IDT79R3041™

Integrated RISController™ for Low-Cost Systems

Hardware User's Manual

July 1, 1995

Revision 1.12

Integrated Device Technology, Inc.

ABOUT THIS MANUAL

This version 1.12 of the IDT79R3041 Hardware User's Manual contains typographic corrections since the publication of version 1.11 (May of 1993).

This manual provides a qualitative description of the functional operation of the IDT79R3041 integrated RISController™.

A quantitative description of the processor electrical interface is provided in the data sheet for this product. Also included in the data sheet is the mechanical description 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.

TABLE OF CONTENTS

List of Figures

List of Tables

INTRODUCTION

The IDT R30xx family is a series of high-performance 32-bit microprocessors featuring a high-level of integration, and targeted to high-performance yet cost sensitive embedded processing applications. The R30xx 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 R30xx family is able to offer 35 MIPS of integer performance at 40 MHz without requiring external SRAM or caches.

Further, the R30xx family brings dramatic power reduction to these embedded applications, allowing the use of low-cost packaging. Thus, the R30xx family allows customer applications to bring maximum performance at minimum cost.

The R3041 extends the range of price/performance achievable with the R30xx family, by dramatically lowering the cost of using the MIPS architecture. The R3041 has been designed to achieve minimal system and components cost, yet maintain the high-performance inherent in the MIPS architecture. The R3041 also maintains pin and software compatibility with the R3051 and R3081.

FEATURES

- Instruction set compatible with IDT 79R3000A and R30xx family RISC CPUs
- High level of integration minimizes system cost
  - — RISC CPU
  - — Multiply/divide unit
  - — Instruction Cache
  - — Data Cache
  - — Programmable bus interface
  - — Programmable port width support
- Dhrystone 2.1 24 MIPS performance
- On-chip 24-bit Timer
- Low cost 84-pin PLCC packaging/100-pin TQFP
- On-chip instruction and data caches
  - — 2kB of Instruction Cache
  - — 512B of Data Cache
- Flexible bus interface allows simple, low cost designs
  - — Superset Pin compatible with R3051
  - — Adds programmable port width interface (8-, 16-, or 32-bit memory sub-regions)
  - — Adds programmable bus interface timing support (Extended address hold, Bus turn around time, read/write masks)
- Single, double-frequency clock input
- 16-33 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
- Pin and Software Compatible family includes R3041, R3051, R3052, R3071 and R3081

DEVICE OVERVIEW

The R30xx family offers a variety of price/performance features in a pin-compatible, software compatible family. Table 1.1 provides an overview of the current members of the R30xx family. Note that the R3051, R3052, and R3081 are also available in pin-compatible versions that include a full-function

Device	Instr. Name	Data Cache	Freq. Cache	MMU (MHz)	Floating Option	Bus Point Options
R3041	2kB	512B	16-25	No	Software Emulation	8-, 16-, and 32-bit port width support Programmable timing support
R3051	4kB	2kB	20-40	"E" Version	Software Emulation	32-bit Mux'ed Address/Data
R3052	8kB	2kB	20-40	"E" Version	Software Emulation	32-bit Mux'ed Address/Data
R3071	16kB 0r 8kB	4kB or 8kB	33-50	"E" Version	Software Emulation	1/2 frequency bus option
R3081	16kB 0r 8kB	4kB or 8kB	20-50	"E" Version	On-chip Hardware	1/2 frequency bus option

Table 1.1. Pin-, Socket-, and Software- Compatible R30xx Family

memory management unit, including 64-entry TLB. The R3051/2 and R3081 are described in separate manuals and data sheets.

Figure 1.1 shows a block level representation of the functional units within the R3041. The R3041 could be viewed as the embodiment of a discrete solution built around the R3000A. 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 R30xx family implements the MIPS-I ISA. In fact, the execution engine of the R3041 is the same as the execution engine of the R3000A. Thus, the R3041 is binary compatible with those CPU engines, as well as compatible with other members of the R30xx family.

System Control Co-Processor

The R3041 also integrates on-chip a System Control Co-processor, CP0. CP0 manages the exception handling capability of the R3041, the virtual to physical address mapping of the R3041, and the programmable bus interface capabilities of the R3041. These topics are discussed in subsequent chapters.

The R3041 does not include the optional TLB found in other members of the R30xx family, but instead performs the same virtual to physical address mapping of the base versions of the R30xx family. These devices still support distinct kernel and user mode operation, but do not require page management software or an on-chip TLB, leading to a simpler software model and a lower-cost processor.

Clock Generator Unit

The R3041 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 R3000A based applications.

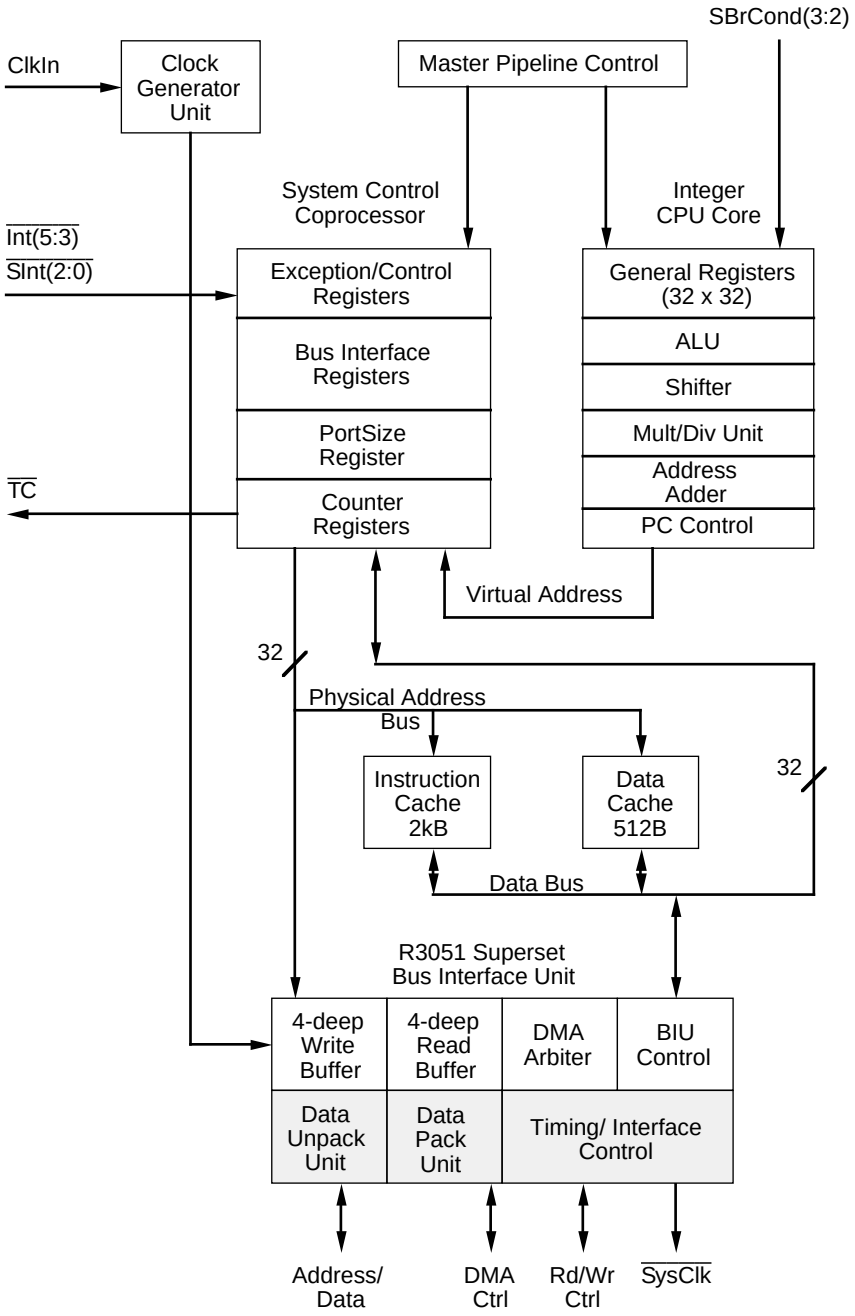


Figure 1.1. Block Diagram

Instruction Cache

The R3041 integrates 2kB of on-chip Instruction Cache, organized with a line size of 16 bytes (four 32-bit entries). This relatively large cache substantially contributes to the performance inherent in the R3041, and allows systems based on the R3041 to achieve high-performance even from low-cost memory systems. 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 and physical tags (rather than virtual addresses or tags), and thus does not require flushing on context switch.

Data Cache

The R3041 incorporates an on-chip data cache of 512B, organized as a line size of 4 bytes (one word). This relatively large data cache contributes substantially to the performance inherent in the R30xx 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 R30xx 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 R30xx 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 A/D bus, and simple handshake signals to process CPU read and write requests. In addition to the read and write interface, the R3041 incorporates a DMA arbiter, to allow an external master to control the external bus.

The R3041 augments the basic R3051 bus interface capability by adding the ability to directly interface with varying memory port widths, for instructions or data. Thus, the R3041 can be used in a system with an 8-bit boot PROM, 16-bit font cartridges, and 32-bit page buffer, transparently to software, and without requiring external data packing, rotation, or unpacking.

In addition, the R3041 incorporates the ability to change some of the interface timing of the bus. These features can be used to eliminate external data buffers, and take advantage of lower speed (lower cost) interface components.

The R3041 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. During main memory writes, the R3041 can break a large datum (e.g. 32-bit word) into a series of smaller transactions (e.g. bytes), according to the width of the memory port being written. This operation is transparent to the software which initiated the store, insuring that the same software can run in true 32-bit memory systems.

The R30xx 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, static 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 R30xx 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.

In addition, the R3041 can perform on-chip data packing when performing large datum reads (e.g. quad words) from narrower memory systems (e.g. 16-bits). Once again, this operation is transparent to the actual software, simplifying migration of software to higher performance (true 32-bit) systems, and simplifying field upgrades to wider memory. Since this capability works for either instruction or data reads, using 8-, 16-, or 32-bit boot PROMs is easily supported by the R3041.

SYSTEM USAGE

The IDT R30xx 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. Embedded systems may also optionally contain static RAMs.

Figure 1.2 shows some of the flexibility inherent in the R3041. In this example system, which is typical of a laser printer, a 32-bit PROM interface is used due to the size of the PDL interpreter. Other embedded systems could optionally use an 8-bit or a 16-bit PROM interface. A 16-bit font cartridge interface is provided for add in cards and a 16-bit page buffer is used for low cost. In this example, a field or manufacturing upgrade to a 32-bit page buffer is supported by the boot software and DRAM controller. Such a system features a very low entry price, with a range of field upgrade options including the ability to upgrade to a more powerful member of the R30xx family.

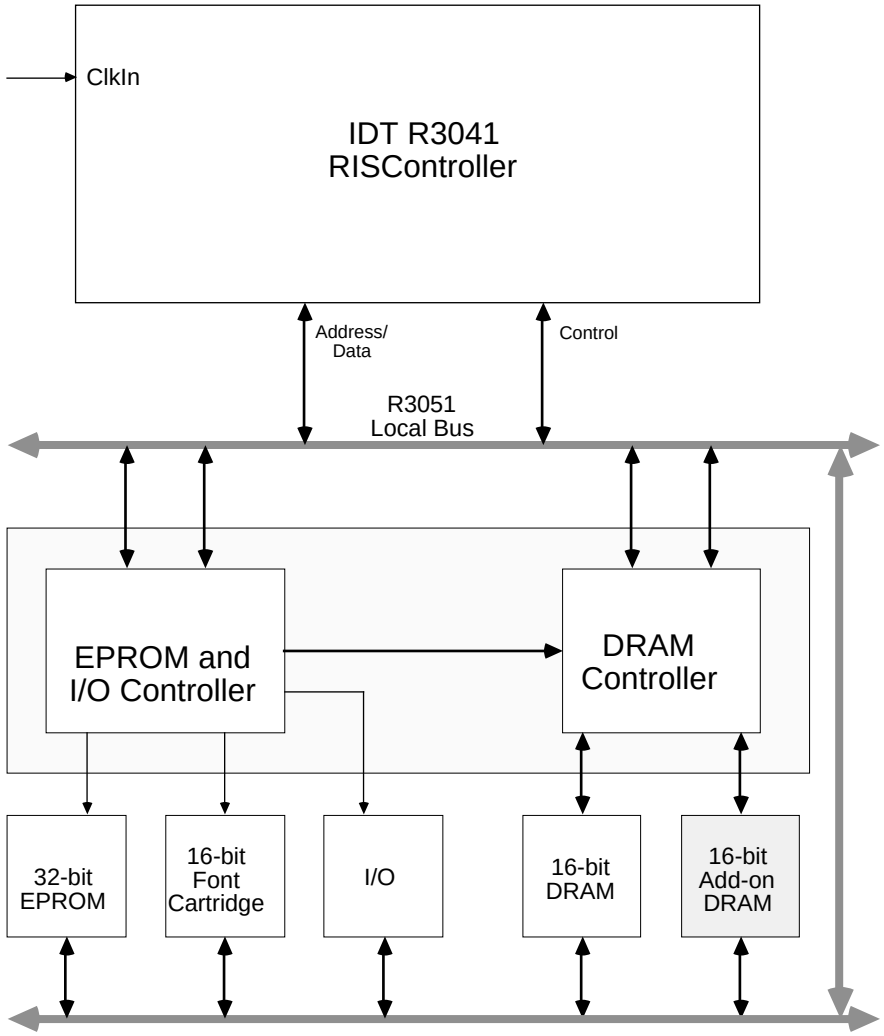


Figure 1.2. Typical R3041 System

DEVELOPMENT SUPPORT

The IDT R30xx 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, sub-system modules, and shrink wrap operating systems. IDT's development support program, called "Advantage IDT", insures the availability of all the tools required to rapidly bring an R30xx-based system rapidly to market.

The R3071 and R3081 are pin and software compatible with many other family members, allowing the system designer to use a single toolchain and methodology for multiple system development efforts.

Figure 1.2 is an overview of the system development process typically used when developing R30xx family applications. The R30xx family is supported in all phases of project development. These tools allow timely, parallel development of hardware and software for R30xx family based applications, and include tools such as:

- Optimizing compilers from MIPS, the acknowledged leader in optimizing compiler technology. The compilers are available in both native and cross environments.
- Cross development tools, available in a variety of development environments and from a number of vendors.
- The high-performance IDT floating point library software, including transcendental functions and IEEE compliant exception handlers.
- IDT Evaluation systems, which includes RAM, EPROM, I/O, and the IDT PROM Monitor.
- IDT Adobe Reference Printer systems, which directly drive low-cost print engines, and run PostScript™ software from Adobe.
- IDT/sim, which implements a full prom monitor (diagnostics, remote debug support, peek/poke, etc.).
- IDT/kit, which implements a run-time support package for R3051 family systems.
- In-circuit Emulator equipment.

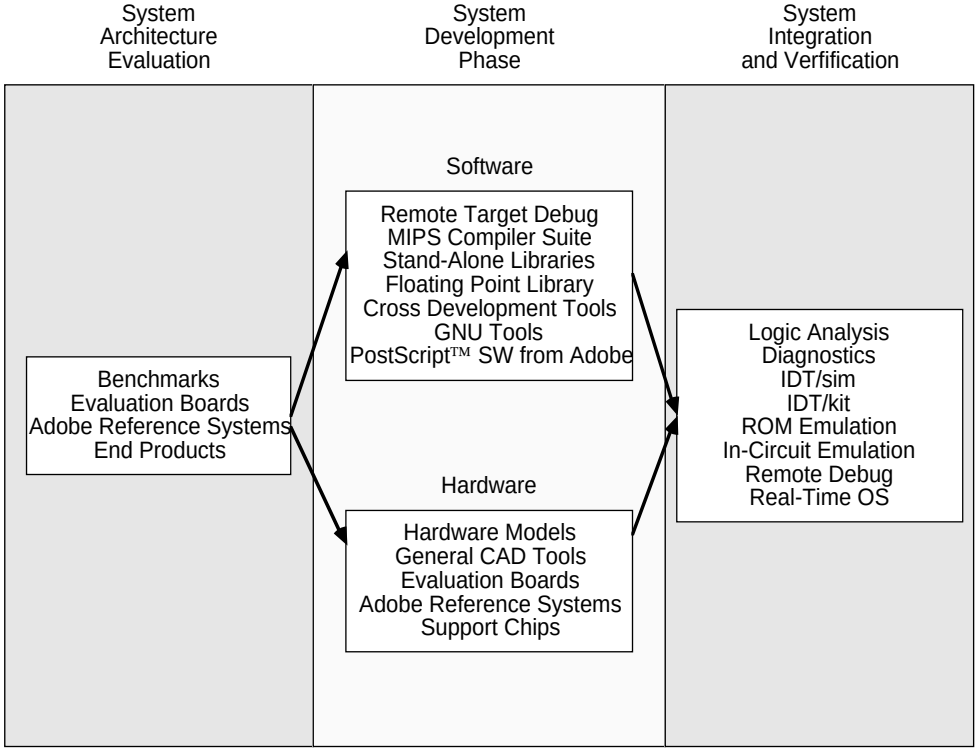


Figure 1.3. Development Support

PERFORMANCE OVERVIEW

The R30xx 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 a 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 R3041 achieves over 16 MIPS performance when operating out of cache.
- Large on-chip caches. The R30xx family contains caches which are substantially larger than those on the majority of embedded microprocessors. These large caches minimize the number of bus transactions required, and allow the R30xx family to achieve actual sustained performance very close to its peak execution rate, even with low cost memory systems.
- Autonomous multiply and divide operations. The R30xx family features an on-chip integer multiplier/divide unit which is separate from the other ALU. This allows the R3041 to perform multiply or divide operations in parallel with other integer operations, using a single multiply or divide instruction rather than with "step" operations.
- Integrated write buffer. The R3041 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 R3041 enables the system designer to utilize page, static or nibble mode RAMs when performing read operations to minimize the main memory read penalty and increase the effective cache hit rates.

The performance differences among the various R30xx family members depends on the application software and the design of the memory system. Different family members feature different cache sizes, and the R3081 features a hardware floating point accelerator. Since all these devices can be used in a pin and software compatible fashion, the system designer has maximum freedom in trading between performance and cost.

INSTRUCTION SET ARCHITECTURE

CHAPTER 2

INTRODUCTION

The IDT R30xx 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 R30xx 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 R30xx family, and discusses the programmers' model for this device. Further detail is available in the book “mips RISC Architecture”, available from IDT.

The R3041 is software compatible with the base versions of the R30xx family. However, to reduce system cost, the TLB functions present in the "E" versions are not available in the R3041; instead, the R3041 features increased control of the system interface, including the ability to control timing relationships of the bus interface, and the ability to directly interface with memory systems of varying widths.

PROCESSOR FEATURES OVERVIEW

The R30xx 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 R30xx 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 R30xx 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 R3041 is compatible with the base versions of the IDT R30xx family, which do not utilize a TLB, but perform fixed segment-based mapping of the virtual space to physical addresses. In addition, the R3041 allows kernel software to configure the "width" of regions of the memory space, to allow direct interface to memory systems of 8, 16, or 32 bits of data width.

CPU REGISTERS OVERVIEW

The IDT R30xx family provides 32 general purpose 32-bit registers, an internal 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.

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.

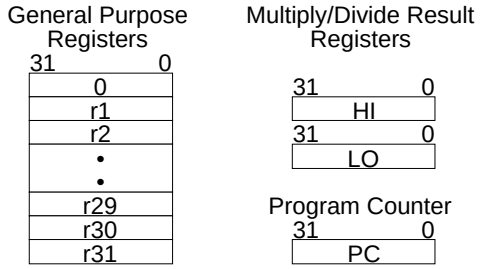


Figure 2.1. CPU Registers

INSTRUCTION SET OVERVIEW

All R30xx 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.

The R30xx 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.

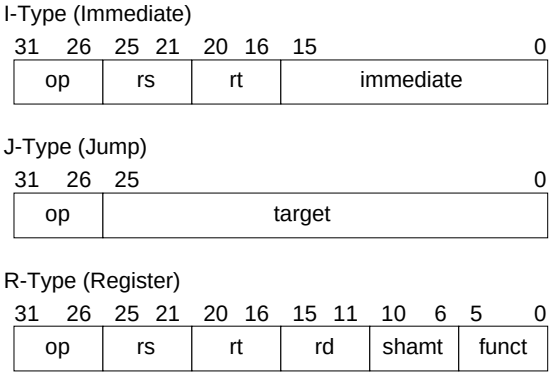
The Jump and Link instructions save a return address in Register r31. These are typically used as subroutine calls, where the subroutine return address is stored into r31 during the call operation.

- **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 R30xx family, the System Control Co-Processor (CP0) contains registers which are used in memory management, system interface control, cache control, and exception handling.

  Additionally, the R30xx family implements 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.



where:

op	is a 6-bit operation code
rs	is a 5-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

Figure 2.2. Instruction Encoding

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 Word 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

†These instructions are not valid with the R3041, which does not include the TLB.

Table 2.1. Instruction Set Mnemonics

Table 2.1 lists the instruction set mnemonics of the R30xx 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.

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 MIPS-I architecture 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 11) 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 shows the ordering of bytes within words and the ordering of words within multiple word structures for the big-endian and little-endian conventions.

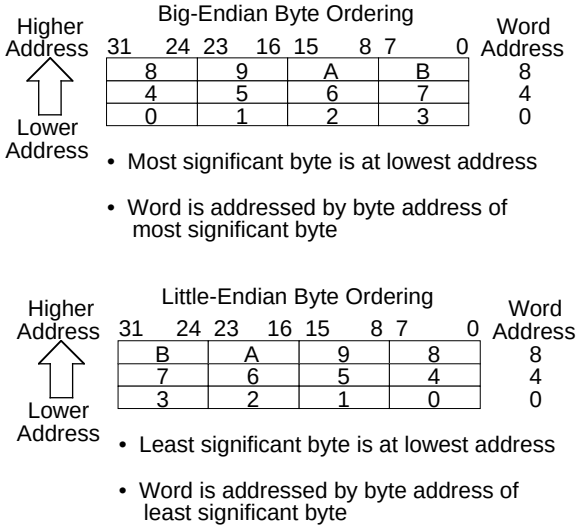


Figure 2.3. Byte Ordering Conventions

The R30xx family uses byte addressing for all accesses, including half-word and word. The MIPS architecture 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.

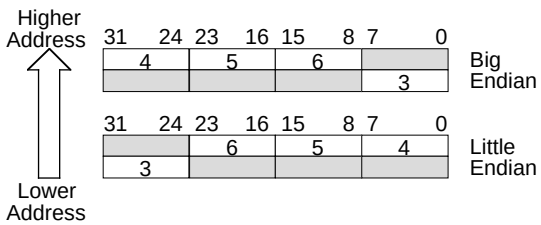


Figure 2.4. Unaligned Words

CPU General Registers

The R30xx 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 as 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.

CP0 Special Registers

In addition to the general CPU registers, the R30xx 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 (System Control), and Chapter 6 (Exception Handling). Note that the MIPS architecture allows CP0 to vary by implementation; the R3041 contains some new CP0 registers not found in other R30xx family members; however, their definition is such that it still remains possible to use a single binary program across all family members.

Number	Mnemonic	Description
0	Reserved(1)	
1	Reserved(1)	
2	BusCtrl(1)	Bus Timing and Interface Control
3	Config(3)	Cache Usage Configuration
4	Reserved(1)	
5-7	Reserved	
8	BadVAddr	Bad Virtual Address
9	Count(2)	Timer Counter Register
10	PortSize(1)	Memory Sub-Region Port Width Control
11	Compare(2)	Timer Compare Register
12	SR	Status Register
13	Cause	Cause of Last Exception
14	EPC	Exception Program Counter
15	PRId	Processor Revision Identifier
16-31	Reserved	

Notes:
1: This register is used in Extended Architecture CPUs to control the TLB and virtual memory system. In the "E" versions, register $2 is "TLB EntryLo", and register $10 is "TLB EntryHi".
2: This register is reserved in other family members.
3: This register has a different meaning in other family members.

Table 2.2. R3041 CP0 Registers

Operating Modes

The R30xx 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 R3041, CP0) from access or modification by the user task.

Finally, the R30xx 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 4 discusses the memory management facilities of the processor.

Pipeline Architecture

The IDT R30xx family uses the same basic pipeline structure as that implemented in the R3000A. 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.

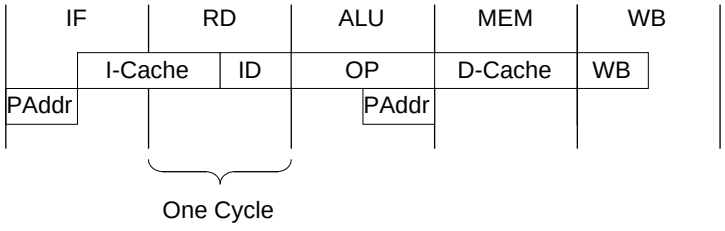


Figure 2.5. 5-Stage Pipeline

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.

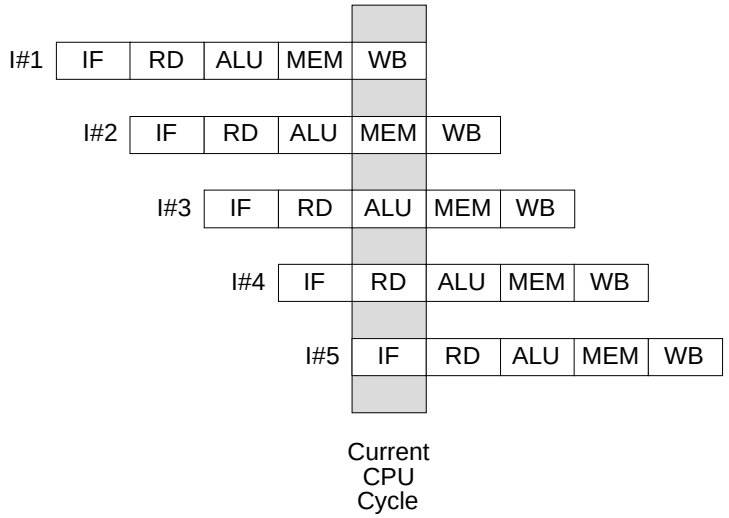


Figure 2.6. 5-Instructions per Clock Cycle

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 R3041, 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 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.

- Jump and Branch instructions have a delay of one cycle before the program

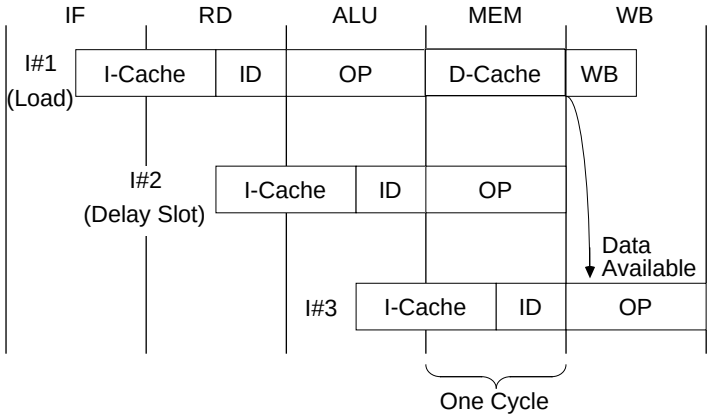


Figure 2.7. Load Delay

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.

The R3041 continues execution, despite the delay in the operation. Thus,

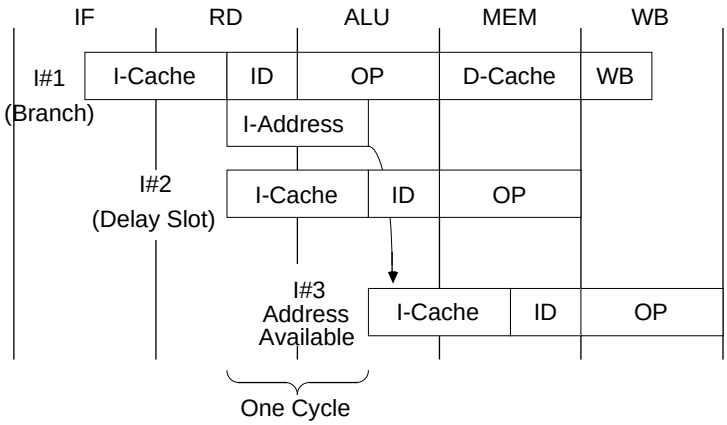


Figure 2.8. Branch Delay

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.

Note that there may also be latencies associated with changes to various of the CP0 registers; for example, changing the bus interface control register may require multiple cycles before the change is actually reflected in the chip interface.

Rather than include extensive pipeline control logic, the MIPS-I instruction set

gives responsibility for dealing with “delay slots” to software. Thus, peephole optimizations (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 6 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 R3041 pipeline is designed to properly manage exceptional events.

INSTRUCTION SET SUMMARY

This section provides an overview of the R30xx 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 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 to this rule is that for the target register for the "load word left" and "load word right" instructions may be specified as the same register used as the destination of a load instruction that immediately precedes it.

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.

Big-Endian (32-bit memory system)

Size	CPU Core VAdrLo(1)	CPU Core VAdrLo(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
Half-word	0	0	Yes	Yes	No	No
Half-word	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 (32-bit memory system)

Size	VAdrLo(1)	VAdrLo(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
Half-word	0	0	No	No	Yes	Yes
Half-word	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

Table 2.3 (a). Byte Addressing in Load/Store Operations (32-bit memory)

Big-Endian (16-bit memory system)

Size	CPU Core VAdrLo(1)	CPU Core VAdrLo(0)	First Transfer BE16(1) Data(31:24)	First Transfer BE16(0) Data(23:16)	Second Transfer BE16(1) Data(31:24)	Second Transfer BE16(0) Data(23:16)
Word	0	0	Yes	Yes	Yes	Yes
Tri-Byte	0	0	Yes	Yes	Yes	No
Tri-Byte	0	1	No	Yes	Yes	Yes
Half-word	0	0	Yes	Yes	N/A	N/A-B
Half-word	1	0	Yes	Yes	N/A	N/A
Byte	0	0	Yes	No	N/A	N/A
Byte	0	1	No	Yes	N/A	N/A
Byte	1	0	Yes	No	N/A	N/A
Byte	1	1	No	Yes	N/A	N/A

Little-Endian (16-bit memory system)

Size	CPU Core VAdrLo(1)	CPU Core VAdrLo(0)	First Transfer BE16(1) Data(15:8)	First Transfer BE16(0) Data(7:0)	Second Transfer BE16(1) Data(15:8)	Second Transfer BE16(0) Data(7:0)
Word	0	0	Yes	Yes	Yes	Yes
Tri-Byte	0	0	Yes	Yes	No	Yes
Tri-Byte	0	1	Yes	No	Yes	Yes
Half-word	0	0	Yes	Yes	N/A	N/A
Half-word	1	0	Yes	Yes	N/A	N/A
Byte	0	0	No	Yes	N/A	N/A
Byte	0	1	Yes	No	N/A	N/A
Byte	1	0	No	Yes	N/A	N/A
Byte	1	1	Yes	No	N/A	N/A

Table 2.3 (b). Byte Addressing in Load/Store Operations (16-bit memory)

Note that the size of the operand requested by the load instruction is independent of the memory width of the addressed memory. Thus, if the actual size of the datum is 32-bits, software can safely use a load or store word instruction, even if the addressed memory is actually only 8- or 16-bits wide. The bus interface unit will interact with CP0 to determine the width of the addressed memory, and will, if necessary, perform multiple datum transfers to satisfy a single load or store instruction.

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 (a, b). Note that certain combinations of access type and low-order address bits can never occur: only the combinations shown in Table 2.3(a, b) are permissible. The R30xx family indicates which bytes are being accessed by the byte-enable ($\overline{BE}$) bus; the R3041 adds the $\overline{BE16}$ bus to simplify the interface to 16-bit wide memory subsystems.

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

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

Table 2.4. Load and Store Instructions

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* 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* 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* 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. Place result in register *rt.*
Set on Less Than Unsigned Immediate	SLTIU *rt, rs, immediate* 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* Zero-extend 16-bit *immediate*, AND with contents of register *rs* and place result in register *rt.*
OR Immediate	ORI *rt, rs, immediate* Zero-extend 16-bit *immediate*, OR with contents of register *rs* and place result in register *rt.*
Exclusive OR Immediate	XORI *rt, rs, immediate* Zero-extend 16-bit *immediate*, exclusive OR with contents of register *rs* and place result in register *rt.*
Load Upper Immediate	LUI *rt, immediate* Shift 16-bit *immediate* left 16 bits. Set least significant 16 bits of word to zeroes. Store result in register *rt.*

Table 2.5a. ALU Immediate Operations

Instruction	Format and Description
Add	ADD *rd, rs, rt* 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* 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* 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* 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* Compare contents of register *rt* to register *rs* (as signed 32-bit integers). If register *rs* is less than *rt*, result = 1; otherwise, result = 0.
Set on Less Than Unsigned	SLTU *rd, rs, rt* 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* Bit-wise AND contents of registers *rs* and *rt* and place result in register *rd*.
OR	OR *rd, rs, rt* Bit-wise OR contents of registers *rs* and *rt* and place result in register *rd*.
Exclusive OR	XOR *rd, rs, rt* Bit-wise Exclusive OR contents of registers *rs* and *rt* and place result in register *rd*.
NOR	NOR *rd, rs, rt* Bit-wise NOR contents of registers *rs* and *rt* and place result in register *rd*.

Table 2.5b. Three Operand Register-Type Operations

Instruction	Format and Description
Shift Left Logical	SLL *rd, rt, shamt* 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* 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* 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* 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* 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* 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*.

Table 2.5c. Shift Operations

Instruction	Format and Description
Multiply	MULT *rs*, *rt* 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* Multiply contents of registers *rs* and *rt* as unsigned values. Place 64-bit result in special registers HI/LO
Divide	DIV *rs*, *rt* 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* 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* Move contents of special register HI to register *rd*.
Move From LO	MFLO *rd* Move contents of special register LO to register *rd*.
Move To HI	MTHI *rd* Move contents of special register rd to special register HI.
Move To LO	MTLO *rd* Move contents of register rd to special register LO.

Table 2.5d. Multiply and Divide Operations

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, 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 r31.

Table 2.6a summarizes the R30xx family Jump instructions and Table 2.6b summarizes the Branch instructions.

Instruction	Format and Description
Jump	J target 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 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* Jump to address contained in register *rs* with a one instruction delay.
Jump and Link Register	JALR *rs, rd* Jump to address contained in register *rs* with a one instruction delay. Place address of instruction following delay slot in *rd*.

Table 2.6a. Jump Instructions

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* Branch to target address if register *rs* equal to *rt*
Branch on Not Equal	BNE *rs, rt, offset* Branch to target address if register *rs* not equal to *rt.*
Branch on Less than or Equal Zero	BLEZ *rs,offset* Branch to target address if register *rs* less than or equal to 0.
Branch on Greater Than Zero	BGTZ *rs,offset* Branch to target address if register *rs* greater than 0.
Branch on Less Than Zero	BLTZ *rs,offset* Branch to target address if register *rs* less than 0.
Branch on Greater than or Equal Zero	BGEZ rs,*offset* Branch to target address if register *rs* greater than or equal to 0.
Branch on Less Than Zero And Link	BLTZAL *rs, offset* 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* 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.

Table 2.6b. Branch Instructions

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 Initiates system call trap, immediately transferring control to exception handler.
Breakpoint	BREAK Initiates breakpoint trap, immediately transferring control to exception handler.

Table 2.7. Special Instructions

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 R30xx 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 for the R3041 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)* 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)* 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* Move contents of CPU register *rt* into co-processor register *rd* of co-processor unit z.
Move from Co-processor	MFCz *rt,rd* Move contents of co-processor register *rd* from co-processor unit z to CPU register *rt*.
Move Control To Co-processor	CTCz *rt,rd* 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* Move contents of control register *rd* of co-processor unit z into CPU register *rt*.
Co-processor Operation	COPz *cofun* 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* 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* 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.

Table 2.8. Co-Processor Operations

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, bus programmability, timer, and exception handling facilities of the processor. Memory management is discussed in Chapter 4; bus programmability and timer features are described in Chapter 5; and exception handling is covered in detail in Chapter 6.

Table 2.9 summarizes the instructions available to work with CP0.

Instruction	Format and Description
Move To CP0	MTC0 *rt, rd* Store contents of CPU register *rt* into register *rd* of CP0. This follows the convention of store operations.
Move From CP0	MFC0 *rt, rd* Load CPU register *rt* with contents of CP0 register *rd*.
Read Indexed TLB Entry	TLBR† Load *EntryHi* and *EntryLo* registers with TLB entry pointed at by *Index* register.
Write Indexed TLB Entry	TLBWI† Load TLB entry pointed at by *Index* register with contents of *EntryHi* and *EntryLo* registers.
Write Random TLB Entry	TLBWR† Load TLB entry pointed at by *Random* register with contents of *EntryHi* and *EntryLo* registers.
Probe TLB for Matching Entry	TLBP† 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 Restore previous interrupt mask and mode bits of *status* register into current status bits. Restore old status bits into previous status bits.

†These operations are undefined/reserved in the R3041, which does not include an on-chip TLB.

Table 2.9. System Control Co-Processor (CP0) Operations

R30XX FAMILY OPCODE ENCODING

Table 2.10 shows the opcode encoding for the MIPS architecture.

	28..26			**OPCODE**				
31..29	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	†	†	†	†

	2..0			**SPECIAL**				
5..3	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	†	†	†	†	†	†	†	†

	18..16			**BCOND**				
20..19	0	1	2	3	4	5	6	7
0	BLTZ	BGEZ						
1								
2	BLTZAL	BGEZAL						
3								
4								

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

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

	2..0			**CP0**				
4..3	0	1	2	3	4	5	6	7
0		TLBR	TLBWI				TLBWR	
1	TLBP							
2	RFE							
3								

Table 2.10. Opcode Encoding

INTRODUCTION

The R30xx 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 R3041. However, as these caches are managed by the R3041 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 longer 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 or virtual address space, 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.

To minimize cost while maintaining high-performance, the R30xx family, including the R3041, integrate a reasonable amount of cache internal to the chip, eliminating the cost and complexity of external caches.

R3041 CACHE ORGANIZATION

There are a number of algorithms possible for managing a processor cache. This section describes the cache organization of the R3041.

Basic Cache Operation

When the processor makes a reference, its 32-bit internal physical 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.

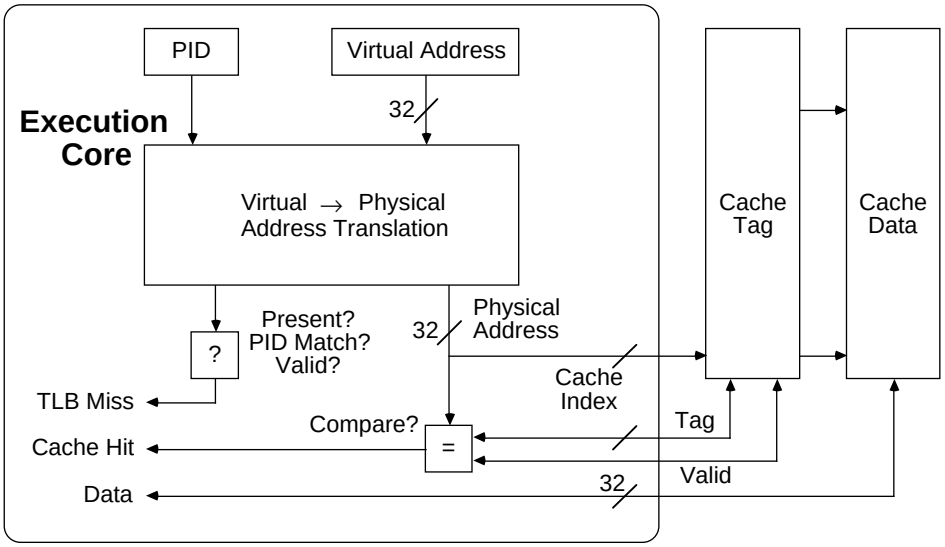


Figure 3.1. Cache Line Selection

To maximize performance, the R3041 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 R3041 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.

This organization, coupled with the relatively large cache sizes resident on the R3041, 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 "alias" 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 virtual to physical address translation in the processor pipeline, physical cache addressing is used with no inherent speed penalty.

Write Policy

The R3041 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 R3041 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 R3041 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. If the cache read results in a hit, the memory interface will see the full word write, rather than the partial word. This allows the designer to observe the actual activity in the on-chip caches.

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 R3041, 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 R3041 can support either true burst accesses or can utilize a simpler, slower memory protocol for quad word reads. Also note that the variable bus sizing capability of the R3041 means that block reads can occur from 8- or 16-bit memory systems. This includes the case of instruction fetches; the bus interface unit will automatically translate the block read protocol into a larger number of sub-word reads, depending on the memory width programmed for the target memory location.

Finally, note that the R3041 performs "streaming" during instruction cache refill. That is, the processor will simultaneously refill the instruction cache and execute the incoming instructions. Streaming contributes an average of 5% of performance.

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-3041; some systems may achieve higher performance through the use of data cache burst refill. No “streaming” occurs on data cache refills.

Summary

The on-chip caches of the R30xx family can be thought of as constructed from discrete devices around the R3000A. Figure 3.2 shows the block diagram of the cache interface for the R3041.

