unused bits of the BREAK instruction (bits 25..6) can be used pass additional information. To examine these bits, load the contents of the instruction pointed at by the *EPC* register. NOTE: If the instruction resides in the branch delay slot, add four to the contents of the *EPC* register to find the instruction.

To resume execution, change the *EPC* register so that the R3051/52 does not execute the BREAK instruction again. To do this, add four to the *EPC* register before returning. NOTE: If a BREAK instruction is in the branch delay slot, the BRANCH instruction must be interpreted in order to resume execution.

### Bus Error Exception

#### Cause

This exception occurs when the Bus Error input to the CPU is asserted by external logic during a read operation. For example, events like bus time-outs, backplane bus parity errors, and invalid physical memory addresses or access types can signal exception. This exception is not maskable.

This exception is used for synchronously occurring events such as cache miss refills. The general interrupt mechanism must be used to report a bus error that results from asynchronous events such as a buffered write transaction.

#### Handling

The R3051/52 branches to the General Exception vector for this exception. When exception occurs, the R3051/52 sets the *IBE* or *DBE* code in the CAUSE register ExcCode field to indicate whether the error occurred during an instruction fetch reference (IBE) or during a data load or store reference (DBE).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the BRANCH instruction that preceded the exception-causing instruction and sets the BD bit of the cause register.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the Status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

#### Servicing

The physical address where the fault occurred can be computed from the information in the CP0 registers:

- If the Cause register's IBE code is set (showing an instruction fetch reference), the virtual address resides in the EPC register.

- If the Cause register's DBE exception code is set (specifying a load or store reference), the instruction that caused the exception is at the virtual address contained in the EPC register (if the BD bit of the cause register is set, add four to the contents of the EPC register). Interpret the instruction to get the virtual address of the load or store reference and then use the TLBProbe(tlbp) instruction and read EntryLo to compute the physical page number.

A kernel should hand the executing process a bus error when this exception occurs. Such an error is usually fatal.

### Co-processor Unusable Exception

#### Cause

This exception occurs due to an attempt to execute a co-processor instruction when the corresponding co-processor unit has not been marked usable (the appropriate CU bit in the status register has not been set). For CP0 instructions, this exception occurs when the unit has not been marked usable and the process is executing in User mode: CP0 is always usable from Kernel mode regardless of the setting of the CP0 bit in the status register. This exception is not maskable.

#### Handling

The R3051/52 branches to the General Exception vector for this exception. It sets the CPU code in the CAUSE register ExcCode field. Only one co-processor can fail at a time.

The contents of the cause register's CE (Co-processor Error) field show which of the four coprocessors (3,2,1, or 0) the R3051/52 referenced when the exception occurred.

The EPC register points at the co-processor instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the co-processor instruction and sets the BD bit of the Cause register.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

#### Servicing

To identify the co-processor unit that was referenced, examine the contents of the Cause register's CE field. If the process is entitled to access, mark the co-processor usable and restore the corresponding user state to the co-processor.

If the process is entitled to access to the co-processor, but the co-processor is known not to exist or to have failed, the system could interpret the co-processor instruction. If the BD bit is set in the Cause register, the BRANCH instruction must be interpreted; then, the co-processor instruction could be emulated with the EPC register advanced past the co-processor instruction.

If the process is not entitled to access to the co-processor, the process executing at the time should be handed an illegal instruction/privileged instruction fault signal. Such an error is usually fatal.

### Interrupt Exception

#### Cause

This exception occurs when one of eight interrupt conditions (software generates two, hardware generates six) occurs.

Each of the eight external interrupts can be individually masked by clearing the corresponding bit in the IntMask field of the status register. All eight of the interrupts can be masked at once by clearing the IEc bit in the status register.

#### Handling

The R3051/52 branches to the General Exception vector for this exception. The R3051/52 sets the INT code in the Cause register's ExcCode field.

The IP field in the Cause register show which of six external interrupts are pending, and the SW field in the cause register shows which two software interrupts are pending. More than one interrupt can be pending at a time.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

#### Servicing

If software generates the interrupt, clear the interrupt condition by setting the corresponding Cause register bit (SW1:0) to zero.

If external hardware generated the interrupt, clear the interrupt condition by alleviating the conditions that assert the interrupt signal.

### Overflow Exception

#### Cause

This exception occurs when an **ADD ADDI, SUB,** or **SUBI** instruction results in two's complement overflow. This exception is not maskable.

#### Handling

The R3051/52 branches to the General Exception vector for this exception. The R3051/52 sets the OV code in the CAUSE register.

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the Branch instruction that preceded the exception-causing instruction and sets the BD bit of the CAUSE register.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

### Reserved Instruction Exception

**Cause**

This exception occurs when the R3051/52 executes an instruction whose major opcode (bits 31..26) is undefined or a Special instruction whose minor opcode (bits 5..0) is undefined.

This exception provides a way to interpret instructions that might be added to or removed from the R3051/52 processor architecture.

**Handling**

The R3051/52 branches to the General Exception vector for this exception. It sets the RI code of the Cause register's ExcCode field.

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the Branch instruction that preceded the reserved instruction and sets the BD bit of the CAUSE register.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

**Servicing**

If instruction interpretation is not implemented, the kernel should hand the executing process an illegal instruction/reserved operand fault signal. Such an error is usually fatal.

An operating system can interpret the undefined instruction and pass control to a routine that implements the instruction in software. If the undefined instruction is in the branch delay slot, the routine that implements the instruction is responsible for simulating the branch instruction after the undefined instruction has been "executed". Simulation of the branch instruction includes determining if the conditions of the branch were met and transferring control to the branch target address ( if required) or to the instruction following the delay slot if the branch is not taken. If the branch is not taken, the next instruction's address is [EPC] + 8. If the branch is taken, the branch target address is calculated as [EPC] + 4 + (Branch Offset * 4).

Note that the target address is relative to the address of the instruction in the delay slot, not the address of the branch instruction. Refer to the description of branch instruction for details on how branch target addresses are calculated.

### Reset Exception

#### Cause

This exception occurs when the R3051/52 RESET signal is asserted and then de-asserted.

#### Handling

The R3051/52 provides a special exception vector for this exception. The Reset vector resides in the R3051/52's unmapped and uncached address space; Therefore the hardware need not initialize the Translation Lookaside Buffer (TLB) or the cache to handle this exception. The processor can fetch and execute instructions while the caches and virtual memory are in an undefined state.

The contents of all registers in the R3051/52 are undefined when this exception occurs except for the following:

- The SWc, KUc, and IEc bits of the Status register are cleared to zero.
- The BEV bit of the Status register is set to one.
- The Random register is initialized to 63.
- For extended versions of the architecture, the TS bit is cleared to zero.
- For base versions of the architecture, the TS bit is frozen at one.

#### Servicing

The reset exception is serviced by initializing all processor registers, co-processor registers, the caches, and the memory system. Typically, diagnostics would then be executed and the operating system bootstrapped. The reset exception vector is selected to appear in the uncached, unmapped memory space of the machine so that instructions can be fetched and executed while the cache and virtual memory system are still in an undefined state.

### System Call Exception

**Cause**

This exception occurs when the R3051/52 executes a SYSCALL instruction.

**Handling**

The R3051/52 branches to the General Exception vector for this exception and sets the SYS code in the CAUSE register's ExcCode field.

The EPC register points at the SYSCALL instruction that caused the exception, unless the SYSCALL instruction is in a branch delay slot: in that case, the EPC register points at the branch instruction that preceded the SYSCALL instruction and the BD bit of the CAUSEregister is set.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

**Servicing**

The operating system transfers control to the applicable system routine. To resume execution, alter the EPC register so that the SYSCALL instruction does not execute again. To do this, add four to the EPC register before returning. NOTE: If a SYSCALL instruction is in a branch delay slot, the branch instruction must be interpreted in order to resume execution.

### TLB Miss Exceptions

There are three different types of TLB misses that can occur:

- If the input Virtual Page Number (VPN) does not match the VPN of any TLB entry, or if the Process Identifier (PID) in EntryHi does not match the TLB entry's PID (and the Global bit is not set), a miss occurs. For KUSEG references, a UTLB Miss exception is taken. For KSEG2 references, a TLB Miss occurs.
- If everything matches, but the valid bit of the matching TLB entry is not set, a TLB Miss occurs.
- If the dirty bit in a matching TLB entry is not set and the access is a write, a TLB MOD exception occurs.

Figure 5.19 (a simplified version of TLB address translation figure used in Chapter 4) illustrates how the three different kinds of TLB miss exceptions are generated. Each of the exceptions is described in detail in the pages that follow.

The TLB exceptions obviously only occur in extended architecture versions of the processor.

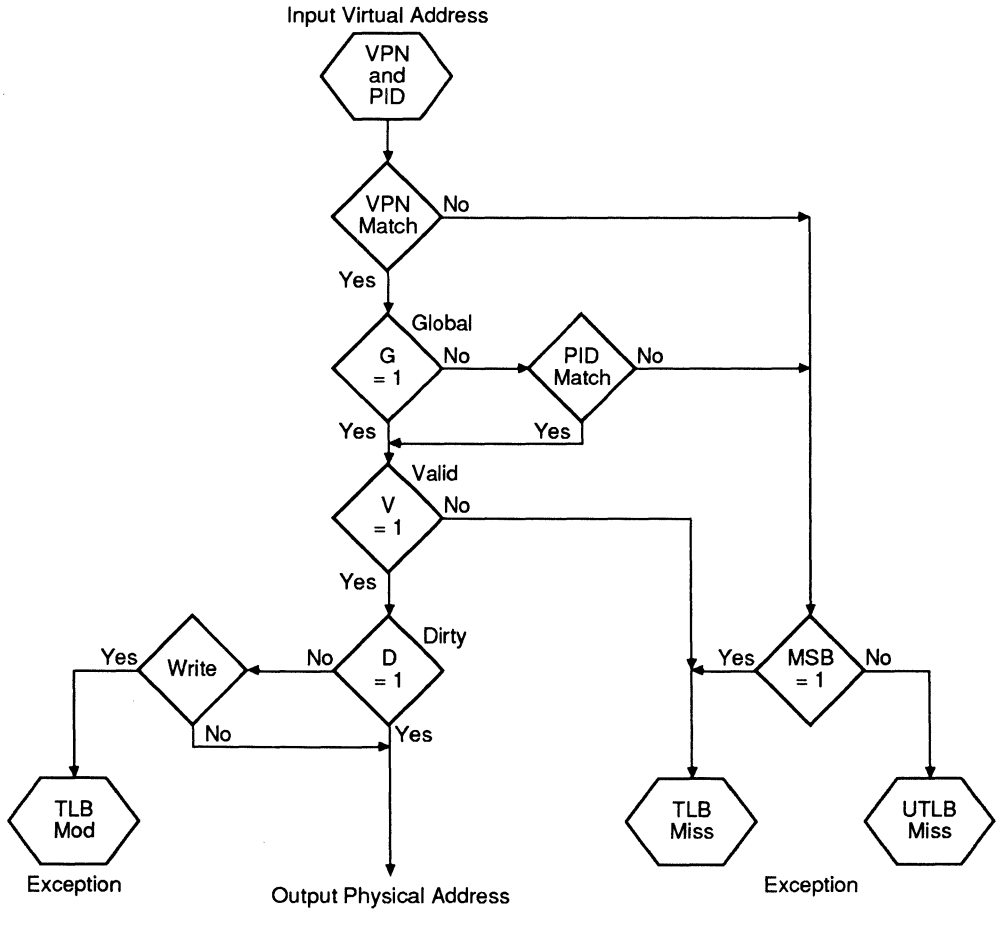



**Figure 5.19. TLB Miss Exceptions**

### TLB Miss Exception

#### Cause

This exception occurs when a Kernel mode virtual address reference to memory is not mapped, when a User mode virtual address reference to memory matches an invalid TLB entry, or when a Kernel mode reference to user memory space matches an invalid TLB entry.

#### Handling

The R3051/52 branches to the General Exception vector for this exception. When the exception occurs, the R3051/52 sets the TLBL or TLBS code in the CAUSE register's ExcCode field to indicate whether the miss was due to an instruction fetch or a load operation (TLBL) or a store operation (TLBS).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the Branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register. The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

When this exception occurs, the BadVAddr, Context, and EntryHi register contain the virtual address that failed address translation. The PID field of EntryHi remains unchanged by this exception. The Random register normally specifies the pseudo-random location where the R3051/52 can put a replacement TLB entry.

#### Servicing

The failing virtual address or virtual page number identifies the corresponding PTE. The operating system should load EntryLo with the appropriate PTE that contains the physical page frame and access control bits and also write the contents of EntryLo and EntryHi into the TLB.

#### Servicing Multiple (nested) TLB Misses

Within a UTLB Miss handler, the virtual address that specifies the PTE contains physical address and access control information that might not be mapped in the TLB. Then, a TLB Miss exception occurs. This case is recognized by noting that the EPC register points within the UTLB Miss handler. The operating system might interpret the event as an address error (when the virtual address falls outside the valid region for the process) or as a TLB Miss on the page mapping table.

This second TLB miss obscures the contents of the BadVAddr, Context, and EntryHi registers as they were within the UTLB Miss handler. As a result, the exact virtual address whose translation caused the first fault is not known unless the UTLB Miss handler specifically saved this address. You can only observe the failing PTE virtual address. The BadVAddr register now contains the original contents of the Context register within the UTLB Miss handler, which is the PTE for the original faulting address.

If the operating system interprets the exception as a TLB Miss on the page table, it constructs a TLB entry to map the page table and writes the entry into the TLB. Then, the operating system can determine the original faulting virtual page number, but not the complete address. The operating system uses this information to fetch the PTE that contains the physical address and access control information. It also writes this information into the TLB.

The UTLB Miss handler must save the EPC in a way that allows the second miss to find it. The EPC register information that the UTLB Miss handler saved gives the correct address at which to resume execution. The"old" KUo and IEo bits of the status register contain the correct mode after the R3051/52 services a double miss. NOTE: You neither need nor want to return to the UTLB Miss handler at this point.

### TLB Modified Exception

**Cause**

This exception occurs when the virtual address target of a store operation matches a TLB entry is marked valid, but not marked dirty. This exception is not maskable.

**Handling**

The R3051/52 branches to the General Exception vector for this exception and sets the MOD exception code in the CAUSE register's ExcCode field.

When this exception occurs, the BadVAddr, Context, and EntryHi registers contain the virtual address that failed address translation. EntryHi also contains the PID from which the translation fault occurred.

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the Branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

**Servicing**

A kernel should use the failing virtual address or virtual page number to identify the corresponding access control information. The identified page might or might not permit write accesses. (Typically, software maintains the "real" write protection in other memory areas.) If the page does not permit write access, a "Write Protection Violation" occurs.

If the page does permit write accesses, the kernel should mark the page frame as dirty in its own data structures. Use the TLBProbe (tlbp) instruction to put the index of the TLB entry that must be altered in the Index register. Then load the EntryLo register with a word that contains the physical page frame and access control bits (with the data bit D set). Finally, use the TLBWrite Indexed (tlbwi) instruction to write EntryHi and EntryLo into the TLB.

### UTLB Miss Exception

#### Cause

This exception occurs from User or Kernel mode references to user memory space when no TLB entry matches both the VPN and the PID. Invalid entries cause a TLB Miss rather than a UTLB Miss. This exception is not maskable.

#### Handling

The R3051/52 uses the special UTLB Miss interrupt vector for this exception. When the exception occurs, the R3051/52 sets the TLBL or TLBS code in the Cause register ExcCode field to indicate whether the miss was due to an instruction fetch or a load operation (TLBL) or a store operation (TLBS).

The EPC register points at the instruction that caused the exception, unless the instruction is in a branch delay slot: in that case, the EPC register points at the Branch instruction that preceded the exception-causing instruction and sets the BD bit of the Cause register.

The R3051/52 saves the KUp, IEp, KUc, and IEc bits of the status register in the KUo, IEo, KUp, and IEp bits, respectively, and clears the KUc and IEc bits.

The virtual address that failed translation is held in the BadVAddr, Context, and EntryHi registers. The EntryHi register also contains the PID (Process Identifier) from which the translation fault occurred. The Random register contains a valid pseudo-random location in which to put a replacement TLB entry.

#### Servicing

The contents of the Context register can be used as the virtual address of the memory word that contains the physical page frame and the access control bits(a Page Table Entry, or PTE) for the failing reference. An operating system should put the memory word in EntryLo and write the contents of EntryHi and EntryLo into the TLB by using a TLB Write Random (tlbwr) assembly instruction.

The PTE virtual address might be on a page that is not resident in the TLB. Therefore, before an operating system can reference the PTE virtual address, it should save the EPC register's contents in a general register reserved for kernel use or in a physical memory location. If the reference is not mapped in the TLB, a TLB Miss exception would occur within the UTLB Miss handler.

A short routine (nine instructions, one load) to service a UTLB miss is shown.

```
mfc0    k0, C0_CTX      # get address of PTE
mfc0    k1, C0_EPC      # get address of failed reference
lw      k0, 0(k0)       # fetch PTE
nop                     # load delay slot
mtc0    k0, C0_TLBLO    # write EntryLo (EntryHi set by chip hardware)
nop                     # effective delay slot due to CP0 move
c0      C0_WriteR       # tlbwr; write random TLB entry
j       k1              # return to EPC
rfe                     # restore context from exception
```

4000 drw 44

**Figure 5.20. User TLB Refill Code**

# INTERFACE OVERVIEW



The IDT R3051 family utilizes a simple, flexible bus interface to its external memory and I/O resources. The interface uses a single, multiplexed 32-bit address and data bus and a simple set of control signals to manage read and write operations. Complementing the basic read and write interface is a DMA Arbiter interface which allows an external agent to gain control of the memory interface to transfer data.

The R3051 family supports the following types of operations on its interface:

- **Write Operations:** The R3051 family utilizes an on-chip write buffer to isolate the execution core from the speed of external memory during write operations. The write interface of the R3051 family is thus designed to allow a variety of write strategies, from fast 2-cycle write operations through multiple wait-state writes.

  The R3051 family supports the use of fast page mode writes by providing an output indicator, $\overline{\text{WrNear}}$, to indicate that the current write may be retired using a page mode access. This facilitates the rapid "flushing" of the on-chip write buffer, since the majority of processor writes will occur within a localized area of memory.

  The write interface is described in detail in Chapter 8.

- **Read Operations:** The processor executes read operations as the result of either a cache miss or an uncacheable reference. As with the write interface, the read interface has been designed to accommodate a wide variety of memory system strategies. There are two types of reads performed by the processor:

  Burst (or quad word) reads occur when the processor requests a contiguous block of four words from memory. Bursts occur in response to instruction cache misses, and may occur in response to a data cache miss. The processor incorporates an on-chip 4-deep read buffer which may be used to "queue up" the read response before passing it through to the high-bandwidth cache and execution core. Read buffering is appropriate in systems which require wait states between adjacent words of a block read. On the other hand, systems which use high-bandwidth memory techniques (such as page mode, static column, nibble mode, or memory interleaving) can effectively bypass the read buffer by providing words of the block at the processor clock rate. Note that the choice of burst vs. read buffering is independent of the initial latency of the memory; that is, burst mode can be used even if multiple wait states are required to access the first word of the block.