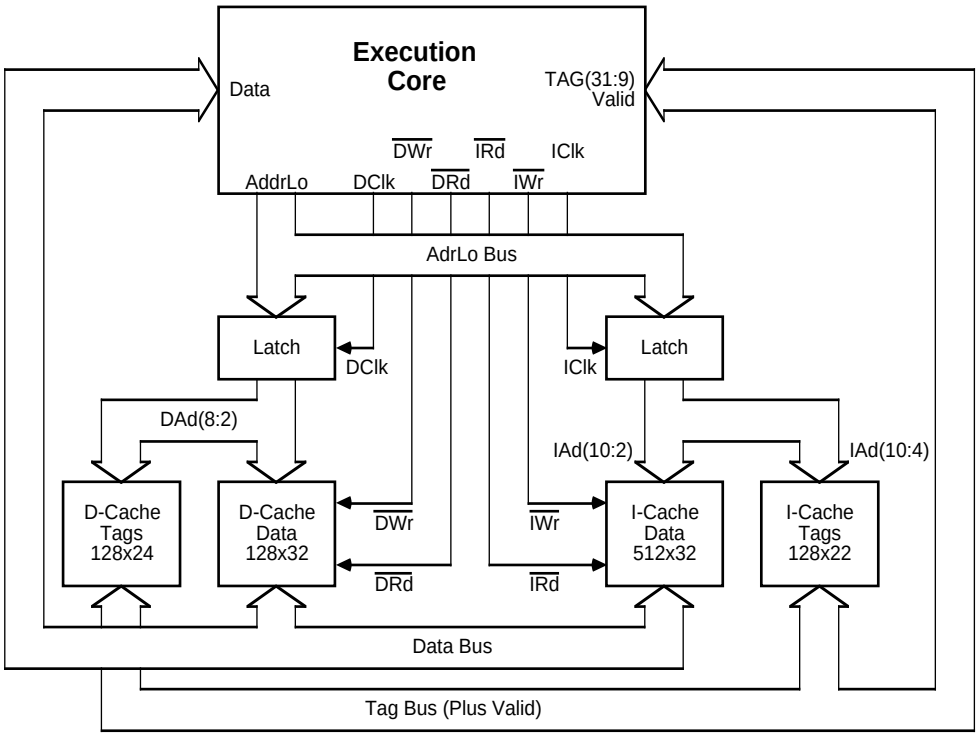


Figure 3.2. R3041 Execution Core and Cache Interface

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/R3500, the R30xx family can access both the instruction and data caches in a single clock cycle, resulting in high 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. Note that data in the cache is organized into 32-bit words, regardless of the width associated with main-memory from which the datum was taken. Thus, cache hits can retrieve a full 32-bits in a single cycle, minimizing the performance impact of the narrower memory system.

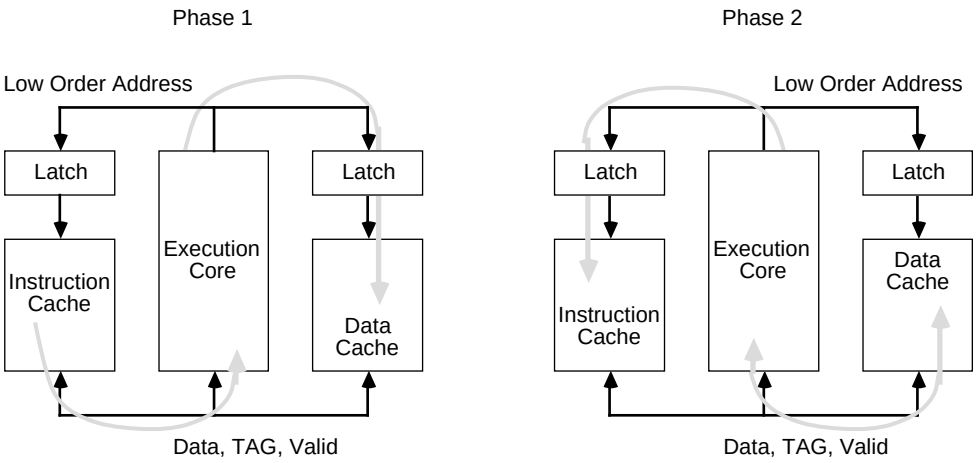


Figure 3.3. Phased Access of Instruction and Data Caches

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 R30xx 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 I/O device status register data 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 R30xx 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). The 5 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 the amount of cache resident on any given R30xx family chip (note that the R3041, R3051, R3052, and R3081 each feature differing amounts of cache on chip). Having software determine the size of the cache at boot time, rather than building static values into the software, allows for maximum flexibility in interchanging various members of the R30xx family, including future devices.

Cache sizing in an R30xx family CPU 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 should:

1: Swap Caches (not needed for D-Cache sizing)
2: Isolate Caches
3: Write a value at location 8000_0000
4: Write a value at location 8000_0200 (8000_0000 + 512B)
   Read location 8000_0000.
   Examine the CM (Cache_Miss) bit of the status register; if it indicates a cache miss, then the cache is 512B; otherwise, the cache is 1kB or larger.
5: Write a value at location 8000_0400 (8000_0000 + 1kB)
   Read location 8000_0000.
   Examine the CM (Cache_Miss) bit of the status register; if it indicates a cache miss, then the cache is 1kB; otherwise, the cache is 2kB or larger.
6. etc...

Of course a more generalized algorithm could be developed to determine the cache size; this may be desirable for compatibility with discrete R3000A/ R3500 systems or other R30xx 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, and smaller cache sizes need to be considered.

Note that this software should operate as uncached. Once this algorithm 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 R30xx family implements physically addressed 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 R30xx family by having the cache isolated, and then writing a partial word quantity into the current data cache. Under these conditions, the CPU 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 128 such writes have been performed (128 in the R3041, more for other R30xx family members).
3: Return the data cache to its normal state by clearing the IsC bit.

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 (128 lines in the R3041, more for other R30xx family members) have been invalidated. Note that treating the R3041 as if it had larger on-chip caches, and flushing/invalidating more than 128 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.

Also, of course it is preferable to use the cache sizing algorithm described earlier to determine the number of lines to be flushed.

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 cacheable 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 R30xx family are key to the inherent performance of the processor. The R30xx 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 R3041 based system.

INTRODUCTION

The R3041 provides the same basic virtual to physical address translation as the rest of the R30xx family base versions (the R3051, R3052, and R3081). These devices 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 of the R30xx family (the R3051E, R3052E, and R3081E) provide a full featured memory management unit (MMU) identical to the MMU structure of the R3000A and R3500. The extended MMU uses an on-chip translation lookaside buffer (TLB) and dedicated registers in CP0 to provide for software management of page tables. There is no Extended Architecture version of the R3041.

This chapter describes the operating states of the processor (kernel and user), and describes the virtual to physical address translation mechanisms provided in the R3041.

VIRTUAL MEMORY IN THE R30XX FAMILY

There are two primary purposes of the memory management capabilities of the R30xx family.

- Various areas of main memory can have individual sets of attributes associated with 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 establishes the rules appropriate for a given virtual address. The R3041 memory manager provides for these mechanisms, without requiring the use of a TLB.
- 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 R30xx family.

Figure 4.1 shows the format of an R30xx 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 4kB pages; in the base versions (and thus in the R3041), only the three highest bits (segment number) are involved in the virtual to physical address translation.

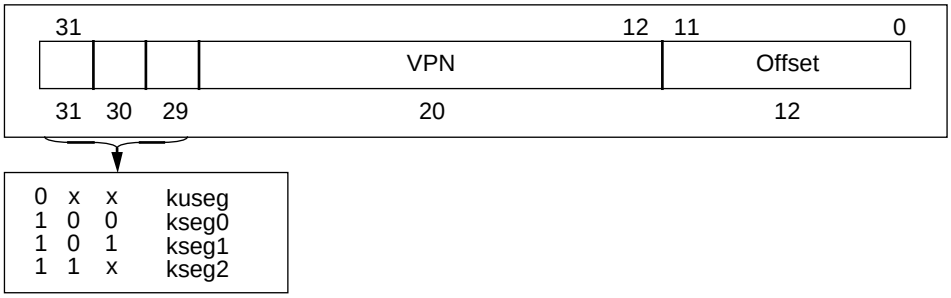


Figure 4.1. Virtual Address Format

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. In addition, the R3041 uses the high-order address bits of the physical address to determine which memory region is being accessed; this information, along with the contents of the CP0 PortSize register, determine the width of the memory system being addressed in a given memory transfer.

PRIVILEGE STATES

The R3041 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 2GB 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 6). Kuseg begins at virtual address 0 and extends linearly for 2GB. This segment is typically used to hold user code and data, and the current user processes.

Also note that the physical address space corresponding to kuseg is independent of the physical address spaces of the various kernel only segments. Thus, systems can be constructed which preclude user tasks from affecting kernel memory. On the other hand, simple systems can, by virtue of the address decode, compress the mapping into a single address region.

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, including the Port Size attributes, is identical with User mode addressing to this segment.
- **kseg0.** Kseg0 is a 512MB segment, beginning at virtual address 0x8000_0000. This segment is always translated to a linear 512MB 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 512MB segment, beginning at virtual address 0xa000_0000. This segment is also translated directly to the 512MB 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 1GB 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 1024MB 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.

Base versions of the R30xx family (including the R3041) are distinguishable from extended versions 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.

The PRId register (described in chapter 6) can be used to distinguish the R3041 (with its variable bus sizing features, among others) from other members of the R30xx family.

R3041 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 2GB 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”, and “01” is translated to “10”, with the remaining 30 bits of the virtual address unchanged.
  Kuseg is broken into 4 equal sub-regions to support the variable width bus interface capability of the R3041. The 2GB of Kuseg is divided into 4 equal 512MB regions (Kuseg[a:d]), whose port widths are indicated in the CP0 Port Size register. Thus, Kuseg can be composed of a mix of memory spaces, of varying widths, independent from the widths of the kernel address space.
- 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 1MB kseg2 physical address space is divided into two equally sized 512MB subregions, whose memory width attributes are controlled by the CP0 PortSize register.
- Virtual addresses in kseg0 and kseg1 are both translated identically to the same physical address region. This 512MB region is subdivided into 8 equal 64MB sub-spaces, whose memory widths are independently selectable in the CP0 Port Size register. This allows the various kernel regions to have varying port widths, independent of kuseg.

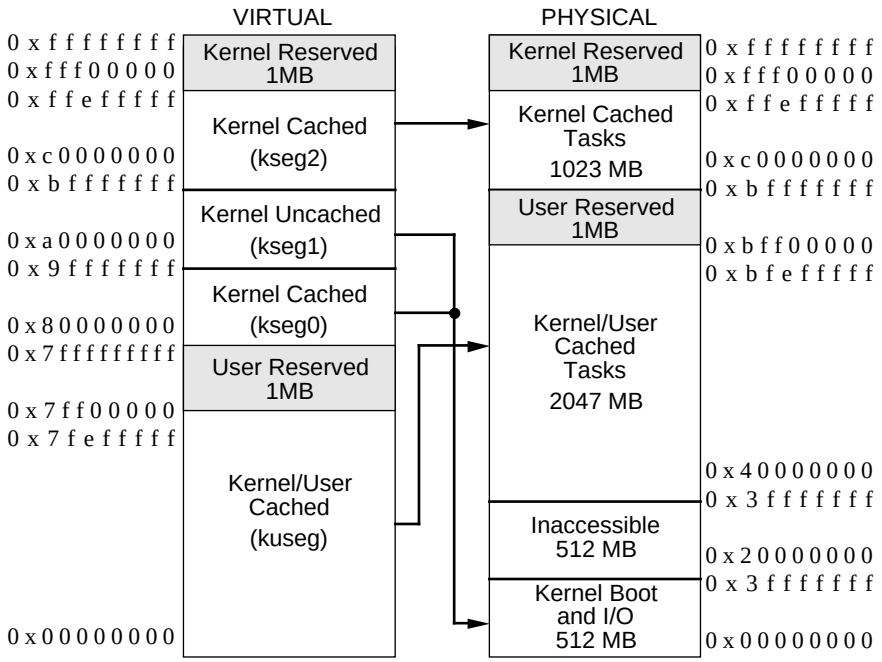


Figure 4.2. Virtual to Physical Address Translation in Base Versions

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.

Note that the reserved areas of the virtual address space shown in figure 4.2 are translated to physical addresses identically with the remainder of their virtual segment; they are indicated as reserved to insure compatibility with future family members which may incorporate on-chip resources in these address spaces.

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 occurred 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

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.

SUMMARY

The R30xx 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. The R3041 only implements the base version address translation, but in addition, subdivides each segment into sub-regions. Each sub-region may be declared, via the CP0 Port Size register, as having either an 8-, 16-, or 32-bit memory interface. The Bus Interface Unit of the R3041 dynamically translates processor core references to the appropriate port width, making the actual software independent of the port width. Both instruction and data fetches can be transferred between memory and the CPU, regardless of the memory port width.

INTRODUCTION

The R3041 bus interface has been designed to minimize system cost by providing a simple, flexible bus interface. In addition, the bus interface has been designed to allow the R3041, R3051, R3052, and R3081 to be easily interchanged in a given design.

To allow the system designer to enjoy maximum flexibility, the bus interface of the R3041 features a number of programmable options. These options are controlled by various registers of the on-chip Co-Processor 0. This chapter describes those registers and their impact on the bus interface.

CO-PROCESSOR 0 BUS INTERFACE CONTROL

Figure 5.1 illustrates the co-processor 0 registers used to control various actions of the bus interface. Note that the MIPS architecture allows the register set of CP0 to vary by implementation; software can easily identify the R3041 (and its CP0 registers) from the R3051 and R3081 by reading the PRId from CP0.

The fields of these registers, and their impact on the bus interface, are described below. Note that software should allow a minimum of 10 instruction cycles for changes to these registers to be reflected in subsequent bus transactions.

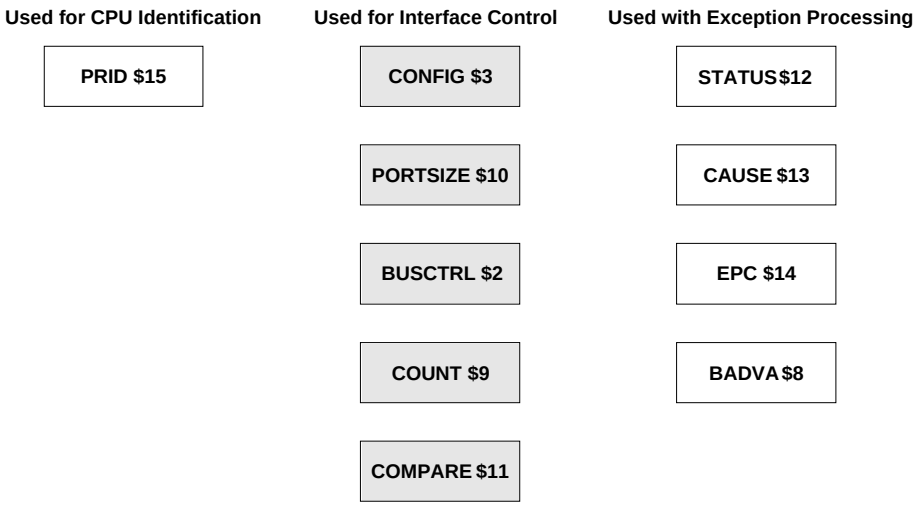


Figure 5.1. R3041 Bus Interface Control Registers

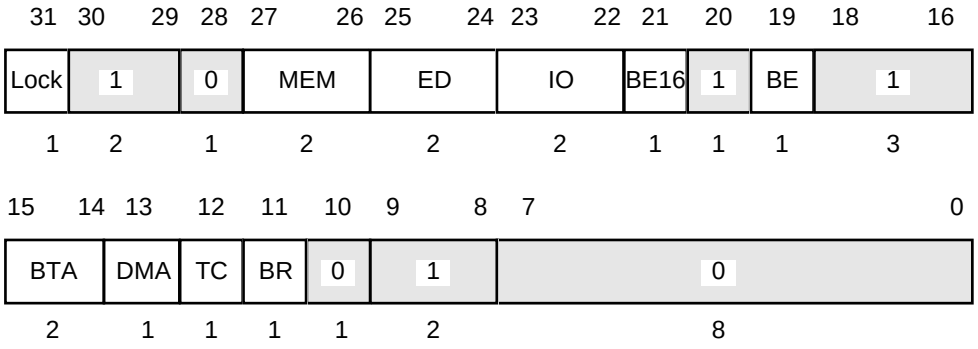
BUS CONTROL REGISTER

The Bus Control register allows the kernel to configure various aspects of the bus interface, simplifying the I/O interface in many systems.

This register controls the use of the $\overline{BE}$(3:0), $\overline{BE16}$(1:0), $\overline{TC}$, and SBrCond(3:2) signals, and also controls the time between back to back transactions.

Figure 5.2 illustrates the various fields of the Bus Control register. The reset defaults for this register have been selected to insure R3051 compatible operation.

The Bus Control register is both readable and writeable.



Lock: Register Write Lock
'1': Reserved: Must be written as '1'
'0': Reserved: Must be written as '0'
MEM: MemStrobe Control
ED: ExtDataEn Control
IO: IOStrobe Control
BE16: BE16 Read Control
BE: BE(3:0) Read Control
BTA: Bus Turn Around Time
DMA: DMA Protocol Control
TC: TC Negation Control
BR: SBrCond(3:2) Control

Figure 5.2. R3041 Bus Control Register

Lock

The lock bit can be used by the kernel to inhibit subsequent write operations to this register, as shown in table 5.1. It is useful in ensuring that operating systems written for other R3000A-based applications, including applications which may run on other R30xx family members, do not inadvertently change the fields of the Bus Control register.

At reset, the register is unlocked (Lock bit is '0'). Thus, the BusCtrl register can be written and re-written as the operating system chooses. Once the Lock bit is written with a '1', subsequent writes to the BusCtrl register will be ignored.

Value	Action
'0'	Leave Unlocked (Default).
'1'	Lock register from future writes.

Table 5.1. R3041 Bus Control Register "Lock" Bit Function

Reserved-High ('1')

This bit is reserved for testing of the R3041. At reset, the bit will be set high ('1'). Writes to the BusCtrl register must maintain these bit fields as high ('1').

Reserved-Low ('0')

These fields are reserved for testing and for future variants of the R3041. At reset, these bit fields are reset ('0'). Writes to the BusCtrl register must maintain these bit fields as low ('0').

MemStrobe Control

These bits control the use of the MemStrobe pin, according to Table 5.2.

Reset initializes this field to '01', which allows the use of MemStrobe on writes.

Value	Action
'00'	MemStrobe remains high on both reads and writes.
'01'	Use MemStrobe on write cycles only (default).
'10'	Use MemStrobe on read cycles only.
'11'	Use MemStrobe on both read and write cycles.

Table 5.2. R3041 MemStrobe Configuration Field

ExtDataEn Control

These bits control the use of the ExtDataEn pin, according to Table 5.3.

These bits depend on the settings of the SBRCond control bit; if the bit is programmed to allow SBrCond(3:2) to be used as outputs, the settings of the table apply. Otherwise, SBrCond(3:2) will be used as inputs, and the value of the ExtDataEn Control field is ignored.

Value	Action
'00'	ExtDataEn remains high on both reads and writes.
'01'	Use as ExtDataEn on write cycles only.
'10'	Use as ExtDataEn on read cycles only.
'11'	Useas ExtDataEn on both read and write cycles (default).

Table 5.3. R3041 ExtDataEn Configuration Field

IOStrobe Control

These bits control the use of the BrCond(3) pin, according to Table 5.4.

These bits depend on the settings of the SBRCond control bit; if the bit is programmed to allow SBrCond(3:2) to be used as outputs, the settings of the table apply. Otherwise, SBrCond(3:2) will be used as inputs, and the value of the IOStrobe Control field is ignored.

Value	Action
'00'	IOStrobe remains high on both reads and writes.
'01'	Use as IOStrobe on write cycles only.
'10'	Use as IOStrobe on read cycles only.
'11'	Use as IOStrobe on both read and write cycles (default).

Table 5.4. R3041 IOStrobe Configuration Field

BE16 Control

When set high ('1'), the $\overline{BE16(1:0)}$ outputs will assert according to the datum size in both read and write transfers. When reset low ('0'), both $\overline{BE16(1:0)}$ outputs will be negated during read transactions; on write transactions, $\overline{BE16(1:0)}$ will assert according to the size of the datum to be transferred.

This feature allows the $\overline{BE16(1:0)}$ outputs to be used as Write Strobes to 16-bit DRAM systems, by directly connecting $\overline{BE16(1:0)}$ to the Write Enables of the memories, and using the $\overline{RAS}$ and $\overline{CAS}$ lines to perform memory selects. $\overline{BE16(1:0)}$ can also be connected to SRAMs and other memories if their chip selects are registered instead of transparently latched.

Reset initializes this field to high ('1').

Value	Action
'0'	$\overline{BE16(1:0)}$ masked as de-asserted during reads.
'1'	$\overline{BE16(1:0)}$ active during reads (default).

Table 5.5. R3041 BE16 Control Field

BE Control

When set high ('1'), the $\overline{BE(3:0)}$ outputs will assert according to the datum size in both read and write transfers. When reset low ('0'), the $\overline{BE(3:0)}$ outputs will be negated during read transactions; on write transactions, $\overline{BE(3:0)}$ will assert according to the size of the datum to be transferred.

This feature allows the $\overline{BE(3:0)}$ outputs to be used as Write Strobes to 32-bit DRAM systems, by directly connecting $\overline{BE(3:0)}$ to the Write Enables of the memories, and using the $\overline{RAS}$ and $\overline{CAS}$ lines to perform memory selects. If $\overline{RAS}$ before $\overline{CAS}$ refreshing is used, then the DRAMs must be 1Mb or less since many 4Mb DRAMs must de-assert their $\overline{WE}$ pin during refreshing. $\overline{BE(3:0)}$ can also be connected to SRAMs and other memories if their chip selects are registered instead of transparently latched.

Reset initializes this field to high ('1'), consistent with R3051 $\overline{BE(3:0)}$.

Value	Action
'0'	$\overline{BE(3:0)}$ masked as de-asserted during reads.
'1'	$\overline{BE(3:0)}$ active during reads (default).

Table 5.6. R3041 BE Control Field

Bus Turn Around

This two-bit field controls the minimum number of clock cycles required between sampling data on a read cycle, and asserting an address for a subsequent transfer. Read response data is provided by memory or I/O devices, which drive the A/D bus for sampling by the processor; during the address phase of a subsequent transfer, the processor drives the A/D bus with a target address. This change in mastership is referred to as "Bus Turn Around". Extending the minimum amount of time for bus turnaround allows relatively slow memory devices to be used without data buffers.

Table 5.7 shows the values supported by the R3041 for this field. At reset, the default value of this field is '11', corresponding to the maximum value of 3.5 cycles.

Value	Action
'00'	No additional delay; 0.5 cycles minimum.
'01'	One additional delay cycle; 1.5 cycles minimum.
'10'	Two additional delay cycles; 2.5 cycles minimum.
'11'	Three additional delay cycles; 3.5 cycles minimum (default).

Table 5.7. R3041 Bus Turn Around Configuration Field

DMA Protocol Control

This bit enables the DMA pulse protocol of the R3041, described in Chapter 10. If this bit is set ('1'), the R3041 may request that an external DMA master relinquish bus mastership back to the CPU during a DMA cycle by negating its $\overline{\text{BusGnt}}$ output, and waiting for the external master to negate the $\overline{\text{BusReq}}$ input.

If this bit is cleared ('0'), R3051 compatible operation will result, and $\overline{\text{BusGnt}}$ will remain asserted throughout the DMA mastership cycle.

The default is '0' on reset.

Value	Action
'0'	R3051 compatible DMA operation (default).
'1'	R3041 DMA pulse protocol enabled.

Table 5.8. R3041 DMA Protocol Control Field

TC Control

This bit controls the waveform seen on the $\overline{\text{TC}}$ (Terminal Count) output pin of the R3041 and defaults to '0' on reset.

Regardless of the bit setting, $\overline{\text{TC}}$ asserts (active low) on the rising edge of $\overline{\text{SysClk}}$, two clock cycles after the Count register equals the Compare register.

If this bit value is cleared low ('0'), $\overline{\text{TC}}$ will then negate on the falling edge of $\overline{\text{SysClk}}$ that is 1.5 clock cycles after the assertion of $\overline{\text{TC}}$, as shown in Figure 5.3. This mode of operation may typically be used for DRAM refresh requests; no software intervention is required to de-assert $\overline{\text{TC}}$.

If this bit value is set high ('1'), $\overline{\text{TC}}$ will remain asserted until software re-writes the Compare register. This mode of operation corresponds to the use of the timer as an interrupt generator; $\overline{\text{TC}}$ may be tied to one of the CPU interrupt inputs, and the interrupt handler will clear $\overline{\text{TC}}$ by re-writing the Compare register. Note that for this mode of operation, the AC parameter propagation delays associated with the assertion and negation of $\overline{\text{TC}}$ use the same values as shown in Figure 5.3; however, the number of clock cycles between the assertion and negation of $\overline{\text{TC}}$ will be longer.

Value	Action
'0'	$\overline{\text{TC}}$ output pulse (default).
'1'	$\overline{\text{TC}}$ output uses software acknowledge.

Table 5.9. R3041 TC Control Field

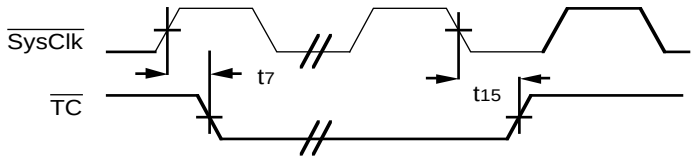


Figure 5.3. R3041 $\overline{\text{TC}}$ Output

BR Control

This bit controls the usage of the SBrCond(3:2) pins. If low (the default on reset), the SBrCond(3:2) pins will function as the SBrCond(3:2) inputs. If set high, the SBrCond(3) and SBrCond(2) pins will function as the $\overline{\text{IOStrobe}}$ and $\overline{\text{ExtDataEn}}$ outputs, respectively.

Value	Action
'0'	SBrCond(3:2) pins are inputs (default).
'1'	SBrCond(3:2) pins are outputs.

Table 5.10. R3041 BR Control Field

CACHE CONFIGURATION REGISTER

The cache configuration register allows the kernel to control various operational aspects of the on-chip caches of the R3041. These features can be used to improve performance and/or implement debug capability for the R3041. The Config register is both readable and writeable.

Figure 5.4 illustrates the various fields of the cache configuration register. The reset defaults for this register insure R3051 compatible operation.

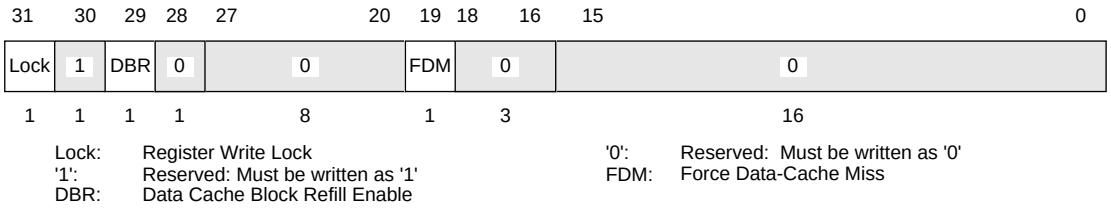


Figure 5.4. R3041 Cache Configuration Register

Lock

The lock bit can be used by the kernel to inhibit subsequent write operations to this register. It is useful in ensuring that operating systems written for other R3000A-based applications do not inadvertently change the fields of the Cache Configuration register.

At reset, the register is unlocked (Lock bit is '0'). Thus, the Config register can be written and re-written as the operating system chooses. Once the Lock bit is written with a '1', subsequent writes to the Config register will be ignored.

Value	Action
'0'	Leaves unlocked (default).
'1'	Lock register from future writes.

Table 5.11. R3041 Cache Configuration Register Lock Field

Reserved-High ('1')

This bit is reserved for testing of the R3041. At reset, the bit will be set high ('1'). Writes to the Config register must maintain this bit as high ('1').

Reserved-Low ('0')

These fields are reserved for testing and for future variants of the R3041. At reset, these bit fields are reset ('0'). Writes to the Config register must maintain these bit fields as low ('0').

DBlockRefill ('DBR')

If this bit is set high ('1'), data cache misses will be processed as a quad (four-word) read, as described in Chapter 7. If this bit is reset low ('0'), data cache misses will be processed as a single word read, as described in Chapter 7. At reset, this bit is reset low ('0').

Value	Action
'0'	Data cache misses use single word refill (default).
'1'	Data cache misses use quad word refill.

Table 5.12. R3041 DBlockRefill Field

ForceDCacheMiss ('FDM')

If this bit is set high ('1'), all cacheable data load references will be forced to miss in the data cache. The data references will then be supplied using the Data Cache miss protocol (including DBlockRefill). Full-word store operations will continue to update the cache, and the cache miss processing will update the cache. However, partial word store operations will NOT update the cache, since a read-write sequence will "miss" on the read. Thus, in general, the "FDM" bit should not be set when executing partial word stores to cachable memory. This bit provides a quick method of initializing the cache or reloading the cache from an external device.

At reset, this bit is reset low ('0'), allowing normal operation of the data cache.

Value	Action
'0'	Normal data cache operation (default).
'1'	Force data cache operations to miss.

Table 5.13. R3041 ForceDCacheMiss Field

COUNT REGISTER

The Count register implements a 24-bit, free running timer as part of the R3041 CP0. Figure 5.5 illustrates the count register.

Reset initializes the Count register to '0'. The count register is then incremented on each $\overline{\text{SysClk}}$ cycle, regardless of processor activity.

The Count register is reset to '0' with the assertion of $\overline{\text{TC}}$, when the Count register equals the value of the Compare register.

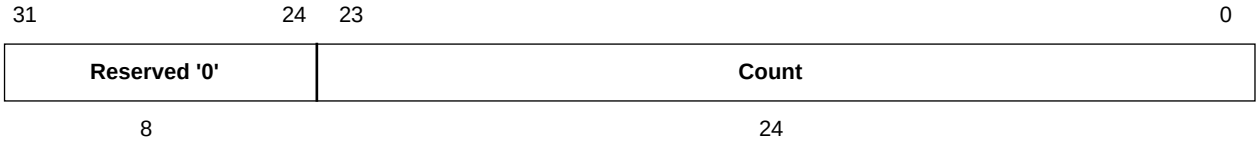


Figure 5.5. R3041 Count Register

The Count register is both readable and writeable. When read with the appropriate "mfc0" instruction, the count register is latched one clock cycle before the memory stage of the pipeline as in the example in Table 5.14. A similar latency effect is seen in writes.

```
/* assume code is running cached w/ no cache misses */
mtc0   zero, count      # load count register with '0'
nop                     # count = 1
mfc0   t0,count         # t0 = 1; count = 2
nop                     # load delay slot; count = 3
mfc0   t1,count         # t1 = 3; count = 4
nop                     # load delay slot; count = 5
```

Table 5.14. Example of Reading and Writing to the Count Register

COMPARE REGISTER

The Compare register is used in conjunction with the Count register to implement a 24-bit timer. When the value of the Count register reaches the value programmed into the Compare register, the $\overline{\text{TC}}$ output pin is asserted and the Count register is reset to '0'. Thus, the $\overline{\text{TC}}$ will be asserted once every "Compare + 1" cycles. Note that the negation of the $\overline{\text{TC}}$ output is controlled by the TC Control bit of the Bus Control register, described above.

At reset, the Compare register is initialized to the value 0x00ff_ffff. The Compare register is both readable and writeable. Writing the Compare register has no effect on the value of the Count register.

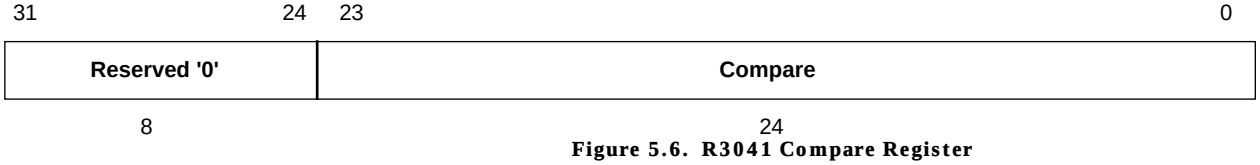


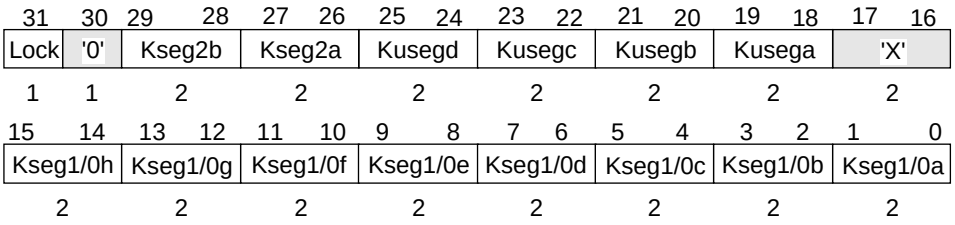
Figure 5.6. R3041 Compare Register

PORTSIZE CONTROL REGISTER

The PortSize control register is used to interface the R3041 to varying width memory regions. The PortSize register divides the physical address space into sub-regions; the data path width of each sub-region is independently programmed into the PortSize register by the operating system at boot time.

The software is then free to presume that all memory has a 32-bit data path; each off-chip reference is looked up in the PortSize register to determine the actual width of memory. The R3041 bus interface unit will then perform the appropriate sequence of transfers between the CPU and memory, depending on the actual size of the datum, and the actual width of the memory.

Figure 5.7 shows the format of the PortSize register. At reset, the initial port width of each memory region is initialized according to the width indicated for the boot PROM; that is, the PortSize register will assume that all memory is the same port width as the boot PROM. The kernel can then later re-program individual memory sub-regions, according to their actual port width.



Lock:	Register Write Lock
'0':	Reserved: Must be written as '0'
'X':	Reserved for future use
Kseg2(b:a):	Subregions of Kseg2
Kuseg(d:a):	Subregions of Kuseg
Kseg1/0(h:a):	Subregions of Kseg1 and Kseg0

Figure 5.7. R3041 PortSize Register

This allows systems to be constructed from a mix of memory widths; for example, an 8-bit boot prom, with 32-bit DRAM memory and 16-bit Font cartridge cards. This maximizes the number of price/performance trade-offs available to the system designer.

In addition, it is possible to construct a system such that its base configuration assumes a narrow memory width (e.g. a 16-bit DRAM system). However, field upgrades to larger memory systems can increase both the width and total amount of memory, increasing the performance of the system, and thus increasing the value of the field upgrade option.

Table 5.15 shows the bit encodings of memory width for each of the memory sub-regions.

Value	Port Width
'00'	32-bit
'01'	8-bit
'10'	16-bits
'11'	Reserved

Table 5.15. R3041 Port Width Encoding for PortSize Register

Table 5.16 shows the correspondence between memory sub-regions, physical addresses, and kernel/user segments of the R3041. From this, a system designer can construct varying memory widths available exclusively to the kernel or also available to the user, and can allow either cacheable or uncacheable references to these regions.

Physical Address Bits(31:26)	PortSize Register Bits	Description
111x xx	29:28	Kseg2(b) 512MB sub-region
110x xx	27:26	Kseg2(a) 512MB sub-region
101x xx	25:24	Kuseg(d) 512MB sub-region
100x xx	23:22	Kuseg(c) 512MB sub-region
011x xx	21:20	Kuseg(b) 512MB sub-region
010x xx	19:18	Kuseg(a) 512MB sub-region
001x xx	17:16	Reserved; inaccessible 512MB
0001 11	15:14	Kseg1/0(h) 64MB sub-region
0001 10	13:12	Kseg1/0(g) 64MB sub-region
0001 01	11:10	Kseg1/0(f) 64MB sub-region
0001 00	9:8	Kseg1/0(e) 64MB sub-region
0000 11	7:6	Kseg1/0(d) 64MB sub-region
0000 10	5:4	Kseg1/0(c) 64MB sub-region
0000 01	3:2	Kseg1/0(b) 64MB sub-region
0000 00	1:0	Kseg1/0(a) 64MB sub-region

Table 5.16. R3041 PortSize Memory Subregions

Lock

The lock bit can be used by the kernel to inhibit subsequent write operations to this register. It is useful in ensuring that operating systems written for other R3000A-based applications do not inadvertently change the fields of the PortSize register.

At reset, the register is unlocked (Lock bit is '0'). Thus, the PortSize register can be written and re-written as the operating system chooses. Once the Lock bit is written with a '1', subsequent writes to the PortSize register will be ignored.

Reserved

These fields are reserved for future variants of the R3041. At reset, these bit fields are set to a default value. Writes to the PortSize register should maintain these values, however, it is not mandatory to do so.

Kseg2(b:a)

These are independent 512MB sub-regions of the kseg2 virtual address space.

KUseg(d:a)

These are independent 512MB sub-regions of the kuseg virtual address space.

Kseg1/0(h:a)

These are independent 64MB sub-regions of both the kseg1 and kseg0 virtual address spaces. In the MIPS architecture, both kseg0 and kseg1 virtual address spaces are translated to the same area of physical memory; the difference between the spaces lies in the fact that references through one space are cacheable, while references through the other are not.

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 R30xx 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 R3041-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.

R30XX FAMILY EXCEPTION MODEL

The exception processing capability of the R30xx 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 R3041 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 6.1 lists the exceptions recognized by the R30xx 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) 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) 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) 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.

†These exceptions will not occur in a R3041, or in any base member of the R30xx family.

Table 6.1. R30xx 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 greatly benefit from a processor model which guarantees precise exception context and cause information. The MIPS architecture, including the R30xx family, implements a precise exception model for all exceptional events.

EXCEPTION PROCESSING

The R3051 family'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 R3041 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 CPU then performs a direct jump into a designated exception handler routine. This insures that the R3041 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 four registers handling exception processing, shown in shaded boxes in Figure 6.1. These are the *Cause* register, the *EPC* register, the *Status* register, and the *BadVAddr* register. A brief description of each follows.

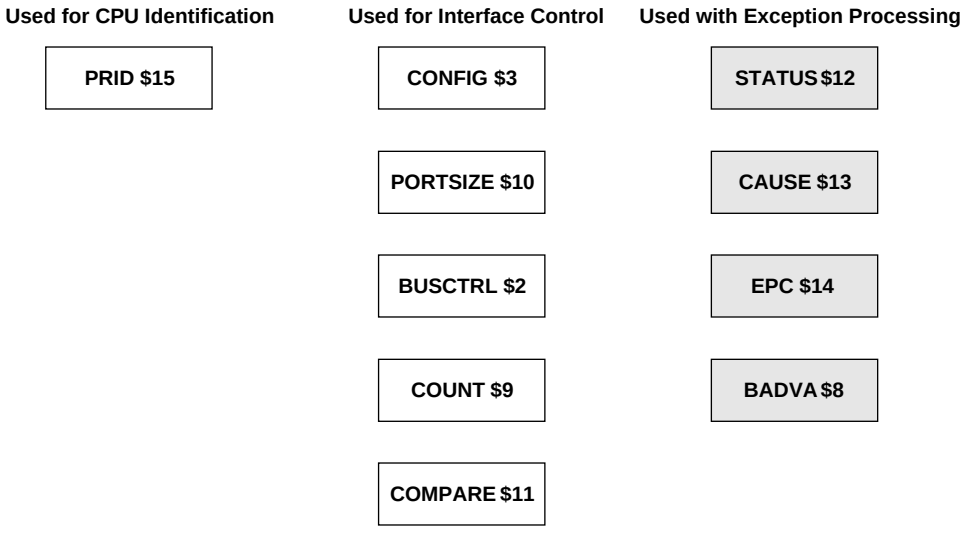


Figure 6.1. The CPO Exception Handling Registers

Table 6.2 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 (Decimal)
Bus Control	$2
Cache Configuration	$3
Bad Virtual Address	$8
Count	$9
PortSize	$10
Compare	$11
Status	$12
Cause	$13
Exception	$14
PRId	$15
Reserved	$0-$2, $4-$7, $16-$31

Table 6.2. 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 6.2 illustrates the format of the Cause register. Table 6.3 details the meaning of the various exception codes.

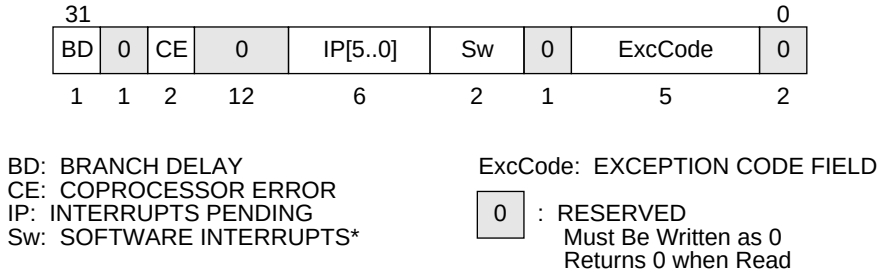


Figure 6.2. The Cause Register

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

†These exceptions will not occur in a R3041

Table 6.3. 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 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.

ExcCode The exception code field indicates the reason for the last exception. Its values are listed in Table 6.3.

The EPC (Exception Program Counter) Register

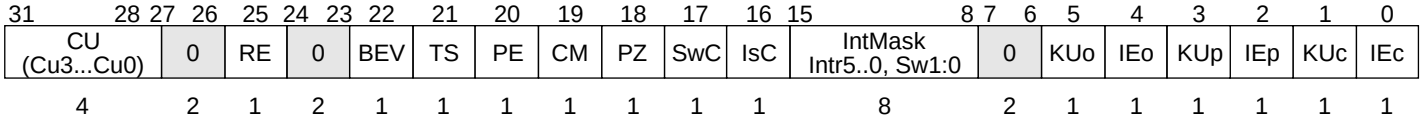
The 32-bit EPC register contains the virtual address of the instruction which took the exception, from which point processing resumes after the exception has been serviced. When the virtual address of the instruction resides in a branch delay slot, the EPC contains the virtual address of the instruction 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.