  Single word reads are used for uncacheable references (such as I/O or boot code) and may be used in response to a data cache miss. The processor is capable of retiring a single word read in as few as two clock cycles.

  The read interface of the R3051 family is described in detail in Chapter 7.

- **DMA Operations:** The R3051 family includes a DMA arbiter which allows an external agent to gain full control of the processor read and write interface. DMA is useful in systems which need to move significant amounts of data within memory (e.g. BitBlT operations) or move data between memory and I/O channels.

  The R3051 family utilizes a very simple handshake to transfer control of its interface bus. This handshake is described in detail in chapter 9.

## MULTIPLE OPERATIONS

It is possible for the R3051 family to have multiple interface activities pending. Specifically, there may be data in the write buffer, a read request (e.g. due to a cache miss), a DMA mastership request, and an ongoing transaction all occurring simultaneously.

In establishing the order in which the requests are processed, the R3051 family is sensitive to possible conflicts and data coherency issues. For example, if the on-chip write buffer contains data which has not yet been written to memory, and the processor issues a read request to the target address of one of the write buffer entries, then the processor strategy must insure that the read request is satisfied by the new, current value of the data.

Thus, in the R3051 family, multiple operations are serviced in the following order:

1: Ongoing transactions are completed without interruption.
2: DMA requests are serviced.
3: Instruction cache misses are processed.
4: Pending writes are processed.
5: Data cache misses or uncacheable reads are processed.

This service order has been designed to achieve maximum performance, minimize complexity, and solve the data coherency problem possible in write buffer systems.

Note that this order assumes that the write buffer does not contain instructions which the processor may wish to execute. The processor does not write directly into the instruction cache: store instructions generate data writes which may change only the data cache. The only way in which an instruction reference may reside in the write buffer is in the case of self modifying code, generated with the caches swapped. If this technique is required in a system, then an uncacheable reference should occur prior to execution of the generated code, to insure that the write buffer is flushed. However, self-modifying code is typically not recommended.

## EXECUTION ENGINE FUNDAMENTALS

This section describes the fundamentals of the processor read interface and its interaction with the execution core. These fundamentals will help to explain the relationship between design tradeoffs in the system interface and the performance achieved in R3051/52 based systems.

**Execution Core Cycles**

The R3051/52 execution core utilizes many of the same operation fundamentals as does the R3000A processor. Thus, much of the terminology used to describe the activity of the R3051/52 is derived from the terminology used to describe the R3000A. In many instances, the activity of the execution core is independent of that of the bus interface unit.

### Cycles

A cycle is the basic timing reference of the R3051/52. Cycles in which forward progress is made (the processor pipeline advances) are called Run cycles. Cycles in which no forward progress occurs are called stall cycles. Stall cycles are used for resolving exigencies such as cache misses, write stalls, and other types of events. All cycles can be classified as either run or stall cycles.

### Run Cycles

Run cycles are characterized by the transfer of an instruction into the processor core, and the optional transfer of data into or out of the execution core. Thus, each run cycle can be thought of as having an instruction and data, or ID, pair.

There are actually two types of run cycles: cache run cycles, and refill run cycles. Cache run cycles (typically referred to as just run cycles) occur while the execution core is executing out of its on chip cache; these are the principal execution mechanism.

Refill run cycles, referred to as streaming cycles, occur when the execution core is executing instructions as they are brought into the on-chip cache. For the R3051/52, streaming cycles are defined as cycles in which data is brought out of the on-chip read buffer into the execution core (rather than defining them as cycles in which data is brought from the memory interface to the read buffer).

### Stall Cycles

There are three types of stall cycles:

**Wait Stall Cycles.** These are commonly referred to simply as stall cycles. During wait stall cycles, the execution core maintains a state consistent with resolving a stall causing event. No cache activity will occur during wait stalls.

**Refill Stall Cycles.** These occur only during memory reads, and are used to transfer data from the on-chip read buffer into the caches.

**Fixup Stall Cycles.** Fixup cycles occur during the final cycle of a stall; that is, one cycle before entering a run cycle or entering another stall. During the final fixup cycle (the one which occurs before finally re-entering run operation), the ID pair which should have been processed during the last run cycle is handled by the processor. The fixup cycle is used to restart the processor and co-processor pipelines, and in general to fixup conditions which caused the stall.

The basic causes of stalls include:

**Read Busy Stalls:** If the processor is utilizing its read interface, either to process a cache miss or an uncacheable reference, then it will be stalled until the read data is brought back to the execution core.

**Write Busy Stalls:** If the processor attempts to perform a store operation while the on-chip write buffer is already full, then the processor will stall until a write transaction is begun on the interface to free up room in the write buffer for the new address and data.

**Multiply/Divide Busy Stalls:** If software attempts to read the result registers of the integer multiply/divide unit (the HI and LO registers) while a multiply or divide operation is underway, the processor execution core will stall until the results are available.

**Micro-TLB Fill Stalls**: These stalls can occur when an instruction translation misses in the instruction TLB cache (the micro-TLB, which is a two-entry cache of the main TLB used to translate instruction references). When such an event occurs, the execution core will stall for one cycle, in order to refill the micro-TLB from the main TLB. Since this is a single-cycle stall, it is of necessity a fixup cycle.

**Multiple Stalls**

Multiple stalls are possible whenever more than one stall initiating event occurs within a single run cycle. An example of such activity is when a single cycle results in both an instruction cache miss and a data cache miss.

The most important characteristic of any multiple stall cycle is the validity of the ID pair processed in the final fixup cycle. The R3051/52 execution core keeps track of nested stalls to insure that orderly operation is resumed once all of the stall causing events are processed.

For the general case of multiple stalls, the service order is:

1: Micro-TLB Miss and Partial Word Store
2: Data Cache Miss or Write Busy Stall
3: Instruction Cache Miss
4: Multiply/Divide Unit Busy

## PIN DESCRIPTION

This section describes the signals used in the above interfaces. More detail on the actual use of these pins is found in other chapters. Note that many of the signals have multiple definitions which are de-multiplexed either by the ALE signal or the $\overline{Rd}$ and $\overline{Wr}$ control signals. Note that signals indicated with an overbar are active low.

### System Bus Interface Signals

These signals are used by the bus interface to perform read and write operations.

#### Address and Data Path

**A/D(31:0)** **I/O**

**Address/Data:** A 32-bit, time multiplexed bus which indicates the desired address for a bus transaction in one cycle, and which is used to transmit data between this device and external memory resources on other cycles.

Bus transactions on this bus are logically separated into two phases: during the first phase, information about the transfer is presented to the memory system to be captured using the ALE output. This information consists of:

**Address(31:4):** The high-order address for the transfer is presented.

**$\overline{BE(3:0)}$:** These strobes indicating which bytes of the 32-bit bus will be involved in the transfer. $\overline{BE(3)}$ indicates that AD(31:24) is used; $\overline{BE(2)}$ indicates that AD(23:16) is used; $\overline{BE(1)}$ indicates that AD(15:8) is used; and $\overline{BE(0)}$ indicates that AD(7:0) is used.

During write cycles, the bus contains the data to be stored and is driven from the internal write buffer. On read cycles, the bus receives the data from the external resource, in either a single word transaction or in a burst of four words, and places it into the on-chip read buffer.

**Addr(3:2)** **O**

**Low Address (3:2)** A 2-bit bus which indicates which word is currently expected by the processor. Specifically, this two bit bus presents either the address bits for the single word to be transferred (writes or single word reads) or functions as a two bit counter starting at '00' for burst read operations.

#### Read and Write Control Signals

**ALE** **O**

**Address Latch Enable:** Used to indicate that the A/D bus contains valid address information for the bus transaction. This signal is used by external logic (transparent latches) to capture the address for the transfer.

**$\overline{DataEn}$** **O**

**Data Input Enable:** This signal indicates that the AD bus is no longer being driven by the processor during read cycles, and thus the external memory

system may enable the drivers of the memory system onto this bus without having a bus conflict occur. During write cycles, or when no bus transaction is occurring, then this signal is negated.

**$\overline{\text{Burst}}$/ $\overline{\text{WrNear}}$** **O**

**Burst Transfer:** On read transactions, this signal indicates that the current bus read is requesting a block of four contiguous words from memory (a burst read). This signal is asserted only in read cycles due to cache misses; it is asserted for all I-Cache miss read cycles, and for D-Cache miss read cycles if selected at device reset time.

**Write Near:** On write transactions, this output tells the external memory system that the bus interface unit is performing back-to-back write transactions to an address within the same 256 entry memory "page" as the prior write transaction. This signal is useful in memory systems which employ page mode or static column DRAMs.

**$\overline{\text{Rd}}$** **O**

**Read:** An output which indicates that the current bus transaction is a read.

**$\overline{\text{Wr}}$** **O**

**Write:** An output which indicates that the current bus transaction is a write.

**$\overline{\text{Ack}}$** **I**

**Acknowledge:** An input which indicates to the device that the memory system has sufficiently processed the bus transaction, and that the processor may either advance to the next write buffer entry or release the execution core to process the read data.

**$\overline{\text{RdCEn}}$** **I**

**Read Buffer Clock Enable:** An input which indicates to the device that the memory system has placed valid data on the AD bus, and that the processor may move the data into the on-chip Read Buffer.

**$\overline{\text{BusError}}$** **I**

**Bus Error:** Input to the bus interface unit to terminate a bus transaction due to an external bus error. This signal is only sampled during read and write operations. If the bus transaction is a read operation, then the CPU will also take a bus error exception.

### Status Information

**Diag(1)** **O**

**Diagnostic Pin 1.** This output indicates whether the current bus read transaction is due to an on-chip cache miss, and also presents part of the miss address. The value output on this pin is time multiplexed:

**Cached:** During the phase in which the A/D bus presents address information, this pin is an active high output which indicates whether the current read is a result of a cache miss. The value of this pin at this time in other than read cycles is undefined.

**Miss Address (3):** During the remainder of the read operation, this output presents address bit (3) of the address the processor was attempting to reference when the cache miss occurred. Regardless of whether a cache miss is being processed, this pin reports the transfer address during this time.

## Diag(0) O

**Diagnostic Pin 0.** This output distinguishes cache misses due to instruction references from those due to data references, and presents the remaining bit of the miss address. The value output on this pin is also time multiplexed:

**I/$\overline{D}$:** If the "Cached" Pin indicates a cache miss, then a high on this pin at this time indicates an instruction reference, and a low indicates a data reference. If the read is not due to a cache miss but rather an uncached reference ("Cached" is negated), then this pin is undefined during this phase.

**Miss Address (2):** During the remainder of the read operation, this output presents address bit (2) of the address the processor was attempting to reference when the cache miss occurred. Regardless of whether a cache miss is being processed, this pin reports the transfer address during this time.

### DMA Arbiter Interface

These signals are involved when the processor exchanges bus mastership with an external agent.

## $\overline{\text{BusReq}}$ I

**DMA Arbiter Bus Request:** An input to the device which requests that the processor tri-state its bus interface signals so that they may be driven by an external master. The negation of this input releases the bus back to the R3051/52.

## $\overline{\text{BusGnt}}$ O

**DMA Arbiter Bus Grant.** An output from the R3051/52 used to acknowledge that a $\overline{\text{BusReq}}$ has been detected, and that the bus is relinquished to the external master.

### Interrupt Interface

Chapter 5 discusses the exception model of the R3051 family.

**BrCond(1:0)**
**SBrCond(3:2)** **I**

**Branch Condition Port:** These external signals are available as an input port to the processor, which can use the Branch on Co-Processor Condition instructions to test their polarity. The SBrCond bus is synchronized by the R3051/52, and thus may be driven by an asynchronous source; the BrCond signals are directly tied to the execution core, and thus must be generated synchronously.

**$\overline{\text{SInt(2:0)}}$**
**$\overline{\text{Int(5:3)}}$** **I**

**Processor Interrupt:** During operation, these signals are the same as the $\overline{\text{Int(5:0)}}$ signals of the R3000. During processor reset, these signals perform mode initialization of the processor. The Synchronized interrupt inputs are internally synchronized by the R3051/52, and thus may be generated by an asynchronous interrupt agent; the direct interrupts must be externally synchronized by the interrupt agent.

### Reset and Clocking

**Clk2xIn** **I**

**Master clock Input:** This is a double frequency input used to control the timing of the processor

**$\overline{\text{SysClk}}$** **O**

**System Reference Clock:** An output from the processor which reflects the clock used to perform bus interface functions. This clock is used to control state transitions in the read buffer, write buffer, memory controller, and bus interface unit. It should be used as a timing reference by the external memory system.

**$\overline{\text{Reset}}$** **I**

**Master Processor Reset:** This signal initializes the processor. Optional features of the processor are established during the last cycle of reset using the interrupt inputs.

### Miscellaneous

**Rsvd(4:0)** **I/O**

**Reserved:** These five signal pins are reserved for testing and for future revisions of this device. Users must not connect these pins.

## INTRODUCTION

The R3051 family read protocol has been designed to interface to a wide variety of memory and I/O devices. Particular care has been taken in the definition of the control signals available to the system designer. These signals allow the system designer to implement a memory interface appropriate to the cost and performance goals of the end application.

This chapter includes both an overview of the read interface as well as provides detailed timing diagrams of the read interface.

## TYPES OF READ TRANSACTIONS

The majority of the execution engine read requests are never seen at the memory interface, but rather are satisfied by the internal cache resources of the processor. Only in the cases of uncacheable references or cache misses do read transactions occur on the bus.

In general, there are only two types of read transactions: quad word reads and single word reads. Note that partial word reads of less than 32-bits can be thought of as a simple subset of the single word read, with only some of the byte enable strobes asserted.

Quad word reads occur only in response to cache misses. All instruction cache misses are processed as quad word reads; data cache misses may be processed as quad word reads or single word reads, depending on the mode selection made during reset initialization of the device.

In processing reads, there are two parameters of interest. The first parameter is the initial latency to the first word of the read. This latency is influenced by the overall system architecture as well as the type of memory system being addressed: time required to perform address decoding, and perform bus arbitration, memory pre-charge requirements, and memory control requirements, as well as memory access time. The initial latency is the only parameter of interest in single word reads.

The second parameter of interest (only in quad word refills) is the repeat rate of data; that is, time required for subsequent words to be processed back to the processor. Factors which influence the repeat rate include the memory system architecture, the types and speeds of devices used, and the sophistication of the memory controller: memory interleaving, the use of page or static column mode, and faster devices all serve to increase the repeat rate (minimize the amount of time between adjacent words).

The R3051 family has been designed to accommodate a wide variety of memory system designs, including no wait state operations (first word available in two cycles) and true burst operation (adjacent words every clock cycle), through simpler, slower systems incorporating many bus wait states to the first word and multiple clock cycles between adjacent words (this is accomplished by use of the on-chip read buffer).

## READ INTERFACE SIGNALS

The read interface uses the signals listed below. Signal names indicates with an overbar are active low.

**$\overline{\text{Rd}}$ O**

This output indicates that a read operation is occurring.

**A/D (31:0) I/O**

During read operations, this bus is used to transmit the read target address to the memory system, and is used by the memory system to return the required data back to the processor. Its function is de-multiplexed using other control signals.

During the addressing portion of the read transaction, this bus contains the following:

| | |
|---|---|
| Address(31:4) | The upper 28 bits of the read address are presented on A/D (31:4). |
| $\overline{\text{BE}}$(3:0) | The byte strobes for the read transaction are presented on A/D(3:0). |

**ALE O**

This output signal is typically connected directly to the latch enable of transparent latches. Latches are typically used to de-multiplex the address and Byte Enable information from the A/D bus.

**Addr(3:2) O**

The remaining bits of the transfer address are presented directly on these outputs. In the case of quad word reads, these pins function as a two bit counter starting at '00', and are used to perform the quad word transfer. In the case of single (or partial word) reads, these pins contain Address (3:2) of the transfer address.

**$\overline{\text{DataEn}}$ O**

This output indicates that the A/D bus is no longer being driven by the processor, and thus the output drivers of the memory system may be enabled.

Special logic on the R3051/52 guarantees the following:

The A/D bus is driven to guarantee that a minimum of 0ns hold time from the negation of ALE.
The R3051/52 A/D bus output drivers will be disabled a minimum of 0ns before the assertion of $\overline{\text{DataEn}}$.

Thus, the system designer is assured that ALE can be used to directly control the latch enable of a transparent latch. Similarly, $\overline{\text{DataEn}}$ can be used to directly control the output enable of memory system drivers.

**$\overline{\text{Burst}}$ O**

This output distinguishes between quad word reads and other reads.

**$\overline{\text{RdCEn}}$ I**

Read Buffer Clock Enable is used by the external memory system to cause the processor to capture the contents of the A/D bus. In the case of quad word reads, this causes the contents of the A/D bus to be strobed into the on-chip read buffer; in the case of single word reads, this causes the processor to capture the read data and may also terminate the read operation.

**$\overline{\text{Ack}}$ I**

Acknowledge is used by the memory system to indicate that it has sufficiently processed the read transaction, and that the internal execution core may begin processing the read data. Thus, $\overline{\text{Ack}}$ can be used by the external memory system to cause the execution core to begin processing the read data simultaneously with the memory system bringing in additional words of the burst refill. The timing of the assertion of $\overline{\text{Ack}}$ by the memory system must be constructed to insure that words not yet retrieved from the memory will be brought in before they are required by the execution core.

When the memory system is able to supply words at the rate of one per clock cycle (after the initial latency), Ack can be asserted simultaneous with the initial $\overline{\text{RdCEn}}$ to achieve the highest levels of performance.

Other systems, which utilize simpler memory system strategies, may ignore the use of $\overline{\text{Ack}}$ in read transactions. The processor will recognize the implicit termination of a read operation by the assertion of the appropriate number (one or four) of $\overline{\text{RdCEn}}$. While this approach is simpler to design, a small loss of performance will result.

**$\overline{\text{BusError}}$ I**

This input can be used to terminate a read operation. It will also cause the processor to take a BusError exception. Read transactions terminated by $\overline{\text{BusError}}$ do not require the assertion of $\overline{\text{Ack}}$ or $\overline{\text{RdCEn}}$.

**Diag(1) O**

During the address phase of the read transaction, this output indicates whether the read is a result of a cache miss or an uncacheable reference.

During the remainder of the transfer, this output indicates Address(3) of the actual address reference which missed in the cache.

This pin is useful in the initial debug of R3051 family based systems.

**Diag(0) O**

During the address phase of the read transaction, this output indicates whether the read is a result of an instruction or data reference.

During the remainder of the transfer, this output indicates Address(2) of the actual address reference which missed in the cache.

This pin is useful in the initial debug of R3051 family based systems.

## READ INTERFACE TIMING OVERVIEW

The read interface is designed to allow a variety of memory strategies. An overview of how data is transmitted from memory and I/O devices to the processor is discussed below. Note that multiplexing the address and data bus does not slow down read transactions: the address is on the A/D bus for only one-half clock cycle, so the data drivers can be enabled quickly; memory and I/O devices initiate their transfers based on addressing and chip enable, not on the availability of the bus. Thus, memory does not need to "wait" for the bus, and no performance penalty occurs.