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 6.3 shows the functionality of the various bits in the status register.



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

Figure 6.3. 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 is set to "1". The rest of the bit fields are undefined after reset.

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

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

RE Reverse Endianness. The R30xx 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. This bit has no effect on kernel mode. Also note that the setting of this bit does not affect the byte lanes used in 16- and 8-bit memory ports; thus, external byte lane shift logic is not required.

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. For the R3041, this bit is frozen at "1".

PE Parity Error. This field should be written with a "1" at boot time. Once initialized, this field will always be read as "0'.

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.

PZ Parity Zero. This field should always be written with a "0".

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 un-swaps the caches. This is useful for certain operations such as instruction cache flushing. This feature is not intended for normal operation with the caches swapped.

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

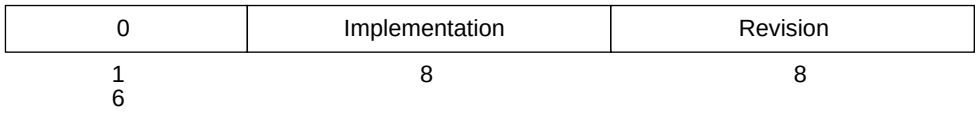
KUc Kernel/User current. This is the current privilege state. A '0' indicates kernel mode.

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 6.4; for the R3041, the value returned is 0x0000_070x. On the R3041, the most significant 4 bits of the Revision field form an extension to the Implementation field. The least significant 4 bits of the Revision field are reserved for manufacturing. This value is different from other members of the R30xx family, so that software can easily determine the CPU type. This facilitates the development of one binary working with all R30xx family members.



0: READ AS 0, MUST BE WRITTEN AS 0
Implementation: EXECUTION ENGINE IMPLEMENTATION CODE
Revision: REVISION LEVEL FOR THIS IMPLEMENTATION

Figure 6.4. Format of Prid Register

EXCEPTION VECTOR LOCATIONS

The R30xx 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 6.4 and 6.5 list the exception vectors for the R30xx 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

Table 6.4. 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

Table 6.5. Exception Vectors When BEV = 1

EXCEPTION PRIORITIZATION

It is important to understand the structure of the R30xx family instruction execution unit in order to understand the exception priority model of the processor. The R30xx family runs instructions through a five stage pipeline, illustrated in Figure 6.5. 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.

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.

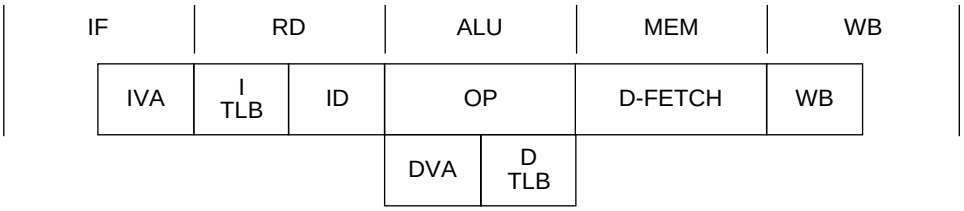


Figure 6.5. Pipelining in the R30xx Family

For example, an illegal instruction exception can only be detected in the instruction decode stage of the R3041; 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 6.6 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)
MOD†	ALU (Data TLB)
TLBL†	ALU (DTLB Miss)
TLBS†	ALU (DTLB Miss)
Ovf	ALU
Int	ALU
Sys	RD (Instruction Decode)
Bp	RD (Instruction Decode)
RI	RD (Instruction Decode)
CpU	RD (Instruction Decode)
TLBL†	I-Fetch (ITLB Miss)
AdEL	IVA (Instruction Virtual Address)
IBE	RD (end of I-Fetch)

†These exceptions will not occur in an R3041, which does not include a TLB.

Table 6.6. R30xx 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 6.6 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 R30XX FAMILY

The R30xx 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 6.6.

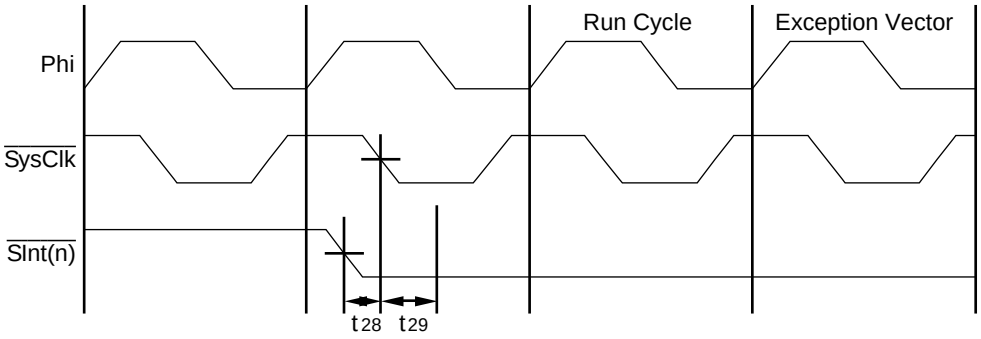


Figure 6.6. Synchronized Interrupt Operation

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 R3041. The operation of these interrupts is illustrated in Figure 6.7.

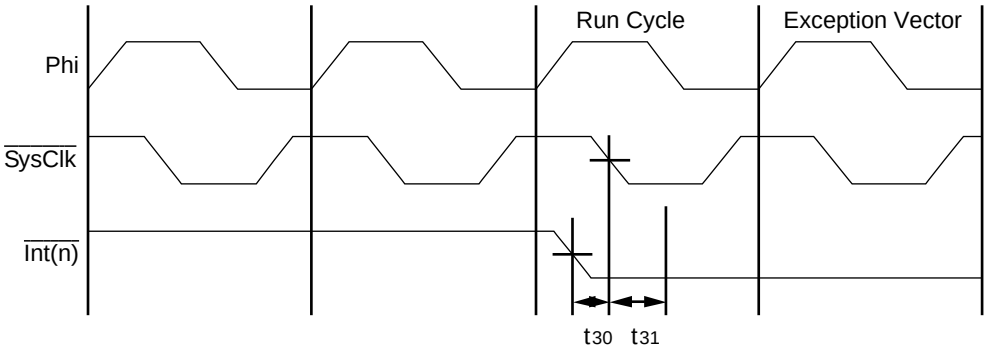


Figure 6.7. Direct Interrupt Operation

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 the R3081 incorporates a hardware floating point accelerator on-chip. The MIPS architecture recommends that $\overline{\text{Int}}$(3) be used to handle the floating point interrupt; thus, the R3081 defaults to this interrupt assignment. However, the R3081 Config register (which differs from the R3041 Config register) can be used to change the assignment. In any case, it is recommended that the system designer reserve one interrupt for the FPA.

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.

The R3041 provides two synchronized BrCond inputs: SBrCond(3:2). Note that BrCond(0), corresponding to the on-chip CP0, and BrCond(1), corresponding to Co-Processor 1 (the FPA, present on the R3081), are not available to the R3041 as user inputs. Instructions that use BrCond(1:0) will always see a '1' on the R3041. Also note that the SBrCond(3:2) on the R3041 may be programmed as output functions for the bus interface, as described in Chapter 5, in which case the SBrCond(3:2) input values are undefined. When programmed to be inputs, the timing requirements of the SBrCond inputs are illustrated in Figure 6.8. Since these inputs are synchronized by the R3041, they do not need to be driven synchronously to the processor.

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. Before executing a Branch Condition instruction (i.e. BCzT, BCzF) the corresponding co-processor usable bit in the CP0 status register must be set; otherwise, a co-processor unusable exception will be signalled.

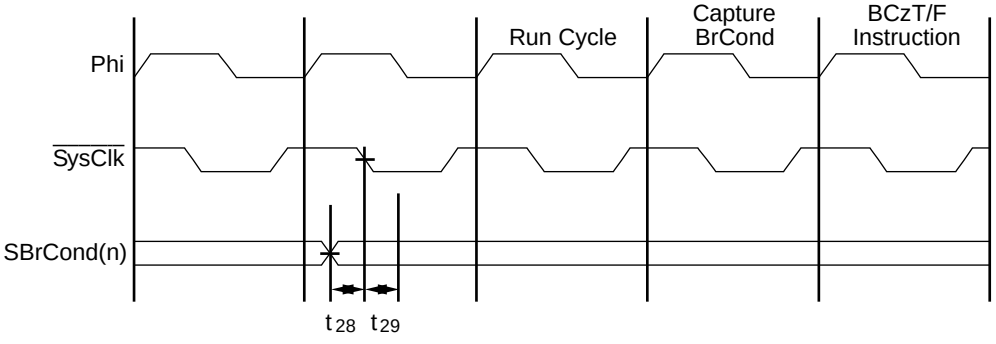


Figure 6.8. Synchronized BrCond Inputs

INTERRUPT HANDLING

The assertion of an unmasked interrupt input causes the R30xx 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 6.9. 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.

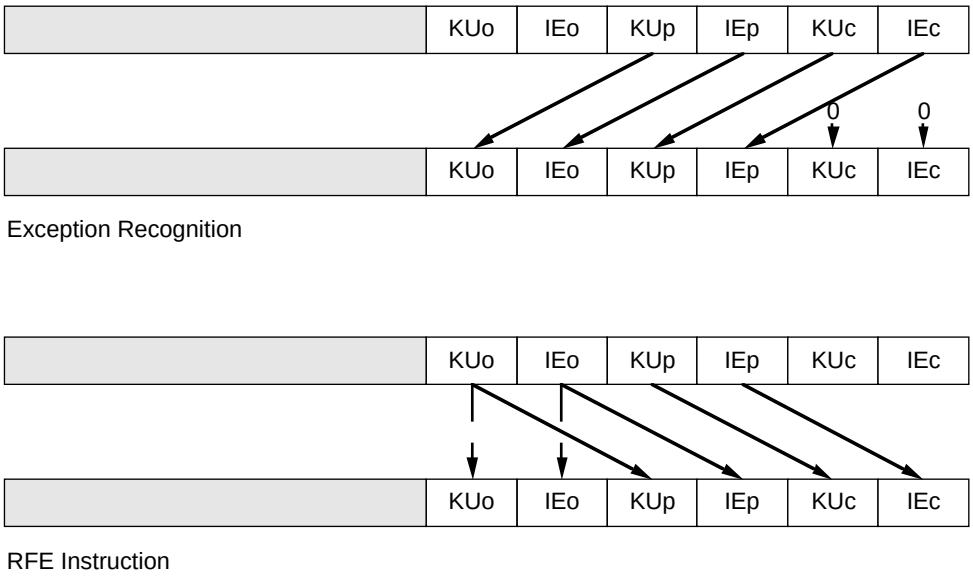


Figure 6.9. Kernel and Interrupt Status Being Saved on Interrupts

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.

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 R30xx 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 6.10 may be used.

It should be noted 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.

```
        .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
```

Figure 6.10. Code Sequence to Initialize Exception Vectors

PRESERVING CONTEXT

The R3041 has the following four 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

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 an address error, software may also preserve the bad virtual address register for later processing.

Note that not all systems need to preserve this information. Since the R30xx 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 6.11. 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.

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.

```
        .set no reorder
        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
        .set reorder

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

Figure 6.11. Preserving Processor Context

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 6.12. 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
```

Figure 6.12. 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 6.13.

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.

```
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                         */
        };
```

Figure 6.13. Exception Service Branch Table

RETURNING FROM EXCEPTIONS

Returning from the exception routine is made through the *rfe* instruction. When the exception first occurs the R3041 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 6.14 is an example of exiting an interrupt handler. The assumption is that registers and context were saved as outlined above.

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.

```
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
```

Figure 6.14. Returning from Exception

SPECIAL TECHNIQUES FOR INTERRUPT HANDLING

There are a number of techniques which take advantage of the R30xx 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 exceptions. 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 R30xx 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 R3041 has external input port signals, the BrCond(3:2) 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 6.15.

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 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 6.16 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 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
```

Figure 6.15. Polling System Using BrCond