### Initiation of Read Request

A read transaction occurs when the processor internally performs a run cycle which is not satisfied by the internal caches. Immediately after the run cycle, the processor enters a stall cycle and asserts the internal control signal $\overline{\text{MemRd}}$. This signals to the internal bus interface unit arbiter that a read transaction is pending.

Assuming that the read transaction can be immediately processed (that is, there are no ongoing bus operations, and no higher priority operations pending), the processor will initiate a bus read transaction on rising edge of $\overline{\text{SysClk}}$ which occurs during phase 2 of the processor stall cycle. Higher priority operations would have the effect of delaying the start of the read by inserting additional processor stall cycles.

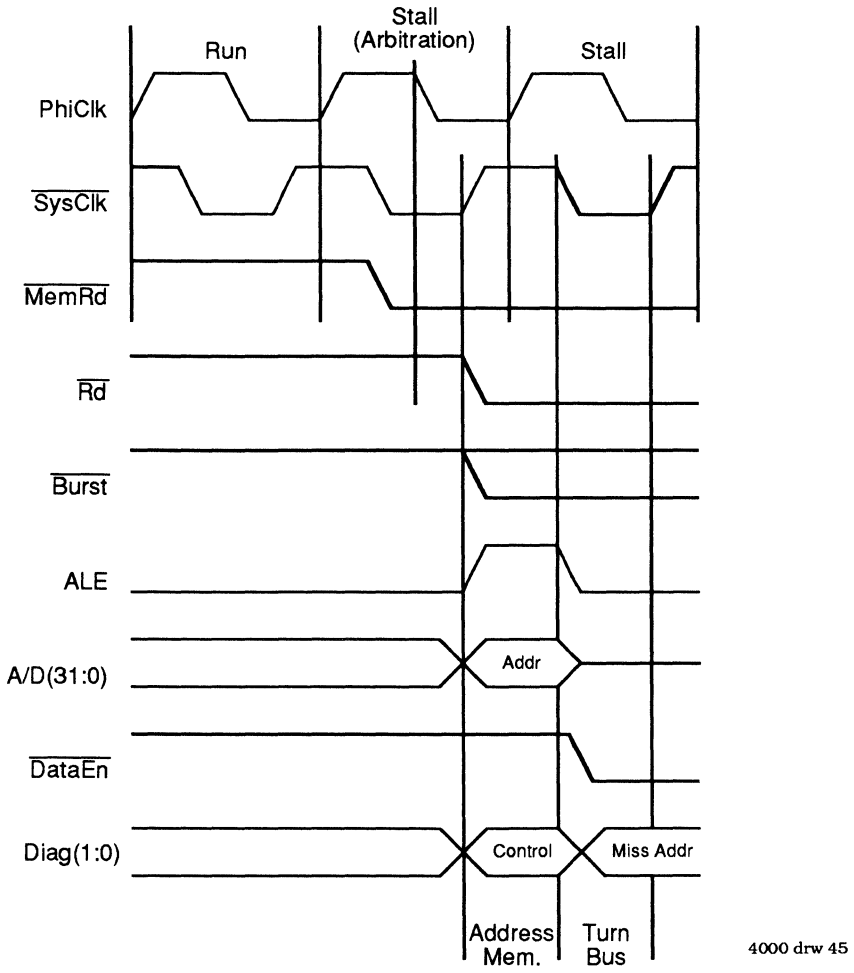



**Figure 7.1. CPU Latency to Start of Read**

Figure 7.1 illustrates the initiation of a read transaction, based on the internal assertion of the MemRd control signal. This figure is useful in determining the overall latency of cache misses on processor operation.

**Memory Addressing**

A read transaction begins when the processor asserts its Rd control output, and also drives the address and other control information onto the A/D and memory interface bus. Figure 7.2 illustrates the start of a processor read transaction, including the addressing of memory and the bus turn around.

The addressing occurs in a half-cycle of the SysClk output. At the rising edge of SysClk, the processor will drive the read target address onto the A/D bus. At this time, ALE will also be asserted, to allow an external transparent latch to capture the address. Depending on the system design, address decoding could occur in parallel with address de-multiplexing (that is, the decoder could start on the assertion of ALE, and the output of the decoder captured by ALE), or could occur on the output side of the transparent latches. During this phase, DataEn will be held high indicating that memory drivers should not be enabled onto the A/D bus.

Concurrent with driving addresses on the A/D bus, the processor will indicate whether the read transaction is a quad word read or single word read, by driving Burst to the appropriate polarity (low for a quad word read). If a quad word read is indicated, the Addr(3:2) lines will drive '00' (the start of the block); if a single word (or subword) is indicated, the Addr(3:2) lines will indicate the word address for the transfer. The functioning of the counter during quad words is described later.

**Bus Turn Around**

Once the A/D bus has presented the address for the transfer, it is "turned around" by the processor to accept the incoming data. This occurs in the second phase of the first clock cycle of the read transaction, as illustrated in Figure 7.2.

The processor turns the bus around by carefully performing the following sequence of events:

- It negates ALE, causing the transparent address latches to capture the contents of the A/D bus.
- It disables its output drivers on the A/D bus, allowing it to be driven by an external agent. The processor design guarantees that the ALE is negated prior to tri-stating the A/D bus.
- The processor then asserts DataEn, to indicate that the bus may be now driven by the external memory resource. The processor design insures that the A/D bus is released prior to DataEn being asserted. DataEn may be directly connected to the output enable of external memory, and no bus conflicts will occur.

Thus, the processor A/D bus is ready to be driven by the end of the second phase of the read transaction. At this time, it begins to look for the end of the read cycle.

**Bringing Data into the Processor**

Regardless of whether the transfer is a quad word read or a single word transfer, the basic mechanism for transferring data presented on the A/D bus into the processor is the same.

Although there are two control signals involved in terminating read operations, only the RdCEn signal is used to cause data to be captured from the bus.

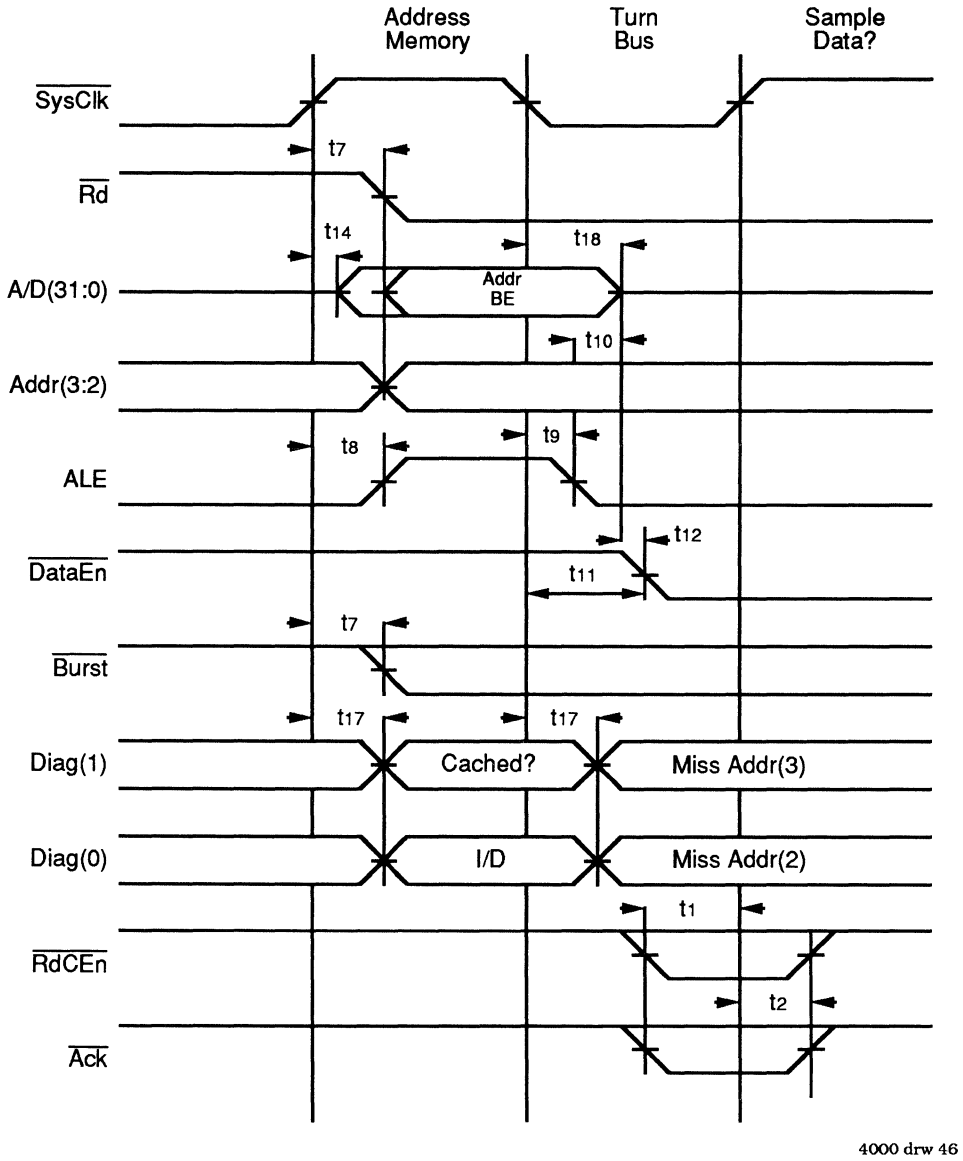
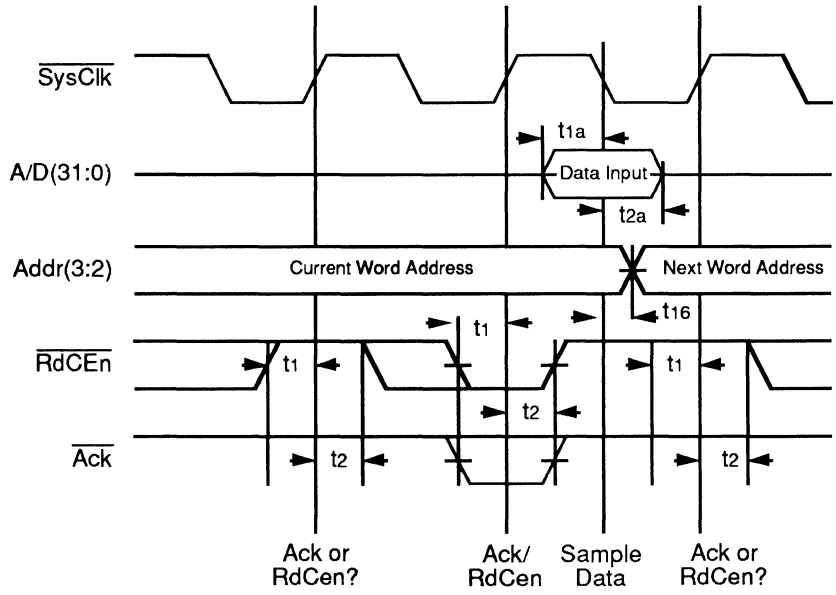



**Figure 7.2. Start of Bus Read Operation**

The memory system asserts $\overline{\text{RdCEn}}$ to indicate to the processor that it has (or will have) data on the A/D bus to be sampled. The earliest that $\overline{\text{RdCEn}}$ can be detected by the processor is the rising edge of $\overline{\text{SysClk}}$ after it has turned the bus around (start of phase 1 of the second clock cycle of the read).

If $\overline{\text{RdCEn}}$ is detected as asserted (with adequate setup and hold time to the rising edge of $\overline{\text{SysClk}}$), the processor will capture (with proper setup and hold time) the contents of the A/D bus on the immediately subsequent falling edge of $\overline{\text{SysClk}}$. This captures the data in the internal read buffer for later processing by the execution core/cache subsystem.

Figure 7.3 illustrates the sampling of data by an R3051/52.




4000 drw 47

**Figure 7.3. Data Sampling on R3051/52**

### Terminating the Read

There are actually three methods for the external memory system to terminate an ongoing read operation:

- It can supply an $\overline{\text{Ack}}$ (acknowledge) to the processor, to indicate that it has sufficiently processed the read request and has or will supply the requested data in a timely fashion. Note that $\overline{\text{Ack}}$ may be signalled to the processor "early", to enable it to begin processing the read data even while additional data is brought from the A/D bus. This is applicable only in burst read operations.

- It can supply a $\overline{\text{BusError}}$ to the processor, to indicate that the requested data transfer has "failed" on the bus, and force the processor to take a bus error exception. Although the system interface behavior of the processor when $\overline{\text{BusError}}$ is presented is identical to the behavior when $\overline{\text{Ack}}$ is presented, no data will actually be written into the on-chip cache. Rather, the cache line will either remain unchanged, or will be invalidated by the processor, depending on how much of the read has already been processed.

- The external memory system can supply the requested data, using $\overline{\text{RdCEn}}$ to enable the processor to capture data from the bus. The processor will "count" the number of times $\overline{\text{RdCEn}}$ is sampled as asserted; once the processor counts that the memory system has returned the desired amount of data (one word or four words), it will implicitly "acknowledge" the read at the same time that it samples the last required $\overline{\text{RdCEn}}$. This approach leads to a simpler memory design at the cost of lower performance.

Throughout this chapter, method one will be illustrated. The other cases can easily be extrapolated from these diagrams (for example, the system designer can directly substitute $\overline{\text{BusError}}$ for $\overline{\text{Ack}}$ in these drawings; similarly, he can assume that $\overline{\text{Ack}}$ is asserted simultaneous with the last $\overline{\text{RdCEn}}$ of a read transfer).

There are actually two phases of terminating the read: there is the phase where the memory system indicates to the processor that it has sufficiently processed the read request, and the internal read buffer can be released to begin refilling the internal caches; and there is the phase in which the read

control signals are negated by the processor bus interface unit. The difference between these phases is due to block refill: it is possible for the memory system to "release" the execution core even though additional words of the block are still required; in that case, the processor will continue to assert the external read control signals until all four words are brought into the read buffer, while simultaneously refilling/executing based on the data already brought on board.

Figure 7.4 shows the timing of the control signals when the read cycle is being terminated.

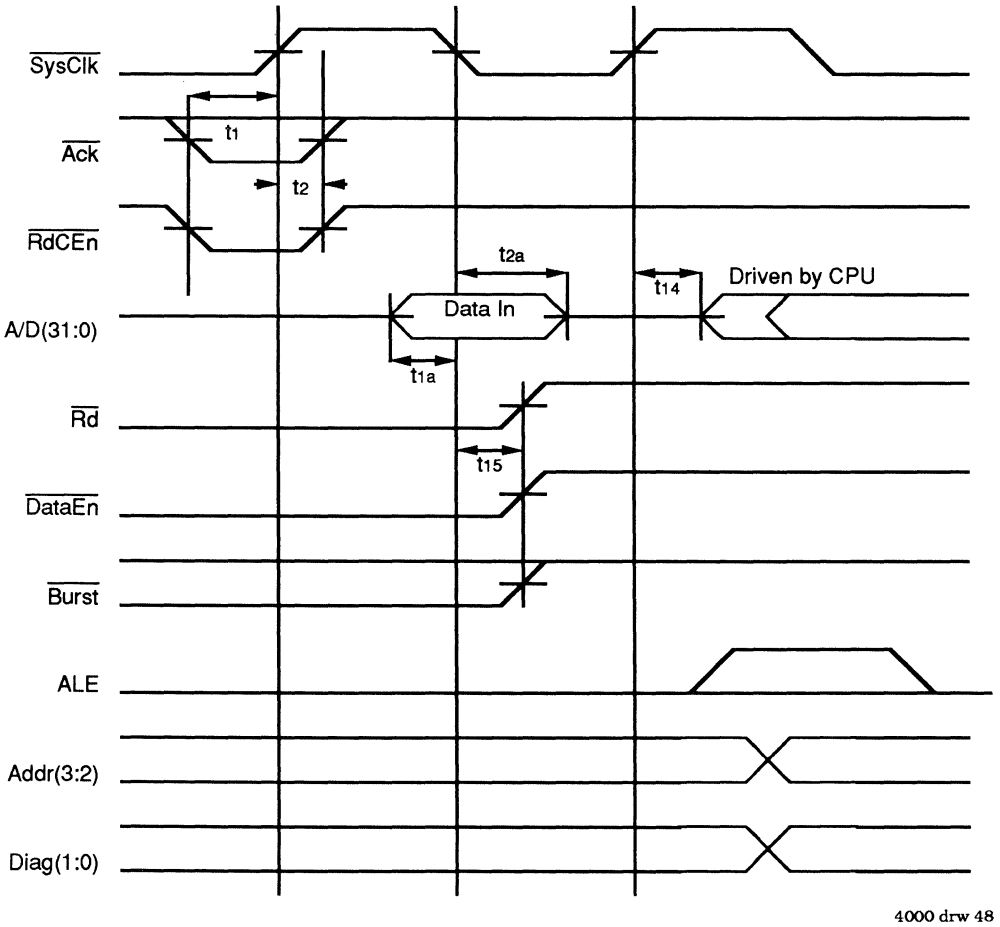
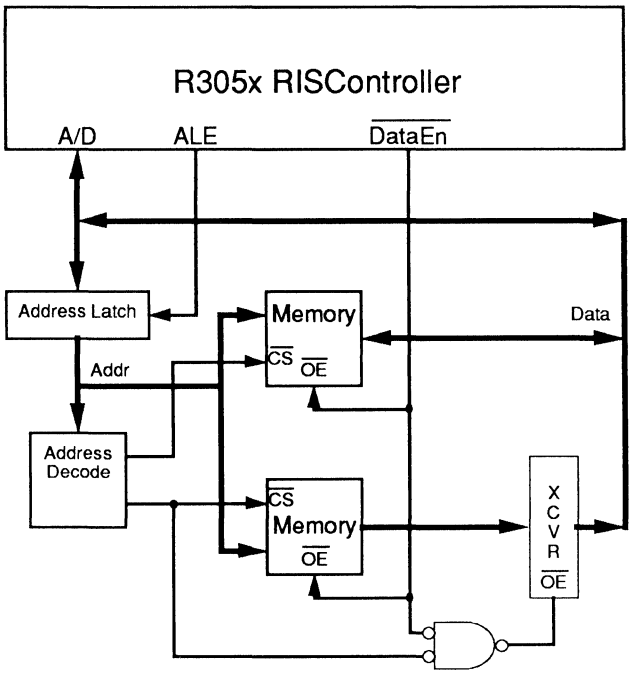



**Figure 7.4. Read Cycle Termination**

### Latency Between Processor Operations

In general, the processor may begin a new bus activity in the phase immediately after the termination of the read cycle. Although this operation may logically be either a read, write, or bus grant, there are no cases where a read operation can be signalled by the internal execution core at this time.

Since a new operation may begin one-half clock cycle after the data is sampled from the bus, it is important that the external memory system cease to drive the bus prior to this clock edge. In order to simplify design, the processor provides the DataEn output, which can be used to control either the Output Enable of the memory device (presuming its tri-state time is fast enough), or to control the Output Enable of a buffer or transceiver between the memory device data bus and the processor A/D bus. This is illustrated in Figure 7.5.



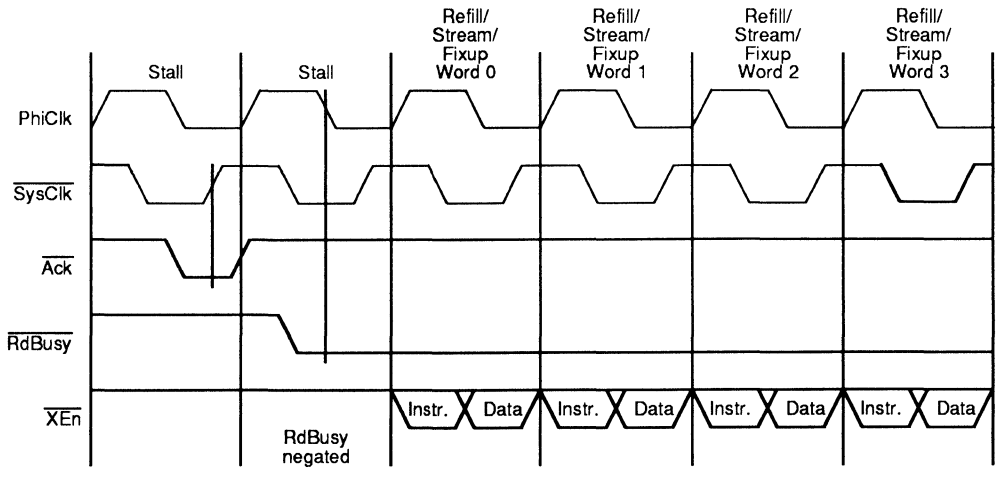

4000 drw 49

**Figure 7.5. Use of DataEn as Output Enable Control**

### Processor Internal Activity

In general, the processor will execute stall cycles until Ack is detected. It will then begin the process of refilling the internal caches from the read buffer.

The system designer should consider the difference between the time when the memory interface has completed the read, and when the processor core has completed the read. The bus interface may have successfully returned all of the required data, but the processor core may still require additional clock cycles to bring the data out of the read buffer and into the caches. Figure 7.6 illustrates the relationship between Ack and the internal activity for a block read.




4000 drw 50

**Figure 7.6. Internal Processor States on Burst Read**

This figure illustrates that the processor may perform either a stream, fixup, or refill cycle in cycles in which data is brought from the read buffer. The difference between these cycles is defined as:

- **Refill.** A refill cycle is a clock cycle in which data is brought out of the read buffer and placed into the internal processor cache. The processor does not execute on this data.

- **Fixup.** A fixup cycle is a cycle in which the processor transitions into executing the incoming data. It can be thought of as a "retry" of the cache cycle which resulted in a miss.

- **Stream.** A stream cycle is a cycle in which the processor simultaneously refills the internal cache and executes the instruction brought out of the read buffer.

When reading the block from the read buffer, the processor will use the following rules:

For uncacheable references, the processor will bring the single word out of the read buffer using a fixup cycle.

For data cache refill, it will execute either one or four refill cycles, followed by a fixup cycle.

For instruction cache refill, it will execute refill cycles starting at word zero until it encounters the miss address, and then transition to a fixup cycle. It will then execute stream cycles until either the entire block is processed, or an event stops execution. If something causes execution to stop, the processor will process the remainder of the block using simple refill cycles. For example, Figure 7.7 illustrates the refill/fixup/stream sequence appropriate for a miss which occurs on the second word of the block (word address 1).

Although this operation is transparent to the external memory system, it is important to understand this operation to gauge the impact of design trade-offs on performance.

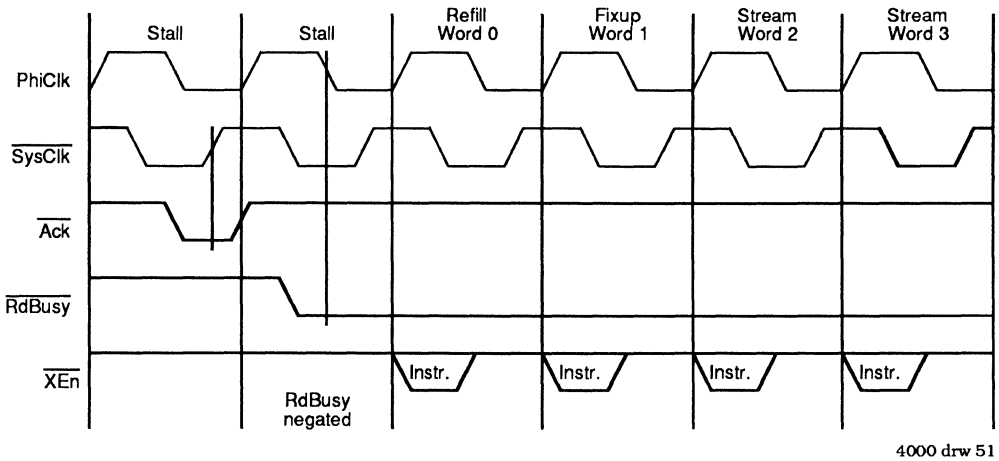


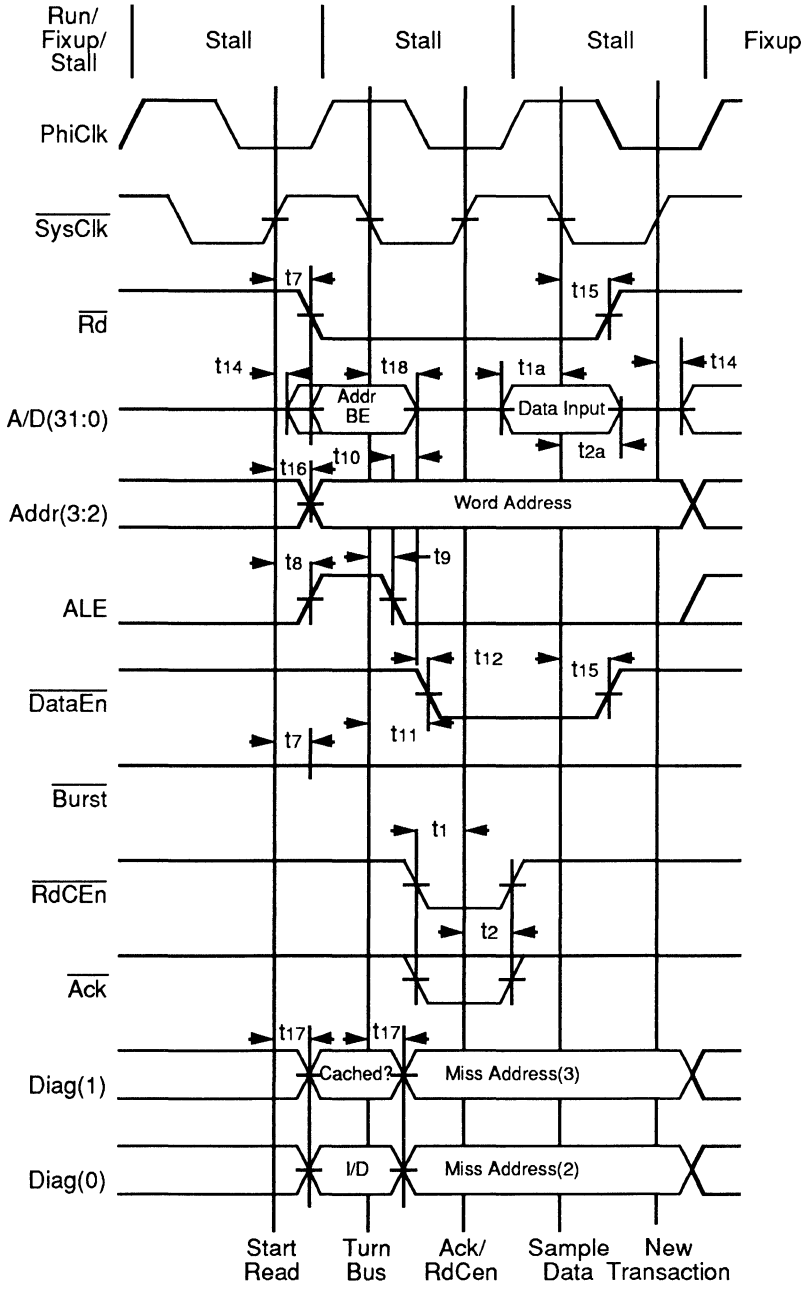

**Figure 7.7. Instruction Streaming Example**

## READ TIMING DIAGRAMS

This section illustrates a number of timing diagrams applicable to R3051 family read transactions. These diagrams reference AC parameters whose values are contained in the R3051/52 data sheet.

### Single Word Reads

Figure 7.8 illustrates the case of a single word read which did not require wait states. Thus, Ack was detected at the rising edge of SysClk which occurred exactly one clock cycle after the rising edge of SysClk which asserted Rd. Data was sampled one phase later, and Rd and DataEn disabled from that falling edge of SysClk. Thus, the execution core required three stall cycles and a fixup to process the internal data.




4000 drw 52

**Figure 7.8. Single Word Read Without Bus Wait Cycles**

Figure 7.9 also illustrates the case of a single word read. However, in this figure, two bus wait cycles were required before the data was returned. Thus, two rising edges of SysClk occurred where neither RdCEn or Ack were asserted. On the third rising edge of SysClk, RdCEn was asserted. Optionally, Ack could also be asserted at this time, although it is not strictly necessary.

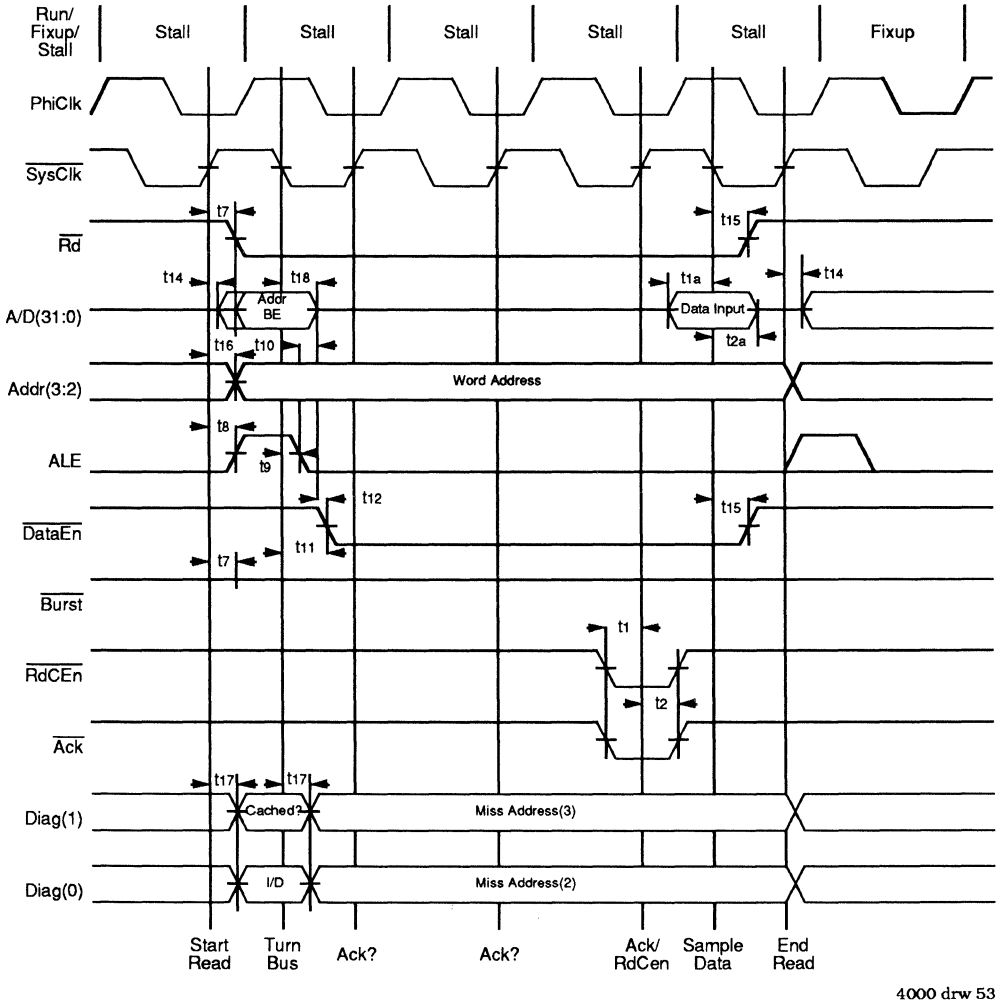
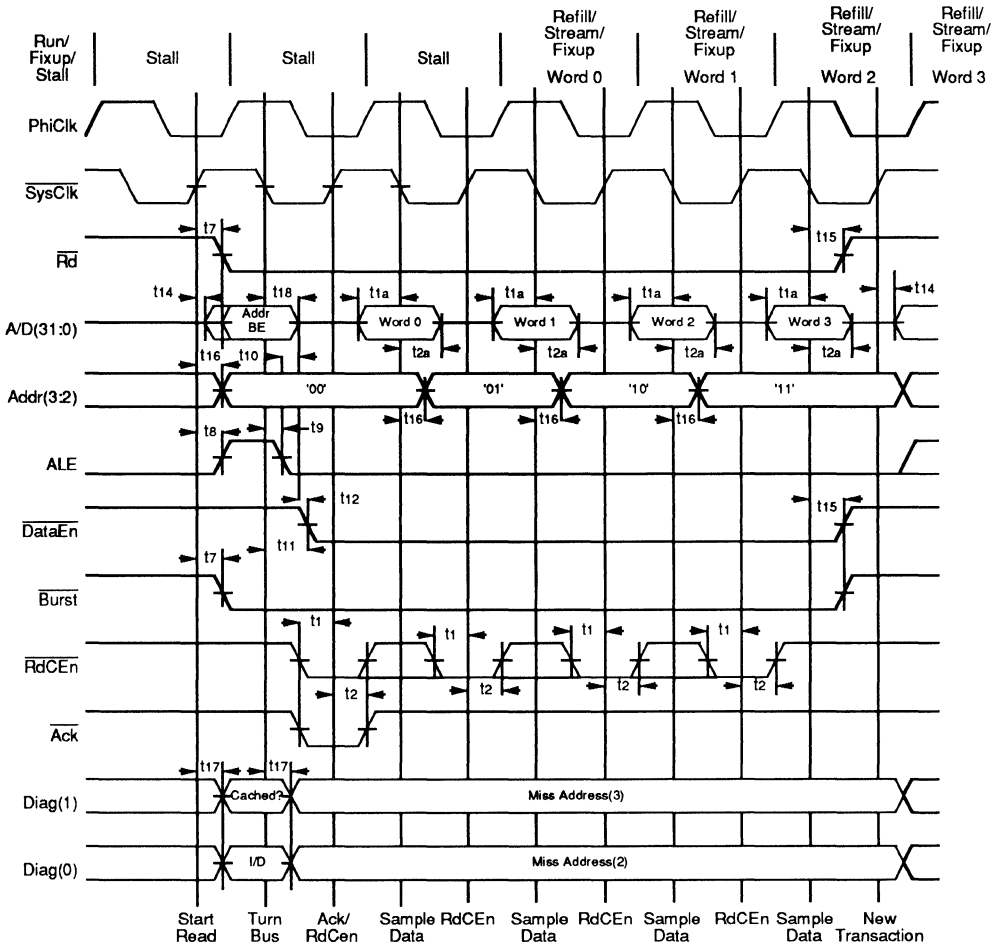



**Figure 7.9. Single Word Read With Bus Wait Cycles**




4000 drw 54

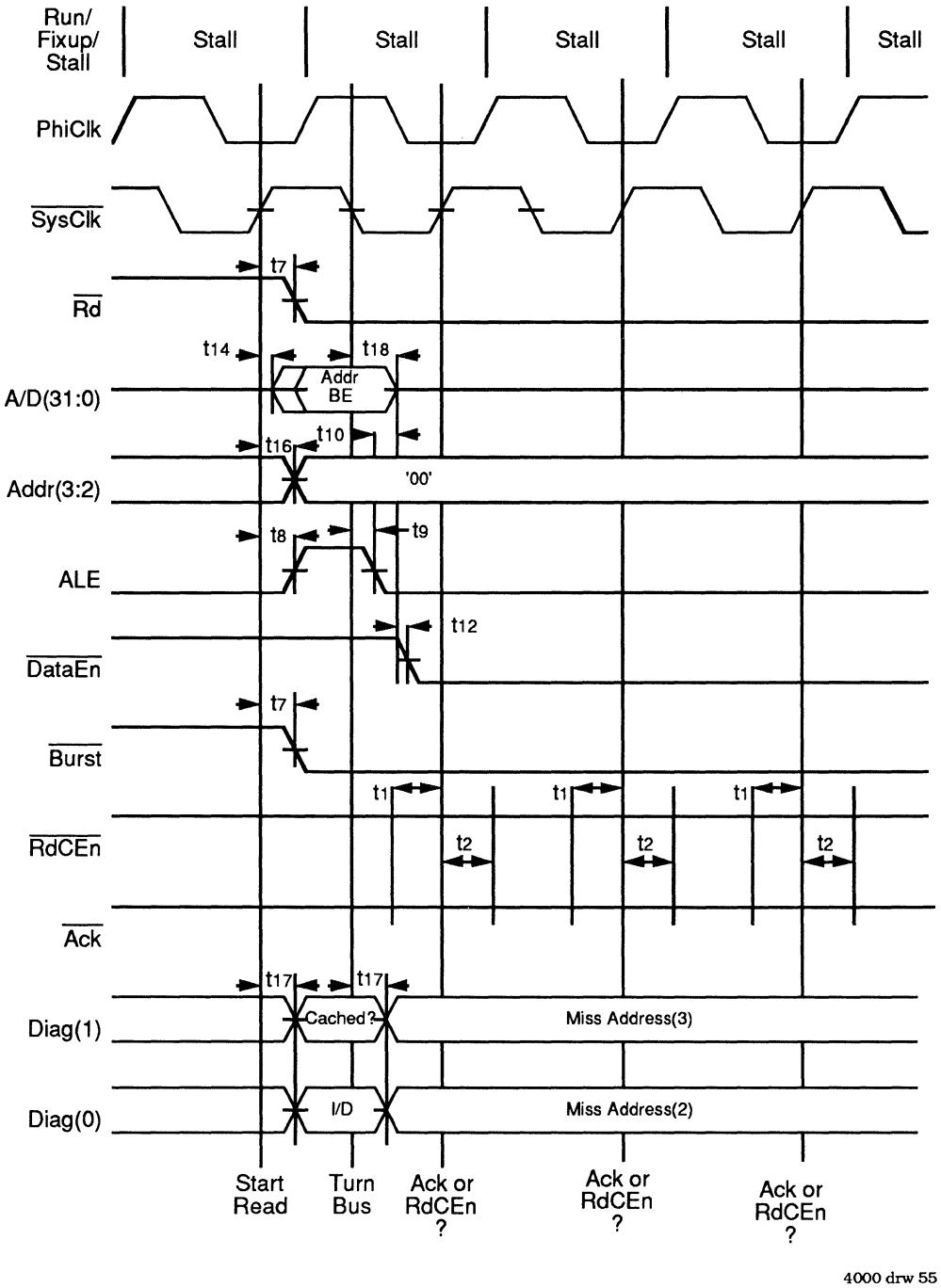
**Figure 7.10. Burst Read With No Wait Cycles**

## Block Reads

Figure 7.10 illustrates the absolute fastest block read. The first word of the block is returned in the second cycle of the read; each additional word is returned in the immediately subsequent clock cycle. Thus, $\overline{Ack}$ can be returned simultaneously with the first $\overline{RdCEn}$, to minimize the number of processor stall cycles.