```
        .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
```

Figure 6.16. 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, and 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 R3041 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 R30xx family branches to the General Exception vector for this exception. When the exception occurs, the R3041 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 R3041 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 R3041 executes the BREAK instruction. This exception is not maskable.

Handling

The R3041 branches to the General Exception vector for the exception and sets the BP code in the CAUSE register ExcCode field.

The R3041 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.

Servicing

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 R3041 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 R3041 branches to the General Exception vector for this exception. When exception occurs, the R3041 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 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 R3041 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 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 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 R3041 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 co-processors (3,2,1, or 0) the R3041 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 R3041 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 R3041 branches to the General Exception vector for this exception. The R3041 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 R3041 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 R3041 branches to the General Exception vector for this exception. The R3041 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 R3041 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 hand the executing process a floating point exception or integer overflow error when this exception occurs. Such an error is usually fatal.

Reserved Instruction Exception

Cause

This exception occurs when the R3041 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 R3041 processor architecture.

Handling

The R3041 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 R3041 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 R3041 RESET signal is asserted and then de-asserted.

Handling

The R3041 provides a special exception vector for this exception. The Reset vector resides in the R3041's un-mapped and un-cached 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 R3041 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 TS bit of the Status register is frozen at one.
- The Config register is unlocked and initialized as described in Chapter 5.
- The PortSize register is unlocked and initialized according to the Reset width of Boot Prom selected at Reset, as described in Chapter 5.
- The BusCtrl is configured for R3051 compatible operation, as described in Chapter 5.
- The Count register is initialized to 0.
- The Compare register is initialized to 0x00ff_ffff.

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, including setting of the PortSize, Config, and BusCtrl registers. The reset exception vector is selected to appear in the uncached, un-mapped 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 R3041 executes a SYSCALL instruction.

Handling

The R3041 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 R3041 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.

INTERFACE OVERVIEW

CHAPTER 7

The IDT R30xx 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. The R3041 bus interface is superset compatible with the R30xx family. Thus the R3041 can use the same interface chips, state machines, and board designs as the rest of the R30xx family. In addition, to the R30xx family bus, the R3041 adds interface options which are capable of reducing system costs. The R3041 adds new control signals and timing options which can simplify memory and I/O controllers. In addition, the memory sub-region CP0 Port Size register allows preassigned memory blocks the capability of handling 16-bit and 8-bit interfaces as well as the R30xx family compatible 32-bit interface. 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 R3041 supports the following types of operations on its interface:

- **Read Operations:** The processor executes an instruction fetch or a data load operation as the result of either a cache miss or an uncacheable reference. The read interface is designed to accommodate a wide variety of memory system strategies. There are two primary types of reads performed by the processor, bursts and single datum reads. An additional type for 16-bit and 8-bit interfaces is also defined, called mini-bursts:

  Burst reads (quad word, octi halfword, or 16 (sexdeci) byte reads corresponding to 32-bit, 16-bit, and 8-bit interfaces, respectively) occur when the processor requests a contiguous block of four words from memory. Bursts occur in response to instruction cache misses, and will occur in response to a data cache miss if the DBlockRefill option in the CP0 Cache Configuration register is enabled. The processor incorporates an on-chip 4-word 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 datums of a block read or in interfacing to memory systems narrower than 32-bits wide. 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 datum of the block.

  Single datum reads (Single word, halfword, or byte reads corresponding to 32-bit, 16-bit, and 8-bit interfaces, respectively) are used for uncacheable references (such as for I/O or boot code) and will be used in response to a 32-bit interface data cache miss if the DBlockRefill option in the CP0 Cache Configuration register is disabled. A single datum reads returns one unit of data per read transaction. The processor is capable of retiring a single datum read in as few as two clock cycles.

Mini-burst reads are a type of read that is in addition to the two primary read types of burst and single datum reads. Only the memory sub-regions using 16-bit and 8-bit interfaces are capable of mini-burst reads. For a 16-bit interface, a mini-burst consists of two halfwords returned within the same read transaction. For an 8-bit interface, a mini-burst consists of two, three, or four bytes returned within the same read transaction.

The read interface of the R3041 is described in detail in Chapter 8.

- **Write Operations:** The R3041 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 R3041 is designed to allow a variety of write strategies, from fast 2-cycle write operations through multiple wait-state writes to 32-bit, 16-bit, and 8-bit memory sub-regions. There is a single primary type of write:

  Single datum writes (word, halfword, or byte writes corresponding to 32-bit, 16-bit and 8-bit interfaces, respectively) are used in response to a data cache miss on the 32-bit interface or possibly for an uncacheable data reference on any of the interface sizes. The processor is capable of retiring a single datum write in as few as two clock cycles.

  Mini-burst writes are a type of write that is in addition to the primary write type of single datum writes. Only the memory sub-regions using 16-bit and 8-bit interfaces are capable of mini-burst writes. For a 16-bit interface, a mini-burst consists of two halfwords sent within the same write transaction. For an 8-bit interface, a mini-burst consists of two, three, or four bytes sent within the same write transaction.

  The R3041 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 to main memory, since the majority of processor writes will occur within a localized area of memory.

  The write interface is described in detail in Chapter 9.

- **DMA Operations:** The R3041 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 R3041 utilizes a very simple handshake to transfer control of its interface bus. This handshake is described in detail in Chapter 10.

MULTIPLE OPERATIONS

It is possible for the R30xx family interface to have multiple 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 R3041 is sensitive to possible conflicts and data coherency issues as well as to performance 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.

There are two levels of priority: that performed by the CPU engine internal to the R3041, and that performed by the bus interface unit. The internal execution engine can be viewed as making requests to the bus interface unit. In the case of multiple requests in the same clock cycle, the CPU core will:

1: Perform the data request first. That is, if both the data cache and instruction cache miss in the same clock cycle, the processor core will request a read to satisfy the data cache first. Similarly, a write buffer full stall will be processed before an instruction cache miss.
2: Perform a read due to an instruction cache miss.

This prioritization is important in maintaining the precise exception model of the MIPS architecture. Since data references are the result of instructions which entered the pipeline earlier, they must be processed (and any exceptions serviced) before subsequent instructions (and their exceptions) are serviced.

Once the processor core internally decides which type of request to make to the bus interface unit, it then presents that request to the bus interface unit.

Thus, in the R3041 Bus Interface Unit, 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.

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 and main memory. 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. However, in order to unswap the caches, an uncacheable instruction which modifies CP0 must be executed; the fetch of this instruction would cause the write buffer to be flushed to memory. Thus, this ordering enforces strong ordering of operations in hardware, even for self modifying code. Of course, software could perform an uncacheable reference to flush the write buffer at any time, thus achieving memory synchronization with software.

EXECUTION ENGINE FUNDAMENTALS

This section describes the fundamentals of the processor interface and its interaction with the execution core. These fundamentals will help to explain the relationship between design trade-offs in the system interface and the performance achieved in R30xx family systems.

Execution Core Cycles

The R30xx family 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 R30xx family 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 R30xx family execution core. 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 execution 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 R30xx family, 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 Instruction/Data (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 Instruction/Data (ID) pair processed in the final fixup cycle. The R3041 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

†Micro-TLB stalls will not occur in the R3041, which does not include an on-chip TLB.

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. Also note that signals indicated with an $\overline{overbar}$ 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

Multiplexed Address/Data Bus: 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 indicate 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. They are valid for the 32-bit port size. For 16-bit or 8-bit port sizes, $\overline{BE(3:0)}$ are not valid, however, they do indicate which bytes will be used sometime during the (multi-datum) transaction. $\overline{BE(3:0)}$ can also be masked (held in-active) during reads by disabling the BE Control read mask bit in the CP0 Bus Control register.

Data(31:0): 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 datum transaction, mini-burst, or burst and places the data into the on-chip read buffer.
Operations using less than 32-bits of data use the data lines as described in Chapter 2 Table 2.3 describing Byte Addressing. The byte addressing in summary requires that 16-bit interfaces use the bytes associated with address offsets 0 and 1, i.e., D(31:16) for big endian and D(15:0) for little endian. 8-bit interfaces use the byte associated with address offset 0, i.e., D(31:24) for big endian and D(7:0) for little endian. These byte lane assignments are dependent on the endianess reset initialization mode, but are independent of the User Mode Reverse Endianess control bit in the CP0 Status register.

Addr(3:0) O

Dedicated Address Bus. The remaining least significant bits of the transfer address are presented directly on these outputs and indicate which word, halfword, or byte is currently expected by the processor.

Specifically, for 32-bit interfaces, Addr(3:2) presents either the address bits for the single word to be transferred (single word reads or writes) or functions as a two bit counter starting at '00' for burst (quad word) read operations. Addr(1:0) are undefined for accesses to 32-bit memory sub-regions.

For 16-bit interfaces, Addr(3:1) presents either the address bits for the single halfword to be transferred (single halfword reads or writes), or functions as a three bit counter starting at '000' for burst (octi halfword) read, and mini-burst (double halfword) read or write operations. Addr(0) is undefined for accesses to 16-bit memory sub-regions.

For 8-bit interfaces, Addr(3:0) presents either the address bits for the single byte to be transferred (single byte reads or writes), or functions as a four bit counter for burst (16 byte) read, and mini-burst (double, tri, or quad byte) read or write operations.

The R3041 Addr(1:0) output pins are designated in the R3051 as the no-connect Rsvd(1:0) pins respectively.

Primary Read and Write Control Signals

ALE O

Address Latch Enable: This active high output signal is used to indicate that the A/D bus contains valid address information for the bus transaction. Typically it is 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.

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

Data Input Enable: This active low output signal indicates that the A/D 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, this signal is negated.

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

Burst Transfer: On read transactions, this active low output signal indicates that the current bus read is requesting a block of four contiguous words (or eight halfwords, or sixteen bytes) 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 "DBR" is selected with the CP0 Cache Configuration register.
Write Near: On write transactions, this active low output signal 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 active low output signal which indicates that the current bus transaction is a read.

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

Write: An active low output signal which indicates that the current bus transaction is a write.

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

Acknowledge: On read transactions, this active low input indicates the internal R3041 execution core can begin to process the data in the read buffer and that the read transaction is near completion.
On write transactions, this active low input indicates to the R3041 that the memory system has sufficiently processed the write data, and that the processor may either advance to the next write data in a mini-burst write and/or that the processor may advance to the next bus transaction.

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

Read Buffer Clock Enable: An active low input which indicates to the R3041 that the memory system has placed valid data on the A/D bus, and that the processor may move the data into the on-chip Read Buffer.

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

Bus Error: An active low input which terminates 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.

Secondary Read and Write Control Signals

$\overline{\text{BE16(1:0)}}$ O

Byte Enable Strobes for 16-Bit Ports: These active low outputs are the byte enable strobes for 16-bit ports. If $\overline{\text{BE16(1)}}$ is asserted then the most significant byte (D(31:24) for big endian or D(15:8) for little endian) is going to be used in this transaction by the R3041. If $\overline{\text{BE16(0)}}$ is asserted then the least significant byte (D(23:16) for big endian or D(7:0) for little endian) is going to be used in this transaction by the R3041. $\overline{\text{BE16(1:0)}}$ can also be masked (held in-active) during reads by disabling the BE16 Control read mask in the CP0 Bus Control register. $\overline{\text{BE16(1:0)}}$ is not necessarily valid for 32-bit or 8-bit ports.

The R3041 $\overline{\text{BE16(1:0)}}$ output pins are designated in the R3051 as the no-connect Rsvd(3:2) pins, respectively.

$\overline{\text{Last}}$ O

Last Datum in Mini-Burst. This active low output indicates that the last datum of a single datum, mini-burst or burst is being read or that the last datum of a single datum or mini-burst is being written. It goes active with $\overline{\text{Rd}}$ or $\overline{\text{Wr}}$ for single datum reads or writes, after the next to last $\overline{\text{RdCEn}}$ is sampled for multiple datum reads, and after the next to last $\overline{\text{Ack}}$ is sampled for mini-burst writes. $\overline{\text{Last}}$ de-asserts when $\overline{\text{Rd}}$ or $\overline{\text{Wr}}$ de-asserts.

The R3041 $\overline{\text{Last}}$ output pin is designated in the R3051 as the Diag(0) output pin.

$\overline{\text{MemStrobe}}$ O

Memory Strobe: This active low output pulses low for each datum read or written. It can be used either as a read strobe, write strobe, data strobe for single datum (non-burst) I/O ports or for a write strobe (burst and non-burst) for SRAM. It can be active for reads, writes, or both depending on the settings in MemStrobe Control bits in the CP0 Bus Control register as described in Chapter 5. After reset, $\overline{\text{MemStrobe}}$ is only active for writes.

The R3041 $\overline{\text{MemStrobe}}$ output pin is designated in the R3051 as the BrCond(0) input pin.

IOStrobe/ O
SBrCond(3) I

The SBrCond(3) pin is used as an input when the SBrCond(3:2) In control bit in the CP0 Bus Control register is asserted. When de-asserted, the pin becomes the IOStrobe output.

Input/Output Strobe: This active low output asserts on the first falling edge of SysClk (1 clock) after ALE de-asserts. It asserts relatively late in the cycle so that addresses and control lines are properly setup. It can be active for reads, writes, or both depending on the setting of the IOStrobe Control bits in the CP0 Bus Control register.
Note that since this signal pin can only become an output after boot PROM initialization has taken place, it cannot be used to control the boot PROM itself. Typical uses include I/O chip select gating, an address mux select for DRAMs, or a data strobe for I/O.
Branch Condition Port 3: This input port to the processor can use the Branch on Co-Processor Condition instructions to test its polarity. The SBrCond(3) input is synchronized by the R3041, and thus may be driven by an asynchronous source.

ExtDataEn/ O
SBrCond(2) I

The SBrCond(2) pin is used as an input when the SBrCond(3:2) In Control bit in the CP0 Bus Control register is asserted. When de-asserted, the pin becomes the ExtDataEn output.

Extended Data Enable: This active low output asserts active low on the first rising edge of SysClk after ALE de-asserts (1/2 clock later). It is extended in that it de-asserts 1/2 clock after Rd de-asserts. ExtDataEn provides extra hold time for data sampling (especially on writes) or for the IOStrobe (if ExtDataEn is used as an extended read/write line. It can be active for reads, writes, or both depending on the setting of the ExtDataEn Control bits in the CP0 Bus Control register.
Note that since this signal pin can only become an output after boot PROM initialization has taken place, it cannot be used to control the boot PROM itself. Typical uses include a write enable control line for data transceivers, a write line for I/O, or an address mux select for DRAMs.
Branch Condition Port 2: This input port to the processor can use the Branch on Co-Processor Condition instructions to test its polarity. The SBrCond(2) input is synchronized by the R3041, and thus may be driven by an asynchronous source.

Status Information and Diagnostics

Diag **O**

Diagnostic Pin: This pin is useful in the initial debug of R3041 based systems. During the address phase of the read transaction, this output indicates whether the read is a result of a cache miss (high) or an uncacheable reference (low).
During the remainder of the transfer, this output indicates whether the read is an instruction (high) or a data reference (low).
The Diag pin is undefined during write transactions.

The R3041 Diag output pin is designated in the R3051 as the Diag(1) output pin.

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

Tri-State All Outputs: An active low input to the device which requests that the processor tri-state all of its outputs. In addition to the outputs which are tri-stated during a DMA operation, $\overline{\text{SysClk}}$, $\overline{\text{TC}}$, and $\overline{\text{BusGnt}}$ are also tri-stated. $\overline{\text{TriState}}$ can be used for in-circuit testing and emulation during board production manufacture.

The R3041 $\overline{\text{TriState}}$ input pin is designated in the R3051 as the no-connect Rsvd(4) pin.

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 active low 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 R3041.

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

DMA Arbiter Bus Grant: An active low output from the R3041 used to acknowledge that a $\overline{\text{BusReq}}$ has been granted, and that the bus is relinquished to the external master. When the DMAProtocol bit in the CP0 Bus Control register is not selected, the DMA device has the highest priority. When the DMAProtocol option is selected, a handshake is invoked that allows the CPU to have an equal priority with the DMA device.

Interrupt Interface

Chapter 5 discusses the exception model of the R3041.

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

Processor Interrupt: These signals are functionally the same as the $\overline{\text{Int(5:0)}}$ signals of the R3000. The Synchronized interrupt inputs are internally synchronized by the R3041, and thus may be generated by an asynchronous interrupt agent; the direct interrupts must be externally synchronized by the interrupt agent.

Reset, Clocking, and Timer

Chapter 4 discusses the internal timer supplied by the R3041. Chapter 11 discusses the Reset and Clock Interface.

ClkIn **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. There is no specific guaranteed AC timing relationship between the ClkIn input clock and the output clock $\overline{\text{SysClk}}$.

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

Terminal Count: An active low output from the processor which pulses low for a minimum of 1.5 clocks whenever the CP0 Timer register equals the CP0 Compare register. Thus $\overline{\text{TC}}$ can be used to initiate a DRAM refresh. If the TC_Ack option is selected in the CP0 Bus Control register, then $\overline{\text{TC}}$ remains low until the CP0 Compare register is written. Thus with the TC_Ack option selected, $\overline{\text{TC}}$ can be used to implement a real-time clock by connecting it to an interrupt pin.

The R3041 $\overline{\text{TC}}$ output pin is designated in the R3051 as the BrCond(1) input pin.

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

Master Processor Reset: This active low input signal initializes the processor. Optional features of the processor are established during the last cycle of reset using the reset configuration mode inputs.

INTRODUCTION

The R3041 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 and 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.

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 programming selection made in the CP0 Cache Configuration register. Uncached instruction fetches or data references are processed as a single word or partial word read.

In processing multiple item reads, there are two parameters of interest. The first parameter is the initial latency to the first data item of the read. This latency is influenced by the overall system architecture and the type of memory system addressed: time required for 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 datum reads when the memory port is sufficiently wide.

The second parameter of interest in burst and mini-burst transfers is the repeat rate of data; that is, time required for subsequent data items 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 R3041 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 data item and multiple clock cycles between adjacent data items, including the ability to process quad word reads as multiple data item reads of a narrow memory subsystem.

The R3041 has a memory sub-region Port Size configuration CP0 register, which allows individual memory blocks to be configured to different size ports. When using a memory block that is configured as a 32-bit port, the R3041 uses single word reads or quad block reads as described above. When using a memory block that is configured as a 16-bit port, the R3041 uses single halfword reads, dual halfword mini-burst reads, or octi halfword burst reads. When using a memory block that is configured as an 8-bit port, the R3041 uses single byte reads, dual, tri or quad byte mini-burst reads, or 16 (sexdeci) byte long burst block reads.

READ INTERFACE SIGNALS

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

$\overline{Rd}$ O

Read Transaction: This active low output indicates that a read operation is occurring. It will assert when the R3041 initiates a read transaction. It will de-assert automatically after all the data has been returned.

A/D (31:0) I/O

Multiplexed Address/Data Bus: 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 by using other control signals. The address phase is at the beginning of the bus transaction and is 1/2 clock long if the ExtAddrHold reset configuration mode is not selected. If the ExtAddr Hold mode is selected, then the address portion is 1 clock long. The data phase occurs during the remaining portion of the read.

During the address 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).

$\overline{BE(3:0)}$ The byte strobes for the read transaction are presented on A/D(3:0). They are only valid for the 32-bit port size. They are not valid for 16-bit or 8-bit port sizes, however, they do indicate which bytes are used sometime during the (multi-datum) transaction. $\overline{BE}$(3:0) can also be masked (held in-active) during reads by disabling the read mask, BE Control field of the CP0 Bus Control register.

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

Data(31:0) The data lines are tri-stated. Operations using less than 32-bits of data use the data lines as described in Table 2.3 describing Byte Addressing. In summary, the byte addressing requires that 16-bit ports use the halfword associated with address offsets 0 and 1, i.e., D(31:16) for big endian and D(15:0) for little endian. 8-bit ports use the byte associated with address offset 0, i.e., D(31:24) for big endian and D(7:0) for little endian. These byte lane assignments are dependent on the endianess selected at reset, but independent of the User Mode Reverse Endianess control bit in the CP0 Status register.

ALE O

Address Latch Enable: This active high 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(3:0) O

Dedicated Address Bus: The remaining least significant bits of the transfer address are presented directly on these outputs. In the case of 32-bit quad word reads, the Addr(3:2) pins function as a two bit counter starting at '00', and are used to perform the quad word transfer. In the case of single datum reads, these pins contain Address (3:2) of the transfer address. Similarly, 16-bit ports use Addr(3:1) and 8-bit ports use Addr(3:0).

Note that Addr(1:0) in the R3041 correspond to the no-connect Rsvd(1:0) pins of the R3051.

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

Data Enable: This active low 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 R3041 guarantees the following:

- The A/D bus is driven to guarantee hold time from the negation of ALE.
- The R3041 A/D bus output drivers will be disabled on reads before the assertion of $\overline{\text{DataEn}}$.

If the ExtAddrHold reset configuration mode is not active, $\overline{\text{DataEn}}$ will be asserted immediately after ALE de-asserts and as soon as the A/D bus is tri-stated.

If the ExtAddrHold reset configuration mode is active, $\overline{\text{DataEn}}$ will be asserted as soon as the A/D bus is tri-stated on the next rising edge of $\overline{\text{SysClk}}$ after ALE de-asserts.

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

Burst Read (multiplexed with Write Near): On read cycles, this active low output distinguishes between 32-bit quad word block and single datum reads. Similarly, on 16-bit reads, this output distinguishes between 16-bit octi halfword block and all other halfword reads. On 8-bit ports this output distinguishes between 8-bit 16 byte long block reads and all other byte reads.

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

Read Buffer Clock Enable: This active low input is used by the external memory system to cause the processor to capture the contents of the A/D bus. In the case of single datum reads, this causes the processor to capture the read data and also terminates the read operation. In the case of multiple data reads, this causes the contents of the A/D bus to be strobed into the on-chip read buffer. When the final datum is captured, it also terminates the read operation.

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

Acknowledge: This active low input 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 data items not yet retrieved from the memory will be brought in before they are required by the execution core.

In general, the highest level of performance is achieved by asserting $\overline{\text{Ack}}$ concurrent with the final $\overline{\text{RdCEn}}$ for single datum and mini-burst block reads and by asserting $\overline{\text{Ack}}$ three clocks before the final $\overline{\text{RdCEn}}$ on burst block reads.

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 of $\overline{\text{RdCEn}}$. While this approach is simpler to design, a loss of performance will result for both single datum and burst reads.

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

Bus Error: This active low input can be used to terminate a read operation if asserted before or concurrently with $\overline{\text{Ack}}$. It will also cause the processor to take a Bus Error exception. Read transactions terminated by $\overline{\text{BusError}}$ do not require the assertion of $\overline{\text{Ack}}$ or $\overline{\text{RdCEn}}$.

$\overline{\text{BE16(1:0)}}$ O

Byte Enable Strobes for 16-bit ports: These active low outputs are the byte enable strobes for 16-bit ports. If $\overline{\text{BE16(1)}}$ is asserted then the most significant byte (D(31:24) for big endian or D(15:8) for little endian) is going to be sampled by the R3041. If $\overline{\text{BE16(0)}}$ is asserted then the least significant byte (D(23:16) for big endian or D(7:0) for little endian) is going to be sampled by the R3041. $\overline{\text{BE16}}$ can also be masked (held in-active) during reads by disabling the read mask, BE16 Control field of the CP0 Bus Control register for direct connection to the write enables in DRAM systems or other systems with gated chip selects. $\overline{\text{BE16}}$ is not valid for 32-bit or 8-bit ports.

The R3041 $\overline{\text{BE16(1:0)}}$ output pins are designated in the R3051 as no-connect Rsvd(3:2) pins, respectively.

$\overline{\text{Last}}$ O

Last Datum in Mini-Burst: This active low output indicates that the last datum of a single datum, mini-burst or burst is being read. It goes active with $\overline{\text{Rd}}$ for single datum reads and after the next to last $\overline{\text{RdCEn}}$ is sampled for multiple datum reads. $\overline{\text{Last}}$ de-asserts when $\overline{\text{Rd}}$ de-asserts.

$\overline{\text{MemStrobe}}$ O

Memory Strobe: This active low output pulses low for each datum read. It can be used either as a read strobe or a data strobe. It can be active for reads, writes, or both depending on the setting of the MemStrobe control field of the CP0 Bus Control Register. After reset, $\overline{\text{MemStrobe}}$ is only active for writes.

The R3041 $\overline{\text{MemStrobe}}$ output pin is designated in the R3051 as the BrCond(0) input pin.

$\overline{\text{IOStrobe}}$ O

Input/Output Strobe: This active low output asserts on the first falling edge of $\overline{\text{SysClk}}$ (1 clock) after ALE de-asserts. It asserts relatively late in the cycle so that addresses and control lines are properly setup. In order for $\overline{\text{IOStrobe}}$ to assert, the transaction must be at least 3 clock cycles long. Thus $\overline{\text{IOStrobe}}$ can be used as an I/O data strobe if $\overline{\text{ExtDataEn}}$ is used as a read/$\overline{\text{write}}$ line or $\overline{\text{IOStrobe}}$ can be used for gating I/O chip selects. It can be active for reads, writes, or both depending on the setting of the IOStrobe Control field of the CP0 Bus Control Register. Since $\overline{\text{IOStrobe}}$ is an input on reset after which it can be configured with the SBrCond(3:2) Control bit to be an output, it cannot be used to control the Boot PROM.

$\overline{\text{ExtDataEn}}$ O

Extended Data Enable: This active low output asserts active low on the first rising edge of $\overline{\text{SysClk}}$ after ALE de-asserts (1/2 clock later). It is extended in that it de-asserts 1/2 clock after $\overline{\text{Rd}}$ de-asserts. $\overline{\text{ExtDataEn}}$ provides extra hold time for data sampling (especially on writes). It can also be configured as an extended read/$\overline{\text{write}}$ line for I/O interfaces. It can be active for reads, writes, or both depending on the setting of the ExtDataEn control field of the CP0 Bus Control Register. Since $\overline{\text{ExtDataEn}}$ is an input on reset after which it can be configured with the SBrCond(3:2) Control bit to be an output, it cannot be used to control the Boot PROM.

Diag O

Diagnostic Pin: This pin is useful in the initial debug of R3041 based systems. During the address phase of the read transaction, this output indicates whether the read is a result of a cache miss (high) or an uncacheable reference (low).
During the remainder of the transfer, this output indicates whether the read is an instruction (high) or a data reference (low).

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 to one clock cycle, so that the system's 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 a 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 the 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 8.1 illustrates the initiation of a read transaction, based on the internal assertion of the $\overline{\text{MemRd}}$ control signal. This figure is useful in determining the overall latency of cache misses on processor operation.

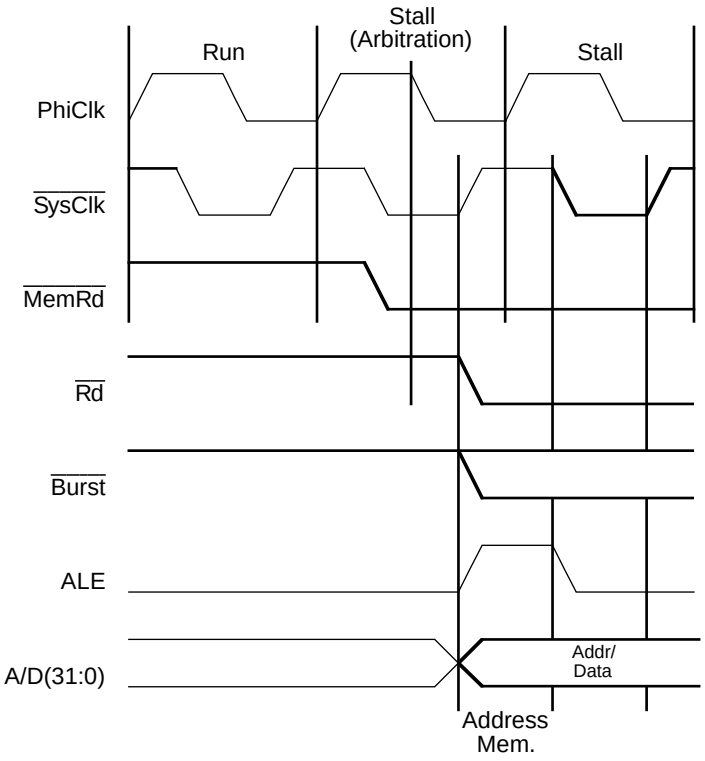


Figure 8.1. CPU Latency to Start of Read

Memory Addressing

A read transaction begins when the processor asserts its $\overline{Rd}$ 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 read transaction, when using the non-Extended Address Hold reset configuration mode option, including the addressing of memory and the intra-transaction bus turn around.

The addressing occurs in a half-cycle of the $\overline{SysClk}$ output. At the rising edge of $\overline{SysClk}$, 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, $\overline{DataEn}$ will be held high indicating that memory drivers should not be enabled onto the A/D bus.

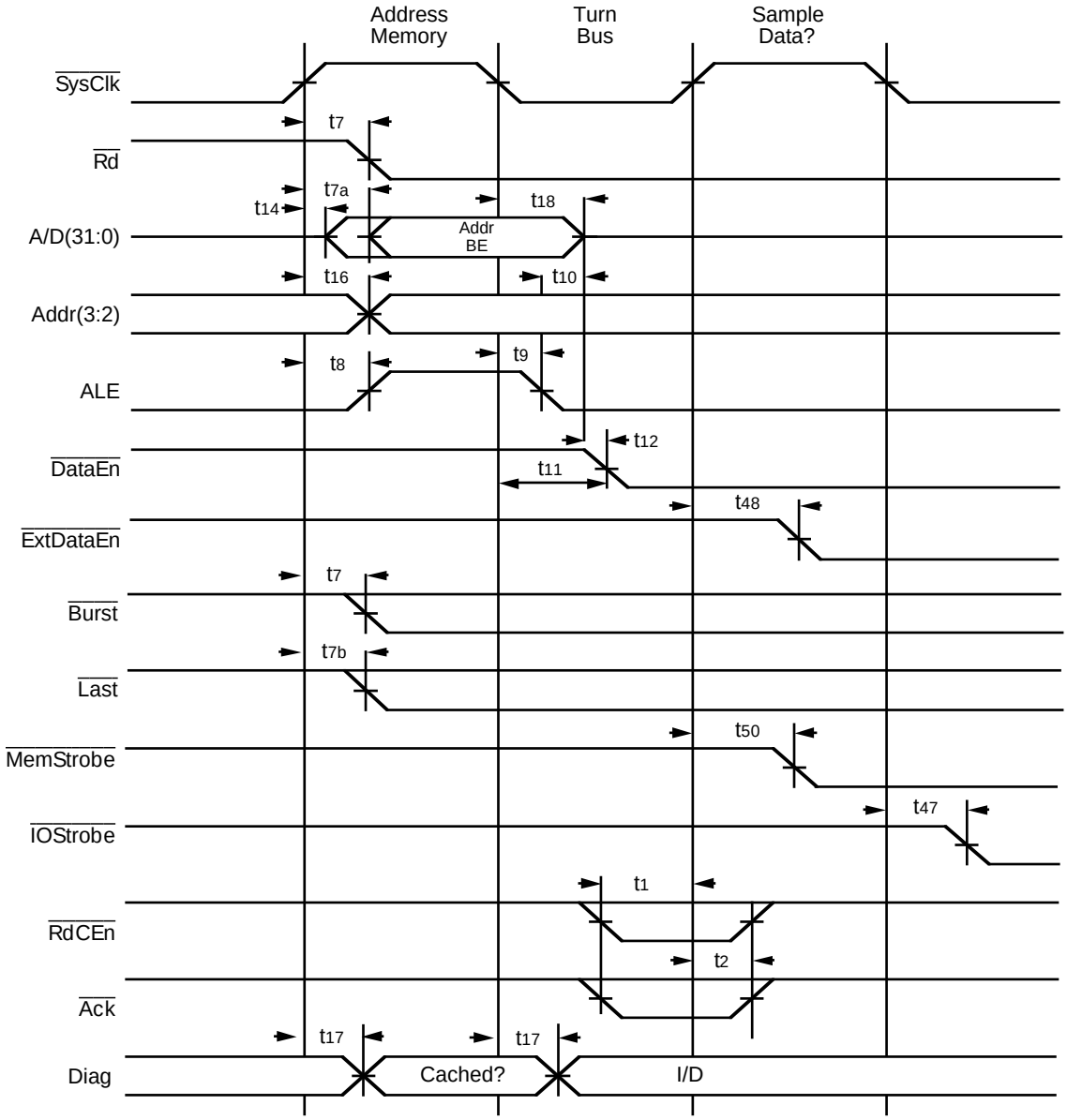


Figure 8.2. Start of Bus Read Operation Without Extended Address Hold

Concurrent with driving addresses on the A/D bus, the processor will indicate whether the read transaction is a burst block read or not, by driving $\overline{\text{Burst}}$ to the appropriate polarity (low for a burst block read). If a quad word read is indicated the Addr bus will drive to the start of the block. If a single datum or mini-burst is indicated, the Addr lines will indicate the address for the transfer. The functioning of the counter during mini-burst and burst reads is also described later.

Figure 8.2 illustrates the initiation of a read transaction when the Extended Address Hold reset configuration mode option, ExtAddrHold is turned on. ExtAddrHold delays the address to data bus turn around for an additional 1/2 clock. Thus the address is held for an extra 1/2 clock and the assertion of $\overline{\text{DataEn}}$ is delayed for 1/2 clock. Since the de-assertion of ALE is unchanged, 1/2 extra clock of address hold time is provided for easier use with ASICs, FPGAs, and other low-cost interfaces.

The remaining figures and examples in this chapter will always be given using the ExtAddrHold reset configuration mode, although either mode is always applicable.

Initiation of the Data Phase

Once the A/D bus has presented the address for the transfer, it is "turned around" by the processor to accept the incoming data. If the ExtAddrHold reset configuration mode is turned off, this occurs in the second phase of the first clock cycle of the read transaction as illustrated in Figure 8.2. If the ExtAddrHold reset mode is turned on, address to data bus turn around occurs in the first phase of the second clock cycle of the read transaction as illustrated in Figure 8.3.

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 exact timing of this depends on the reset setting of the Extended Address Hold feature, as described above.

- The processor then asserts $\overline{\text{DataEn}}$, 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 $\overline{\text{DataEn}}$ being asserted. $\overline{\text{DataEn}}$ 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 if the ExtAddrHold reset configuration mode is turned off and by the end of the first phase of the second clock if the ExtAddrHold mode is turned on. At this time, it begins to look for data to sample.

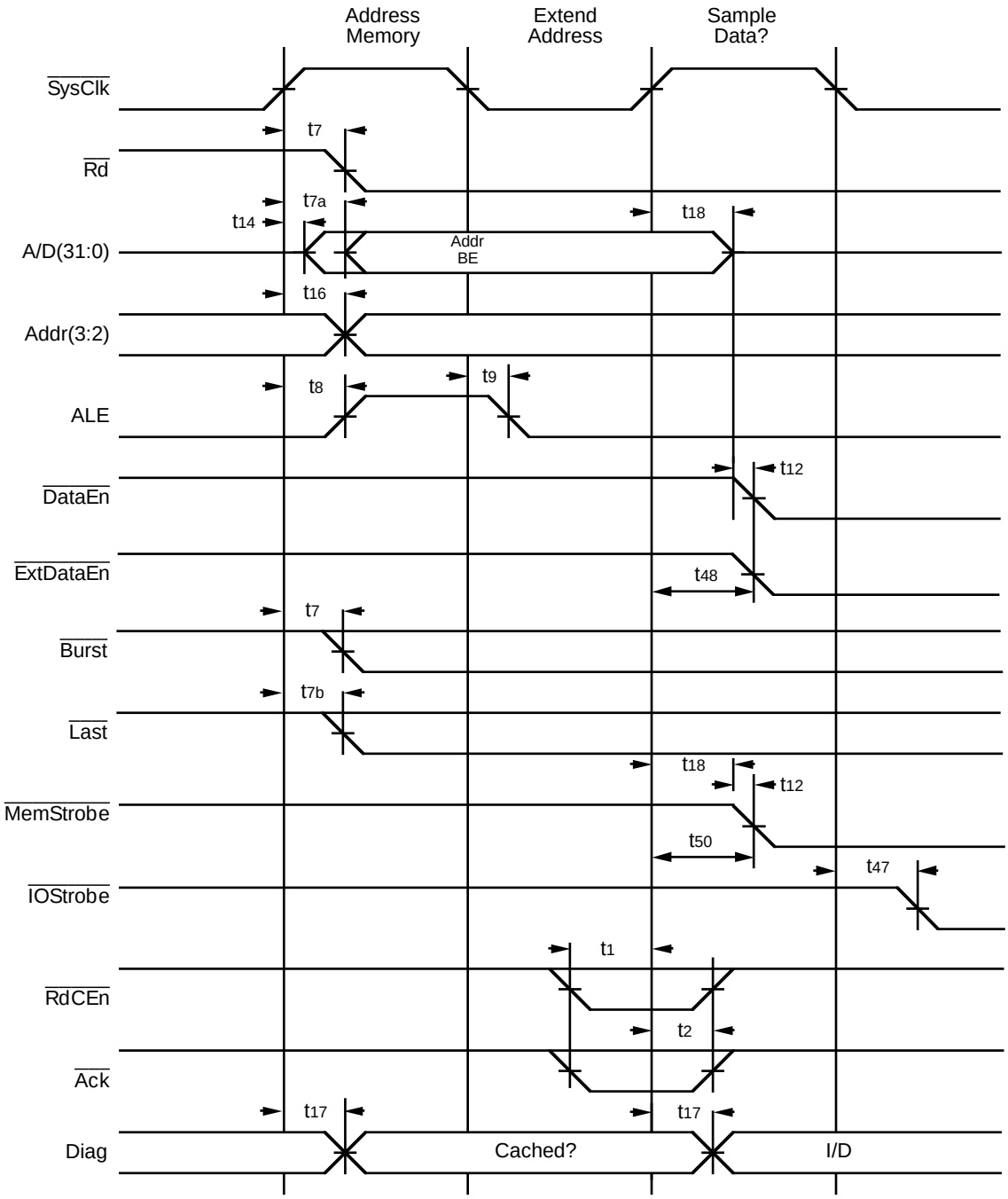


Figure 8.3. Start of Bus Read Operation with Extended Address Hold

Bringing Data into the Processor

Regardless of whether the transfer is a burst read or a single datum 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.

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

If RdCEn is detected as asserted (with adequate setup and hold time to the rising edge of 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 SysClk. This captures the data in the internal read buffer for later processing by the execution core/cache subsystem.

The R3041 integrates on-chip a 4-word read buffer, capable of acting as a speed-matching FIFO between the system interface and the execution core. This bus interface then performs byte or half-word gathering, and assembles them into 32-bit words for the read buffer. Thus, the bus interface supports 8-, 16-, and 32-bit memory subsystems, even for quad word reads, with no real system impact.

Figure 8.4 illustrates the sampling of data by the R3041.

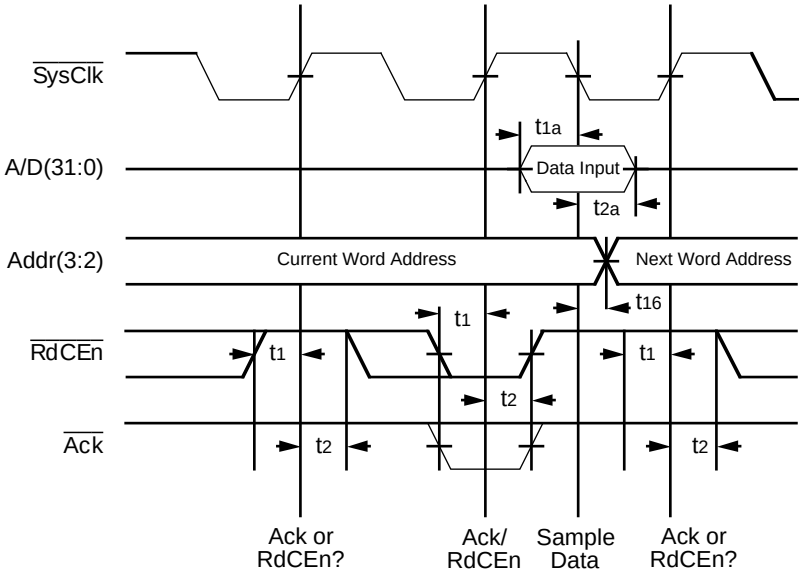


Figure 8.4. Data Sampling on the R3041

During the data phase, these three control signals may also assert:

- When programmed via the ExtDataEn and SBrCond(3:2) Control bits in the CP0 Bus Control register, ExtDataEn asserts one clock cycle after Rd asserts and remains asserted 1/2 clock cycle after Rd de-asserts. Although primarily intended for being programmed to assert on Wr cycles, ExtDataEn can also be used as a DRAM address multiplexor select if configured to assert on both reads and writes.

- When programmed via the MemStrobe Control bits in the CP0 Bus Control register, MemStrobe asserts one clock after Rd asserts. It de-asserts 1/2 clock after every RdCEn is sampled. If more datum are being read within the same transaction (i.e., on a mini-burst or burst read), MemStrobe asserts again 1/2 clock after the last de-assertion and remains asserted until the next RdCEn occurs. The (de)-assertions continue until all datum are sampled. See Figure 8.13 for an example.

- When programmed via the IOStrobe and SBrCond(3:2) Control bits in the CP0 Bus Control register, $\overline{\text{IOStrobe}}$ asserts 1.5 clock cycles after $\overline{\text{Rd}}$ asserts and remains asserted until $\overline{\text{Rd}}$ de-asserts. It will only assert if there are at least three clocks in the transaction. Thus this signal is useful for I/O reads if disabled during writes. $\overline{\text{IOStrobe}}$ can be used as an I/O data strobe if $\overline{\text{ExtDataEn}}$ is configured as a read/$\overline{\text{write}}$ signal. $\overline{\text{IOStrobe}}$ can also be used as a DRAM address multiplexor select if configured to assert on both reads and writes.

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 quad-word 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 similar 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 byte to four words), it will implicitly "acknowledge" the read after 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 assume that $\overline{\text{Ack}}$ is asserted simultaneous with the last $\overline{\text{RdCEn}}$ of a single word read transfer and 3 clocks before the last $\overline{\text{RdCEn}}$ of a burst 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.

To determine the end of the read transaction one of these methods may be used:

Systems that only use 32-bit memory sub-region ports as with the rest of the R30xx family only have single datum reads or burst reads and can either count the number of wait-cycles or use the de-asserting edge of $\overline{Rd}$ to end the transaction.

Systems that use 16 or 8-bit ports must in general support mini-burst reads. Memory controllers for such systems can use the de-asserting edge of $\overline{Rd}$ to reset the controller. The memory controller can also look for $\overline{Last}$ to assert. When $\overline{Last}$ asserts, the controller knows that it is handling the final datum of the transaction. It is also possible to decode $\overline{BE(3:0)}$ (if enabled on reads) to determine how many datum are to be returned.

Figure 8.5 shows the timing of the control signals when the read cycle is being terminated.

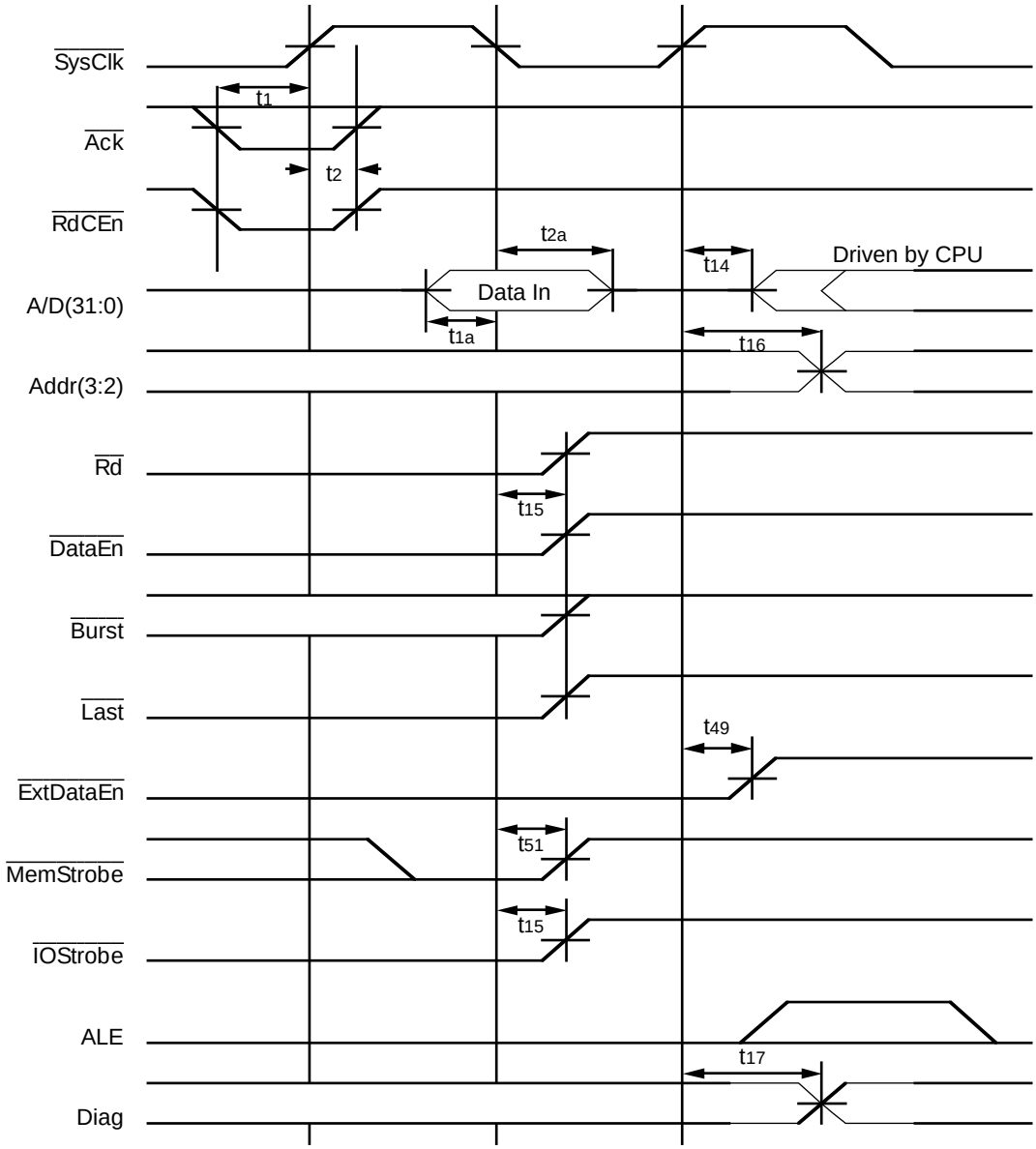


Figure 8.5. Read Cycle Termination

Latency Between Processor Operations

In general, the processor may begin a new bus activity as soon as 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. To simplify design, the processor provides the $\overline{\text{DataEn}}$ 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, as illustrated in Figure 8.6.

The R3041 also adds a new feature to the R30xx family to enable the system designer to lengthen the amount of time available for bus turn-around. The Bus Turn Around control field of the CP0 Bus Control register enables the system designer to extend the minimum guaranteed amount of time available for bus turn-around. This enables the system designer to eliminate some transceiver devices and/or use slower system components, without worrying about bus conflicts.

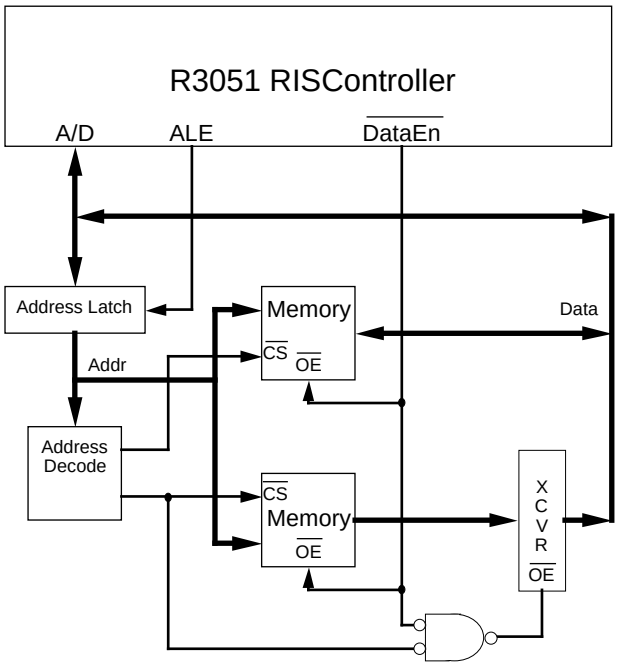


Figure 8.6. Use of $\overline{\text{DataEn}}$ as Output Enable Control

Processor Internal Activity

In general, the processor will execute stall cycles until $\overline{Ack}$ 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 8.7 illustrates the relationship between $\overline{Ack}$ and the internal activity for a block

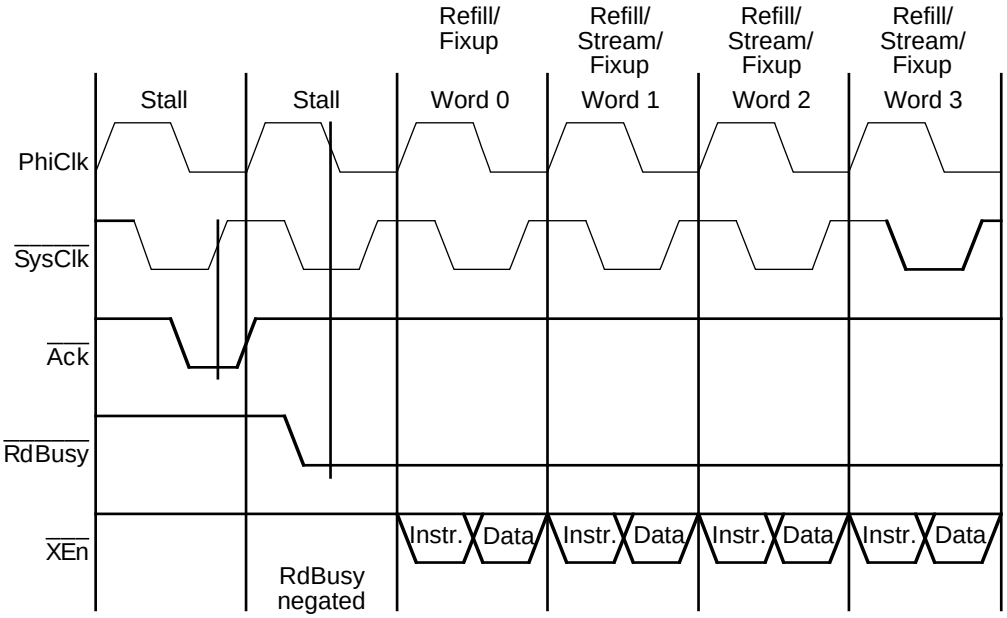


Figure 8.7. Internal Processor States on Burst Read

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 instruction 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 8.8 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.

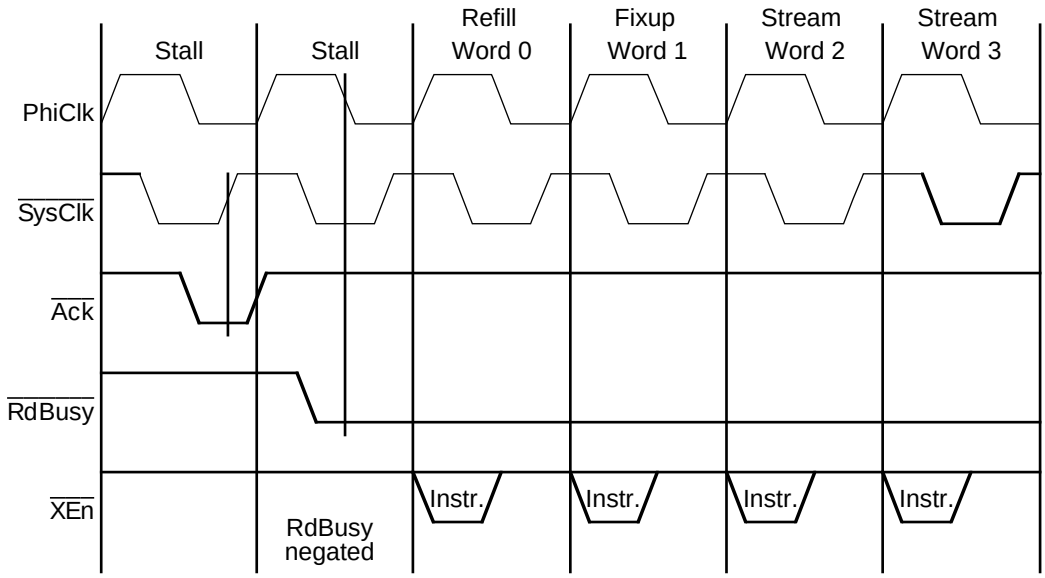


Figure 8.8. Instruction Streaming Example

32-BIT READ TIMING DIAGRAMS