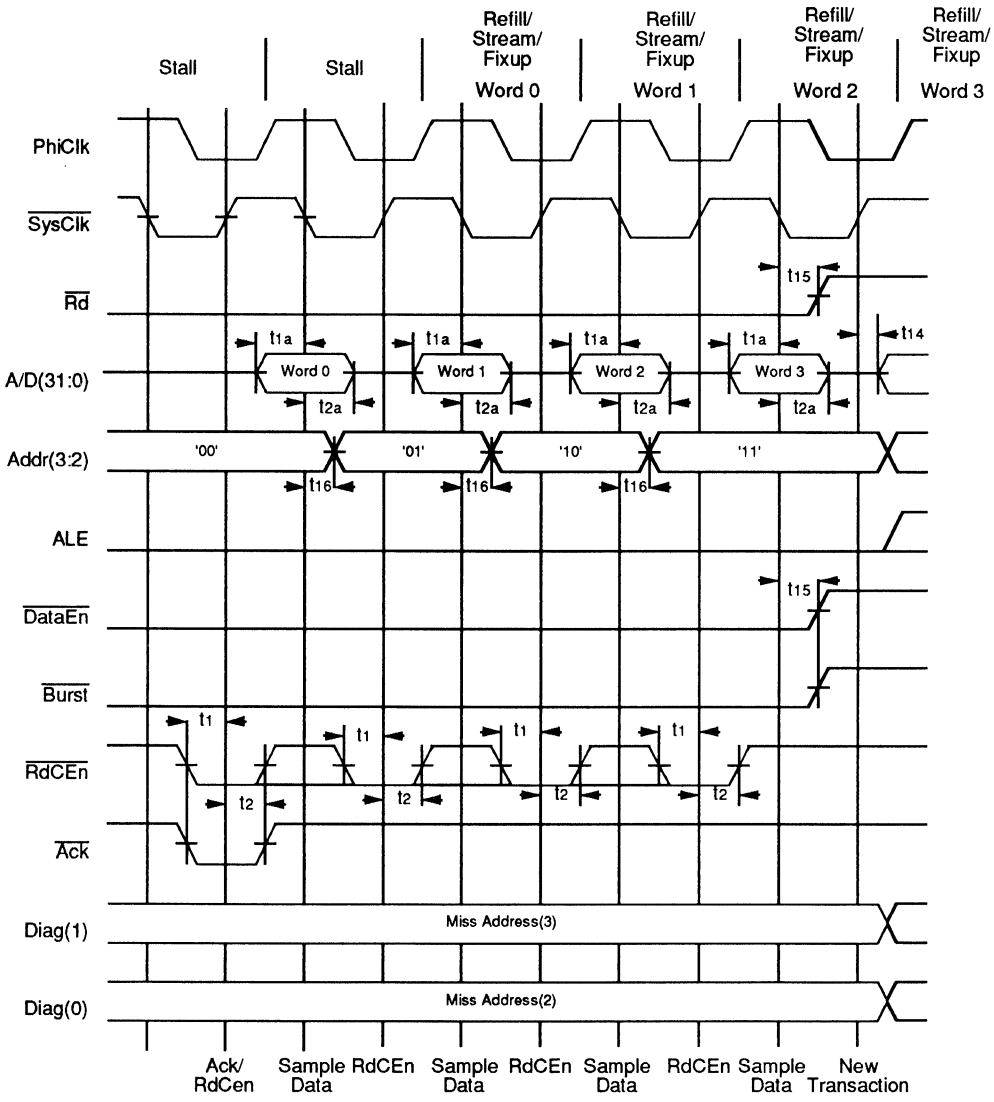
Note that although $\overline{Ack}$ is brought in the first data cycle, a number of clock cycles are required before the processor negates the Rd control output. Thus, the system designer is assured that $\overline{Rd}$ remains active as long as the processor continues to expect data.

Figure 7.11 (a, b) illustrates a block read in which bus wait cycles are required before the first word is brought to the processor, but in which additional words can be brought in at the processor clock rate. Thus, as with the no wait cycle operation, $\overline{Ack}$ is returned simultaneously with the first $\overline{RdCEn}$. Figure 7.11 (a) illustrates the start of the block read, including initial wait cycles to the first word; Figure 7.11 (b) illustrates the activity which occurs as data is brought onto the chip and the read is terminated.




4000 drw 55

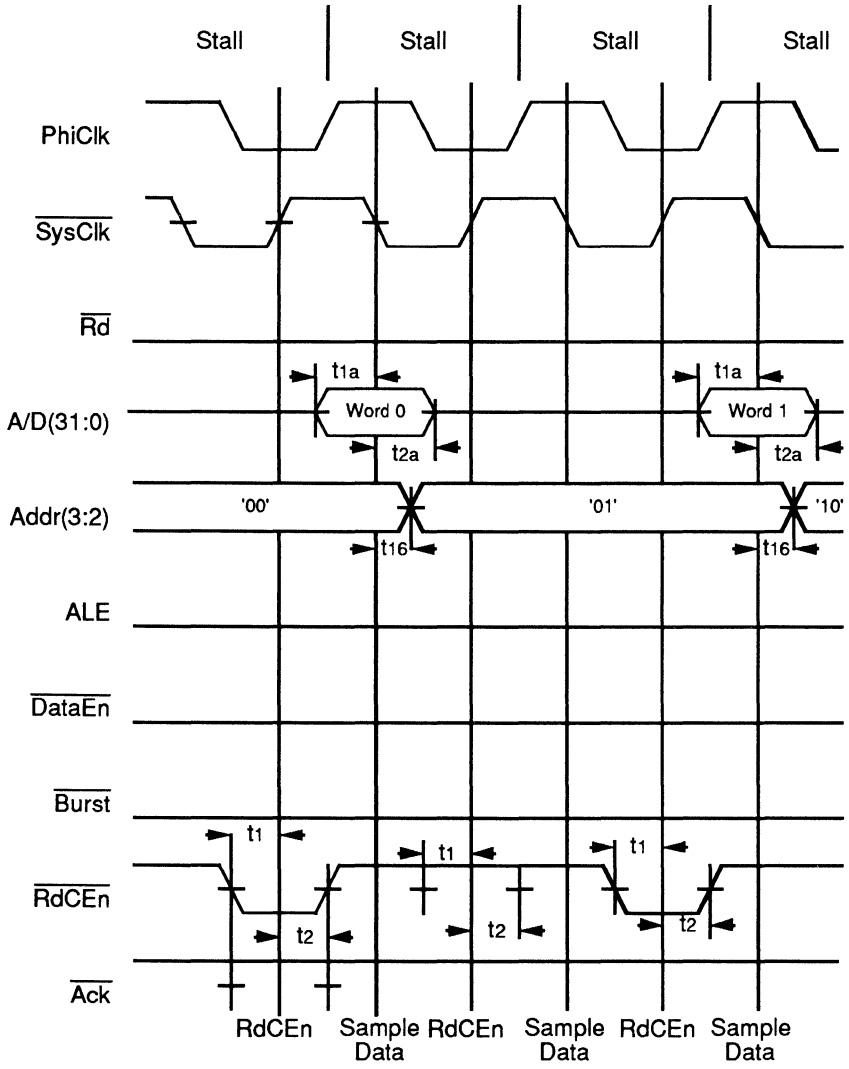
**Figure 7.11a. Start of Burst Read With Initial Wait Cycles**




4000 drw 56

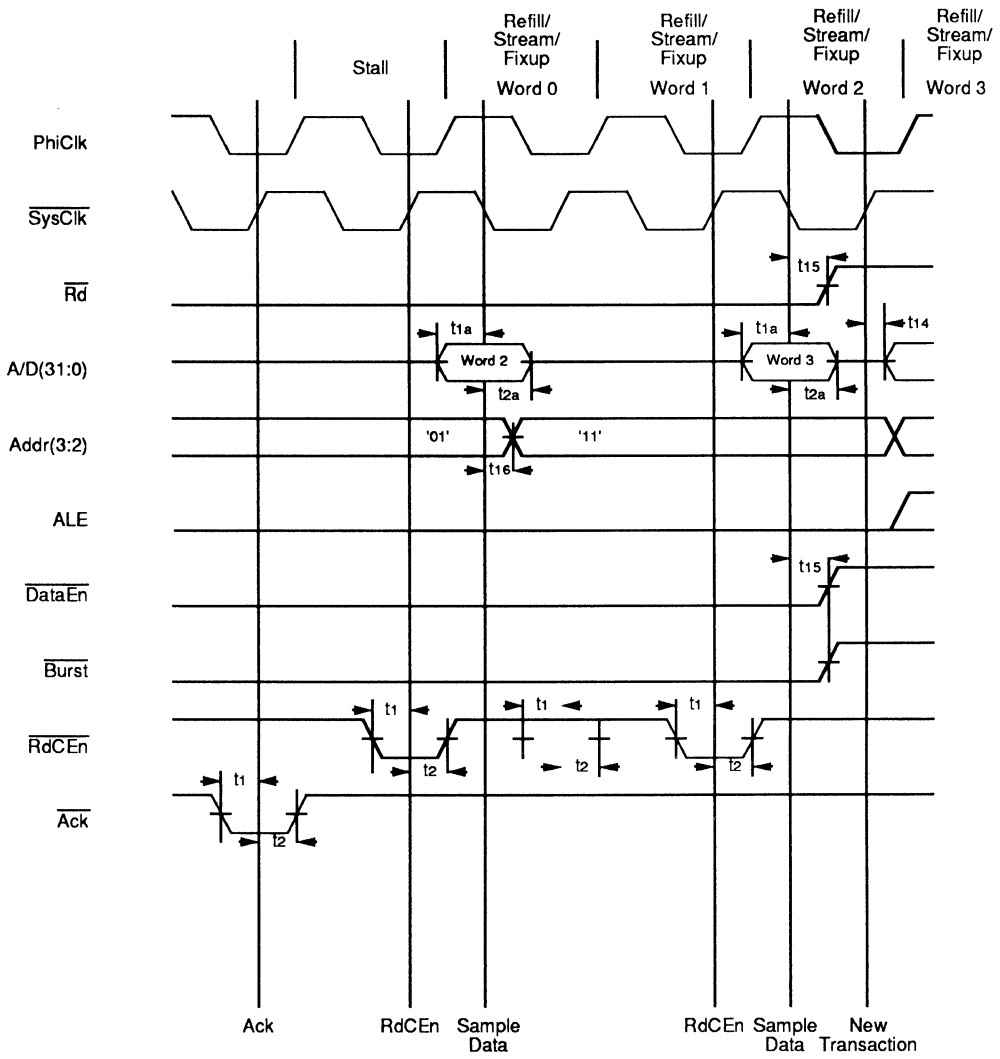
**Figure 7.11b. End of Burst Read**

Figure 7.12 (a, b) illustrates a block read in which bus wait cycles are required before the first word is returned, and in which wait cycles are required between subsequent words: figure 7.12 (a) illustrates the the first two words of the block being brought on chip; figure 7.12 (b) illustrates the last two words of the read, including the optimum timing of $\overline{Ack}$, and the negation of the read control signals.




4000 drw 57

**Figure 7.12a. First Two Words of "Throttled" Quad Word Read**




4000 drw 58

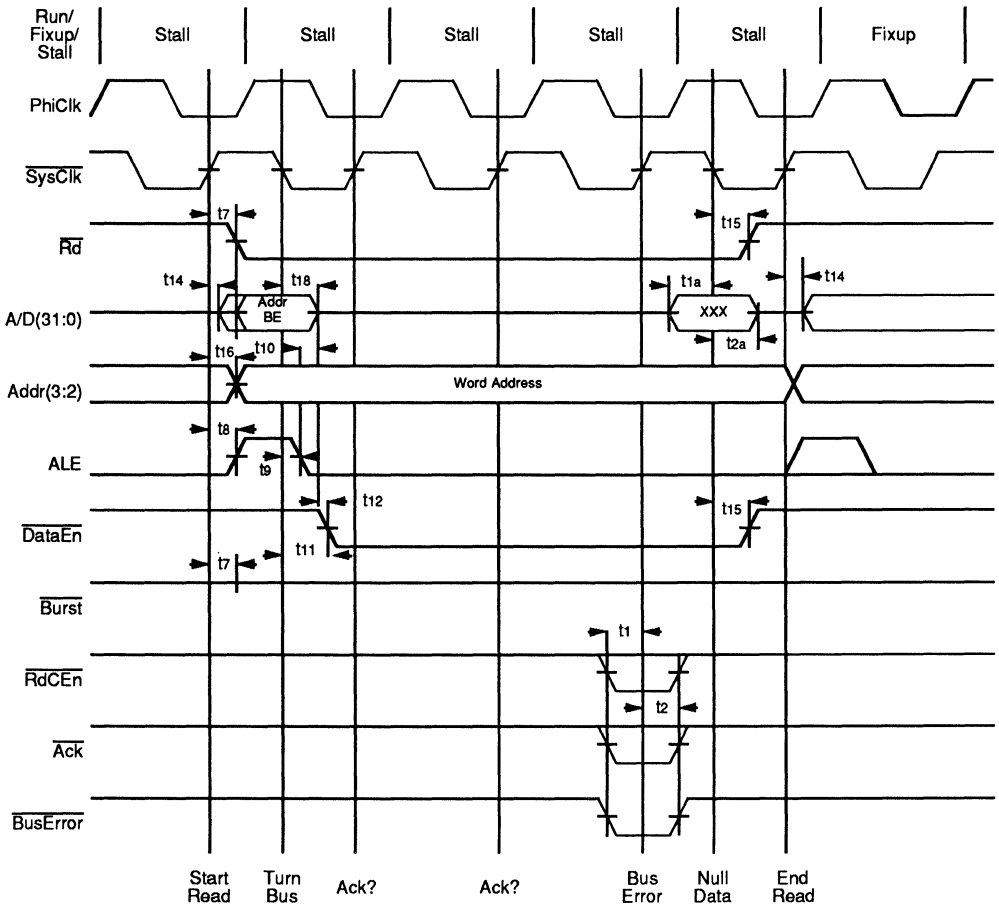
**Figure 7.12b. End of Throttled Quad Word Read**

In this diagram, the memory system returns $\overline{Ack}$ according to when the processor will empty the read buffer. In order to determine the optimum time to return $\overline{Ack}$, the system designer must look at when the processor would read the fourth word from the read buffer. Align this cycle with one clock cycle after the memory system will return the fourth word to the processor. As shown in figure 5.12(b), the memory system should return $\overline{Ack}$ five cycles prior to when the execution core requires the fourth word. The system designer should also insure that the third, second, etc. words of the read cycle are available to the read buffer before the execution core removes them to the caches.

### Bus Error Operation

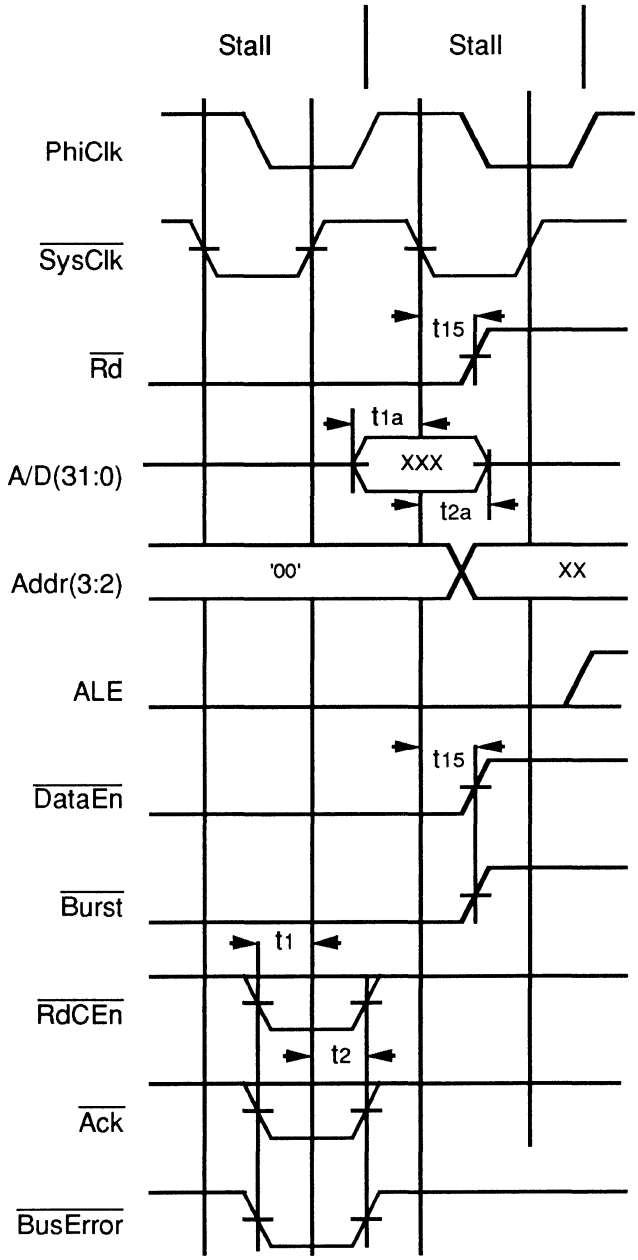
Figure 7.13 is a modified version of Figure 7.9 (single word read with wait cycles), in which BusError is used to terminate the read cycle. In this diagram, note that RdCEn does not need to be asserted, since the processor will insure that the contents of the A/D bus do not get written into the cache or executed. In single word reads, BusError can be asserted anytime up until Ack is asserted. If BusError and Ack are asserted simultaneously, the BusError will be processed; if BusError is asserted after Ack is sampled, it will be ignored.

Figure 7.14 shows the impact of BusError on block reads. The assertion of BusError is allowed up until the assertion of Ack. Once BusError is asserted (sampled on a rising edge of SysClk), the read cycle will be terminated immediately, regardless of how many words have been written into the read buffer. Note that this means that the external memory system should stop cycling RdCen at this time, since a late RdCEn may be erroneously detected as part of a subsequent read. Note that if BusError and Ack are asserted simultaneously, BusError processing will occur. If BusError is asserted after Ack, the BusError will be ignored.




4000 drw 59

**Figure 7.13. Single Word Read Terminated by Bus Error**




4000 drw 60

**Figure 7.14. Block Read Terminated by Bus Error**

## INTRODUCTION

The write protocol of the R3051 family has been designed to complement the read interface of the processor. Many of the same signals are used for both reads and writes, simplifying the design of the memory system control logic.

This chapter includes both an overview of the write interface as well as provides detailed timing diagrams of the write interface.

## IMPORTANCE OF WRITES IN R3051 FAMILY SYSTEMS

The design goal of the write interface was to achieve two things:

Insure that a relatively slow write cycle does not unduly degrade the performance of the processor. To this end, a four deep write buffer has been incorporated on chip. The role of the write buffer is to decouple the speed of the memory interface from the speed of the execution engine. The write buffer captures store information (data, address, and transaction size) from the processor at its clock rate, and later presents it to the memory interface at the rate it can perform the writes. Four such buffer entries are incorporated, thus allowing the processor to continue execution even when performing a quick succession of writes. Only when the write buffer is filled must the processor stall; simulations have shown that significantly less than 1% of processor clock cycles are lost to write buffer full stalls.

Allow the memory system to optimize for fast writes. To this end, a number of design decisions were made: the $\overline{\text{WrNear}}$ signal is provided to allow page mode writes to be used in even simple memory systems; the A/D bus presents the data to be written in the second phase of the first clock cycle of a write transaction; and writes can be performed in as few as two clock cycles.

All though it may be counter-intuitive, a significant percentage of the bus traffic will in fact be processor writes to memory. This can be demonstrated if one assumes the following:

**Instruction Mix:**

| | |
|---|---|
| ALU Operations | 55% |
| Branch Operations | 15% |
| Load Operations | 20% |
| Store Operations | 10% |

**Cache Performance**

| | |
|---|---|
| Instruction Hit Rate | 98% |
| Data Hit Rate | 96% |

Under these assumptions, in 100 instructions, the processor would perform:

2 Reads to process instruction cache misses on instruction fetches
4% x 20 = 0.8 reads to process data cache misses on loads
10 store operations to the write through cache
Total: 2.8 reads and 10 writes

Thus, in this example, over 75% of the bus transactions are write operations, even though only 10 instructions were store operations, vs. 100 instruction fetches and 20 data fetches.

## TYPES OF WRITE TRANSACTIONS

Unlike instruction fetches and data loads, which are usually satisfied by the on-chip caches and thus are not seen at the bus interface, all write activity is seen at the bus interface as single write transactions. There is no such thing as a "burst write"; the processor performs a word or subword write as a single autonomous bus transaction; however, the WrNear output does allow successive write transactions to be processed using page mode writes. This is particularly important when "flushing" the write buffer before performing a data read.

Thus, there really is only one type of write transaction: however, the memory system may elect to take advantage of the assertion of WrNear during a write to perform quicker write operations than would otherwise be performed. Alternately, a high-performance DRAM controller may utilize a different strategy for performing page mode transactions (read or write) to the DRAM.

In processing writes, there is only one parameter of interest: the latency of the write. This latency is influenced by the overall system architecture as well as the type of memory system being addressed: time required to perform address decoding and bus arbitration, memory pre-charge requirements, and memory control requirements, as well as memory access time. WrNear may be used to reduce the latency of successive write operations.

The R3051 family has been designed to accommodate a wide variety of memory system designs, including no wait cycle operations (write completed in two cycles) through simpler, slower systems incorporating many bus wait cycles.

### Partial Word Writes

When the processor issues a store instruction which stores less than a 32-bit quantity, a partial word store occurs. The R3051 family processes partial word stores using a two clock cycle sequence:

It attempts a cache read to see if the store address is cache resident. If it is, it will merge the partial word with the word read from the cache, and write the resulting word back into the cache.

It will use a second clock cycle to allow the write buffer to capture the partial word data and target address and update the cache if appropriate.

## WRITE INTERFACE SIGNALS

The write interface uses the following signals:

**$\overline{Wr}$ O**

This output indicates that a write operation is occurring.

**A/D (31:0) I/O**

During write operations, this bus is used to transmit the write target address to the memory system, and is also used to transmit the store data to the memory system. Its function is de-multiplexed using other control signals.

During the addressing portion of the write transaction, this bus contains the following:

| | |
|---|---|
| Address(31:4) | The upper 28 bits of the write address are presented on A/D (31:4). |
| $\overline{BE}$(3:0) | The byte strobes for the write transaction are presented on A/D(3:0). |

During the data portion of the write transaction, the A/D bus contains the store data on the appropriate data lines, as indicated by the $\overline{BE}$ strobes during the addressing phase.

**ALE O**

This output signal is typically connected directly to the latch enable of transparent latches. Latches are typically used to de-multiplex the address and Byte Enable information from the A/D bus.

**Addr(3:2) O**

The remaining bits of the transfer address are presented directly on these outputs. During write transactions, these pins contain Address (3:2) of the transfer address.

**$\overline{DataEn}$ O**

This output will remain high throughout the write transaction. It is typically used by the memory system to enable output drivers; the CPU will maintain this output as high throughout write transactions, thus disabling memory system output drivers.

**$\overline{WrNear}$ O**

This output is driven valid during the addressing phase of the write transaction. It is asserted if:

1: The store target address of this write operation has the same Addr(31:10) as the previous write transaction, and

2: No read or DMA transaction has occurred since the last write.

If one or both of these conditions are not met, the $\overline{WrNear}$ output will not be asserted during the write transaction.

**$\overline{\text{Ack}}$** I

Acknowledge is used by the memory system to indicate that it has sufficiently processed the write transaction, and that the CPU may terminate the write transaction (and cease driving the write data).

**$\overline{\text{BusError}}$** I

This input can also be used to terminate a write operation. $\overline{\text{BusError}}$ asserted during a write will not cause the processor to take a $\overline{\text{BusError}}$ exception. If the system designer would like the occurrence of a $\overline{\text{BusError}}$ to cause a processor exception, he must use it to externally generate an interrupt to the processor. Write transactions terminated by $\overline{\text{BusError}}$ do not require the assertion of $\overline{\text{Ack}}$. $\overline{\text{BusError}}$ can be asserted at at any time the processor is looking for $\overline{\text{Ack}}$ to be asserted, up to and including the cycle in which the memory system does signal $\overline{\text{Ack}}$.

## WRITE INTERFACE TIMING OVERVIEW

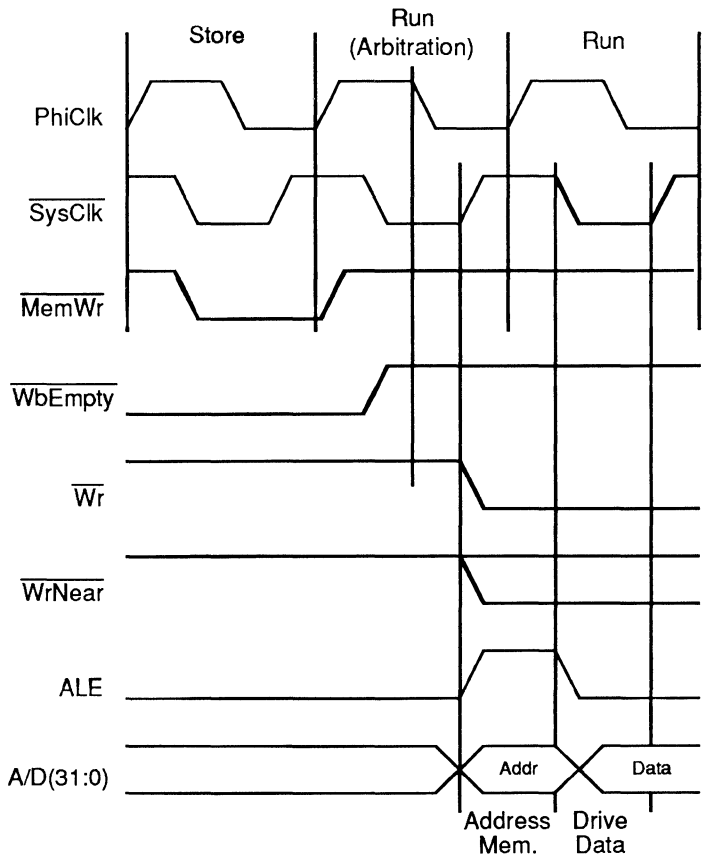
The protocol for transmitting data from the processor to memory and I/O devices is discussed below.

### Initiating the Write

A write transaction occurs when the processor has placed data into the write buffer, and the bus interface is either free, or write has the highest priority. Internally, the processor bus arbiter uses the NotEmpty indicator from the write buffer to indicate that a write is being requested.

Assuming that the write transaction can be processed (that is, there are no ongoing bus operations, and no higher priority operations pending), the processor will initiate a bus write transaction on the next rising edge of $\overline{\text{SysClk}}$. Higher priority operations would have the effect of delaying the start of the write.

Figure 8.1 illustrates the initiation of a write transaction, based on the internal assertion of the $\overline{\text{WbEmpty}}$ control signal. This figure applies when the processor is performing a write, and the write buffer is otherwise empty. If the write buffer already had data in it, it would continually request the use of the bus until it was emptied; it would be up to the bus interface unit arbiter to decide the priority of the request relative to other pending requests. Additional stores would be captured by other write buffer entries, until the write buffer was filled.




4000 drw 61

**Figure 8.1. Start of Write Operation – BIU Arbitration**

### Memory Addressing

A write transaction begins when the processor asserts its $\overline{Wr}$ control output, and also drives the address and other control information onto the A/D and memory interface bus. Figure 8.2 illustrates the start of a processor write transaction, including the addressing of memory and presenting the store data on the A/D bus.

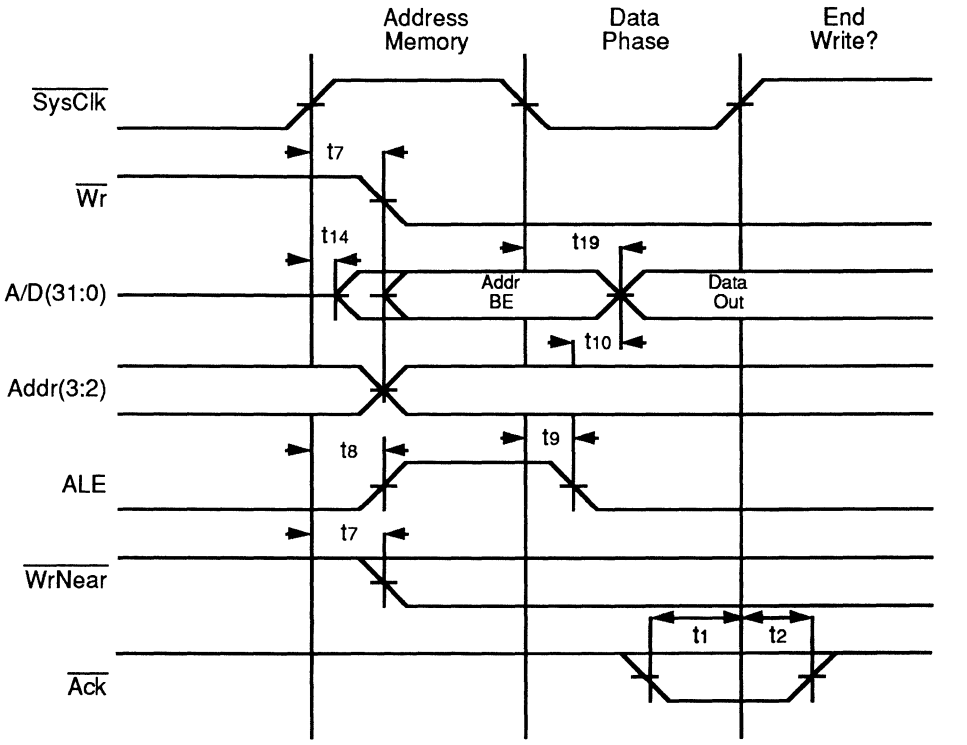
The addressing occurs in a half-cycle of the $\overline{SysClk}$ output. At the rising edge of $\overline{SysClk}$, the processor will drive the write target address onto the A/D bus. At this time, ALE will also be asserted, to allow an external transparent latch to capture the address. Depending on the system design, address decoding could occur in parallel with address de-multiplexing (that is, the decoder could start on the assertion of ALE, and the output of the decoder captured by ALE), or could occur on the output side of the transparent latches. During this phase, $\overline{WrNear}$ will also be determined and driven out by the processor.

### Data Phase

Once the A/D bus has presented the address for the transfer, the address is replaced on the A/D bus by the store data. This occurs in the second phase of the first clock cycle of the write transaction, as illustrated in Figure 8.2.

The processor enters the data phase by performing the following sequence of events:

- It negates ALE, causing the transparent address latches to capture the contents of the A/D bus.
- It internally captures the data in a register in the bus interface unit, and enables this register onto its output drivers on the A/D bus. The processor design guarantees that the ALE is negated prior to the address being removed from the A/D bus.




4000 drw 62

**Figure 8.2. Memory Addressing and Start of Write**

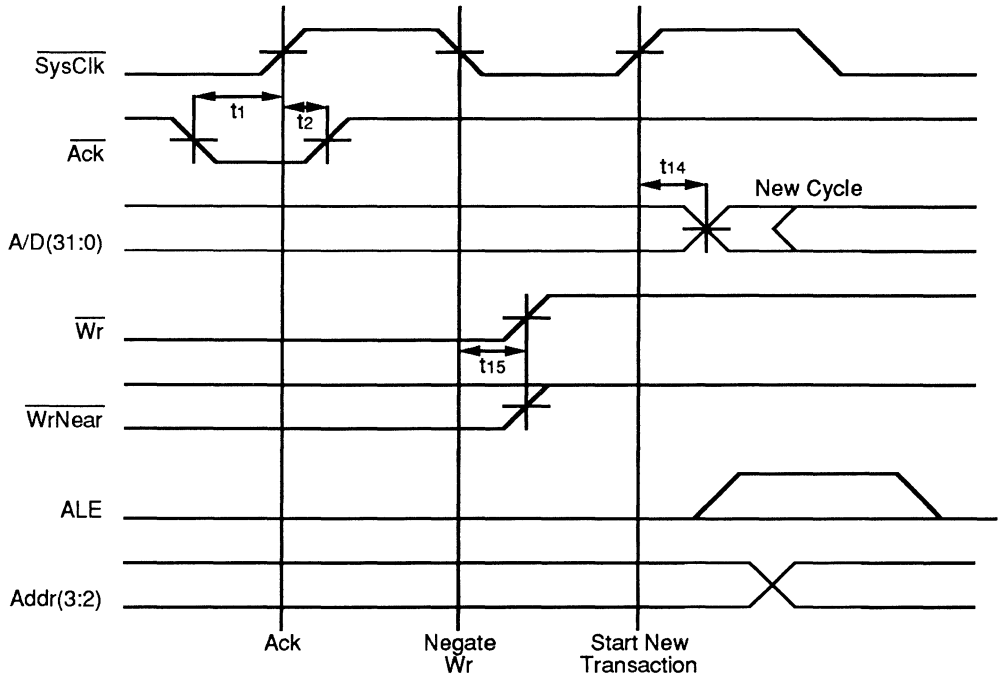
Thus, the processor A/D bus is driving the store data by the end of the second phase of the write transaction. At this time, it begins to look for the end of the write cycle.

### Terminating the Write

There are only two methods for the external memory system to terminate a write operation:

- It can supply an $\overline{\text{Ack}}$ (acknowledge) to the processor, to indicate that it has sufficiently processed the write request, and the processor may terminate the write.
- It can supply a $\overline{\text{BusError}}$ to the processor, to indicate that the requested data transfer has "failed" on the bus. The system interface behavior of the processor when $\overline{\text{BusError}}$ is presented is identical to the behavior when Ack is asserted. In the case of writes terminated by BusError, no exception is taken, and the data transfer cannot be retried.

Figure 8.3 shows the timing of the control signals when the write cycle is being terminated.




4000 drw 63

**Figure 8.3. End of Write**

### Latency Between Processor Operations

In general, the processor may begin a new bus activity in the phase immediately after the termination of the write cycle. This operation may be either a read, write, or bus grant.

Since a new operation may begin one-half clock cycle after the data is sampled from the bus, it is important that the external memory system not rely on the store data still being present on the bus at this time.

### Write Buffer Full Operation

It is possible that the execution core on occasion may be able to fill the on-chip write buffer. If the processor core attempts to perform a store to the write buffer while the buffer is full, the execution core will be stalled by the write buffer until a space is available. Once space is made available, the execution core will use a fixup cycle to "retry" the store, allowing the data to be captured by the write buffer. It will then resume execution.

The write buffer can actually be thought of as "four and one-half" entries: it contains a special data buffer which captures the data being presented by an ongoing bus write transaction. Thus, when the bus interface unit begins a write transaction, the write buffer slot containing the data for that write is freed up in the second phase of the write transaction. If the processor was in a write busy stall, it will be released to write into the now available slot at this time, regardless of how long it takes the memory system to retire the ongoing write.

This operation is illustrated in figure 8.4.

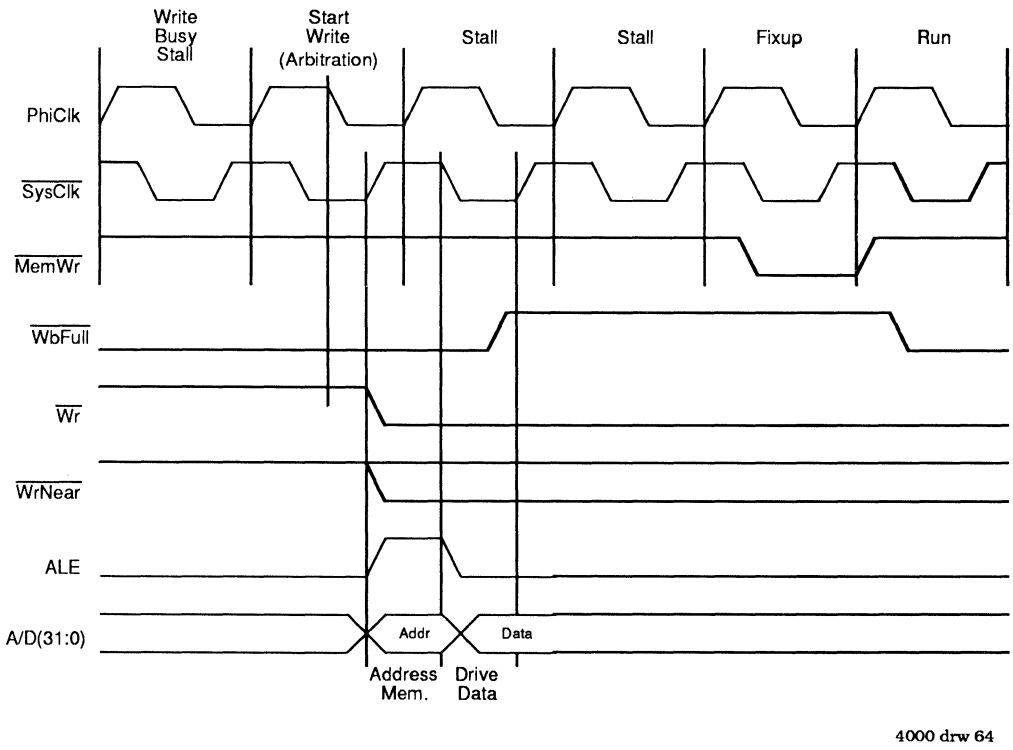



**Figure 8.4. Write Buffer Full Operation**

## WRITE TIMING DIAGRAMS

This section illustrates a number of timing diagrams applicable to R3051 family writes. The values for the AC parameters referenced are contained in the R3051 family data sheet.