This section illustrates a number of timing diagrams applicable to R3041 32-bit read transactions. These diagrams reference AC parameters whose values are contained in the R3041 data sheet. Note that these timing diagrams assume MemStrobe, IOStrobe, and ExtDataEn are all enabled for read operations and that the ExtAddrHold reset configuration mode is enabled.

Table 8.1 shows the kinds of reads of 32-bit wide memory that occur for various conditions of the processor.

Internal Activity	Read Size
Instruction Cache Miss	Quad Word
Uncached Instruction Fetch	Single Word
Data Cache Miss	Single or Quad Word (depends on DBR setting)
Uncached Data Fetch	Single Word

Table 8.1. 32-Bit Reads Resulting from Internal Processor Activity

Single Word Reads

Figure 8.9 illustrates the case of a single word read which did not require wait states. Thus, RdCEn and Ack were 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.

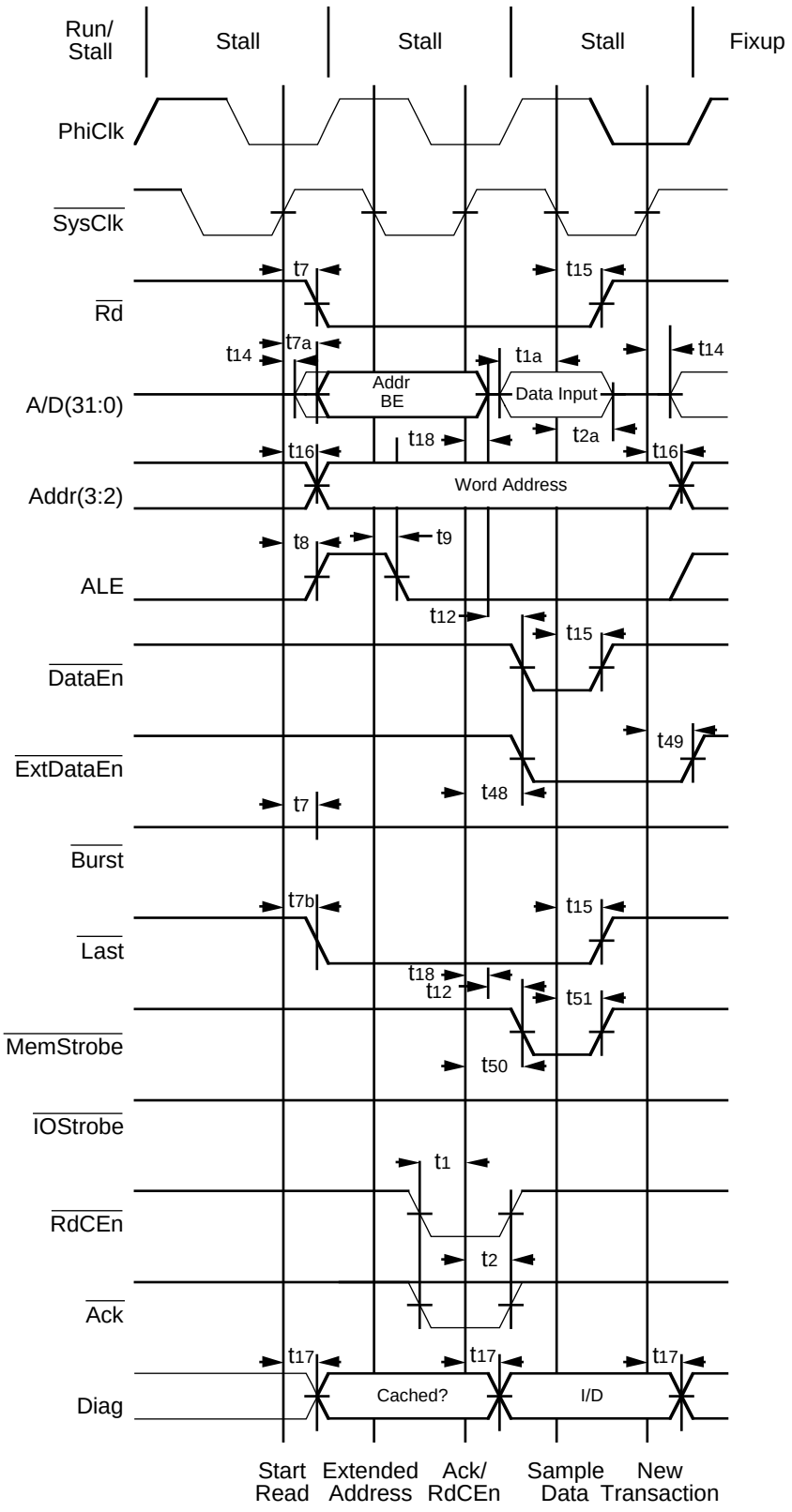


Figure 8.9. Single Word Read Without Bus Wait Cycles

Figure 8.10 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 nor Ack were asserted. On the third rising edge of SysClk, RdCEn was asserted. Ack should also be asserted at this time to optimally restart the pipeline.

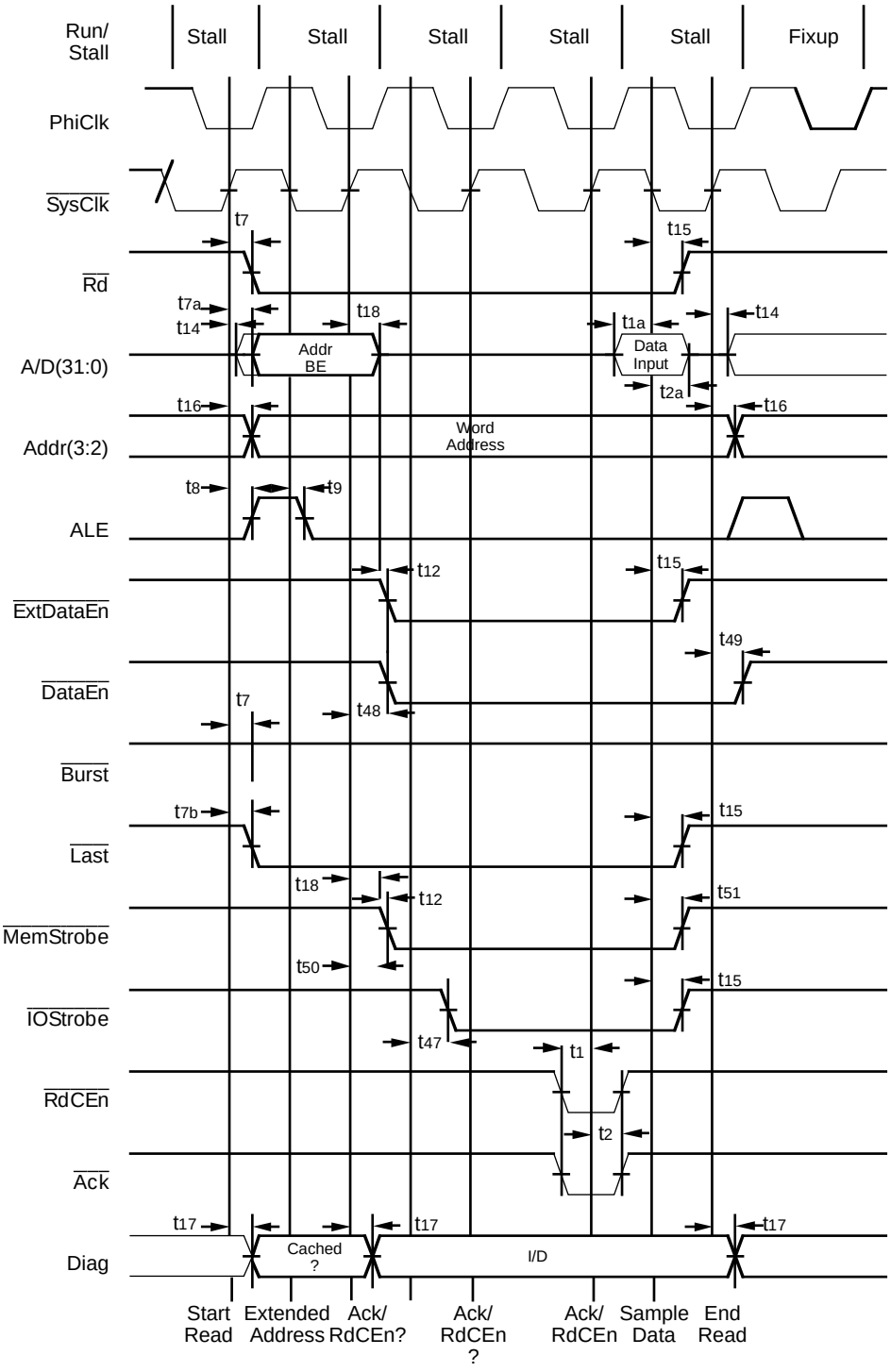


Figure 8.10. Single Word Read With Bus Wait Cycles

Block Reads

Figure 8.11 illustrates the absolute fastest 4 word 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. In this example, $\overline{Ack}$ can be returned simultaneously with the first $\overline{RdCEn}$, to minimize the number of processor stall cycles.

Although $\overline{Ack}$ is brought in 3 clocks before the last $\overline{RdCEn}$, a number of clock cycles are required before the processor negates the $\overline{Rd}$ control output. Thus, the system designer is assured that $\overline{Rd}$ remains active as long as the processor continues to expect data.

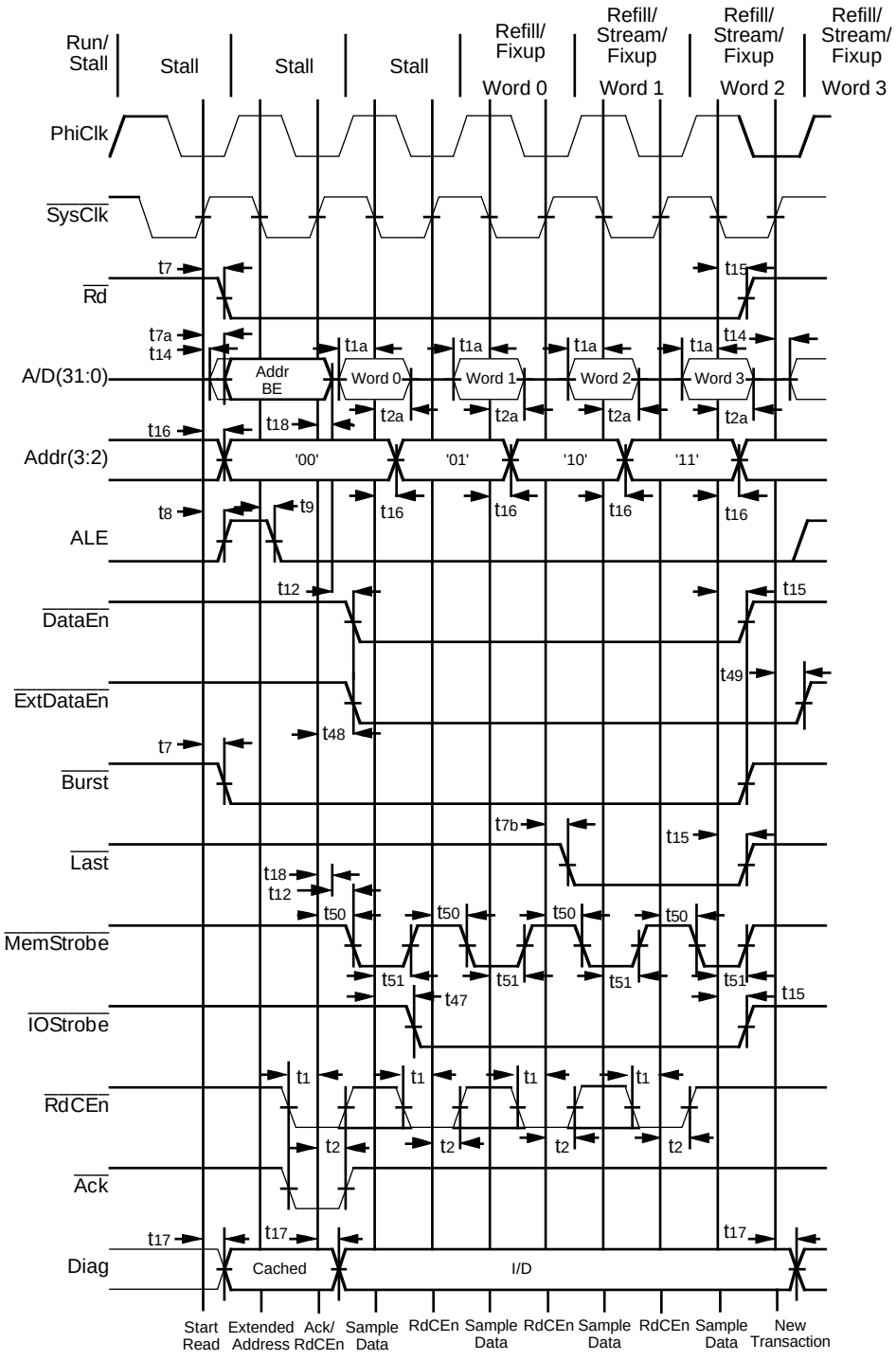


Figure 8.11. Burst Read With No Wait Cycles

Figure 8.12(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 3 clocks before the last $\overline{RdCEn}$. Figure 8.12(a) illustrates the start of the block read, including initial wait cycles to the first word; Figure 8.12(b) illustrates the activity which occurs as data is brought onto the chip and the read is terminated.

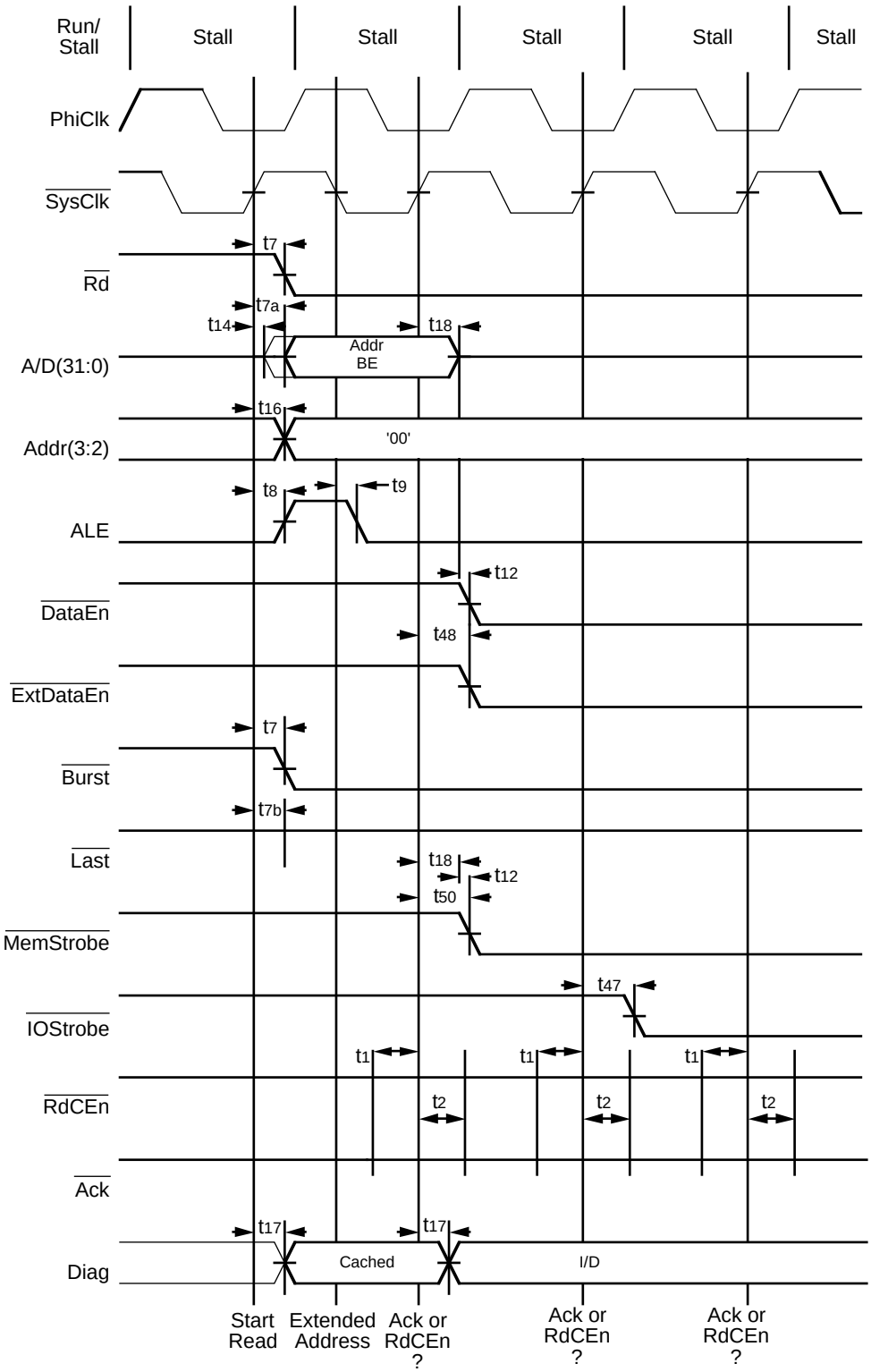


Figure 8.12(a). Start of Burst Read With Initial Wait Cycles

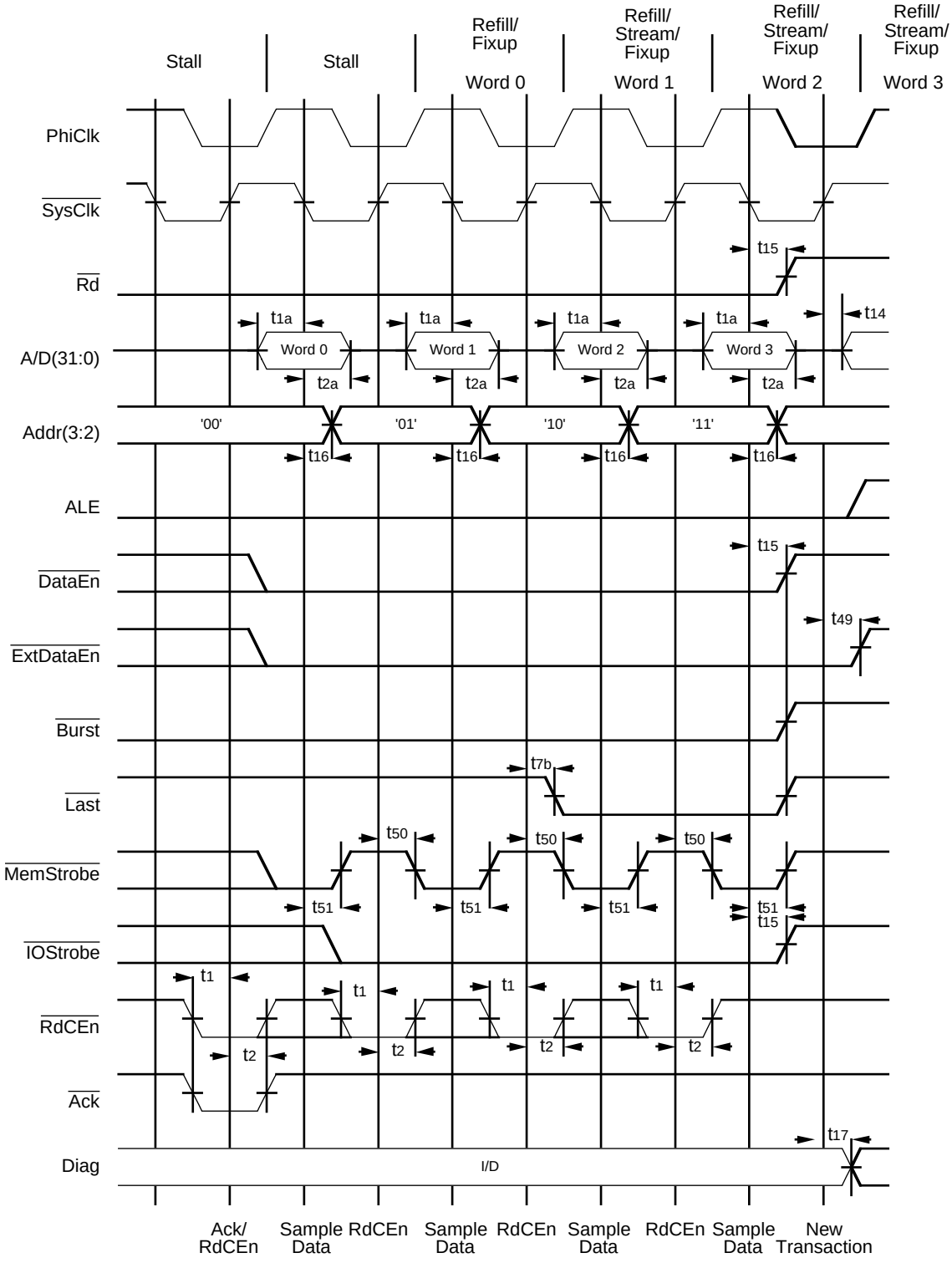


Figure 8.12(b). End of Burst Read

Figure 8.13(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 8.13(a) illustrates the first two words of the block being brought on chip; Figure 8.13(b) illustrates the last two words of the read, including the optimum timing of $\overline{Ack}$, and the negation of the read control signals.

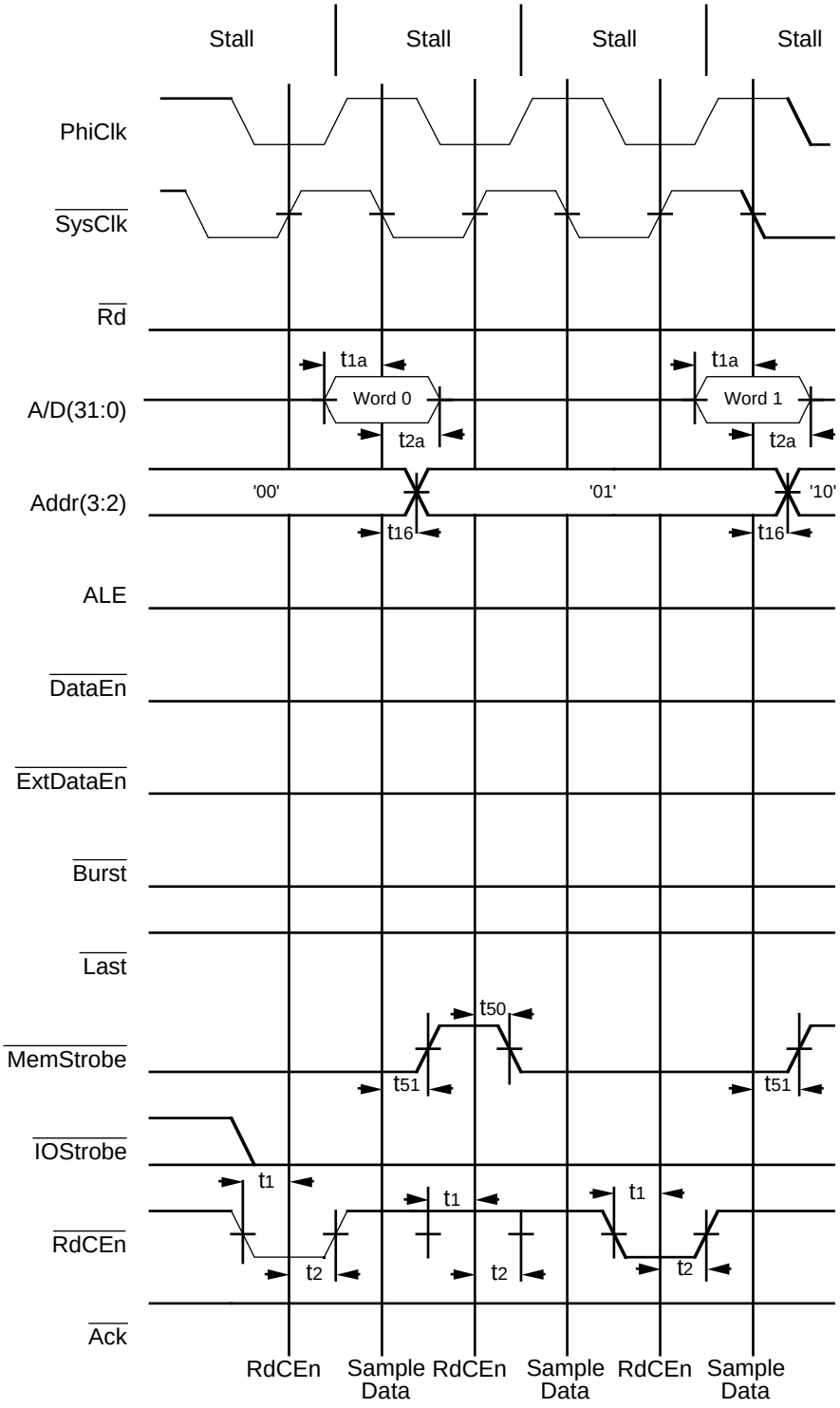


Figure 8.13(a). First Two Words of Throttled Quad Word Read

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 8.13(b), the memory system should return $\overline{Ack}$ five cycles prior to when the execution core requires the fourth word, which is the equivalent of three cycles prior to the last $\overline{RdCEn}$. 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.

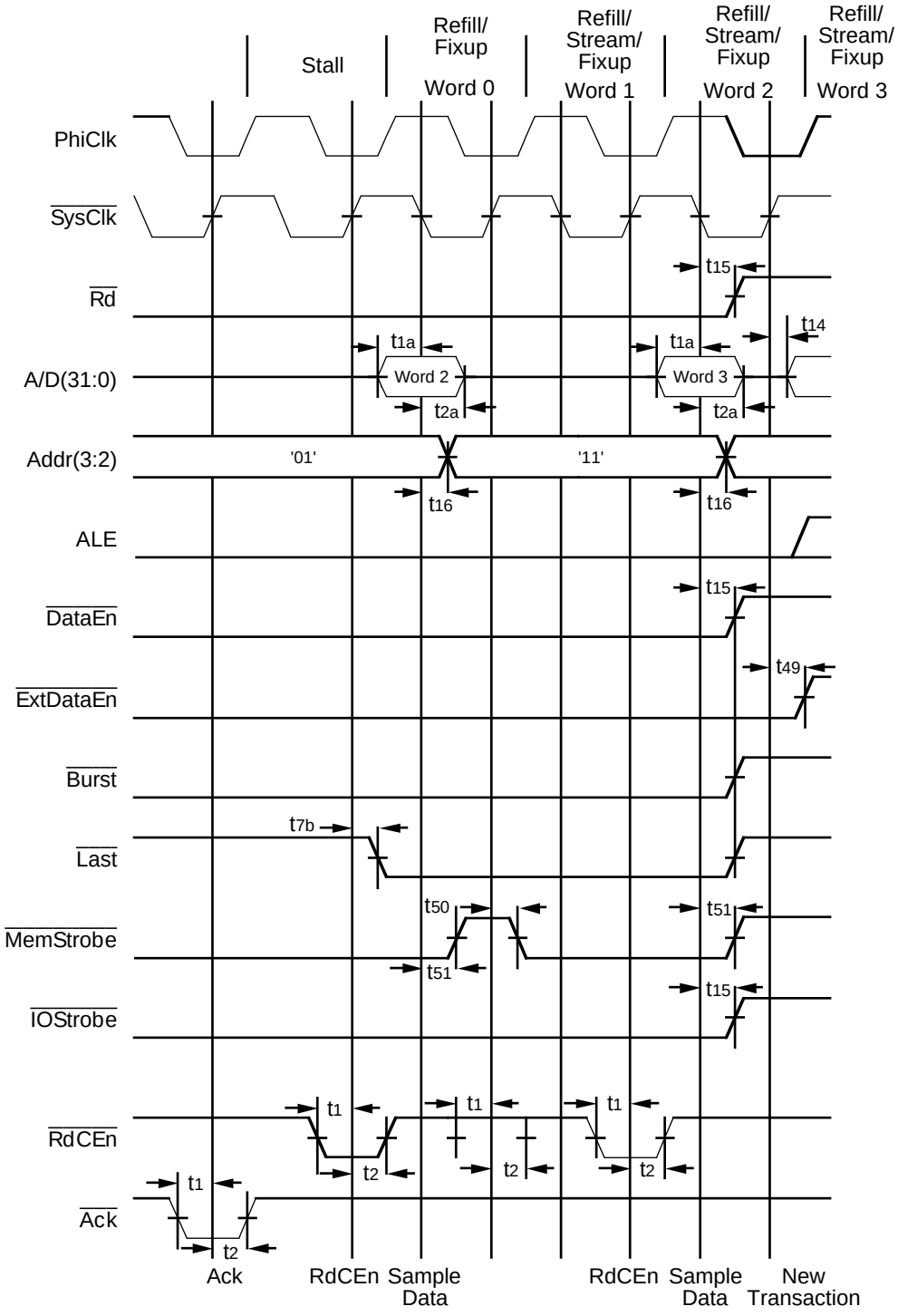


Figure 8.13(b). End of Throttled Quad Word Read

Bus Error Operation

Figure 8.14 is a modified version of Figure 8.10 (single word read with wait cycles), in which $\overline{\text{BusError}}$ is used to terminate the read cycle. In this diagram, note that $\overline{\text{RdCEn}}$ 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, $\overline{\text{BusError}}$ can be asserted anytime up until $\overline{\text{Ack}}$ is asserted. If $\overline{\text{BusError}}$ and $\overline{\text{Ack}}$ are asserted simultaneously, the $\overline{\text{BusError}}$ will be processed; if $\overline{\text{BusError}}$ is asserted after $\overline{\text{Ack}}$ is sampled, it will be ignored.

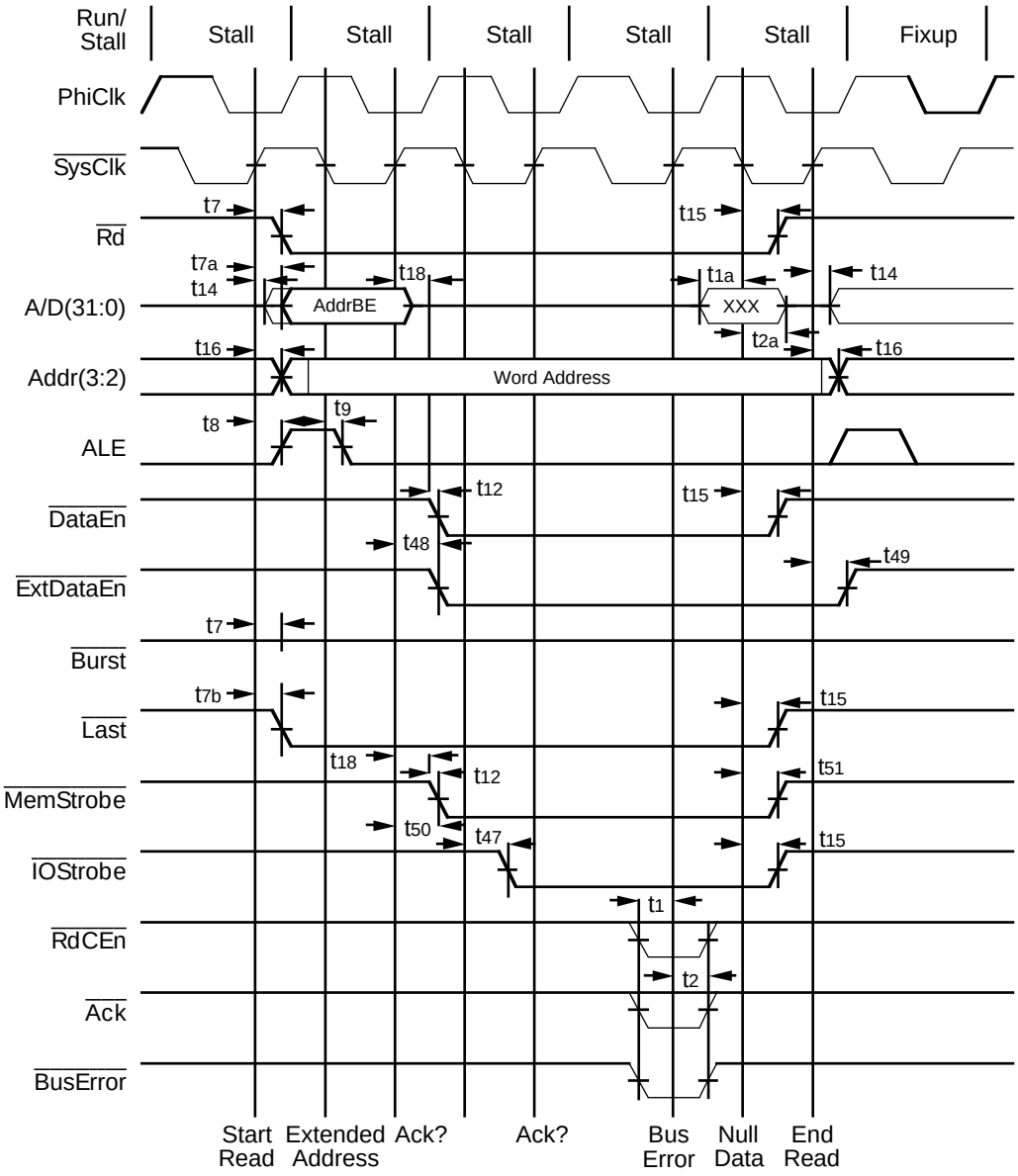


Figure 8.14. Single Word Read Terminated by Bus Error

Figure 8.15 shows the impact of $\overline{\text{BusError}}$ on block reads. The assertion of $\overline{\text{BusError}}$ is allowed up until the assertion of $\overline{\text{Ack}}$. Once $\overline{\text{BusError}}$ is asserted (sampled on a rising edge of $\overline{\text{SysClk}}$), 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 $\overline{\text{RdCEn}}$ at this time, since a late $\overline{\text{RdCEn}}$ may be erroneously detected as part of a subsequent read. Note that if $\overline{\text{BusError}}$ and $\overline{\text{Ack}}$ are asserted simultaneously, $\overline{\text{BusError}}$ processing will occur. If $\overline{\text{BusError}}$ is asserted after $\overline{\text{Ack}}$, the $\overline{\text{BusError}}$ will be ignored.

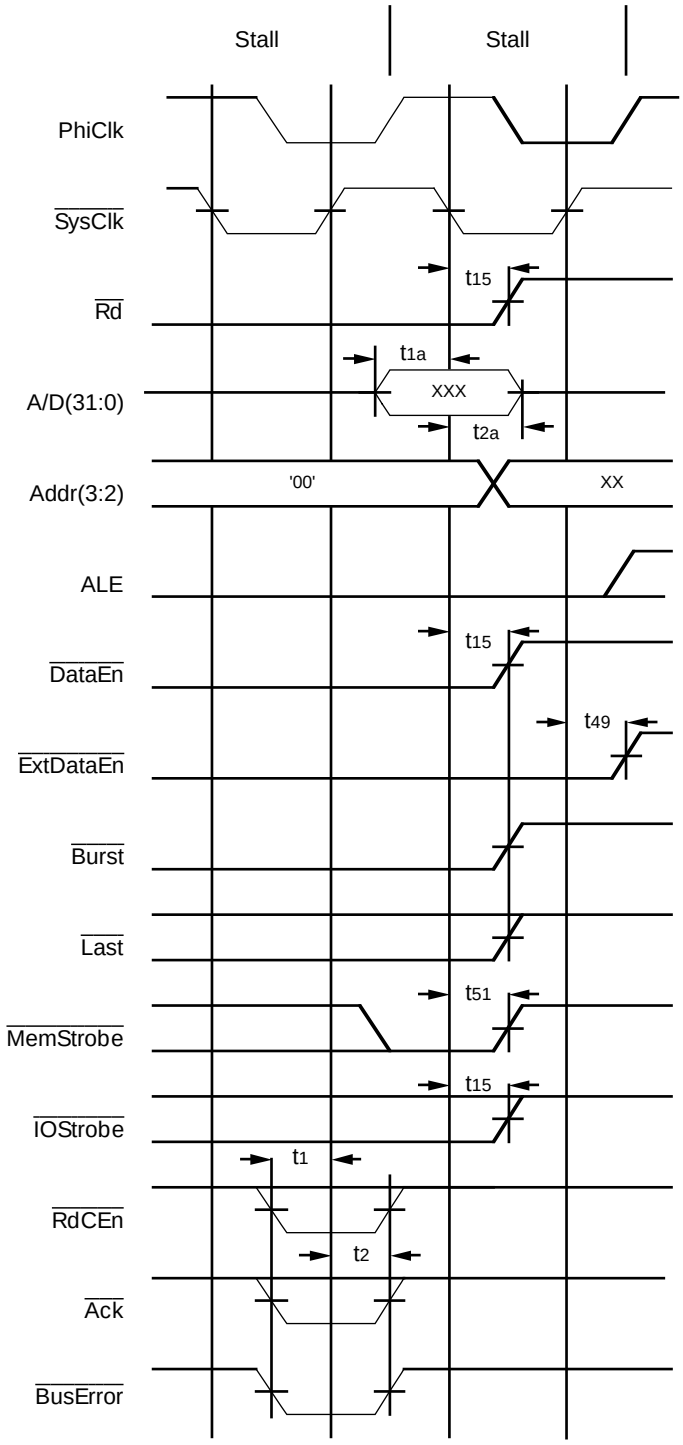


Figure 8.15. Block Read Terminated by Bus Error

16-BIT READ TIMING DIAGRAMS

This section illustrates a number of timing diagrams applicable to R3041 read transactions when a 16-bit port has been selected via the CP0 Port Size register. These diagrams reference AC parameters whose values are contained in the R3041 data sheet.

These timing diagrams assume that $\overline{\text{MemStrobe}}$, $\overline{\text{IOStrobe}}$, and $\overline{\text{ExtDataEn}}$ are enabled for read transactions and that the ExtAddrHold reset configuration mode is enabled.

Also, regardless of the Address 1 value, the half of the A/D bus used during the data phase (A/D(31:16) for big endian or A/D(15:0) for little endian) is constant, according to the system byte ordering (endianness) selected at reset.

Table 8.2 shows the types of reads of 16-bit memory ports, depending on the processor internal activity.

Internal Activity	Read Size
Instruction Cache Miss	Octa Half-word
Uncached Instruction Fetch	Dual Half-word
Data Cache Miss	Dual or Octa Half-word (depends on DBR setting)
Uncached Data Fetch	Single or Dual Half-Word

Table 8.2. 16-Bit Reads Resulting from Internal Processor Activity

Single Halfword Reads

Figure 8.16 illustrates the case of a single halfword read which did not require wait states. Thus, $\overline{\text{RdCEn}}$ and $\overline{\text{Ack}}$ were detected at the rising edge of $\overline{\text{SysClk}}$ which occurred exactly one clock cycle after the rising edge $\overline{\text{SysClk}}$ which asserted $\overline{\text{Rd}}$. Data was sampled one phase later, and $\overline{\text{Rd}}$ and $\overline{\text{DataEn}}$ disabled from that falling edge of $\overline{\text{SysClk}}$. Thus, the execution core required three stall cycles and a fixup to process the internal data. In the cases where only one byte of data is needed, the 16-bit byte enables, $\overline{\text{BE16}}$(1:0) indicate which bytes are being used in this transaction.

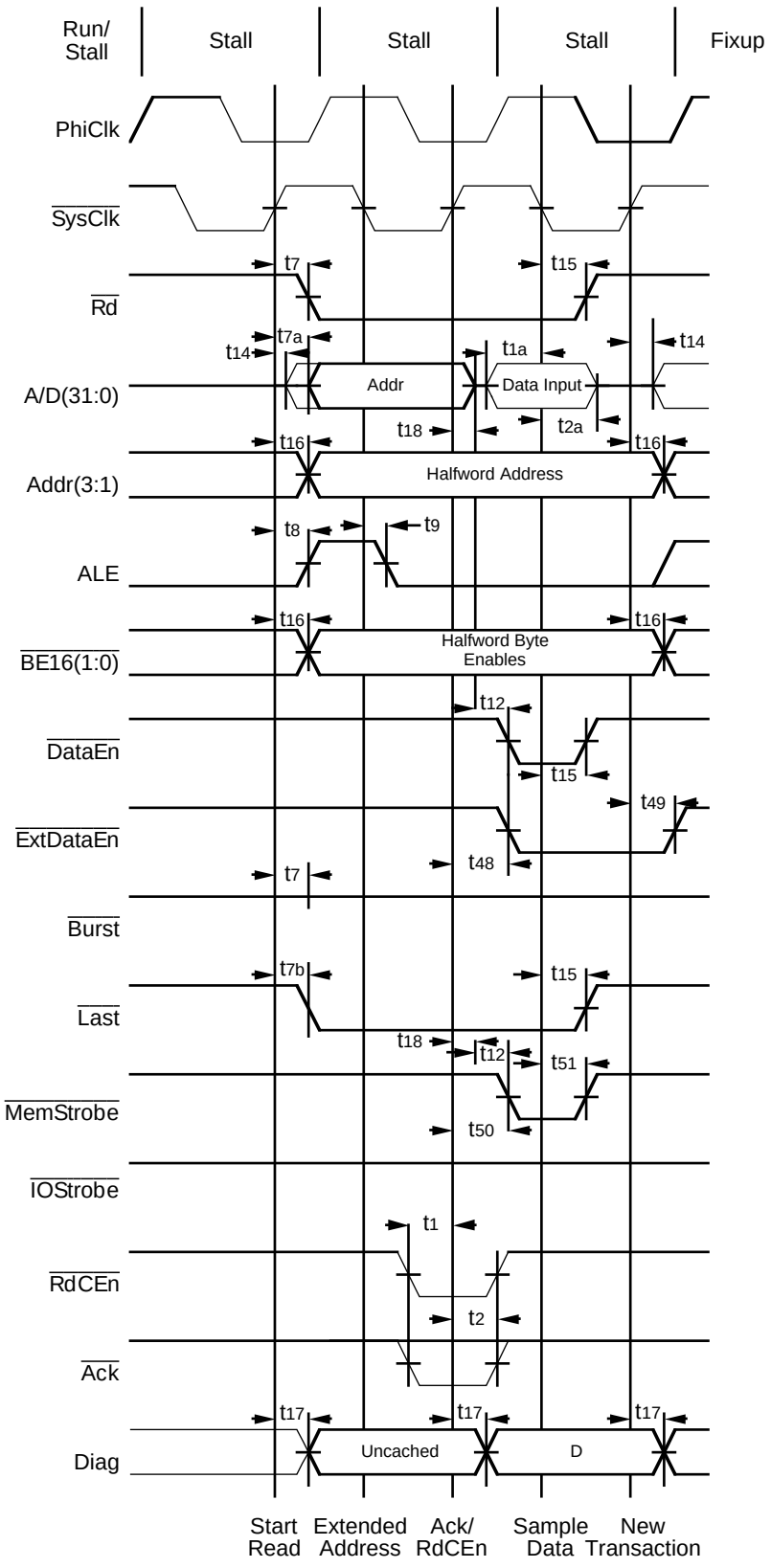


Figure 8.16. Single Halfword Read Without Bus Wait Cycles

Figure 8.17 also illustrates the case of a single halfword read. However, in this figure, one bus wait cycle is required before the data is returned. Thus, one rising edge of SysClk occurred where neither RdCEn or Ack were asserted. On the second rising edge of SysClk, RdCEn was asserted. The timing of Ack in a single datum read should occur with the final RdCEn in order to optimally restart the internal pipeline.

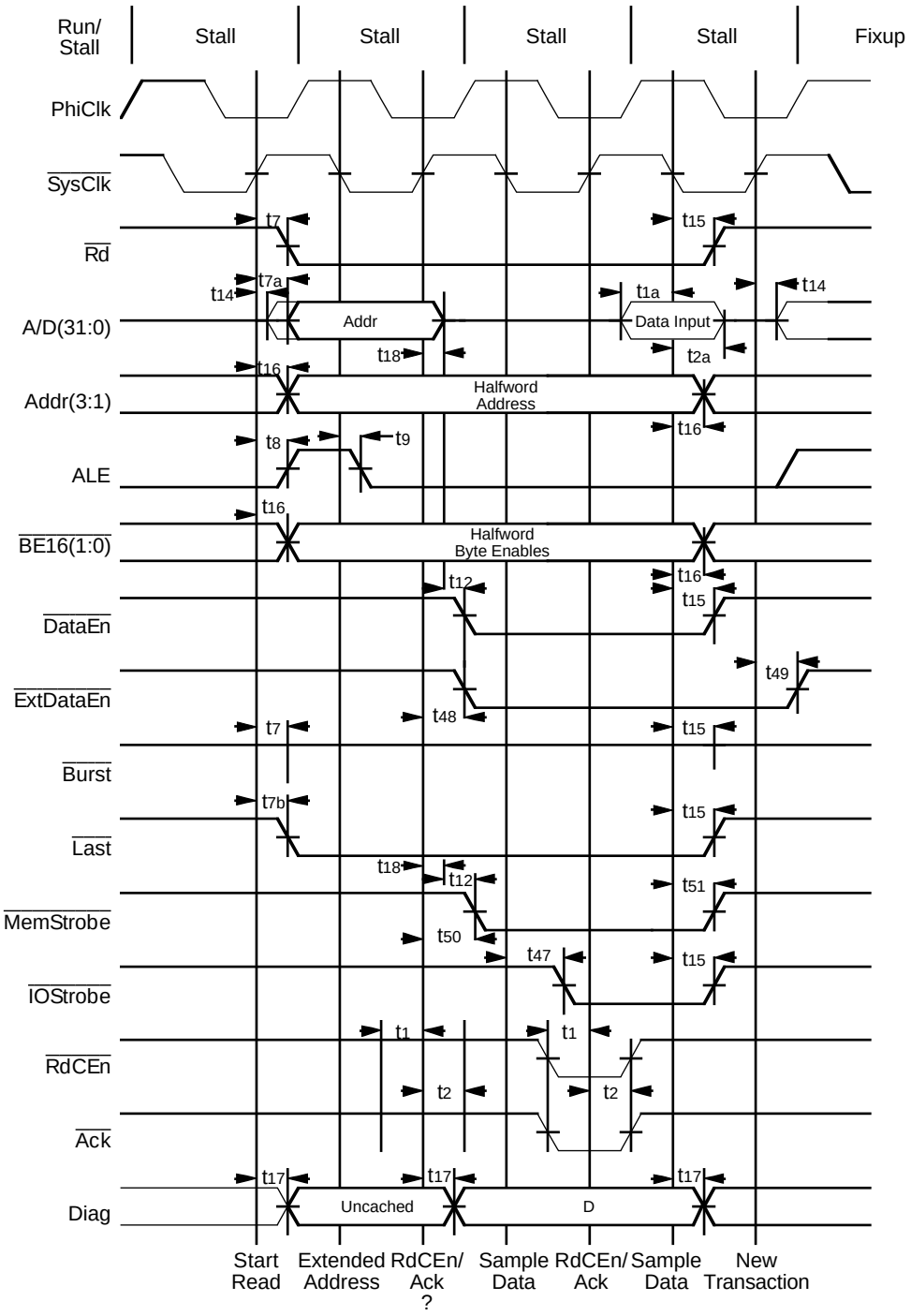


Figure 8.17. Single Halfword Read With Bus Wait Cycle

Mini-Burst Halfword Reads

Mini-burst halfword reads require two halfwords to be returned within the same read cycle as in Figure 8.18. After the second halfword is read, $\overline{Rd}$ will de-assert. Alternatively, external wait state machine controllers can find the start of the final halfword of the mini-burst as indicated by the assertion of $\overline{Last}$. In a mini-burst, the $\overline{Burst}$ line remains de-asserted, since $\overline{Burst}$ is only used to indicate an octi(8) halfword read corresponding to a four word block. Note that during either of the halfwords in a mini-burst may have both or just one of its byte enable, $\overline{BE16}$(1:0) signals asserted. These three cases correspond to instructions which generate tri-byte (addresses 0,1,2 or 1,2,3) and word (addresses 0,1,2,3) loads or fetches.

The timing of $\overline{Ack}$ in a mini-burst read should occur with the final $\overline{RdCEn}$ in order to optimally restart the internal pipeline.

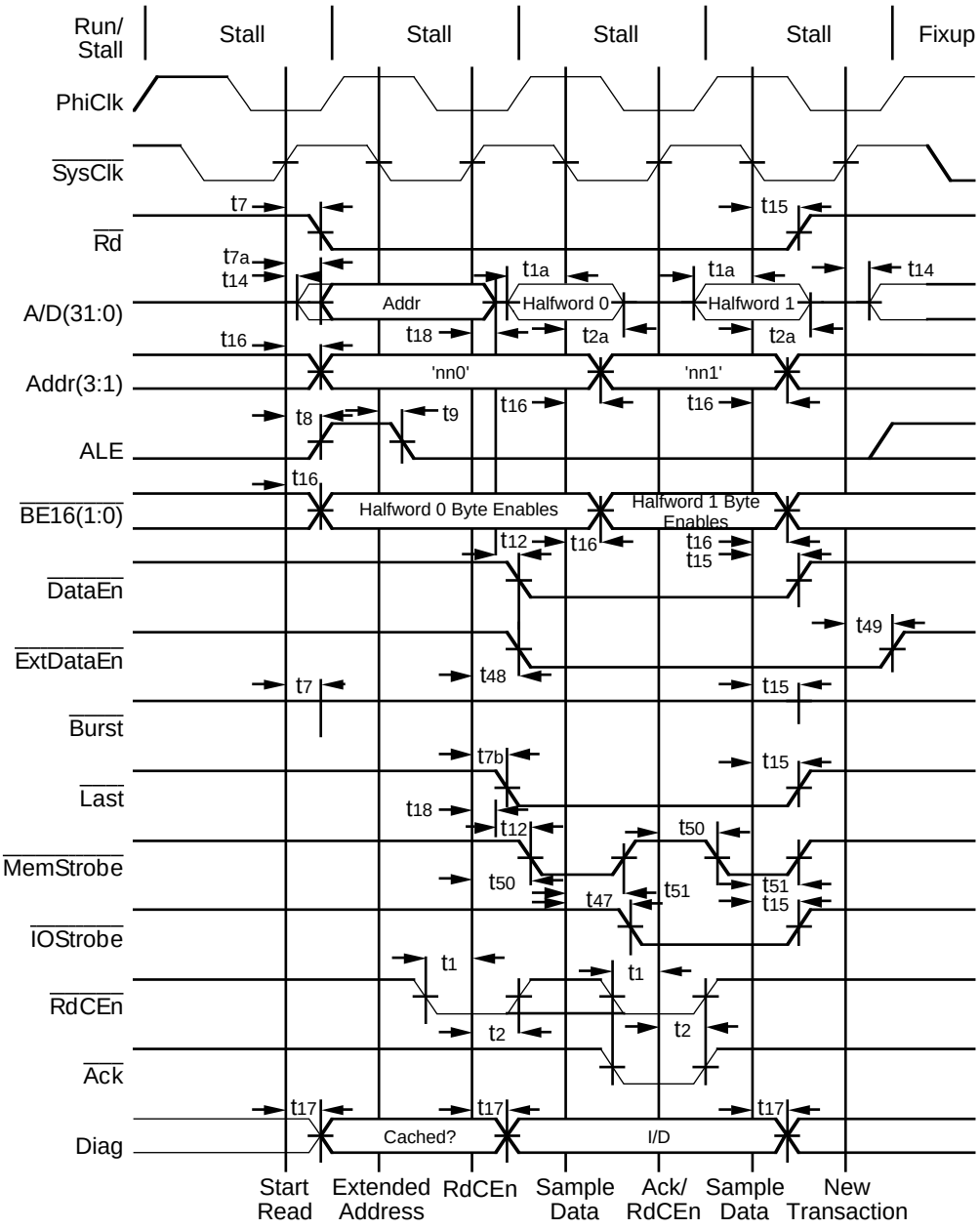


Figure 8.18. Mini-Burst Halfword Read Without Bus Wait Cycles

16-Bit Block Reads

16-bit block reads involve a total of 8 halfwords of data. Figure 8.19(a) illustrates the beginning of the absolute fastest halfword block read. Figure 8.19(b) illustrates the ending of the absolute fastest halfword block read. The first halfword of the block is returned in the second cycle of the read; each additional halfword is returned in the immediately subsequent clock cycles. Thus, $\overline{Ack}$ can be returned on the 3rd clock prior to the last $\overline{RdCEn}$, to minimize the number of processor stall cycles.

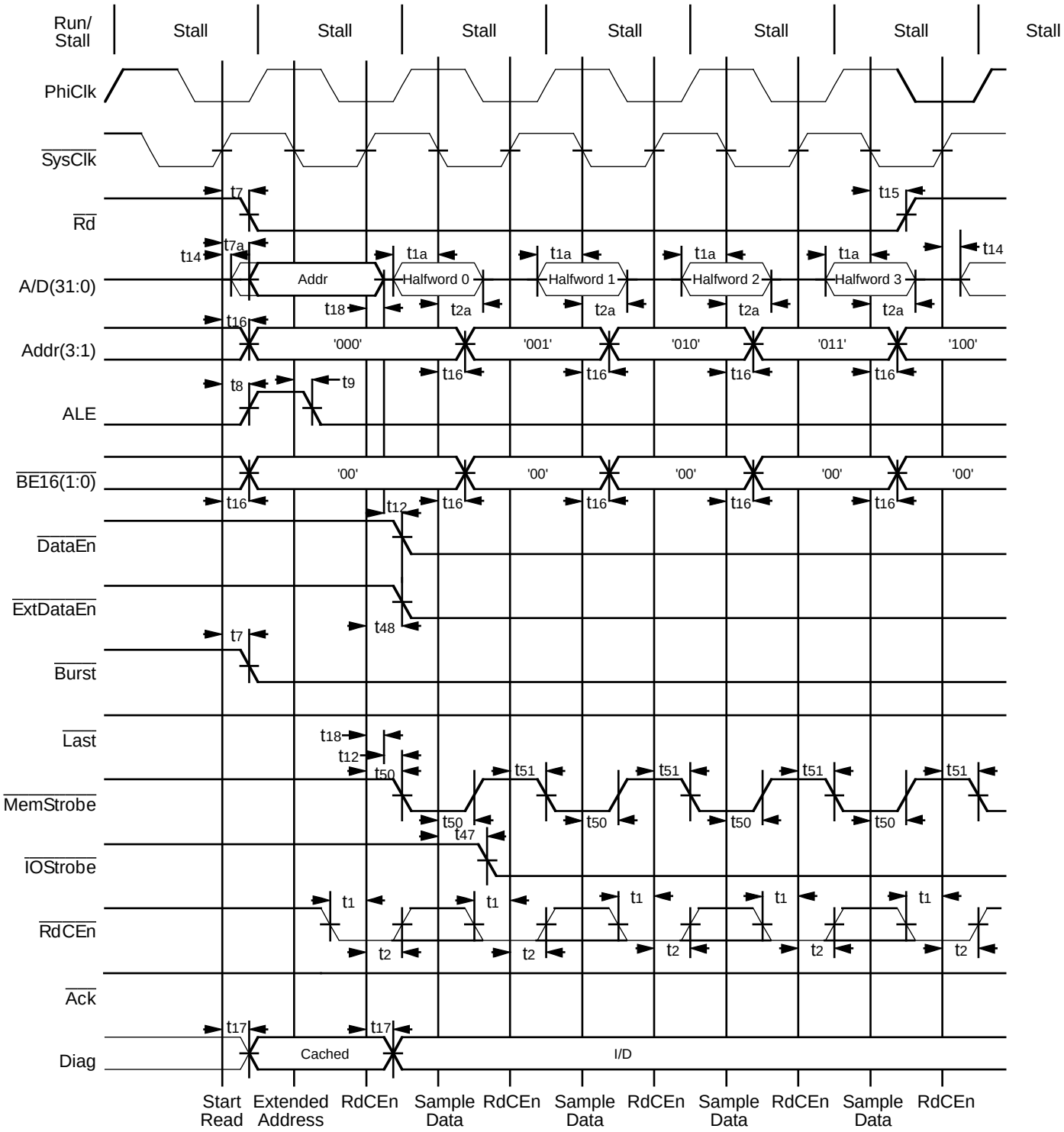


Figure 8.19(a). Start of Burst Block Halfword Read Without Bus Wait Cycles

Note that although $\overline{Ack}$ is brought low in the 3rd clock from the end clock cycle, a number of clock cycles are required before the processor negates the $\overline{Rd}$ control output. Thus, the system designer is assured that $\overline{Rd}$ remains active as long as the processor continues to expect data.

Halfword block reads can insert bus wait cycles just like the 32-bit block reads. Thus bus wait cycles can be inserted before the first halfword and/or between subsequent halfwords simply by delaying the assertion of $\overline{RdCEn}$ until the data is ready. In these cases, $\overline{Ack}$ must be timed so that the pipeline restarts in time to read the last halfword. Thus the optimal placement of $\overline{Ack}$ is no sooner than the 3rd clock from the last $\overline{RdCEn}$.

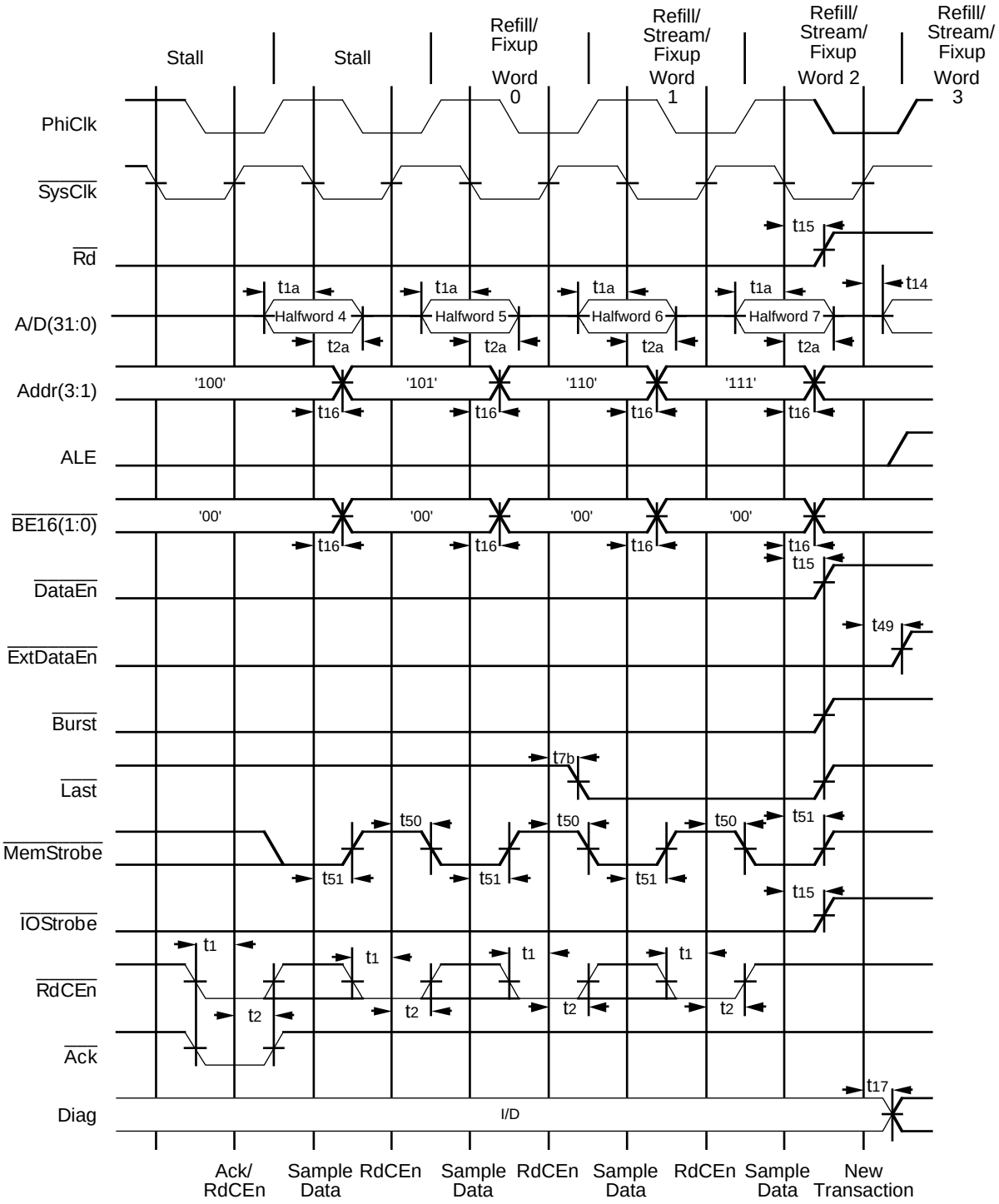


Figure 8.19(b). End of Burst Block Halfword Read Without Bus Wait Cycles

Bus Error Operation

Bus errors for 16-bit halfword ports operate the same as 32-bit bus errors. In single halfword 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.

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 halfwords 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.

8-BIT READ TIMING DIAGRAMS

This section illustrates a number of timing diagrams applicable to R3041 read transactions when an 8-bit port has been selected via the CP0 Port Size register. These diagrams reference AC parameters whose values are contained in the R3041 data sheet.

These diagrams assume that MemStrobe, IOStrobe, and ExtDataEn are enabled for reads and that the ExtAddrHold reset configuration mode is enabled.

The byte lane used for a transfer is not dependent on the address bit 0, but rather on the system byte ordering (endianness) selected at reset. A/D(31:24) is used for big endian systems, and A/D(7:0) is used for little endian systems.

Table 8.3 shows the types of reads from 8-bit memories that result from internal R3041 activity.

Internal Activity	Read Size
Instruction Cache Miss	16 Byte
Uncached Instruction Fetch	4 Byte
Data Cache Miss	4 or 16 Byte (depends on DBR setting)
Uncached Data Fetch	1 to 4 Byte

Table 8.3. 8-Bit Reads Resulting from Internal Processor Activity

Single Halfword Reads

Figure 8.20 illustrates the case of a single byte 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 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.

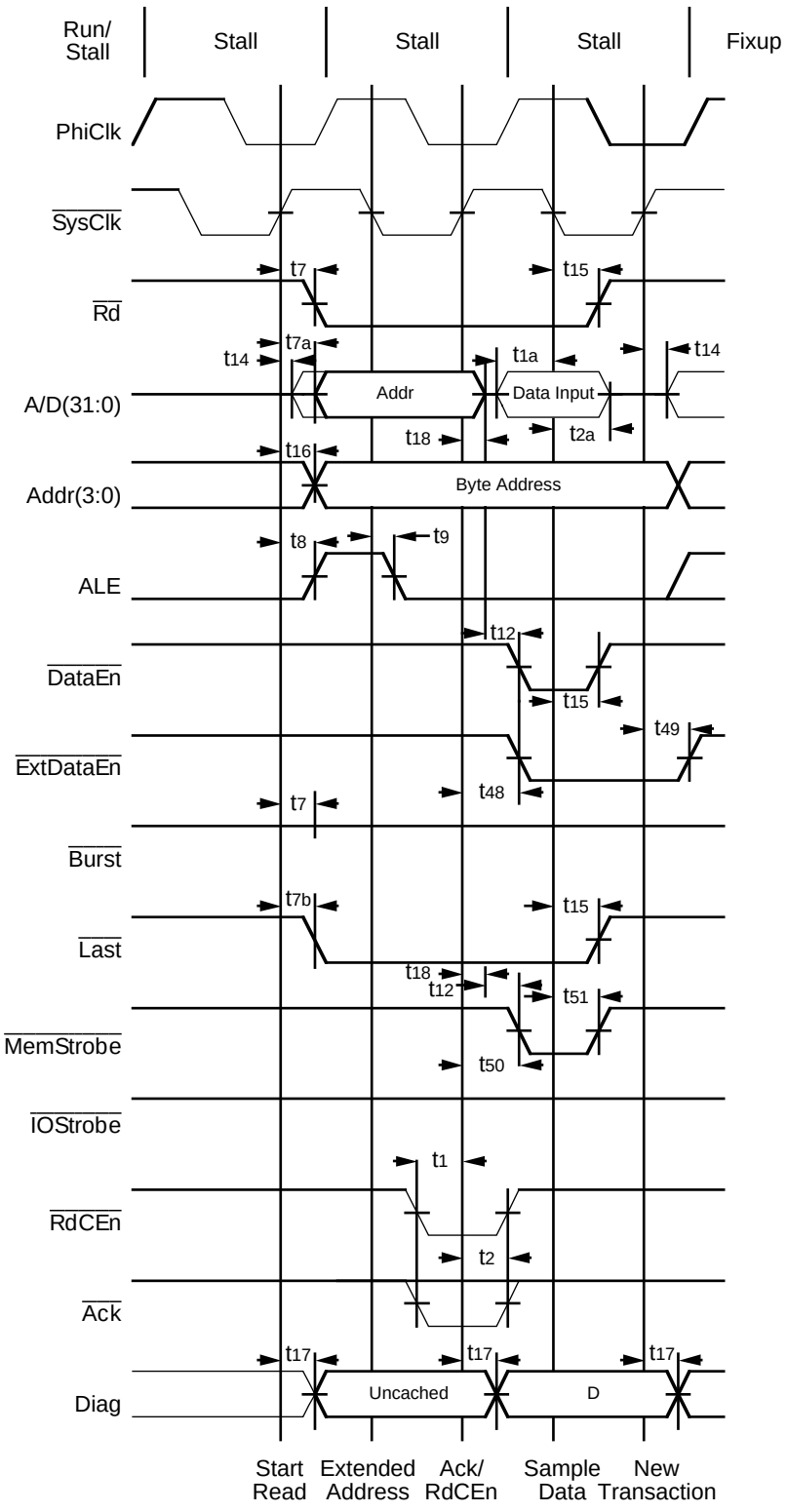


Figure 8.20. Single Byte Read Without Bus Wait Cycles

Figure 8.21 also illustrates the case of a single byte read. However, in this figure, one bus wait cycle was required before the data is 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. The timing of Ack in a single datum read should occur with the final RdCEn in order to optimally restart the internal pipeline.

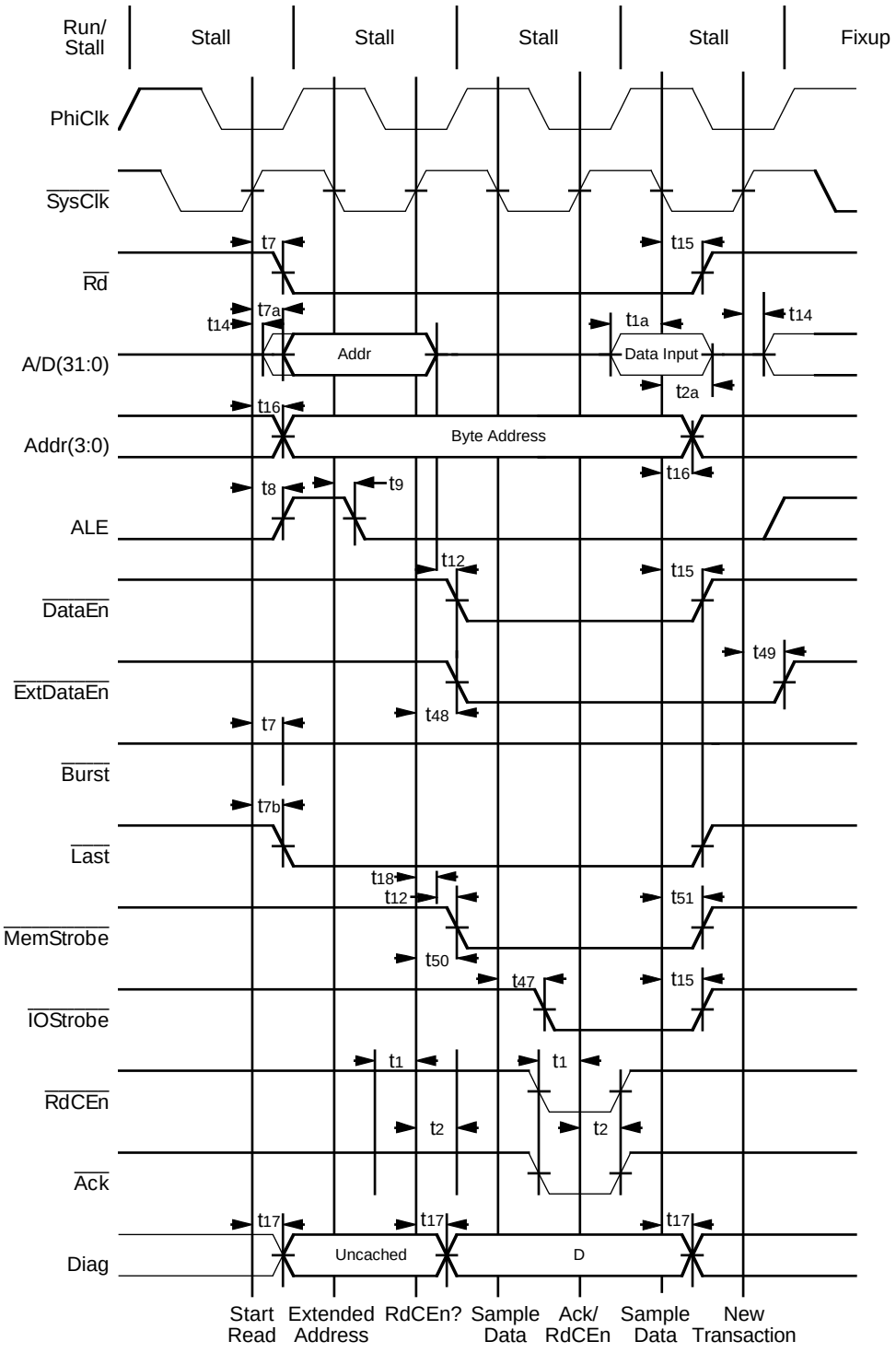


Figure 8.21. Single Byte Read With Bus Wait Cycles

Mini-Burst Byte Reads

Mini-burst byte reads require two, three, or four bytes to be returned within the same read cycle as illustrated in Figures 8.22, 8.23, and 8.24. After the last byte is read, $\overline{Rd}$ will de-assert. Alternatively, external wait state machine controllers can find the start of the final byte of the mini-burst as indicated by the assertion of $\overline{Last}$. In a mini-burst, the $\overline{Burst}$ line remains de-asserted, since $\overline{Burst}$ is only used to indicate a 16 byte block read corresponding to a four word block. Note that the starting address of a mini-burst is not necessarily 0. For example, it could be a '1' if the load or fetch corresponds to a tri-byte operation.

The timing of $\overline{Ack}$ in a mini-burst read should occur with the final $\overline{RdCEn}$ in order to optimally restart the internal pipeline.

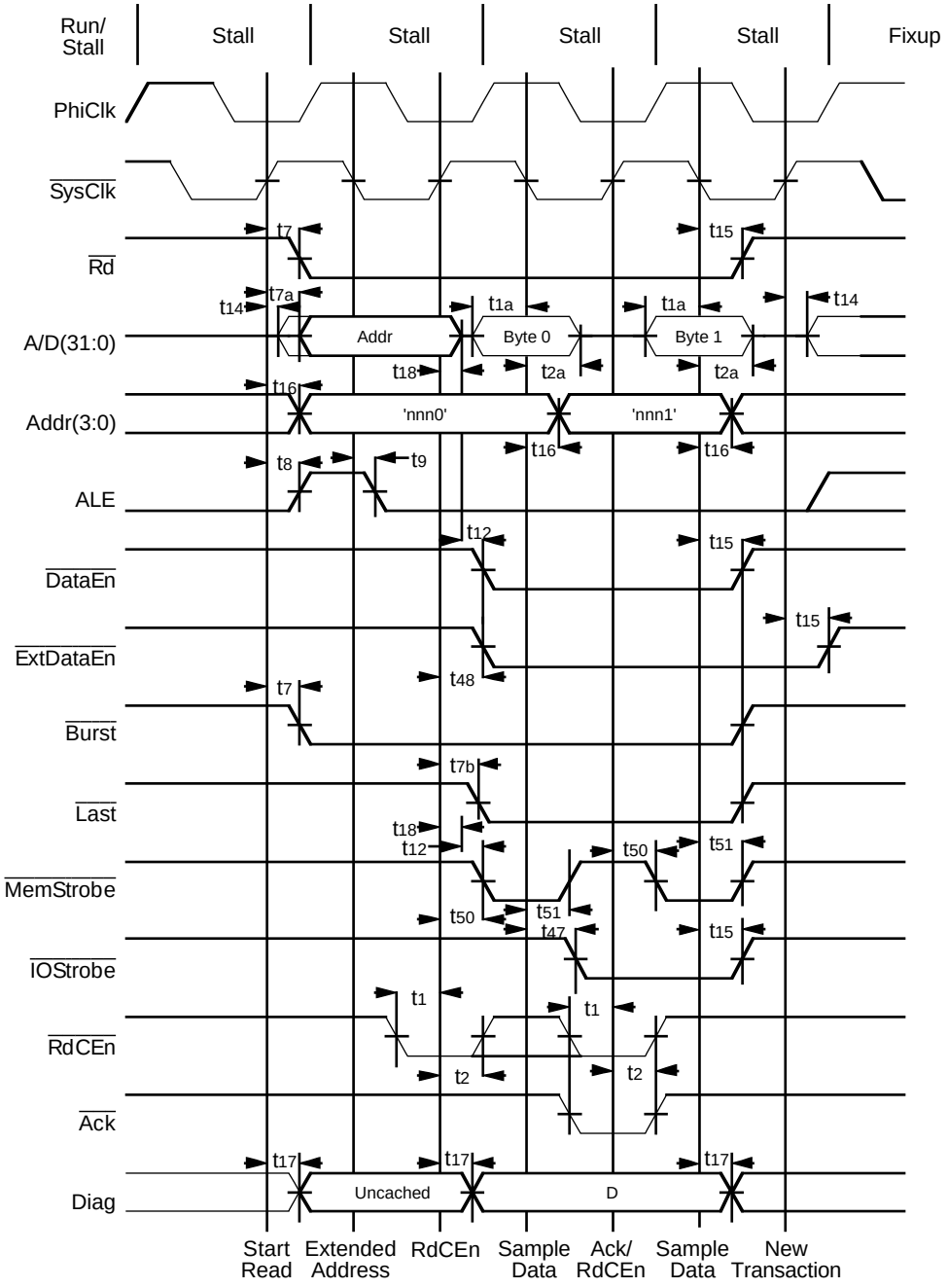


Figure 8.22. Double Byte Read Without Bus Wait Cycles

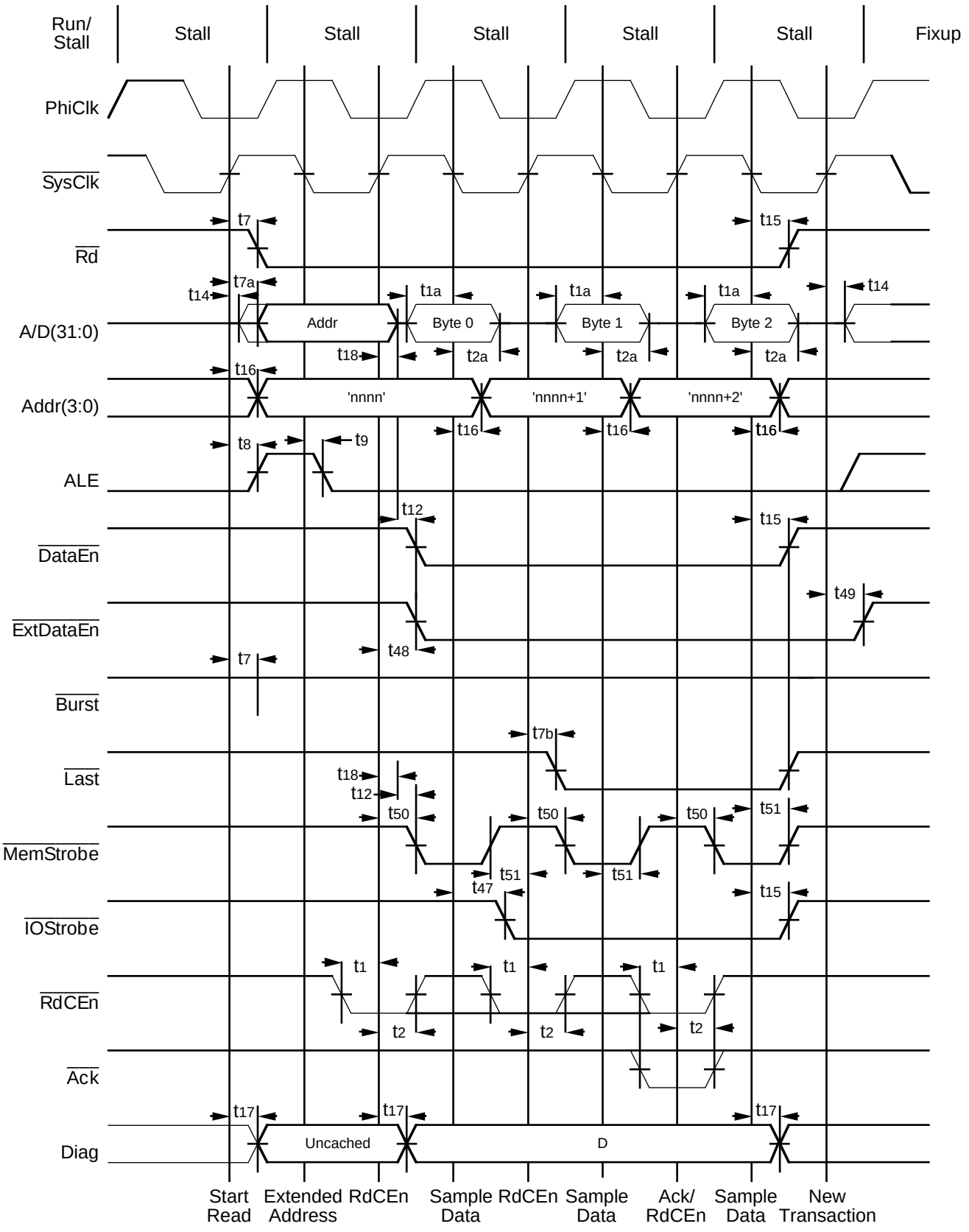


Figure 8.23. Tri-Byte Read Without Bus Wait Cycles

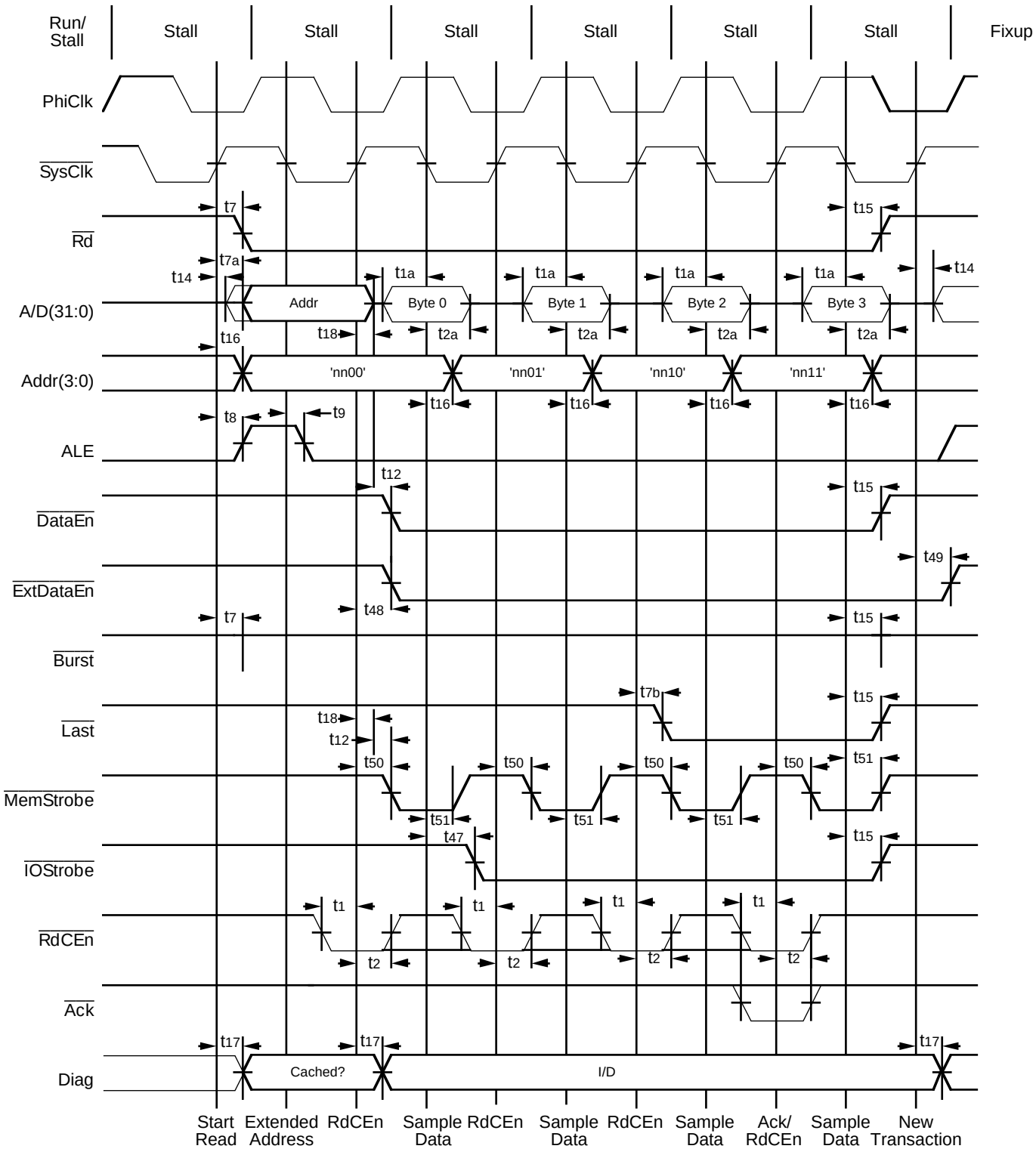


Figure 8.24. Quad-Byte Read Without Bus Wait Cycles

8-Bit Quad Word Reads

8-bit block reads involve a total of 16 bytes of data. Figure 8.25(a) illustrates the beginning of the absolute fastest byte block read. Figure 8.25(b) illustrates the ending of the absolute fastest byte block read. Intervening bytes 5 through 11 are similar. The first byte of the block is returned in the second cycle of the read; each additional byte is returned in the immediately subsequent clock cycles. Thus, $\overline{Ack}$ can be returned on the 3rd clock prior to the last $\overline{RdCEn}$, to minimize the number of processor stall cycles.

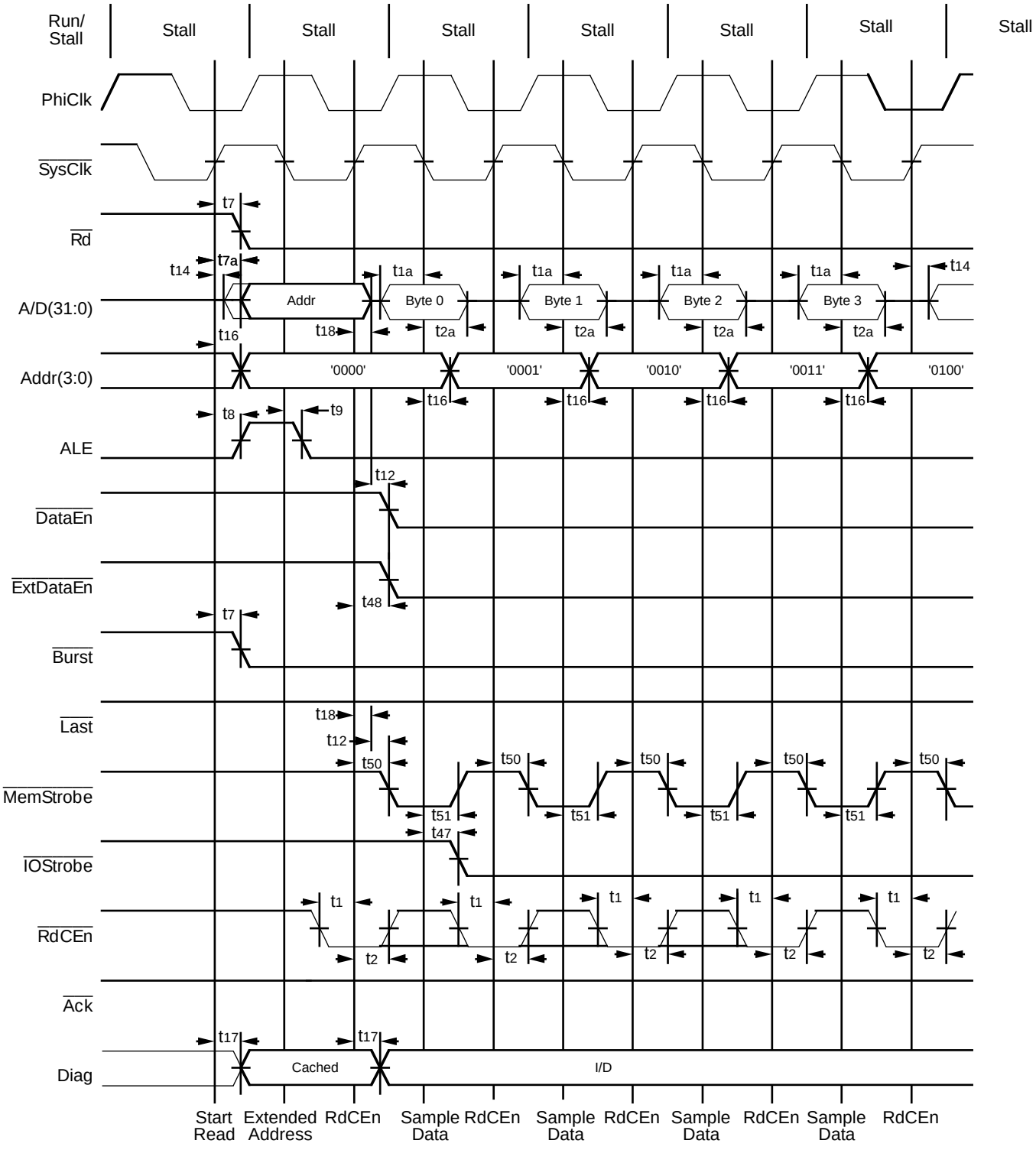


Figure 8.25 (a). Start of 16 Byte Burst Read Without Bus Wait Cycles

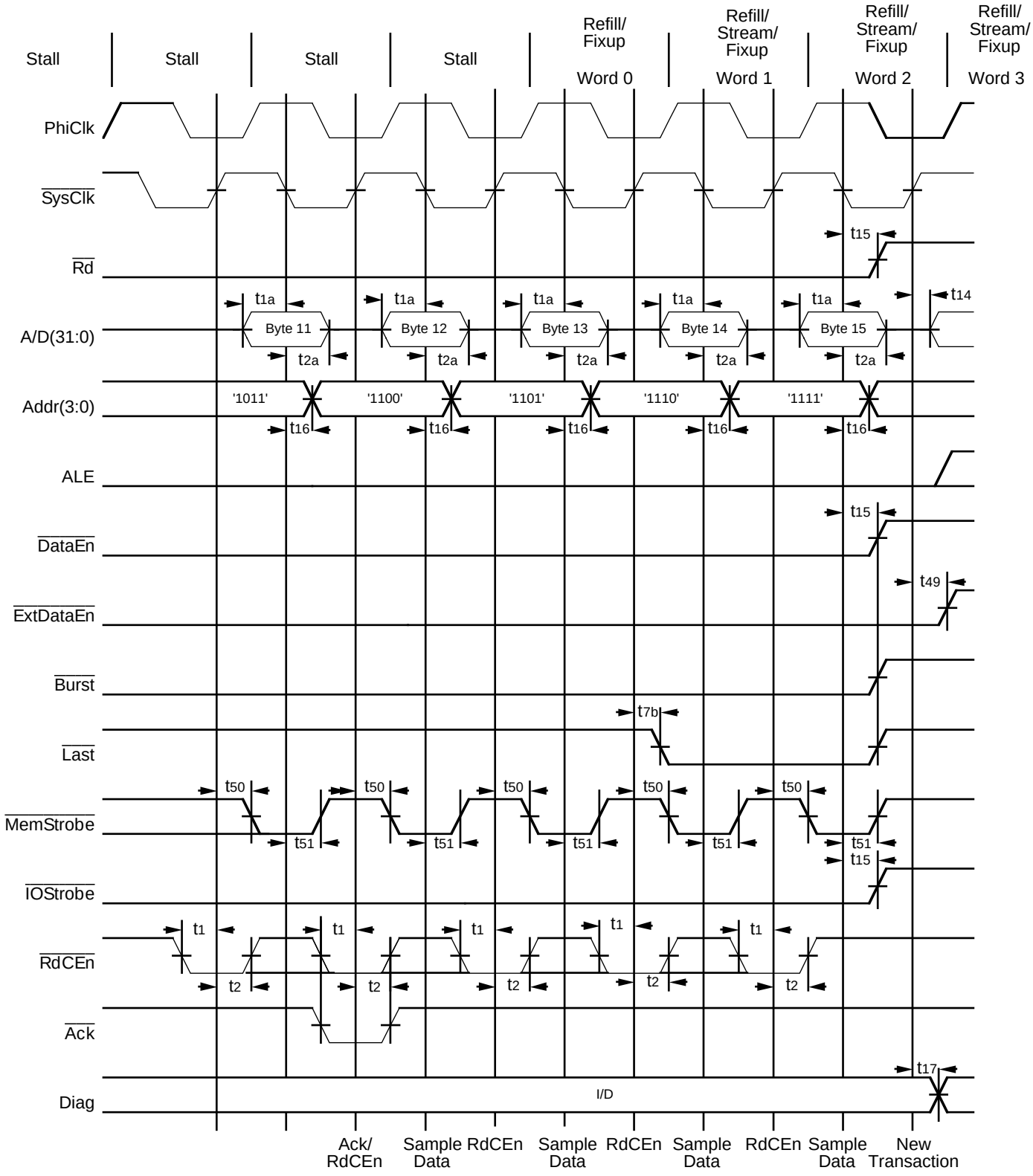


Figure 8.25 (b). End of 16 Byte Burst Read Without Bus Wait Cycles

Note that although $\overline{Ack}$ is brought low in the 3rd clock from the end clock cycle, a number of clock cycles are required before the processor negates the $\overline{Rd}$ control output. Thus, the system designer is assured that $\overline{Rd}$ remains active as long as the processor continues to expect data.

Byte block reads can insert bus wait cycles just like the 32-bit block reads. Thus bus wait cycles can be inserted before the first byte and/or between subsequent bytes simply by delaying the assertion of RdCEn until the data is ready. In these cases, Ack must be timed so that the pipeline restarts in time to read the last byte. Thus the optimal placement of Ack is no sooner than the 3rd clock from the last RdCEn. Note that if Ack is not given at all, an implicit Ack will be generated one clock after the last RdCEn.

Bus Error Operation

Bus errors for 8-bit byte ports operate the same as 32-bit bus errors. In single halfword 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. 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 bytes 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,

INTRODUCTION

The write protocol of the R3041 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 R3041 SYSTEMS

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

Insure that a relatively slow write cycle does not 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 already filled will 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. Thus, a number of design decisions were made: the $\overline{\text{WrNear}}$ signal is provided to allow page mode writes to be used even in simple memory systems; the A/D bus presents the store data as early as the second phase of the first clock cycle of a write; and writes can be performed in as few as two clock cycles.

Although 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	95%
Data Hit Rate	90%

For these assumptions, in 100 instructions, the bus would see:

5 Reads to process instruction cache misses on instruction fetches
10% x 20 = 2 reads to process data cache misses on loads
10 store operations to the write through cache
Total: 7 reads and 10 writes

Thus, in this example, about 60% 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

The R3041 has two basic types of write transactions depending on the port size selected in the CP0 Port Size Configuration register for each memory sub-region. 32-bit ports only use the single word write type. 16-bit ports can use the single halfword write or the mini-burst (double halfword) write type. 8-bit ports can use the single byte write or the mini-burst (double, tri, or quad byte) write type.

Types of 32-Bit 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 32-bit write activity is seen at the bus interface as single write transactions. There is no such thing as a "four word block burst write"; the processor performs a word or sub-word write as a single autonomous bus transaction; however, the $\overline{\text{WrNear}}$ 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.

Uncached writes which contain only 1, 2, or 3 bytes of data assert the appropriate byte enables, $\overline{\text{BE(3:0)}}$ during the address phase. Thus, there really is only one type of 32-bit write transaction. However, the memory system may elect to take advantage of the assertion of $\overline{\text{WrNear}}$ 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 32-bit 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. $\overline{\text{WrNear}}$ may be used to reduce the latency of successive write operations.

The R3041 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.

Types of 16-Bit Transactions

When the R3041 uses a 16-bit port, it does its writes in halfword size increments. Thus if the data contains 8 or 16 bits (1 or 2 bytes), it will be handled with a single halfword write with the appropriate byte enables, $\overline{\text{BE16(1:0)}}$ asserted. If the data contains 24 or 32 bits (3 or 4 bytes), it will handled with a double halfword write mini-burst with the appropriate byte enables, $\overline{\text{BE16(1:0)}}$ for each halfword asserted. A mini-burst puts both halfwords out in the same write transaction. The memory system simply returns an $\overline{\text{Ack}}$ for each halfword datum which will automatically increment Addr(3:1) and change $\overline{\text{BE16(1:0)}}$ if appropriate. Similar to a read mini-burst, a write mini-burst can be detected using the $\overline{\text{Last}}$ signal to determine when the final halfword datum is being returned or by using the de-assertion of the $\overline{\text{Wr}}$ line. The R3041 is designed to accommodate a wide variety of different memory bandwidths, including DRAM systems that need precharge wait cycles for the first halfword and then use a fast page mode access for bursting the second halfword.

The data lines used in 16-bit ports are always A/D(31:16) for big endian systems and A/D(15:0) for little endian systems. This is regardless of the Reverse Endianess bit in the CP0 Status register. For big endian systems, $\overline{\text{BE16(1)}}$ corresponds to the byte lane in A/D(31:24) and $\overline{\text{BE16(0)}}$ corresponds

to A/D(23:16). Similarly, for little endian systems, $\overline{BE16(1)}$ corresponds to the byte lane in A/D(15:8) and $\overline{BE16(0)}$ corresponds to A/D(7:0).

Types of 8-Bit Transactions

When the R3041 uses an 8-bit port, it does its writes in byte size increments. Thus if the data contains 1 byte, it will be handled with a single byte write. If the data contains 2, 3, or 4 bytes, it will handled with a double, tri, or quad byte write mini-burst, respectively. A mini-burst puts 2, 3, or 4 bytes out in the same write transaction. The memory system simply returns an $\overline{Ack}$ for each byte datum which will automatically increment Addr(3:0). Similar to a read mini-burst, a write mini-burst can be detected using the $\overline{Last}$ signal to determine when the final byte datum is being returned or by using the de-assertion of the $\overline{Wr}$ line. The R3041 is designed to accommodate a wide variety of different memory bandwidths, including DRAM systems that need precharge wait cycles for the first byte and then use a fast page mode access for bursting subsequent bytes.

The data lines used in 8-bit ports are always A/D(31:24) for big endian systems and A/D(7:0) for little endian systems. This is regardless of the Reverse Endianess bit in the CP0 Status register. There is no "BE8" signal since bytes written are always valid and should always be enabled.

Partial Word Writes

When the processor issues a store instruction which stores less than a 32-bit quantity, a partial word store occurs. The R3041 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 and the store is cacheable, 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 data and target address. Cacheable stores update or invalidate the cache as appropriate.

WRITE INTERFACE SIGNALS

The write interface uses the following signals:

$\overline{Wr}$ O

Write: This active low output indicates that a write operation is occurring. It will assert when the R3041 write buffer initiates a write transaction. It will de-assert automatically after all the data has been acknowledged.

A/D (31:0) O

Multiplexed Address/Data Bus: 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 32-bit write transaction are presented on A/D(3:0). $\overline{BE(3)}$ 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. $\overline{BE}$(3:0) can be held inactive during reads by using the BE(3:0) Control read mask in the CP0 Bus Control register as might be done for direct connection from the address latch to the $\overline{WE}$ pins in systems using 1M bit or smaller DRAMs. These byte strobes are only valid for 32-bit ports. They are not valid for 16 or 8-bit ports, however, they do indicate which bytes are used sometime during the (multi-datum) transaction.

During the data portion of the write transaction, the A/D bus contains:

Data(31:0) The R3041 drives the store data on the appropriate data lines, as indicated by the byte enable strobes during the addressing phase. Operations using less than 32-bits of data use the data lines as described in Chapter 2 Table 2.3 describing Byte Addressing. In summary, the byte addressing requires that 16-bit ports use the halfword associated with address offsets 0 and 1, i.e., D(31:16) for big endian and D(15:0) for little endian. 8-bit ports use byte associated with address offset 0, i.e., D(31:24) for big endian and D(7:0) for little endian. These byte lane assignments are dependent on the endianess selected at reset, but are independent of the User Mode Reverse Endianess control bit in the CP0 Status register. Note that unused data lines during 8- or 16-bit writes are not necessarily driven.

ALE O

Address Latch Enable: This active high 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(3:0) O

Dedicated Address Bus: The remaining bits of the transfer address are presented directly on these outputs. During 32-bit write transactions, these pins contain Address (3:2) of the transfer address. During 16-bit transactions, these pins contain Address(3:1) of the transfer address which act as a counter during halfword mini-bursts. During 8-bit transactions, these pins contain Address(3:0) of the transfer address which act as a counter during byte mini-bursts.

The R3041 Addr(1:0) output pins are designated in the R3051 as the no-connect Rsvd(1:0) pins respectively.

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

Data Enable: This active low output will remain high throughout the write transaction. It is typically used by the memory system to enable read-side output drivers; the CPU will maintain this output as high throughout write transactions, thus disabling memory system output drivers.

$\overline{\text{WrNear}}$ O

Write Near (multiplexed with Burst): This active low output is driven valid during the address phase of the write transaction. It is asserted if:

1: The store target address of this write operation has the same Addr(31:8) 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 $\overline{\text{WrNear}}$ output will not be asserted during the write transaction. Note that for 16-bit and 8-bit ports, $\overline{\text{WrNear}}$ only asserts if the entire mini-burst meets the above conditions.

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

Acknowledge: This active low input is used by the memory system to indicate that it has sufficiently processed the write transaction, and that if it was a single datum write, the CPU may terminate the write transaction (and cease driving the write data). If the transaction was a mini-burst write, Addr(3:0) and $\overline{\text{BE16(1:0)}}$ will be changed appropriately for the next datum.

BusError I

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

BE16(1:0) O

Byte Enable Strobes for 16-bit ports: These active low outputs are the byte enable strobes for 16-bit ports. If BE16(1) is asserted then the most significant byte (D(31:24) for big endian or D(15:8) for little endian) is going to contain valid data. If BE16(0) is asserted then the least significant byte (D(23:16) for big endian or D(7:0) for little endian) is going to contain valid data. BE16 can also be masked (held in-active) during reads by disabling the read mask, BE16 Control bit in the CP0 Bus Control register. Using the read mask is useful for direct connection of BE16 to the WE pins of DRAM (1Mb or smaller) systems or other systems with gated chip selects for 8-bit ports.
BE16 is not necessarily valid for 32-bit or 8-bit ports. However, if the BE16 read mask in the CP0 Bus Control Register is negated, then BE16(1) for BigEndian (BE16(0) for Little Endian) will be negated high during 8-bit reads, and asserted low for 8-bit writes.

The R3041 BE16(1:0) outputs pins are designated in the R3051 as the no-connect Rsvd(3:2) pins, respectively.

Last O

Last Datum in Mini-Burst: This active low output indicates that the last datum of a single datum or mini-burst is being written. It goes active with Wr for single datum writes and after the next to last Ack is sampled for multiple datum writes. Last de-asserts when Wr de-asserts.

The R3041 Last output pin is designated in the R3051 as the Diag(0) output pin.

MemStrobe O

Memory Strobe: This active low output pulses low for each datum written. It first goes low 1 clock after the beginning of a write. It then de-asserts 1/2 clock after an Ack is received. If there are more datum to be written (as in a mini-burst write) then MemStrobe will assert again 1/2 clock after the previous de-assertion. MemStrobe will continue to (de)-assert until all datum have been written. See Figure 9-18 for an example. It can be used either as a write strobe or a data strobe for single datum (non-burst) I/O ports or for a write strobe (single or mini-burst) for SRAM. It can be active for reads, writes, or both depending on the settings in the MemStrobe Control bits in the CP0 Bus Control register. After reset, MemStrobe is only active for writes.

The R3041 MemStrobe output pin is designated in the R3051 as the BrCond(0) input pin.

$\overline{\text{IOStrobe}}$ O

Input/Output Strobe: This active low output asserts on the first falling edge of $\overline{\text{SysClk}}$ (1 clock) after ALE de-asserts. It asserts relatively late in the cycle so that addresses and control lines are properly setup. It de-asserts with at the end of the write along with $\overline{\text{Wr}}$. It can be active for reads, writes, or both depending if the read and write masks are enabled in the IOStrobe Control bit field in the CP0 Bus Control register. Note that the assertion of $\overline{\text{IOStrobe}}$ requires that the transaction last at least 3 clock cycles.

The $\overline{\text{IOStrobe}}$ pin is software configurable as an input by using the SBrCond(3:2) Control bit in the CP0 Bus Control register. The pin defaults to an input after reset.

$\overline{\text{ExtDataEn}}$ O

Extended Data Enable: This active low output asserts active low on the first rising edge of $\overline{\text{SysClk}}$ after ALE de-asserts (1/2 clock later). It is extended in that it de-asserts 1/2 clock after $\overline{\text{Wr}}$ de-asserts. $\overline{\text{ExtDataEn}}$ provides extra hold time for data sampling (especially on writes) or for the $\overline{\text{IOStrobe}}$ (if $\overline{\text{ExtDataEn}}$ is used as an extended read/$\overline{\text{write}}$ line). It can be active for reads, writes, or both depending if the read and write masks are enabled in the ExtDataEn Control bit field of the CP0 Bus Control register.

The $\overline{\text{ExtDataEn}}$ pin is software configurable as an input by using the SBrCond(3:2) Control bit in the CP0 Bus Control register. The pin defaults to an input after reset.

WRITE INTERFACE TIMING OVERVIEW

The protocol for transmitting data from the processor to memory and I/O devices is discussed below. Throughout this chapter it is assumed that ExtDataEn and IOStrobe are configured as output pins and that they and MemStrobe are enabled for writes.

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 SysClk. Higher priority operations would have the effect of delaying the start of the write.

Figure 9.1 illustrates the initiation of a write transaction, based on the internal negation of the 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, the buffer 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.

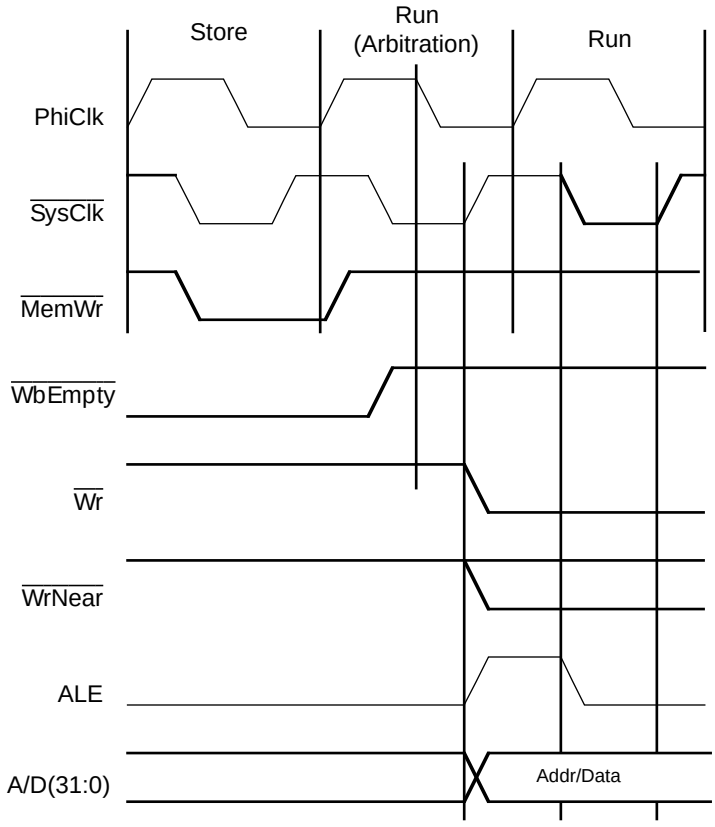


Figure 9.1. Start of Write Operation – BIU Arbitration

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. The R3041 has two types of address phases. If the ExtAddrHold reset configuration mode is not selected, the address is driven with ALE. The data is driven as soon as ALE de-asserts. Figure 9.2 illustrates the start of this type of processor write transaction, including the addressing of memory and presenting the store data on the A/D bus. If the ExtAddrHold reset configuration mode is selected, the address is driven for 1/2 clock past the de-assertion of ALE. Figure 9.3 illustrates the start of this type of processor write transaction. The remaining timing diagrams in this section will only be shown with the ExtAddrHold option asserted even though either mode is always applicable to every type of write transaction.

In either addressing mode, 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.

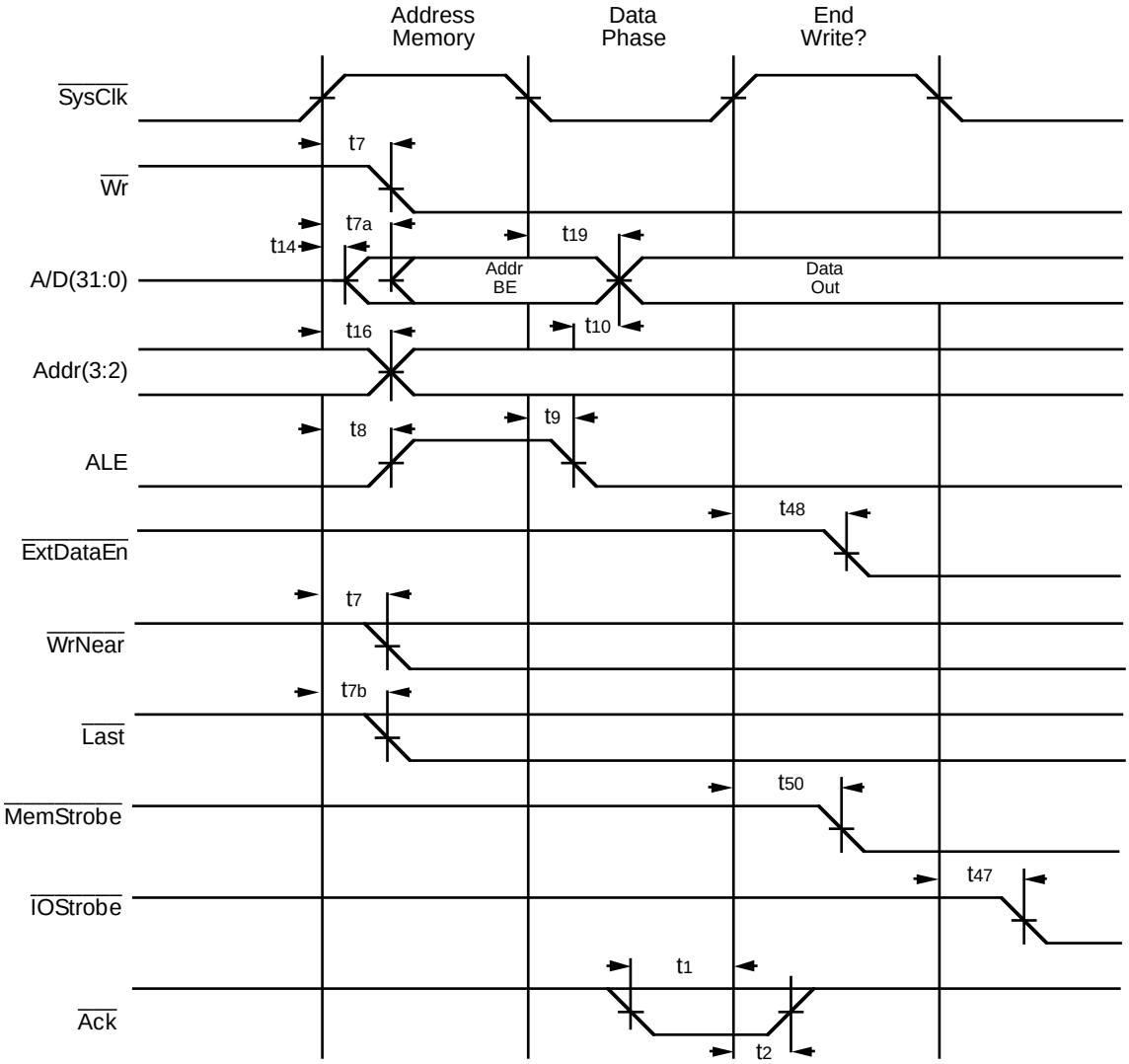


Figure 9.2. Memory Addressing and Start of Write for Non-ExtAddrHold Mode

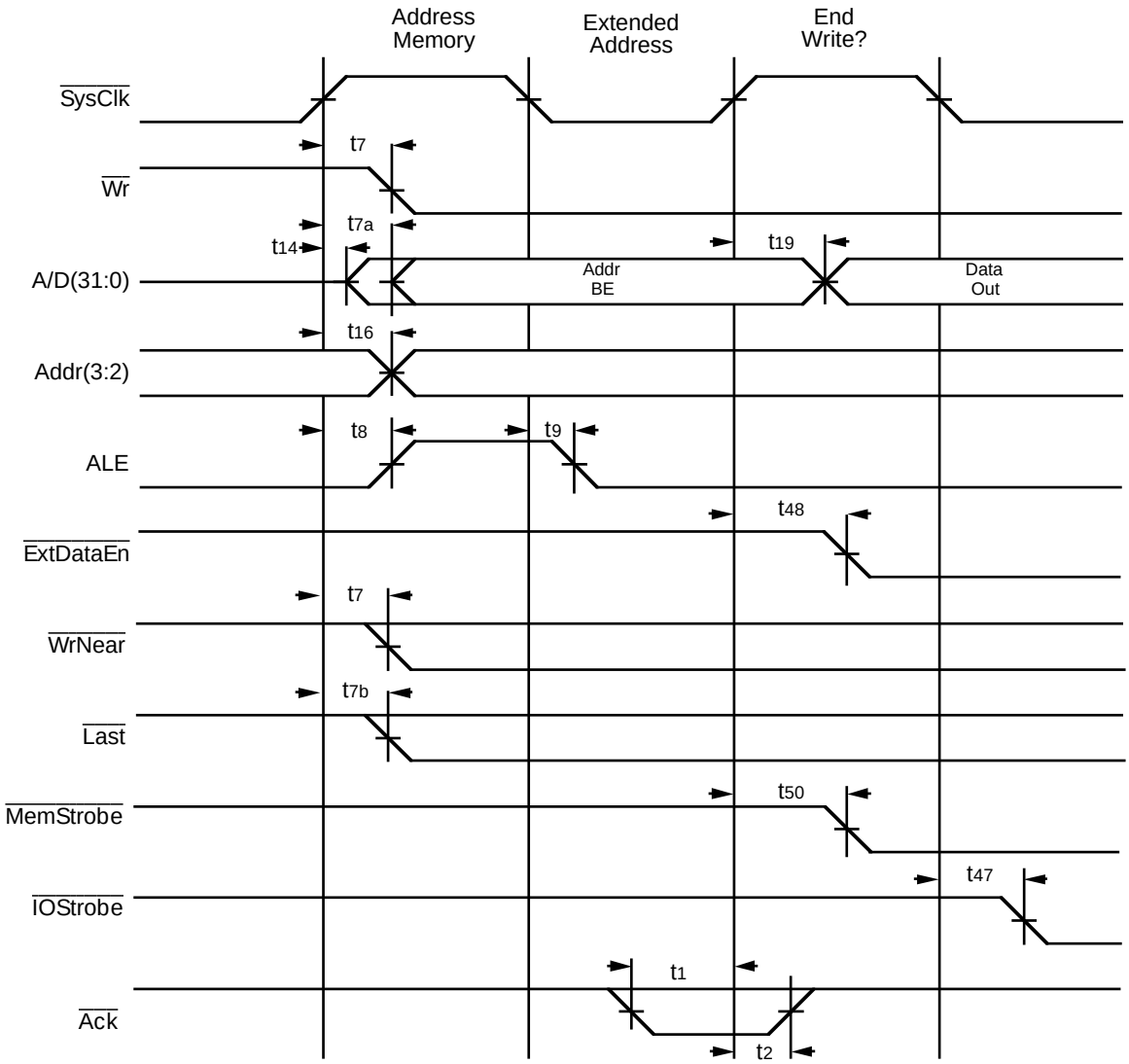


Figure 9.3. Memory Addressing and Start of Write for ExtAddrHold Mode

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 9.2 for the non-ExtAddrHold reset configuration mode, or in the first phase of the second clock cycle for the ExtAddrHold mode, as illustrated in Figure 9.3.

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.

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.

During the data phase, these three control signals may also assert:

When programmed via the ExtDataEn and SBrCond(3:2) Control bits in the CP0 Bus Control register, ExtDataEn asserts one clock cycle after Wr asserts and remains asserted 1/2 clock cycle after Wr de-asserts. Thus this signal is useful for enabling write data transceivers to allow extra hold time or for acting as an I/O read/write signal with extra hold time.

When programmed via the MemStrobe Control bits in the CP0 Bus Control register, MemStrobe asserts one clock after Wr asserts. It de-asserts for 1/2 clock after every Ack is sampled. After 1 clock past an Ack, if more datum are being written within the same transaction, MemStrobe asserts again and so on until all datum are acknowledged.

When programmed via the IOStrobe and SBrCond(3:2) Control bits in the CP0 Bus Control register, IOStrobe asserts 1.5 clock cycles after Wr asserts and remains asserted until Wr de-asserts. It will only assert when the write cycle is at least 3 clocks long. Thus this signal is useful for I/O writes if disabled during reads. IOStrobe can be used as an I/O data strobe if ExtDataEn is configured as a read/write signal. IOStrobe can also be used as a DRAM address multiplexor select if configured to assert on both reads and writes.

Terminating the Write

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

- It can supply the appropriate number of Acks (acknowledges) to the processor, to indicate that it has sufficiently processed the write request, and that the processor may terminate the write.

- It can supply a BusError to the processor, to indicate that the requested data transfer has "failed" on the bus. The system interface behavior of the processor when BusError is presented is identical to the behavior when the last Ack is asserted. In the case of writes terminated by BusError, no exception is taken, and the data transfer cannot be retried.

Figure 9.4 shows the timing of the control signals when the write cycle is being terminated.

To determine the end of the write cycle one of these methods may be used:

Systems that only use 32-bit memory sub-region ports as with the rest of the R30xx family only have single datum writes and either count the number of wait-cycles or use the de-asserting edge of $\overline{Wr}$ to end the transaction.

Systems that use 16 or 8-bit ports must in general support mini-burst writes. Memory controllers for such systems can use the de-asserting edge of $\overline{Wr}$ to reset the controller. The memory controller can also look for $\overline{Last}$ to assert. When $\overline{Last}$ asserts, the controller knows that it is handling the final datum of the transaction. It is also possible to decode $\overline{BE(3:0)}$ to determine how many datum are to be returned.

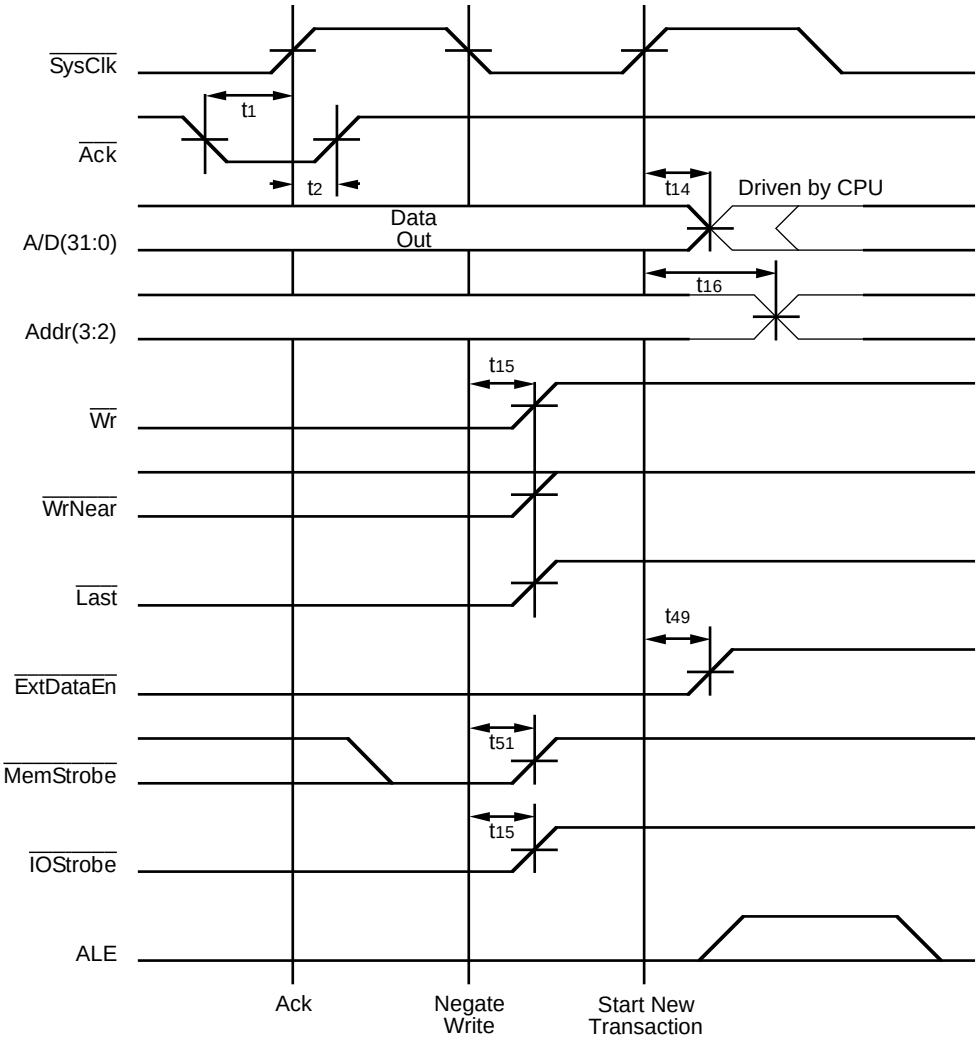


Figure 9.4. End of Write

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. A new operation may begin as soon as one clock cycle after the final $\overline{Ack}$ is sampled from the interface.

Also note that bus turn around after a write transaction does not occur. That is, the processor continues to drive the A/D bus throughout the write transaction (both address and data phases) , and will also drive the A/D bus during the start of either a subsequent read or write transaction. Since no change in bus ownership occurs, the Bus Turn Around field of the CP0 Bus Control register does not apply after write transactions.

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 an internal 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.

Note that each entry of the write buffer is one internal 32-bit word wide, but

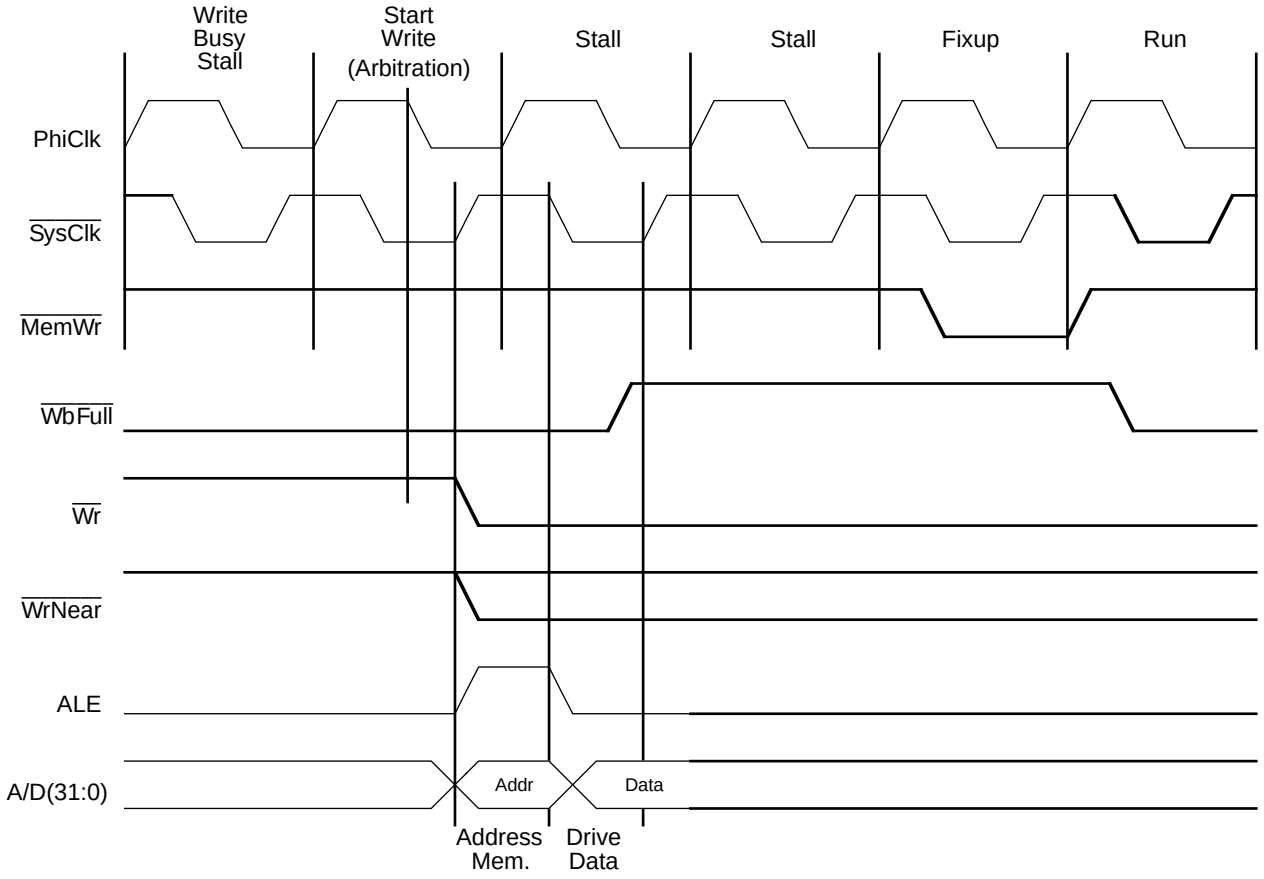


Figure 9.5. Write Buffer Full Operation

each entry can only hold the result of one store operation. Thus a 32-bit port can store 4 words while a 16-bit port can store up to 8 halfwords when using store word operands. However, if for example, four store byte operations are done, each byte takes a full entry.

The write buffer full operation is illustrated in Figure 9.5.

WRITE TIMING DIAGRAMS

This section illustrates a number of timing diagrams applicable to R3041 writes. The values for the AC parameters referenced are contained in the R3041 data sheet. Throughout this chapter it is assumed that $\overline{\text{ExtDataEn}}$ and $\overline{\text{IOStrobe}}$ are configured as output pins and that they and $\overline{\text{MemStrobe}}$ are enabled for writes. Although using the non-ExtAddrHold reset configuration mode option is always applicable, these timing diagrams are all shown using the ExtAddrHold mode.

32-Bit Basic Write

Figure 9.6 illustrates the case of a write operation which did not require wait states. Thus, $\overline{\text{Ack}}$ was detected at the rising edge of $\overline{\text{SysClk}}$ which occurred exactly one clock cycle after the rising edge of $\overline{\text{SysClk}}$ which asserted $\overline{\text{Wr}}$.

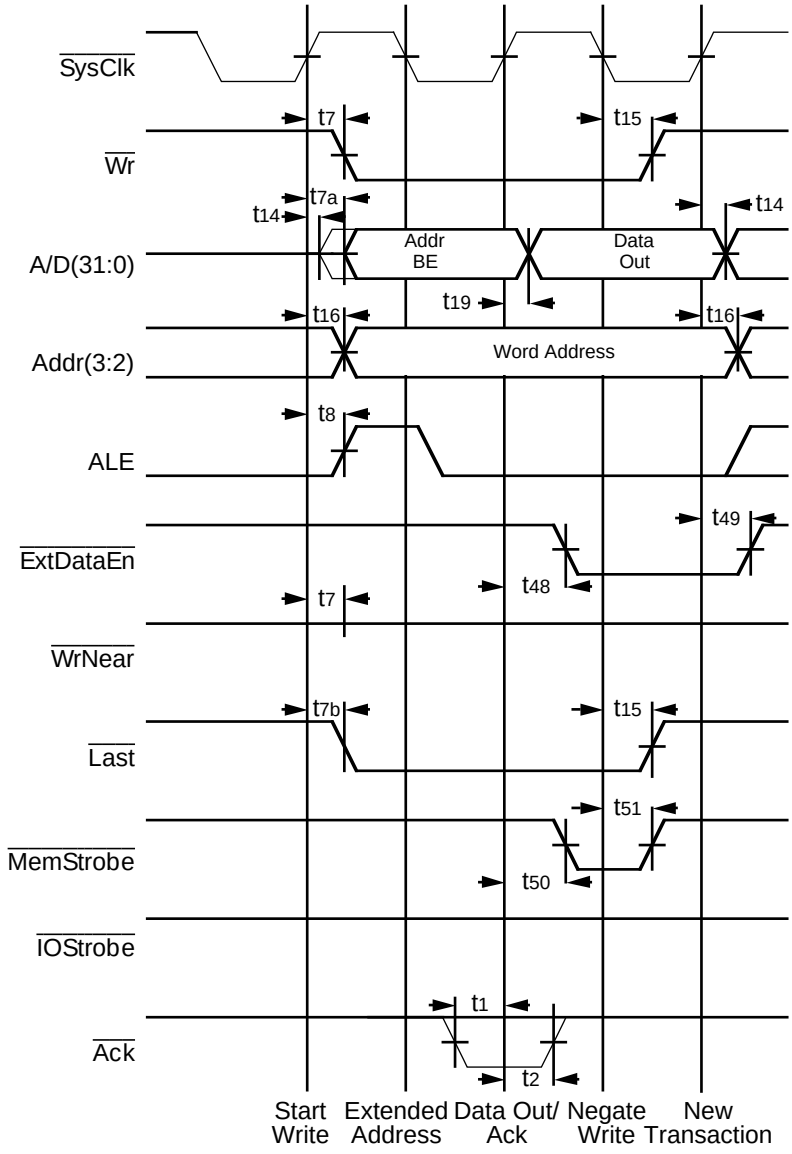


Figure 9.6. Basic 32-Bit Port Write with No Wait Cycles

Figure 9.7 also illustrates the case of a 32-bit memory sub-region basic write. However, in this figure, two bus wait cycles were required before the data was retired. Thus, two rising edges of $\overline{\text{SysClk}}$ occurred where $\overline{\text{Ack}}$ was not asserted. On the third rising edge of $\overline{\text{SysClk}}$, $\overline{\text{Ack}}$ was asserted, and the write operation was terminated.

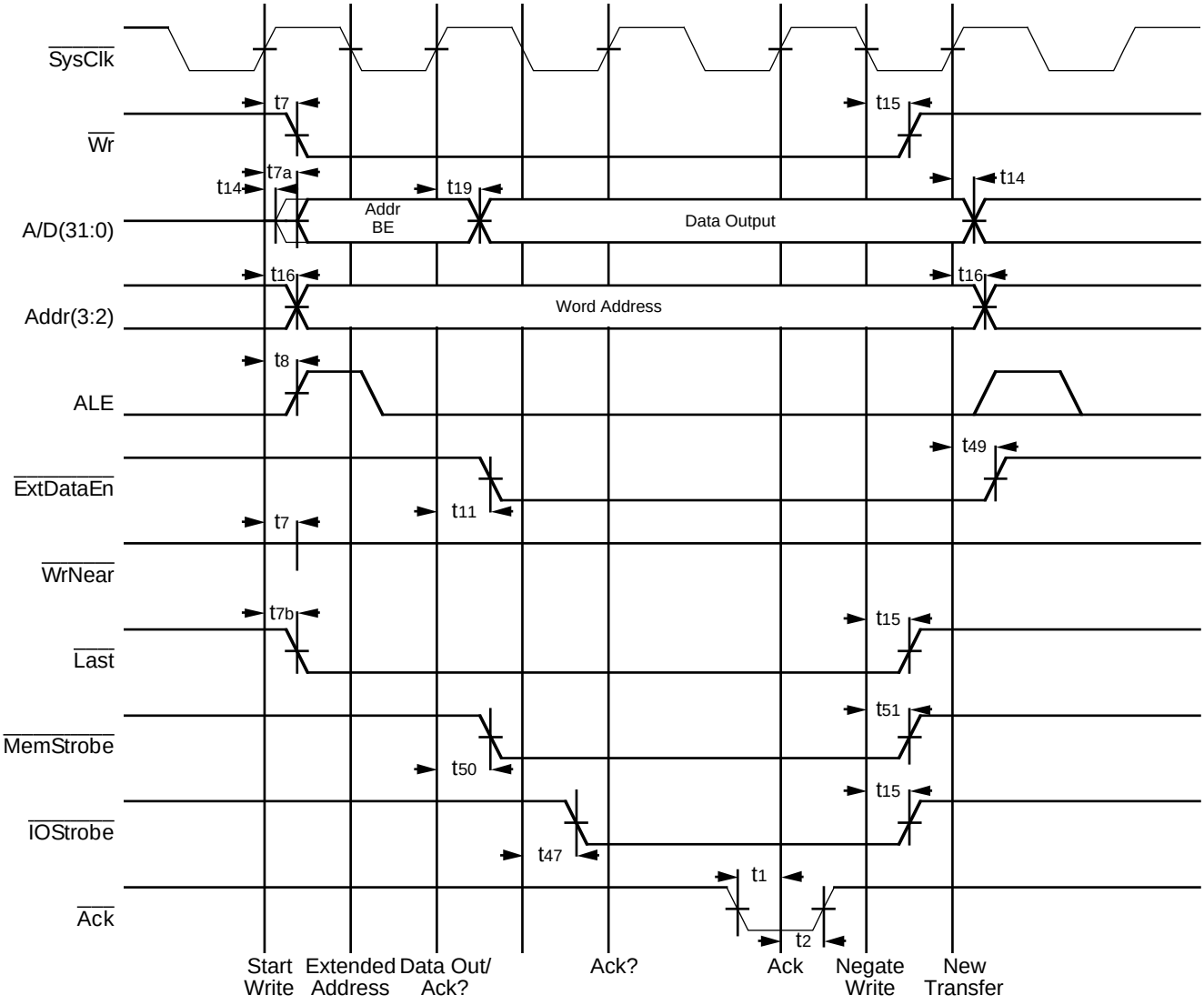


Figure 9.7. Basic 32-Bit Port Write with Wait Cycles

Bus Error Operation

Figure 9.8 is a modified version of Figure 9.7 (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.

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.

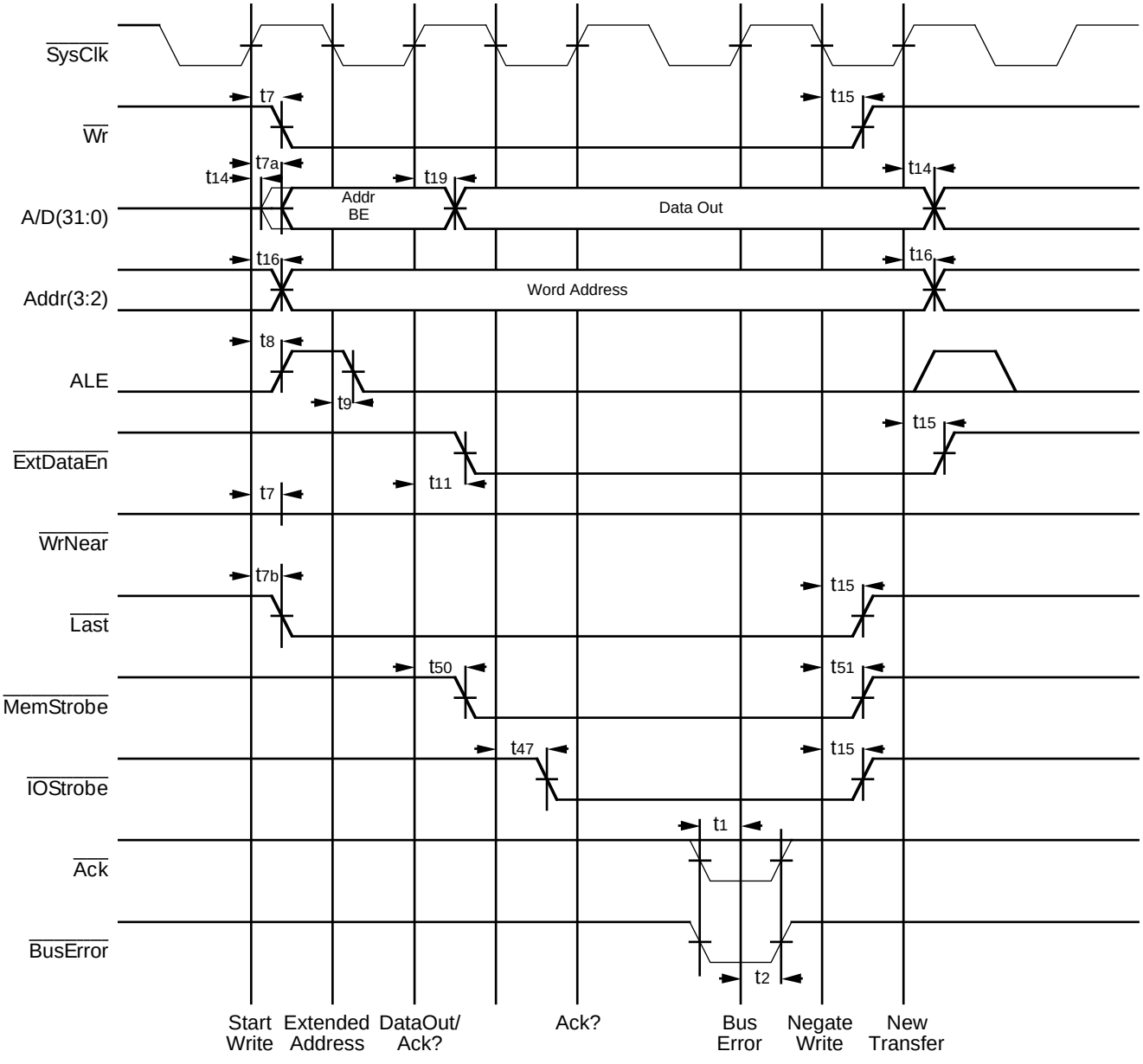


Figure 9.8. Basic Write Terminated by Bus Error

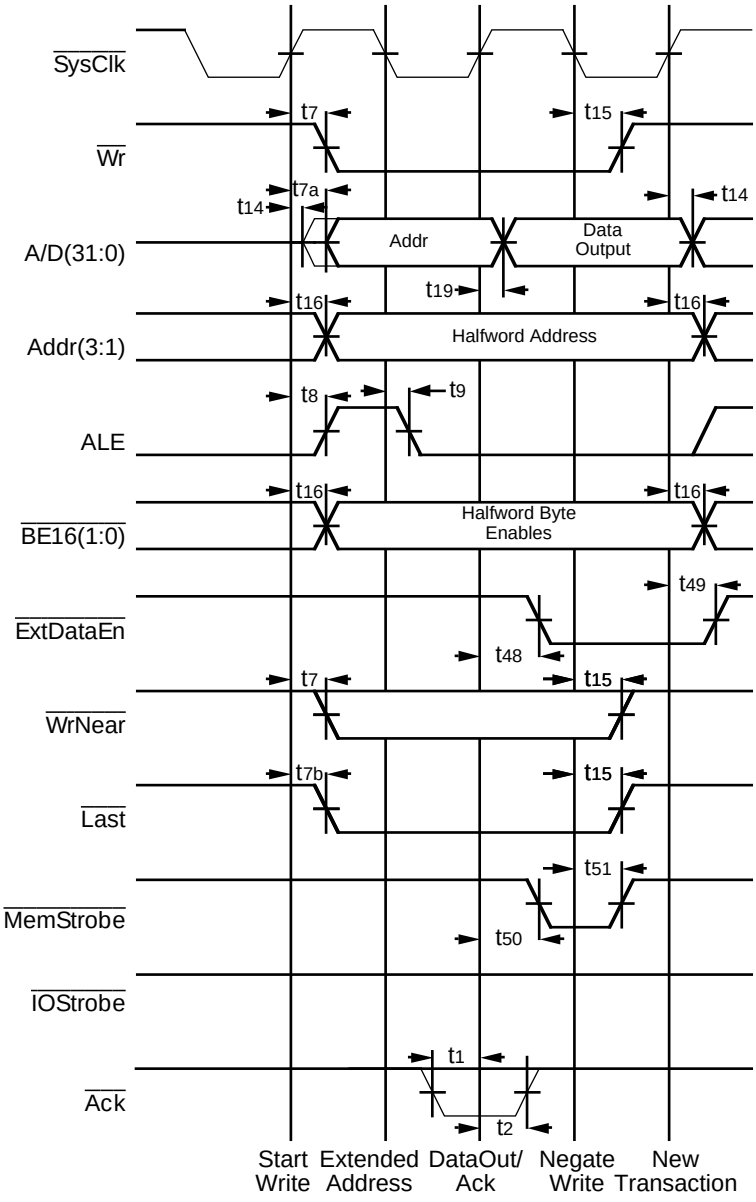


Figure 9.9. Single Datum 16-Bit Port Write with No Wait Cycles

16-Bit Write Timing Diagrams

This section illustrates a number of timing diagrams applicable to R3041 write transactions when a 16-bit port has been selected via the memory sub-region configuration Port Size CP0 Control register. These diagrams reference AC parameters whose values are contained in the R3041 data sheet. It is assumed that $\overline{\text{ExtDataEn}}$ and $\overline{\text{IOStrobe}}$ are configured as output pins and that they and $\overline{\text{MemStrobe}}$ are enabled for writes. Although using the non-ExtAddrHold reset configuration mode option is always applicable, these timing diagrams are all shown using the ExtAddrHold mode.

16-Bit Basic Write

Figure 9.9 illustrates the case of a byte or halfword write operation to a 16-bit port which did not require wait states. Thus, $\overline{\text{Ack}}$ was detected at the rising edge of $\overline{\text{SysClk}}$ which occurred exactly one clock cycle after the rising edge of $\overline{\text{SysClk}}$ which asserted $\overline{\text{Wr}}$. The 16-bit byte enables, $\overline{\text{BE16(1:0)}}$ indicate which bytes are being used in this transaction.

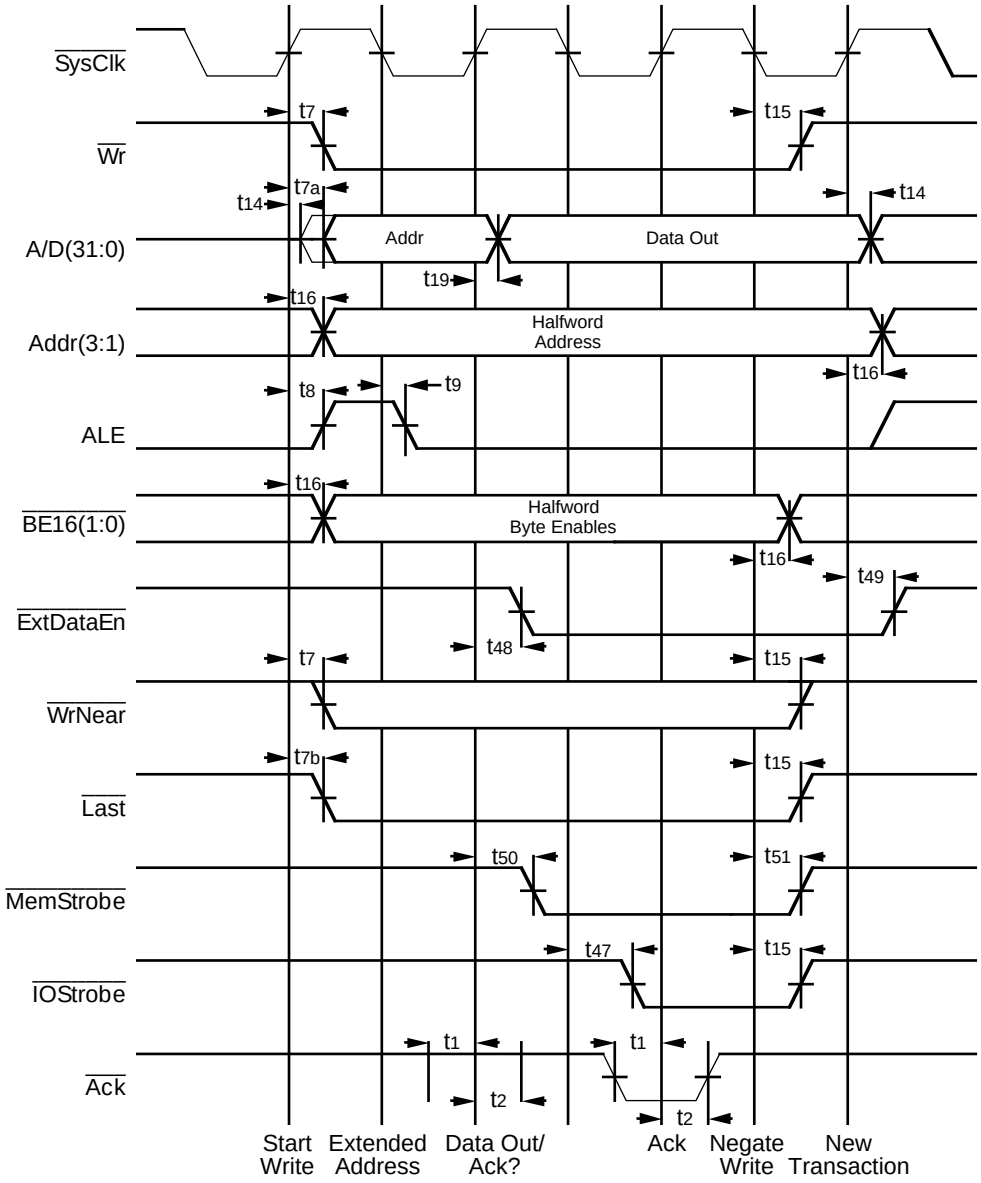


Figure 9.10. Single Datum 16-Bit Port Write with Wait Cycles

Figure 9.10 also illustrates the case of a basic halfword write. However, in this figure, two bus wait cycles were required before the data was retired. Thus, two rising edges of $\overline{\text{SysClk}}$ occurred where $\overline{\text{Ack}}$ was not asserted. On the third rising edge of $\overline{\text{SysClk}}$, $\overline{\text{Ack}}$ was asserted, and the write operation was terminated.

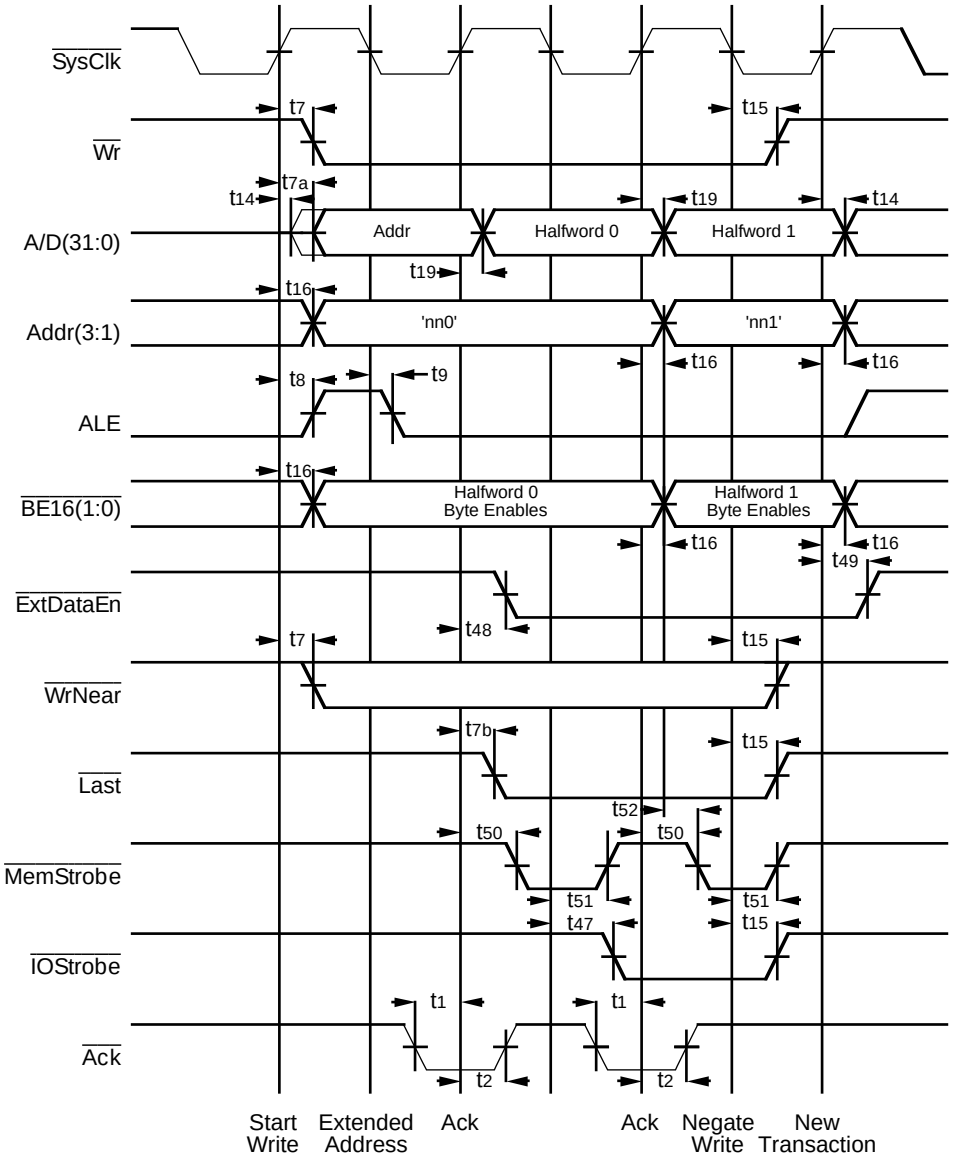


Figure 9.11. Mini-Burst 16-Bit Port Write

Figure 9.11 illustrates the case of a double halfword write operation which did not require wait states. After the first $\overline{Ack}$ is sampled, $\overline{Last}$ asserts to indicate that the second datum is the final datum. Also Addr(3:1) increments and the $\overline{BE16}$(1:0) change if appropriate. As with the single halfword write, bus wait cycles can be inserted for either the first of second datum simply by delaying the assertion of the corresponding $\overline{Ack}$.

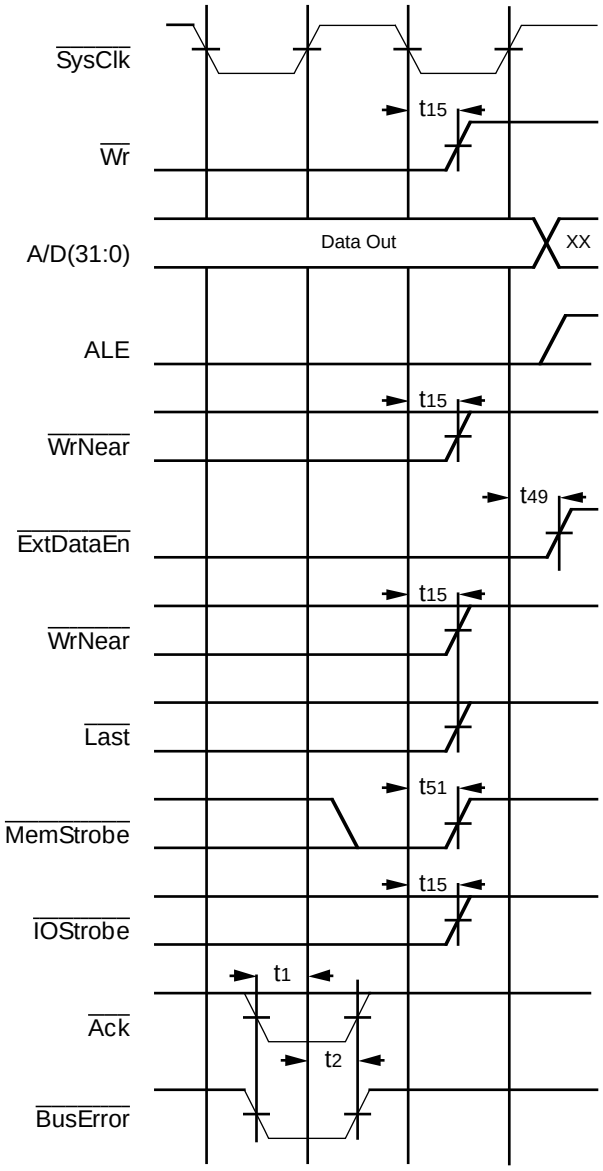


Figure 9.12. 16-Bit Write Terminated by Bus Error

Bus Errors for 16-bit writes are handled similar to 32-bit writes.The BusError input is sampled whenever Ack is sampled. Bus errors which occur before the end of a mini-burst will abandon any unsent datum. A case where BusError is used to signal the end of a write transaction is illustrated in Figure 9.12.

8-Bit Write Timing Diagrams

This section illustrates a number of timing diagrams applicable to R3041 write transactions when a 8-bit port has been selected via the memory sub-region configuration Port Size CP0 Control register. These diagrams reference AC parameters whose values are contained in the R3041 data sheet. It is assumed that ExtDataEn and IOStrobe are configured as output pins and that they and MemStrobe are enabled for writes. Although using the non-ExtAddrHold reset configuration mode option is always applicable, these timing diagrams are all shown using the ExtAddrHold mode.

8-Bit Basic Write

Figure 9.13 illustrates the case of a single byte write operation to an 8-bit port 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.

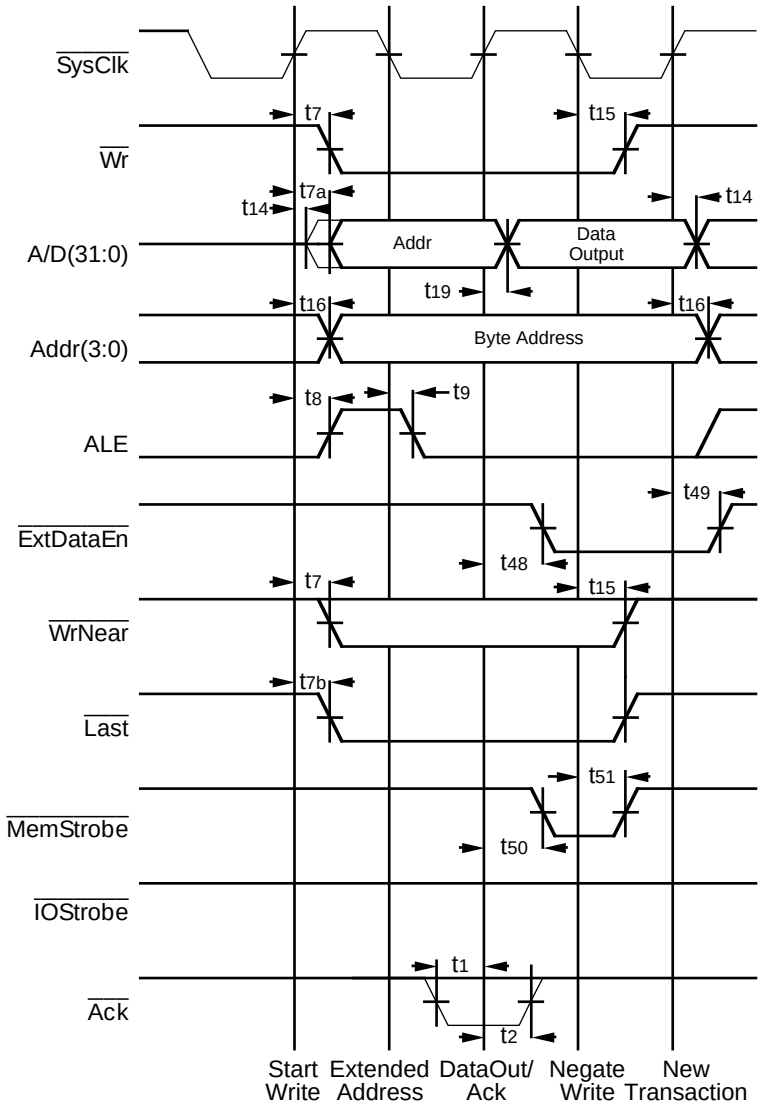


Figure 9.13. Single Byte 8-Bit Port Write with No Wait Cycles

Figure 9.14 also illustrates the case of a basic single byte 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.

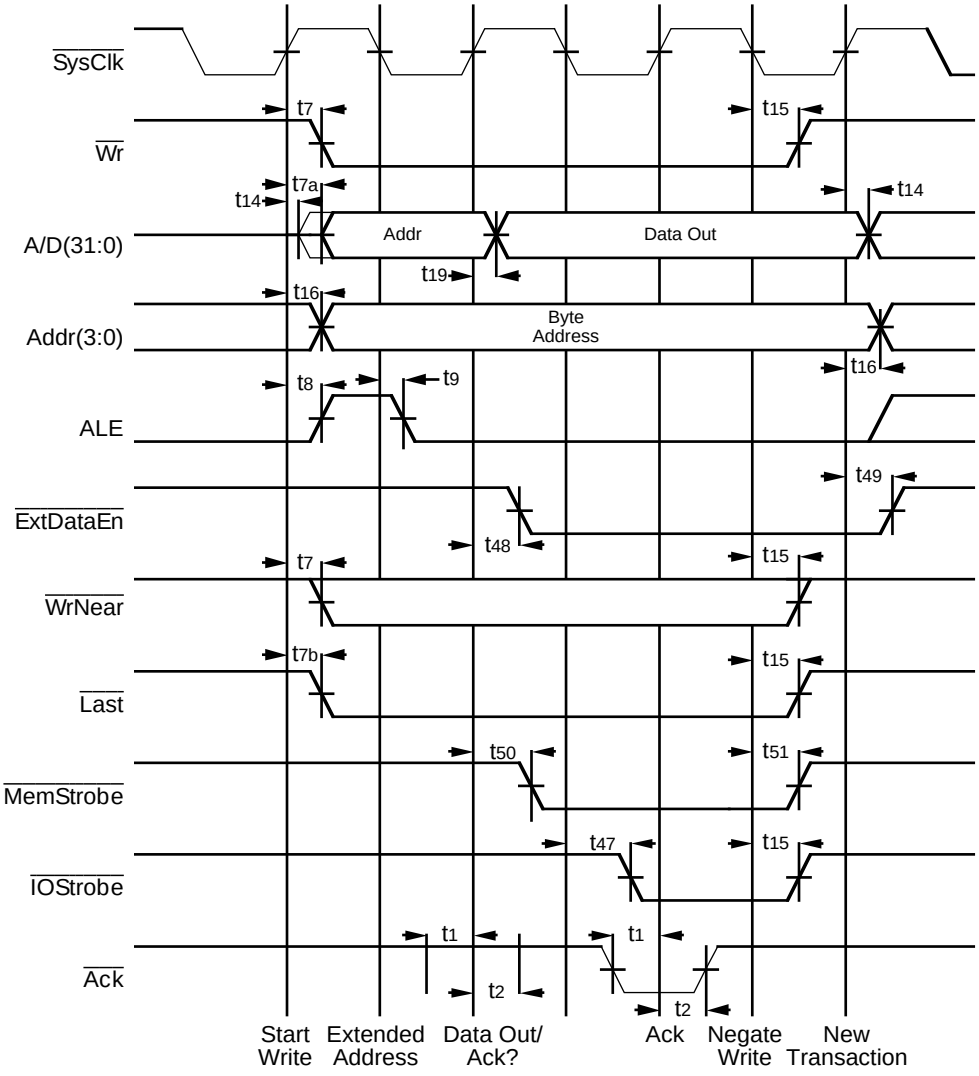


Figure 9.14. Single Byte 8-Bit Port Write with Wait Cycles

Figures 9.15, 9.16, and 9.17 illustrate the cases of a double, tri, and quad byte write operation respectively. These cases did not require wait states. After the second to last $\overline{Ack}$ is sampled, $\overline{Last}$ asserts to indicate that the next datum is the final datum. Also Addr(3:0) increments. As with the single halfword write, bus wait cycles can be inserted for any of the datum simply by delaying the assertion of the corresponding $\overline{Ack}$.

Bus Errors for 8-bit writes are handled similar to 32-bit writes. The $\overline{BusError}$ input is sampled whenever $\overline{Ack}$ is sampled. Bus errors which occur before the end of a mini-burst will abandon any unsent datum.

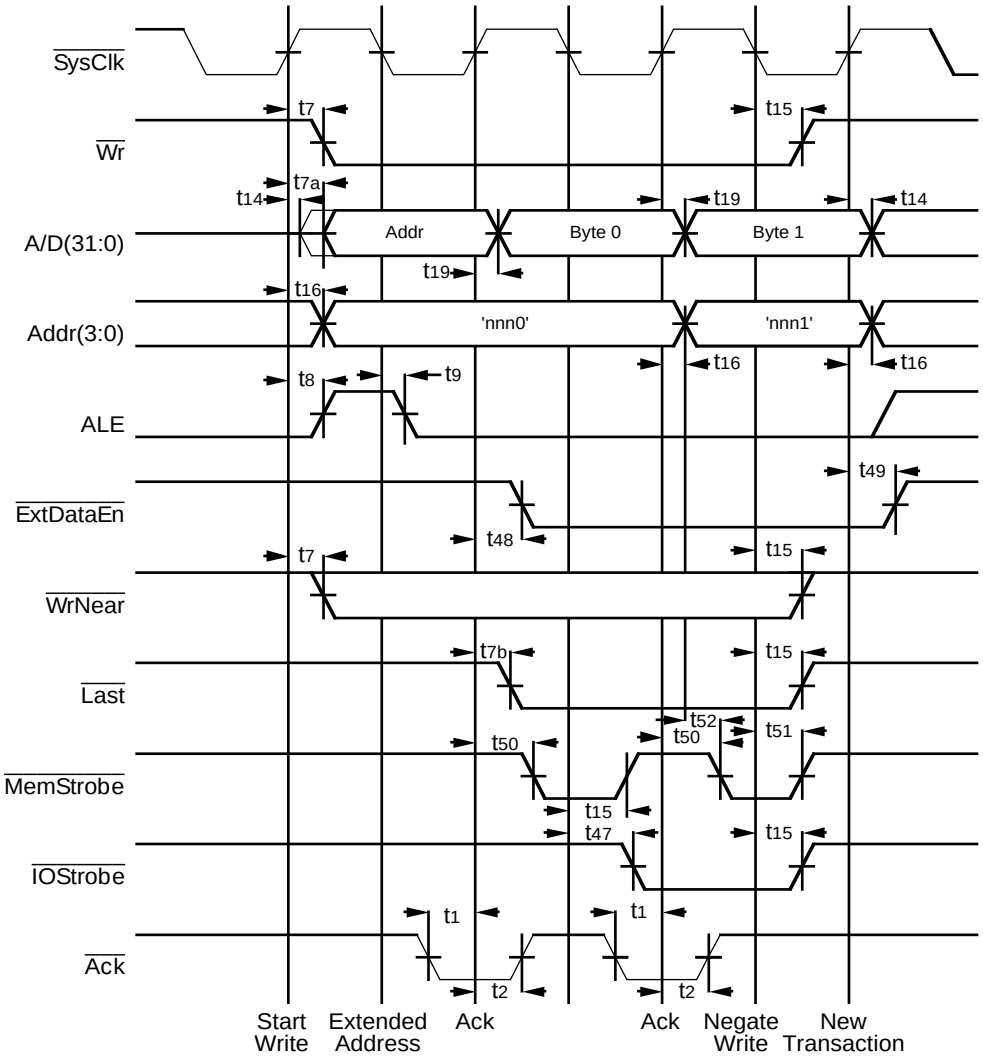


Figure 9.15. Two Byte 8-Bit Port Write with Wait Cycles

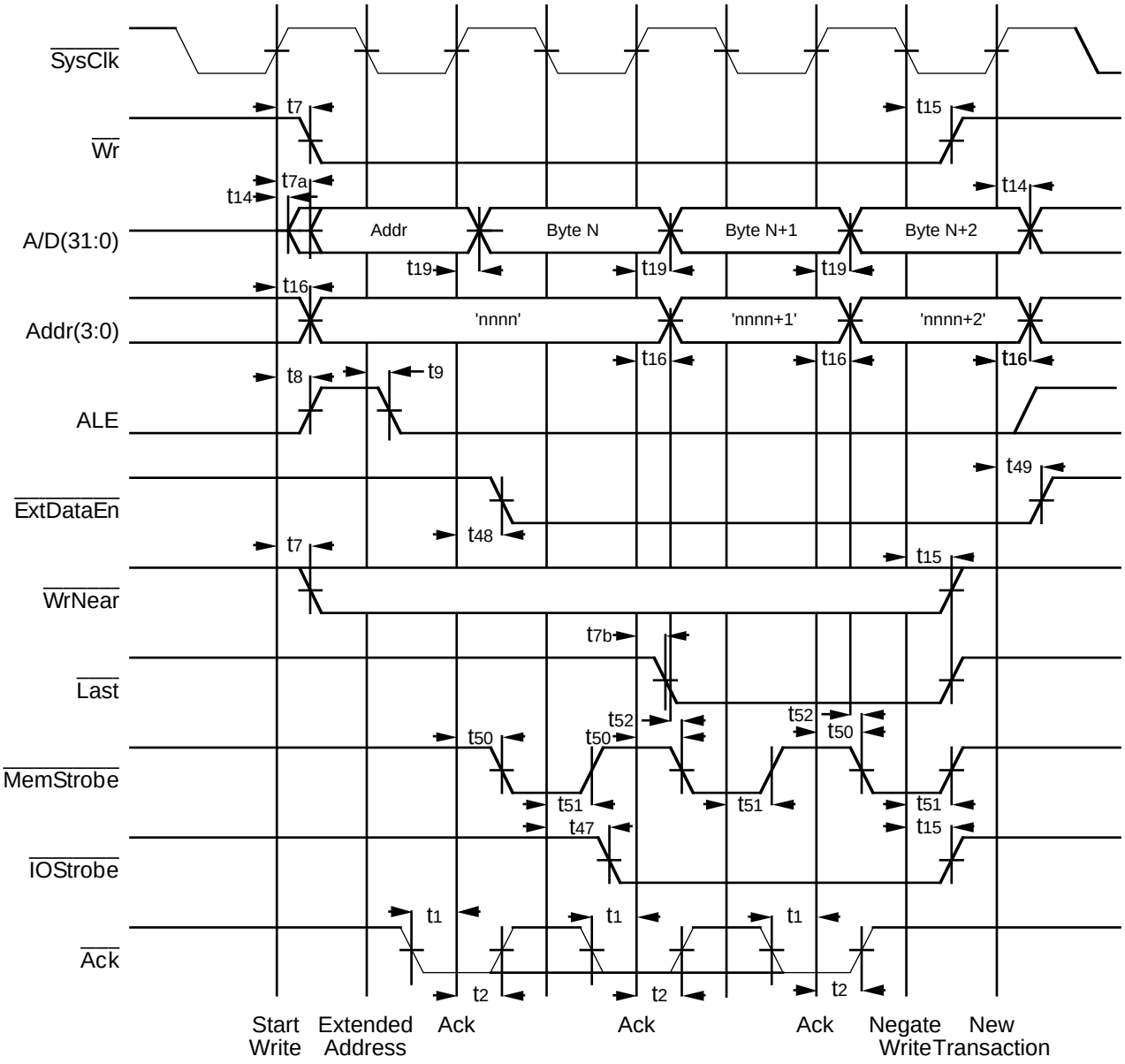


Figure 9.16. Three Byte Mini-Burst 8-Bit Port Write

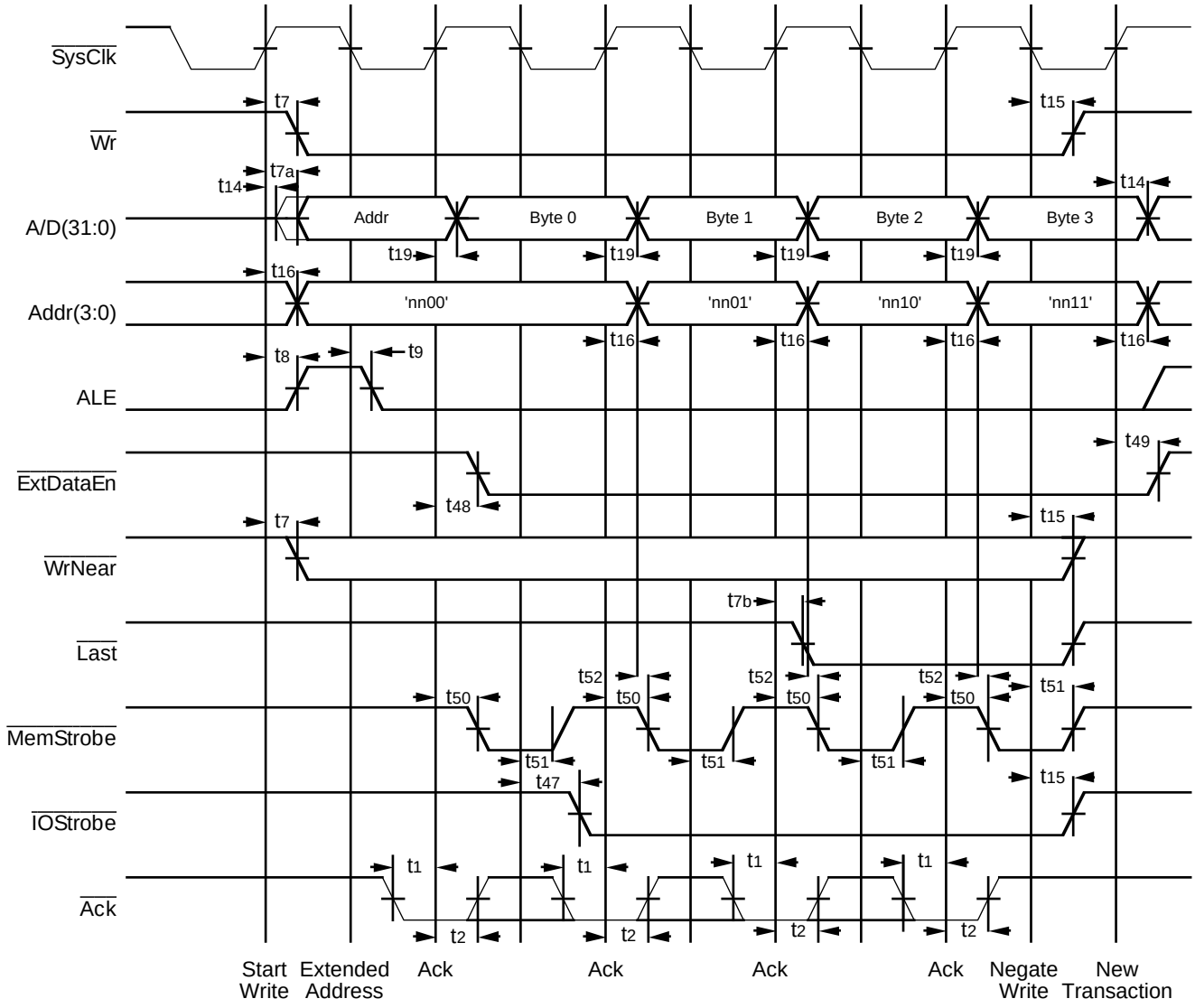


Figure 9.17. Four Byte Mini-Burst 8-Bit Port Write

INTRODUCTION

The R30xx family contains provisions to allow an external agent to remove the processor from its memory bus, and thus perform transfers on its own by a direct memory access (DMA). These provisions use the internal DMA arbiter interface 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.

The R3041 expands on the basic capability of the R30xx family DMA Arbiter by supporting an optional mode whereby the CPU can ask an external DMA master to relinquish the bus. On the other hand, the R3041 can use the default DMA mode in an R3051 compatible fashion.

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(3:0)
- Interface control signals: $\overline{\text{Rd}}$, $\overline{\text{Wr}}$, $\overline{\text{DataEn}}$, $\overline{\text{Burst}}$/$\overline{\text{WrNear}}$, and ALE
- Other control signals: $\overline{\text{Last}}$, $\overline{\text{BE16}}$(1:0), and $\overline{\text{MemStrobe}}$
- If enabled as outputs: $\overline{\text{ExtDataEn}}$ and $\overline{\text{IOStrobe}}$
- Diag

The extended data enable signal, $\overline{\text{ExtDataEn}}$ is slightly different from the other tri-statable signals in that it tri-states 1/2 clock period after the other signals. This allows it to do its primary function of staying asserted 1/2 clock longer than the other signals and yet de-assert before tri-stating.

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.

During DMA cycles, the processor does not tri-state the following memory interface signals:

- $\overline{\text{BusGnt}}$
- $\overline{\text{SysClk}}$
- $\overline{\text{TC}}$

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. Since $\overline{\text{SysClk}}$ and $\overline{\text{TC}}$ do not tri-state, they can be used to continue to clock the main memory state machine and to initiate DRAM refreshes during

DMA, respectively. Thus, DMA can occur at the same speed at which the R3041 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 R3041 has two protocols for de-asserting $\overline{\text{BusGnt}}$. The protocol must be selected using the DMA Protocol bit in the CP0 Bus Control register. If DMA Protocol is not selected then this default R30xx family equivalent mode causes $\overline{\text{BusGnt}}$ during DMA to remain asserted until $\overline{\text{BusReq}}$ is removed. If the DMA Protocol is selected, then during DMA, $\overline{\text{BusGnt}}$ will return high if the CPU makes an internal request for the bus. In order to de-assert, $\overline{\text{BusGnt}}$ must have first been asserted for at least 1.5 clocks. In both protocols, the CPU does not begin driving the bus until it is given control of the bus back. As detailed below in Figure 10.1, the bus control is returned to the CPU when the external DMA agent de-asserts $\overline{\text{BusReq}}$.