### Basic Write

Figure 8.5 illustrates the case of a write operation which did not require wait states. Thus, Ack was detected at the rising edge of SysClk which occurred exactly one clock cycle after the rising edge of SysClk which asserted Wr.

Figure 8.6 also illustrates the case of a basic write. However, in this figure, two bus wait cycles were required before the data was retired. Thus, two rising edges of SysClk occurred where Ack was not asserted. On the third rising edge of SysClk, Ack was asserted, and the write operation was terminated.

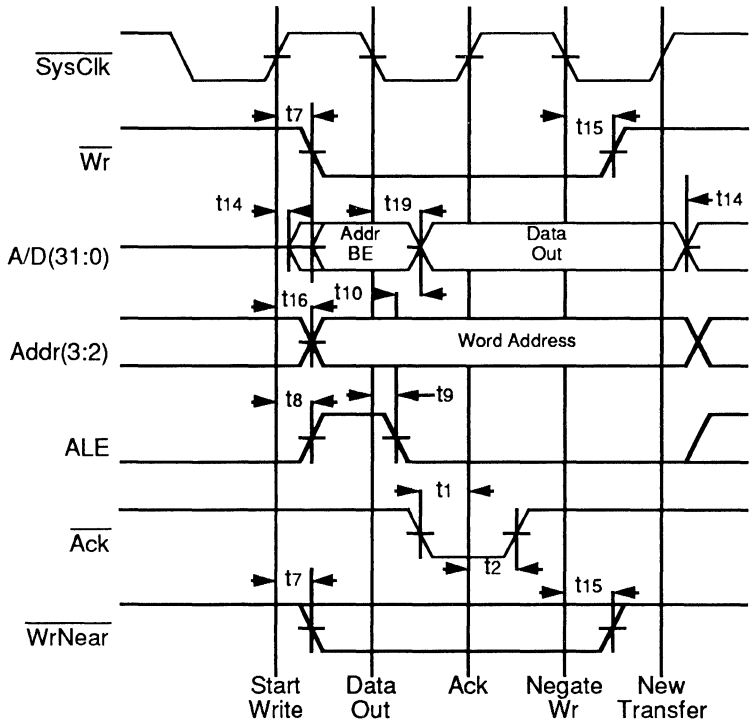



**Figure 8.5. Bus Write With No Wait Cycles**

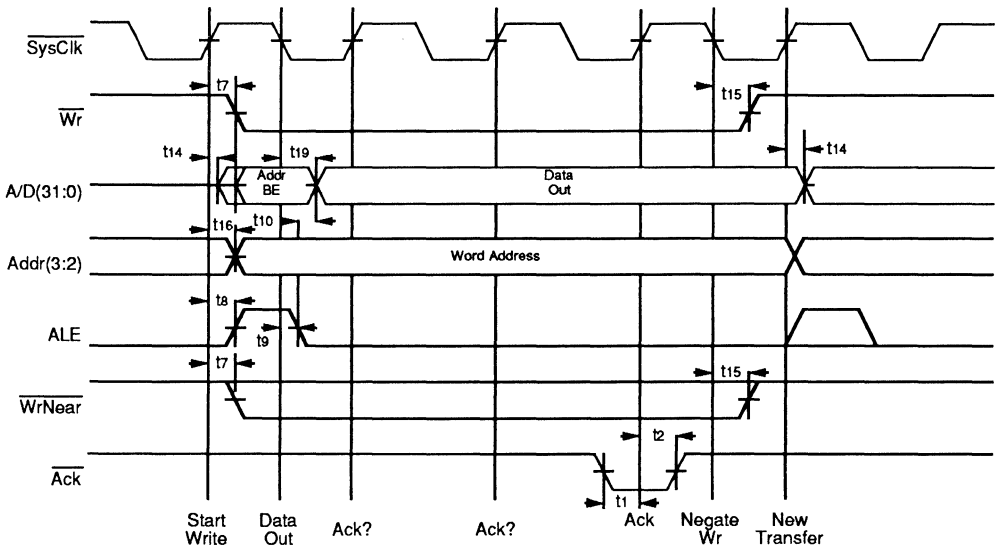



**Figure 8.6. Write With Bus Wait Cycles**

## Bus Error Operation

Figure 8.7 is a modified version of Figure 8.6 (basic write with wait cycles), in which BusError is used to terminate the write cycle. If BusError and Ack are asserted simultaneously, the BusError will be processed; if BusError is asserted after Ack is sampled, it will be ignored.

No exception is taken because such an exception would violate the precise exception model of the processor. Since writes are buffered, the processor program counter will no longer be pointing to the address of the store instruction which requested the write, and other state information of the processor may have been changed. Thus, if the system designer would like the processor core to take an exception as a result of the bus error, he should externally latch the BusError signal, and use the output of the latch to cause an interrupt to the processor.

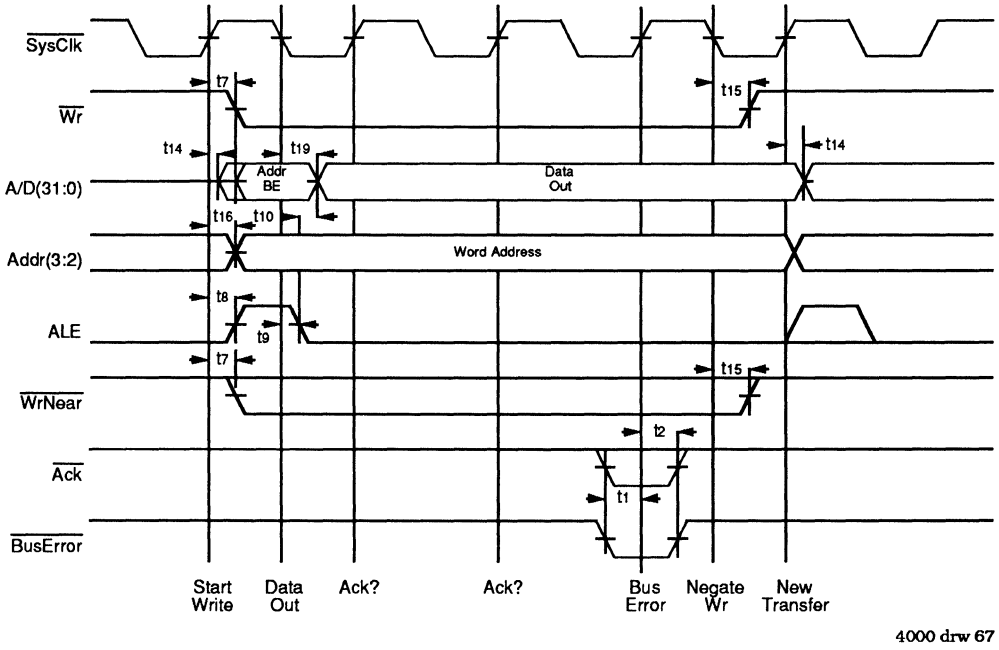



**Figure 8.7. Bus Error on Write**

## INTRODUCTION

The R3051 family contains provisions to allow an external agent to remove the processor from its memory bus, and thus perform transfers (DMA). These provisions use the DMA arbiter to coordinate the external request for mastership with the CPU read and write interface.

The DMA arbiter interface uses a simple two signal protocol to allow an external agent to obtain mastership of the external system bus. Logic internal to the CPU synchronizes the external interface to the internal arbiter unit to insure that no conflicts between the internal synchronous requesters (read and write engines) and external asynchronous (DMA) requester occurs.

## INTERFACE OVERVIEW

An external agent indicates the desire to perform DMA requests by asserting the $\overline{\text{BusReq}}$ input to the processor. DMA requests have the highest priority, and thus, once the request is detected, is guaranteed to gain mastership at the next arbitration.

The CPU indicates that the external DMA cycle may begin by asserting its $\overline{\text{BusGnt}}$ output on the rising edge of $\overline{\text{SysClk}}$ after $\overline{\text{BusReq}}$ is detected with appropriate set-up time to the external rising edge of $\overline{\text{SysClk}}$. During DMA cycles, the processor holds the following memory interface signals in tri-state:

- A/D Bus
- Addr(3:2)
- Interface control signals: $\overline{\text{Rd}}$, $\overline{\text{Wr}}$, $\overline{\text{DataEn}}$, $\overline{\text{Burst}}$/$\overline{\text{WrNear}}$, and ALE
- Diag(1:0)

In addition to tri-stating these signals, the CPU will ignore transitions on $\overline{\text{RdCEn}}$, $\overline{\text{Ack}}$, and $\overline{\text{BusError}}$ during DMA cycles.

Thus, the DMA master can use the same memory control logic as that used by the CPU; it may use $\overline{\text{Burst}}$, for example, to obtain a burst of data from the memory; it may use $\overline{\text{RdCEn}}$ to detect whether the memory has satisfied its request, etc. Thus, DMA can occur at the same speed at which the R3051 family allows data transfers on its bus (a peak of one word per clock cycle). During DMA cycles, the processor will continue to operate out of cache until it requires the bus.

The external agent indicates that the DMA transfer has terminated by negating the $\overline{\text{BusReq}}$ input to the processor, which is sampled on the rising edge of $\overline{\text{SysClk}}$. $\overline{\text{BusGnt}}$ is negated on a falling edge of $\overline{\text{SysClk}}$, so that it will be negated before the assertion of $\overline{\text{Rd}}$ or $\overline{\text{Wr}}$ for a subsequent transfer. On the next rising edge of $\overline{\text{SysClk}}$, the processor will resume driving tri-stated signals.

Note that there is no hardware coherency mechanism defined for DMA transfers relative to either the internal caches or the write buffer. Software must explicitly manage DMA transfers to insure that data conflicts are avoided. This is an appropriate tradeoff for the vast majority of embedded applications.

## DMA ARBITER INTERFACE SIGNALS

**BusReq I**

This signal is an input to the processor, used to request mastership of the external interface bus. Mastership is granted according to the assertion of this input, and taken back based on its negation.

**BusGnt O**

This signal is an output from the processor, used to indicate that it has relinquished mastership of the external interface bus.

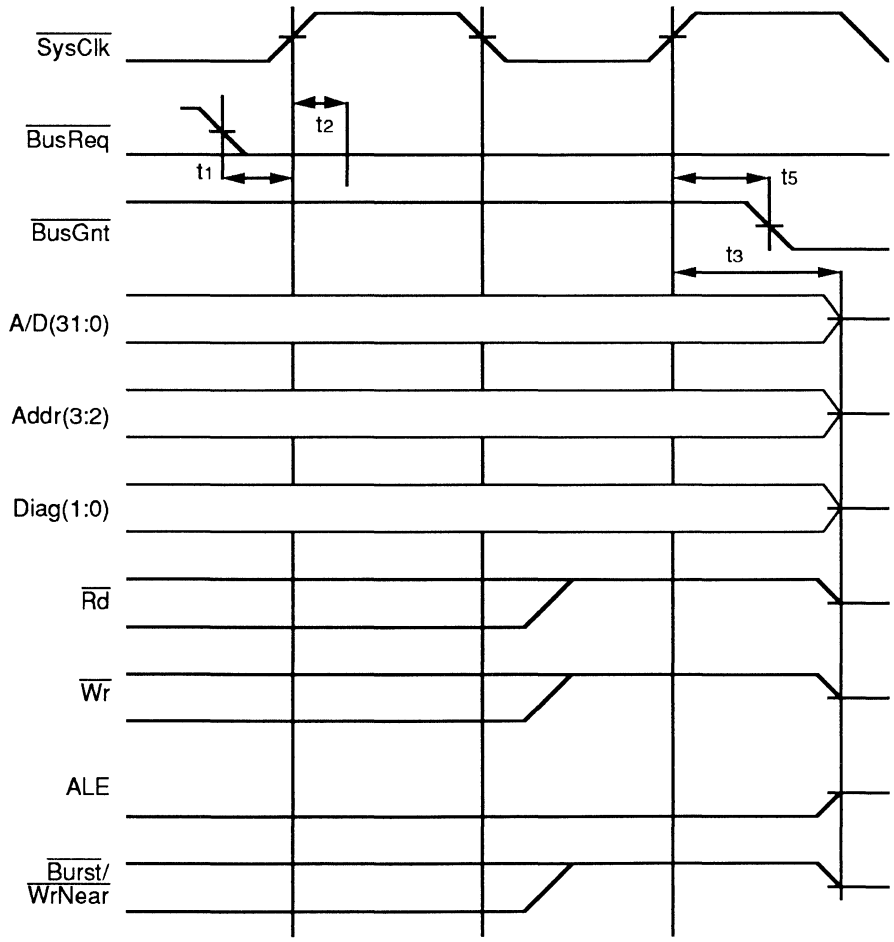
## DMA ARBITER TIMING DIAGRAMS

These figures reference AC timing parameters whose values are contained in the R3051 family data sheet.

### Initiation of DMA Mastership

Figure 9.1 shows the beginning of a DMA cycle. Note that if $\overline{\text{BusReq}}$ were asserted while the processor was performing a read or write operation, $\overline{\text{BusGnt}}$ would be delayed until the next bus slot after the read or write operation is completed.

To initiate DMA, the processor must detect the assertion of $\overline{\text{BusReq}}$ with proper set-up time to $\overline{\text{SysClk}}$. Once $\overline{\text{BusReq}}$ is detected, and the bus is free, the processor will grant control to the requesting agent by asserting its $\overline{\text{BusGnt}}$ output, and tri-stating its output drivers, from a rising edge of $\overline{\text{SysClk}}$. The bus will remain in the control of the external master until it negates $\overline{\text{BusReq}}$, indicating that the processor is once again the bus master.



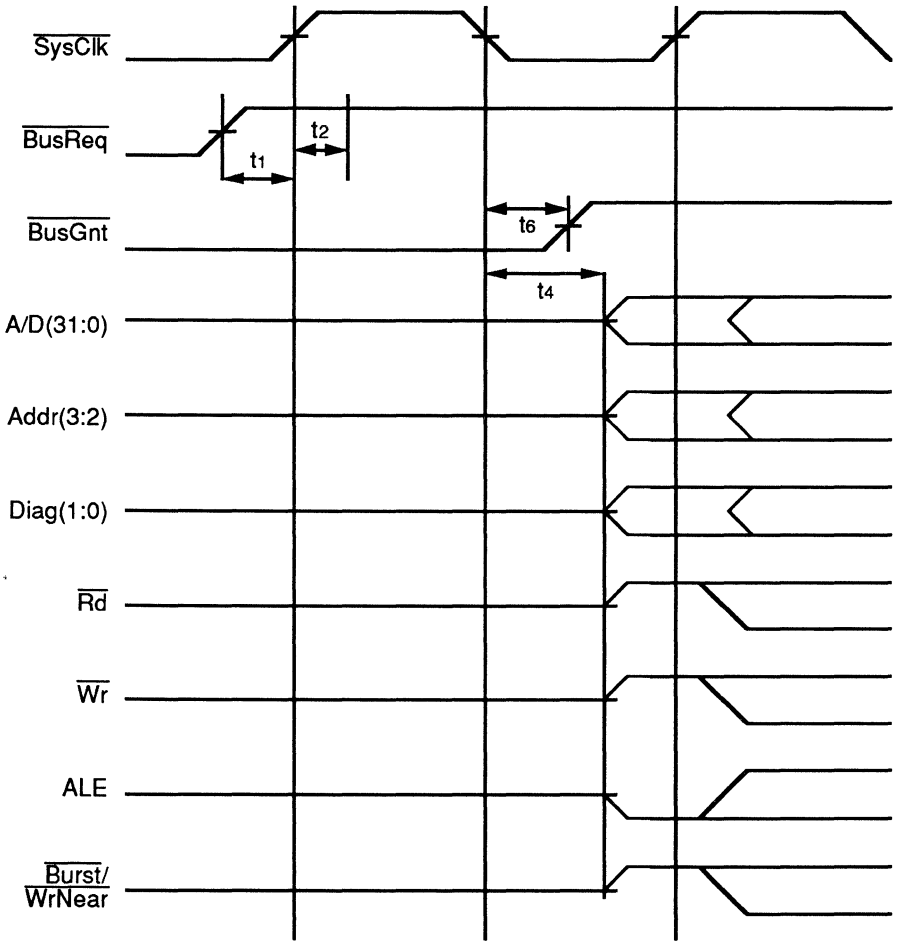

4000 drw 68

**Figure 9.1. Bus Grant and Start of DMA Transaction**

### Relinquishing Mastership Back to the CPU

Figure 9.2 shows the end of a DMA cycle. The next rising edge of SysClk after the negation of BusReq is sampled may actually be the beginning of a processor read or write operation.

To terminate DMA, the external master must negate the processor BusReq input. Once this is detected (with proper setup and hold time), the processor will negate its BusGnt output on the next falling edge of SysClk. It will also re-enable its output drivers. Thus, the external agent must disable its output drivers by this clock edge, to avoid bus conflicts.




4000 drw 69

**Figure 9.2. Regaining Bus Mastership**

# RESET INITIALIZATION AND INPUT CLOCKING

CHAPTER 10

## INTRODUCTION

This chapter discusses the reset initialization sequence required by the R3051 family. Also included is a discussion of the mode selectable features of the processor, and of the software requirements of the boot program.

There are a small number of selectable features in the R3051 family. These mode selectable features are determined by the polarity of the appropriate Interrupt inputs when the rising edge of $\overline{\text{Reset}}$ occurs.

## RESET TIMING

Unlike the R3000, which requires the use of a state machine during the last four cycles of reset to initialize the device and perform mode selection, the R3051 family requires a very simple reset sequence. There are only two concerns for the system designer:

- The set-up time and hold requirements of the interrupt inputs (mode selectable features) with respect to the rising edge of $\overline{\text{Reset}}$ are met.
- The minimum $\overline{\text{Reset}}$ pulse width is satisfied.

## MODE SELECTABLE FEATURES

The R3051 family has features which are determined at reset time. This is done using a latch internal to the CPU: this latch samples the contents of the Interrupt bus ($\overline{\text{SInt}}$(5:3) and $\overline{\text{Int}}$(2:0)) at the negating edge of $\overline{\text{Reset}}$. The encoding of the mode selectable features on the interrupt bus is described in Table 10.1.

| Interrupt Pin | Mode Feature |
|---|---|
| $\overline{\text{SInt}}$(5:3) | Reserved |
| $\overline{\text{Int}}$(2) | DBlockRefill |
| $\overline{\text{Int}}$(1) | $\overline{\text{Tri-State}}$ |
| $\overline{\text{Int}}$(0) | BigEndian |

4000 tbl 23

**Table 10.1. R3051 Family Mode Selectable Features**

### Reserved

Reserved mode bits must be driven high.

### DBlockRefill

If asserted (active high), data cache misses will be processed using quad word refills. If negated, data cache misses will be processed using single word reads. This mode bit does not affect the processing of instruction cache misses (always handled as quad word reads) or uncacheable references (always handled as single word reads).

### Tri-State

If asserted (active low) at the end of reset, all CPU outputs (except $\overline{\text{SysClk}}$) will remain in tri-state after reset. They will remain in tri-state until another reset occurs (with tri-state disabled).

This mode input has the unique feature that it can be used to force the CPU outputs to tri-state during the entire reset period. That is, if Tri-State is asserted while Reset is asserted, the processor outputs will be tri-stated through the reset period. If Tri-State is negated during reset, the output drivers will be enabled. Again, note that the Tri-State mode does not affect SysClk, which is driven regardless of the tri-state mode.

Thus, it is possible to hold tri-state low during the majority of reset, and bring it high only during the last four cycles of reset. The CPU outputs would be tri-state through the reset, but the processor would operate normally after reset. This is useful in board testing, and also for in-circuit emulators.

**BigEndian**

If asserted (active high), the processor will operate as a big-endian machine, and the RE bit of the status register would then allow little-endian tasks to operate in a big-endian system. If negated, the processor (and kernel) will be little-endian, and the RE bit will allow big-endian tasks to operate on a little-endian machine.

**R3000A Equivalent Modes**

The R3000A features a number of modes, which are selected at Reset time. Although most of those modes are irrelevant, a number of equivalences can be made:

- IBlkSize = 4 word refill.
- DBlkSize = 1 or 4 word refill, depending on the DBlockRefill mode selected.
- Reverse Endianness enabled.
- Instruction Streaming enabled.
- Partial Word Stores enabled.

Other modes of the R3000A primarily pertain to its cache interface, which is incorporated within the R3051 family and thus transparent to users of these processors.

## RESET BEHAVIOR

The R3051 family samples for the negation of Reset relative to a falling edge of SysClk. The processor will initiate a read request for the instruction located at the Reset Vector at the 6th rising edge of SysClk after the negation of Reset is detected. These cycles are a result of:

- Reset input synchronization performed by the CPU. The Reset input uses special synchronization logic, thus allowing Reset to be negated asynchronously to the processor. This synchronization logic introduces a two cycle delay between the external negation of Reset and the negation of Reset to the execution core.
- Internal clock cycles in which the execution core flushes its pipeline, before it attempts to read the exception vector.
- One additional cycle for the read request to propagate from the internal execution core to the read interface, as described in Chapter 7.

## BOOT SOFTWARE REQUIREMENTS

Basic mode selection is performed using hardware during the reset sequence, as discussed in the mode initialization section. However, there are certain aspects of the boot sequence that must be performed by software.

The assertion and subsequent negation of reset forces the CPU to begin execution at the reset vector, which is address 0x1FC0_0000. This address resides in uncached, unmapped memory, and thus does not require that the caches or TLB be initialized for the processor to execute boot code.

The processor needs to perform the following activities during boot:

- **Initialize the caches**
  The processor needs to determine the sizes of the on-chip caches, and flush each entry, as discussed in Chapter 3. This must be done before the processor attempts to execute cacheable code.

- **Initialize the TLB**
  The processor needs to examine the TLB Shutdown bit to determine if a TLB is present. If this is an extended architecture version of the processor, software must sequence through all 64 TLB entries, giving them either a valid translation, or marking them as not Valid. This must be done before software attempts to reference through mapped space.

- **Initialize CP0 Registers**
  The processor should establish appropriate values in various CP0 registers, including:

  The PID field of EntryHi.

  The IM bits of the status register.

  The BEV bit.

  Initialize KUp/IEp so that user state can be entered using a RFE instruction

- **Enter User State**

  Branch to the first user task, and perform an RFE.

## DETAILED RESET TIMING DIAGRAMS

The timing requirements of the processor reset sequence are illustrated below. The timing diagrams reference AC parameters whose values are contained in the R3051 family data sheet.

### Reset Pulse Width

There are two parameters to be concerned with: the power on reset pulse width, and the warm reset pulse width.

Figure 10.1 illustrates the power on reset requirements of the R3051 family.

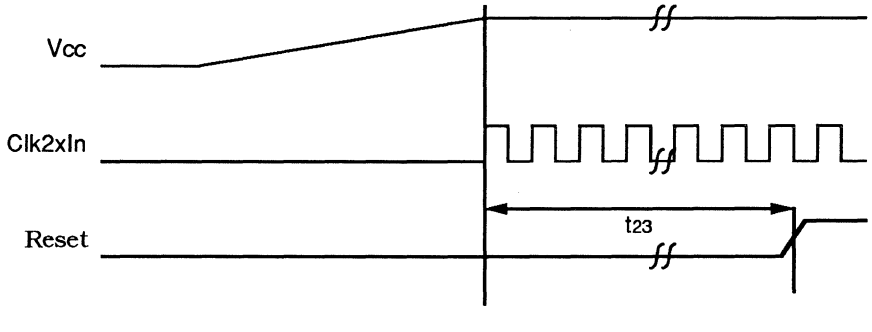



**Figure 10.1. Cold Start**

Figure 10.2 illustrates the warm reset requirements of the processor.

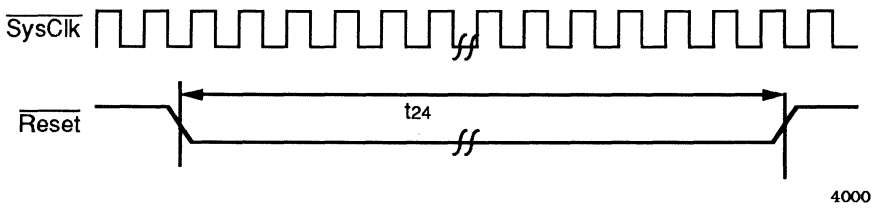



**Figure 10.2. Warm Reset**

### Mode Initialization Timing Requirements

The mode initialization vectors are sampled by a transparent latch, whose output enable is directly controlled by the Reset input of the processor. The internal structure of the processor is illustrated in Figure 10.3.

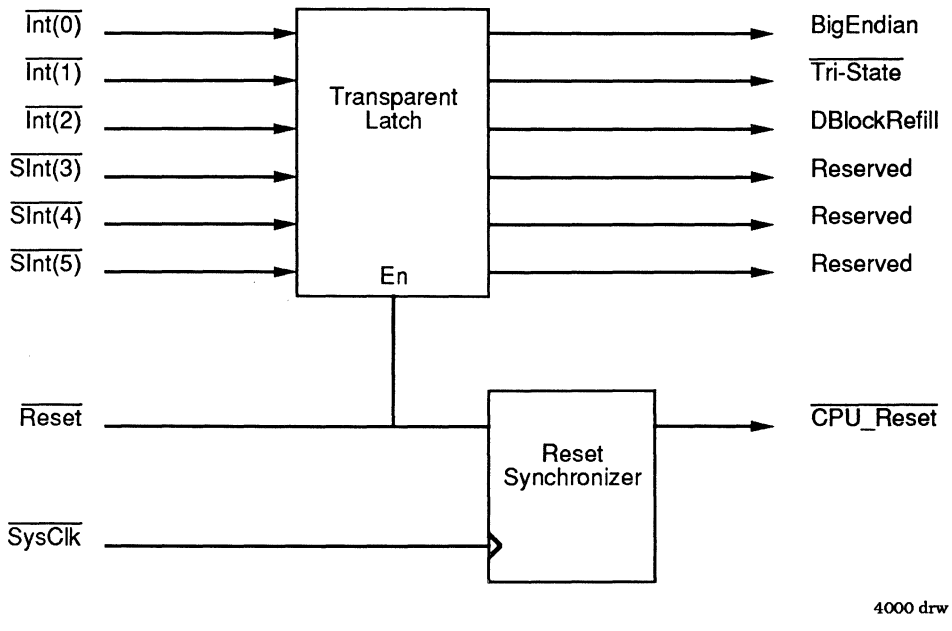
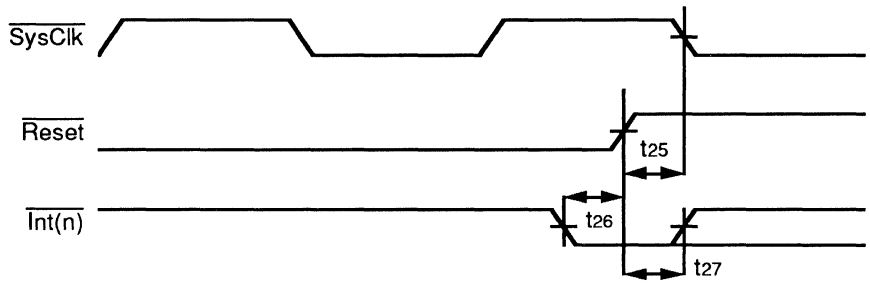



**Figure 10.3. Mode Vector Logic**

Thus, the mode vectors have a set-up and hold time with respect to the rising edge of Reset, as illustrated in Figure 10.4.




4000 drw 73

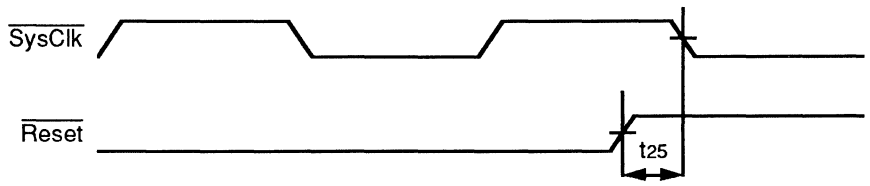
**Figure 10.4. Mode Vector Timing**

### Reset Setup Time Requirements

The reset signal incorporates special synchronization logic which allows it to be driven from an asynchronous source. This is done to allow the processor Reset signal to be derived from a simple circuit, such as an RC network with a time constant long enough to guarantee the reset pulse width requirement is met.

The Reset set-up time parameter can then be thought of as the amount of time Reset must be negated before the rising edge of SysClk for it to be guaranteed to be recognized; failure to meet this requirement will not result in improper operation, but rather will have the effect of delaying the internal recognition of the end of reset by one clock cycle. This does not affect the timing of the sampling of the mode initialization vectors.

Figure 10.5 illustrates the set-up time parameter of the R3051 family.

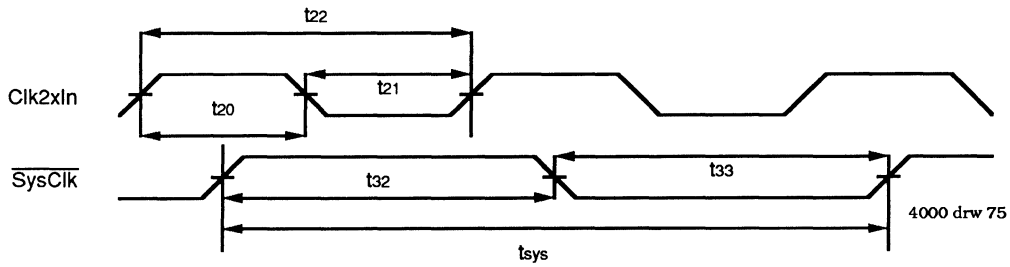



4000 drw 74

**Figure 10.5. Reset Timing**

### Clk2xIn Requirements

The input clock timing requirements are illustrated in Figure 10.6. The system designer does not need to be explicitly aware of the timing relationship between Clk2xIn and SysClk. Note that SysClk is driven even during the Reset period, (regardless of the Tri-state mode), as long as Clk2xIn is provided.




4000 drw 75

**Figure 10.6. R3051 Family Clocking**

## INTRODUCTION

This chapter discusses particular features of the R3051 included to facilitate debugging of R3051-based systems. Although many of these features are intended to be used by an In-Circuit Emulator, the features documented in this chapter are also useful in environments which use a logic analyzer or similar tool.

## OVERVIEW OF FEATURES

The features described in this chapter include:

- The ability of the processor to display internal instruction addresses on its A/D bus during idle bus cycles. This mode facilitates the trace of instruction streams operating out of the internal cache.
- The ability of the processor to have instruction cache misses forced, thus allowing control to be brought to the bus interface. This mode is useful for breaking into infinite loops, and is also useful for "jamming" an alternate instruction stream (such as a debug monitor) into the instruction stream.

Other features useful in debug and In-Circuit Emulation are contained in the definition of the DIAG pins, described in an earlier chapter.

Note that the features described in this chapter are performed on the "Reserved" pins of the processor. Thus, future variants of the chip may or may not incorporate these features in the same fashion. The features described in this chapter are intended for initial debug, rather than continued use in a production system.

## DEBUG MODE ACTIVATION

Debug mode in the R3051 family is activated by driving the Reserved(2) pin high. This mode can be selected any time that the part is running, or may be selected while the part is being reset. Again, it is not recommended that logic driving Reserved(2) be placed on the production board, since future variants of the product may use this signal for a different function.

## ADDRESS DISPLAY

Activating the debug mode forces the CPU to display Instruction stream addresses on its A/D bus during idle bus cycles. Refer to figure 11.1 regarding the timing relationship between instruction initiation in the on-chip cache and the output address. Note that the address is driven out, but ALE is not asserted. This is to reduce the impact of this mode on system designs which may use the initiation of ALE to start a state machine to process the bus cycle. Instead of ALE, external logic should use the rising edge of $\overline{\text{SysClk}}$ to latch the current contents of the address bus.

The address displayed is determined by capturing the low order address bits used to address the instruction cache, and then capturing the TAG response from the cache one-half clock cycle later. These address lines are concatenated, and presented as follows (Note AddrLo(1:0) will be '00' in all Instruction Cache cycles):

- A/D(31:11) displays TAG(31:11)
- A/D(10:4) displays AddrLo(10:4)
- A/D(3:2) displays AddrLo(12:11)
- A/D(1:0) is reserved for future use.
- Addr(3:2) displays AddrLo(3:2)

This mode is intended to allow gross, rather than fine, instruction trace. Specifically, branches taken while a write or DMA operation occurs may not be displayed, and there is no indication that an exception has occurred (and thus that initiated instructions have been aborted). Additionally, erroneous addresses may be presented in cycles where internal processor stalls occur, such as those for integer multiply/divide interlocks or μTLB misses.

Finally, note that the cycle immediately before a read may contain an erroneous address, and the cycle immediately after a read or write may not produce the address with appropriate timing. It is recommended that these cycles be ignored when tracing execution.

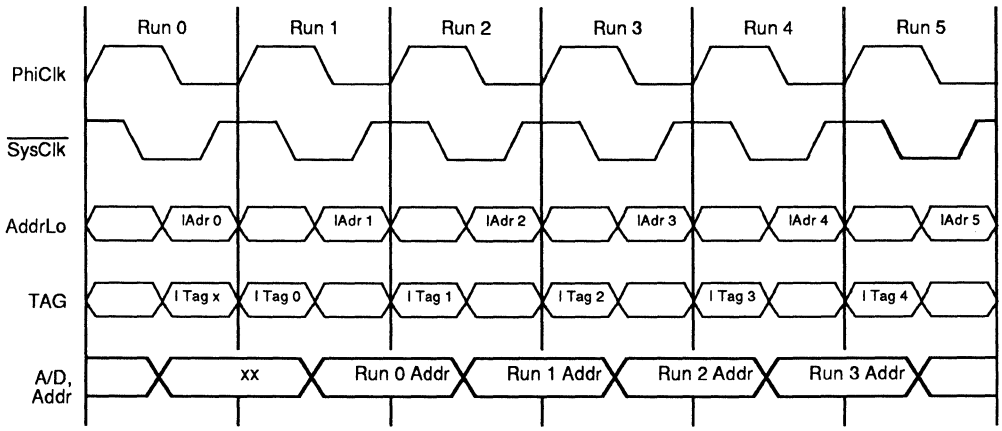



**Figure 11.1. R3051 Debug Mode Instruction Address Display**

## FORCING INSTRUCTION CACHE MISSES

Another feature for debugging is the ability to force an instruction cache miss from an external signal pin. As with debug mode itself, this mode is not intended for use in a production environment.

Forcing an instruction cache miss is a relatively simple operation with the R3051 family. With the device in debug mode (Reserved(2) high), drive Reserved(1) high, to be sampled on a falling edge of SysClk. This will force the next instruction reference to "miss" in the cache, forcing a read operation to the bus. Figure 11.2 illustrates a "jam" operation.

When jamming the instruction cache, a couple of things must be considered:

- The "Jam" input is sampled relative to the falling edge of SysClk. However, IDT does not guarantee the setup and hold time parameters for this signal—it is recommended that a relatively conservative design be used here, since the set-up and hold time of this input are probably slightly larger than the parameters for other inputs.
- Due to the possibility of other bus activities (such as writes), the "Jam" input should be asserted at least until a read is detected on the bus.
- The Jam input does not affect the value of the Valid bit written into the cache on cache line refill. However, it is recommended that the Jam input be negated prior to the Acknowledge of the read (either implicit, by RdCEn, or explicit, by Ack), to avoid unwanted subsequent miss cycles.
- If an instruction other than the target of the read is forced onto the A/D bus for the read, it is the responsibility of that debug monitor to use software cache operations to fix-up the internal instruction cache before resuming normal execution.

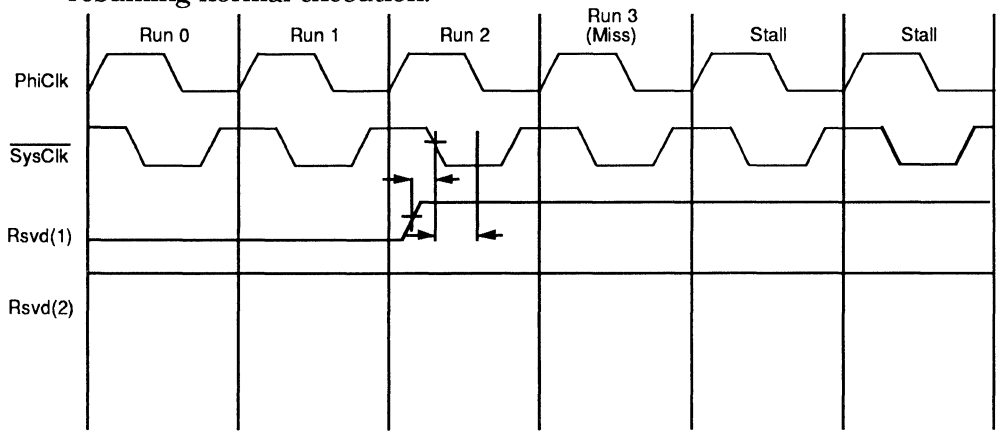



**Figure 11.2. Forcing an Instruction Cache Miss in Debug Mode**

Integrated Device Technology

Integrated Device Technology
3236 Scott Blvd.
Santa Clara, CA 95054-3090
(408) 727-6116
FAX: (408) 492-8674



MAN-RISC-00051