```
/*  BusGntn and BusReqn are for the CPU BusGntn line.
    BusGntn1 is for the highest priority device (DRAM refresher).
    BusGntn2 is for the lowest priority device (DMA controller).
*/

/*  BusGntn1 has the highest priority, even over the CPU.
    Line 3 state feedback gives BusGntn1 the default style
    BusGntn priority by ignoring the !BusGntn signal after
    it gets the bus.
*/
!BusGntn1 := Resetn and BusGntn1 and (                  /* 1 */
                (!BusReqn1 and !BusGntn)                /* 2 */
                or (!BusReqn1 and !BusGntn1)            /* 3 */
          );

/*  BusGnt2n has the lowest priority, equal to that of the CPU.
    Line 2 puts it request below the priority of the Device 1 request.
    Line 3 allows the CPU to take back the bus.
    This assumes that Device 2 will disconnect from the bus
    immediately after the current DMA cycle is done and that it
    will later restart gracefully.
*/
!BusGntn2 := Resetn and BusGntn1 and (                  /* 1 */
                 (!BusReqn2 and BusReqn1 and !BusGntn)  /* 2 */
                or (!BusReqn2 and !BusGntn2 and !BusGntn) /* 3 */
          );

/*  In this example, Device 2 and the CPU will alternate bus
    mastership back and forth until done.
    Line 3 allows the CPU to get the bus back after BusGntn1
    is removed and Device 2 acknowledges by removing its
    BusReqn2.
    Device 2 should remove BusReqn2 for at least 2 clocks
    when it loses its BusGntn2.  If it can't then the
    BusGntn term is needed.
*/
!BusReqn := Resetn and (                                /* 1 */
                (!BusReqn1)                             /* 2 */
                or (!BusReqn2 /* and BusGntn */)        /* 3 */
          );
```

Figure 10.1. Example DMA Arbiter PLA Equations using the DMA Protocol Mode

The external agent indicates that the DMA transfer has terminated by negating the BusReq input to the processor, which is sampled on the rising edge of SysClk. In the default mode with DMA Protocol turned off, BusGnt is negated on a falling edge of SysClk, so that it will be negated before the assertion of Rd or Wr for a subsequent transfer. In the DMA Protocol mode, BusGnt will be de-asserted on a falling edge of SysClk if it has not already done so. In either mode, on the next rising edge of SysClk after BusReq has been sampled as de-asserted, the processor will resume driving tri-stated signals.

Thus the DMA system can operate with the highest bus priority or it can use the DMA Protocol to give DMA and the CPU equal priority. See Figure 10.1 for example PLA equations that implement a typical external DMA arbitration unit.

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 trade-off for the vast majority of embedded applications.

DMA ARBITER INTERFACE SIGNALS

BusReq I

Bus Request: This active low 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

Bus Grant: This active low signal is an output from the processor and has two modes. In the default mode where the DMA Protocol bit in the CP0 Bus Control register is not selected, BusGnt is used to indicate that the CPU has relinquished mastership of the external interface bus. When the DMA Protocol is selected, BusGnt goes low initially for at least 1.5 clocks to indicate that the CPU has relinquished mastership of the external interface bus. After going low, BusGnt returns high either when the CPU makes an internal request for the bus or after BusReq is de-asserted.

DMA ARBITER TIMING DIAGRAMS

These figures reference AC timing parameters whose values are contained in the R3041 data sheet. These figures assume that ExtDataEn and IOStrobe are enabled as outputs instead of as SBrCond(3:2) inputs.

Initiation of DMA Mastership

Figure 10.2 shows the beginning of a DMA cycle. Note that if BusReq were asserted while the processor was performing a read or write operation, 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 BusReq with proper set-up time to SysClk. Once BusReq is detected, and the bus is free, the processor will grant control to the requesting agent by asserting its BusGnt output, and tri-stating its output drivers, from a rising edge of SysClk. The bus will remain under the control of the external master until it negates BusReq, indicating that the processor is once again the bus master.

If ExtDataEn is driven during DMA, then the DMA master can choose to externally delay BusGnt by 1 clock. The tri-stating of ExtDataEn is delayed by 1/2 clock so that it can be driven high first.

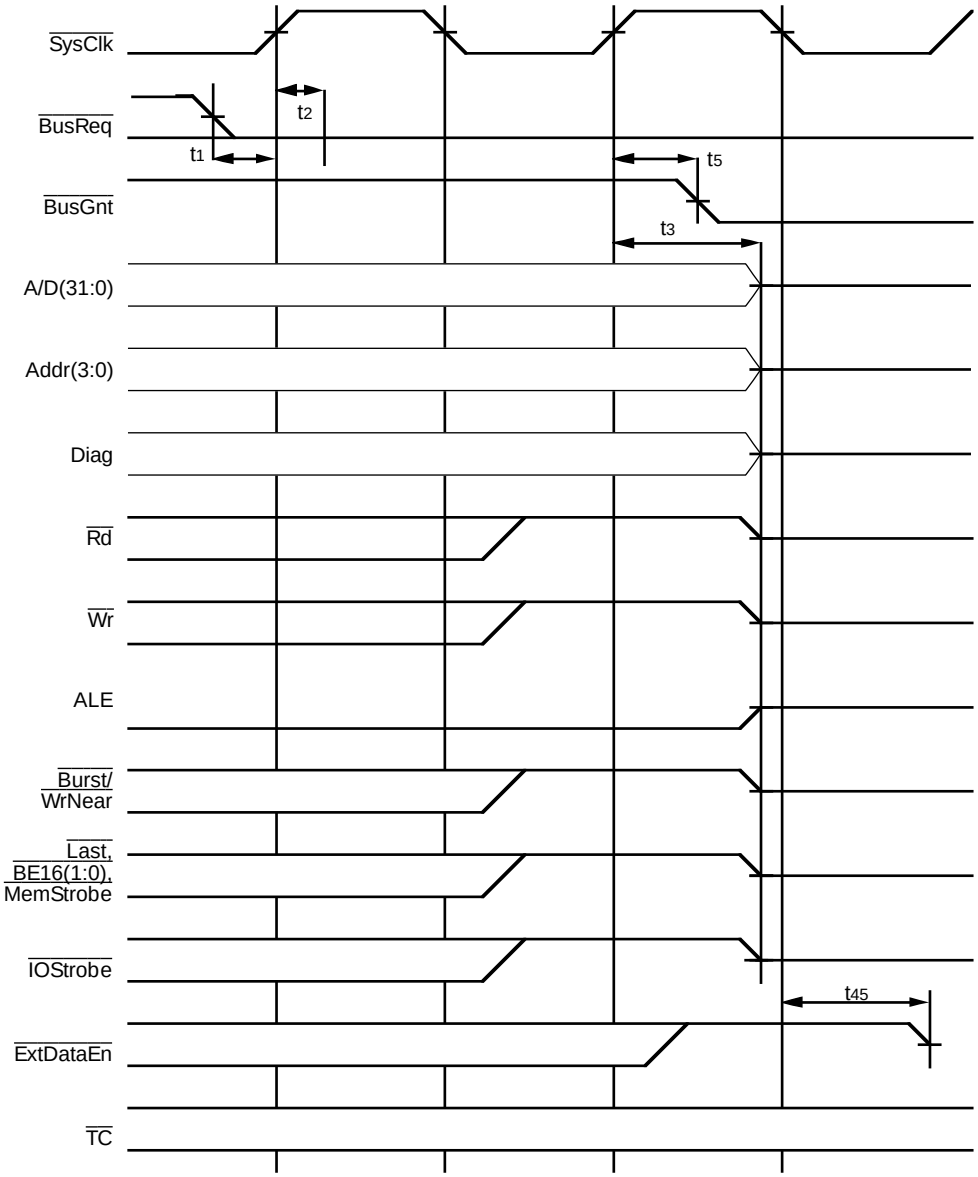


Figure 10.2. Bus Grant and Start of DMA Transaction

Relinquishing Mastership Back to the CPU

Figure 10.3 shows the end of a DMA cycle when not using the DMA Protocol mode. 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 if it hasn't already done so. 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.

Bus Grant Protocol CPU Initiated Bus Grant De-assertion

Figure 10.4 shows the middle of a DMA cycle when using the DMA Protocol mode. If BusGnt has been low for at least 1.5 clock periods and the CPU has a pending external bus request due to either a cache miss or uncached memory reference, then on the next rising edge of SysClk, BusGnt will be de-asserted. Even when this occurs, the mastership is not given back to the CPU until the DMA terminates the present transaction by releasing BusReq.

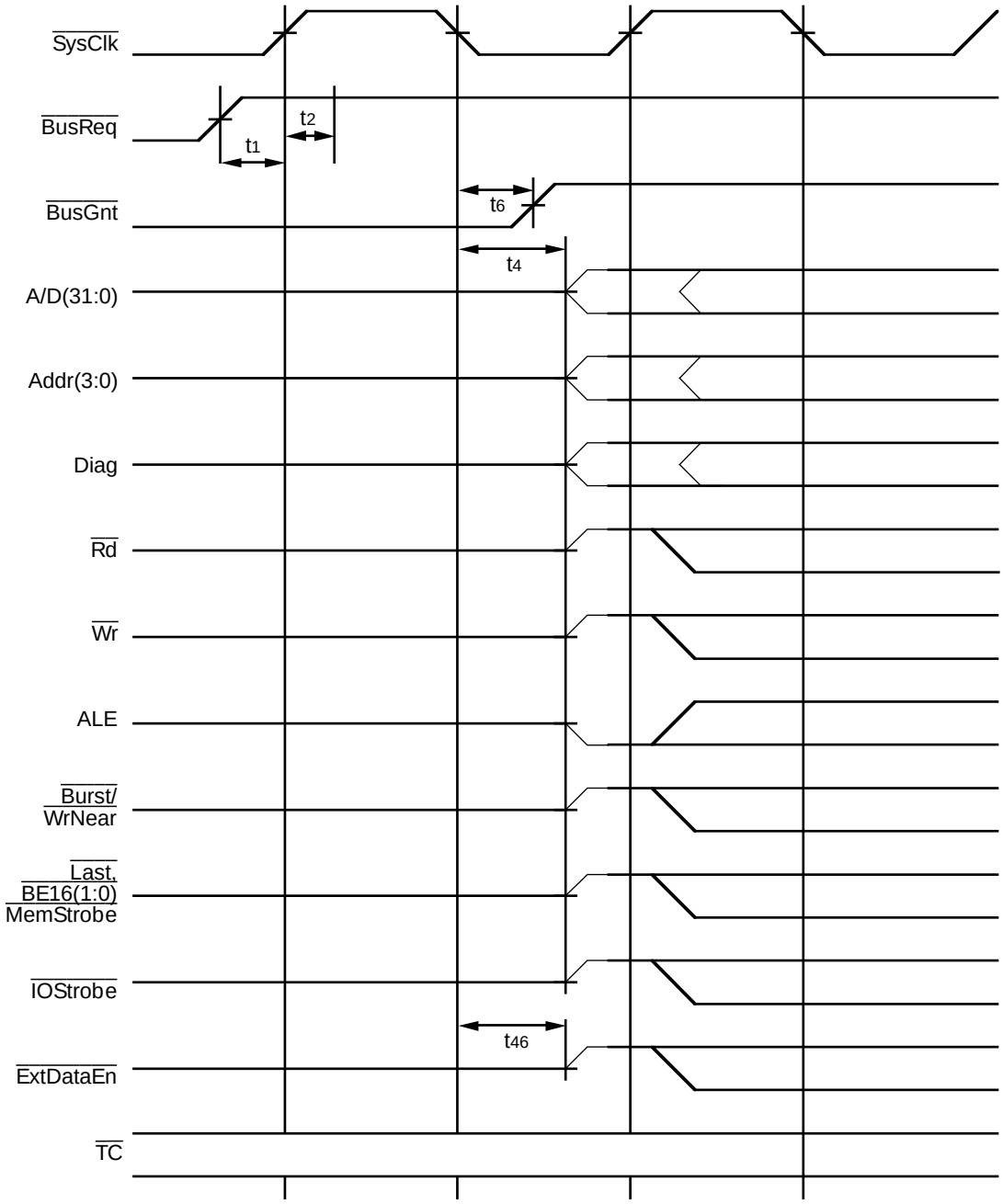


Figure 10.3. Regaining Bus Mastership

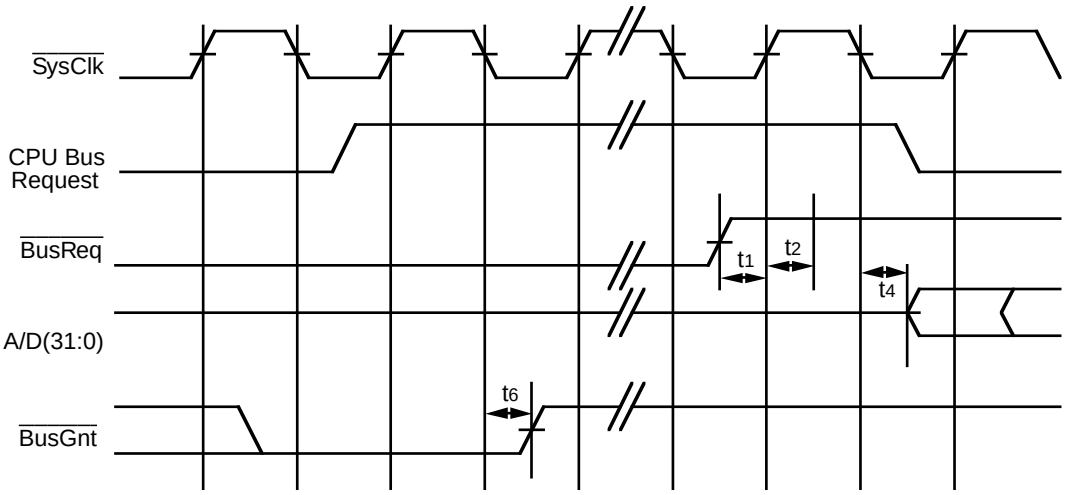


Figure 10.4. DMA Protocol BusGnt De-assertion

RESET INITIALIZATION AND INPUT CLOCKING

INTRODUCTION

This chapter discusses the reset initialization sequence required by the R3041. Also included is a discussion of the configuration mode selectable features of the processor, and of the software requirements of the boot program.

There are a number of selectable features in the R3041. These mode selectable features are determined by the polarity of the appropriate reset configuration mode 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 R3041 requires a very simple reset sequence. There are only two concerns for the system designer:

- That the set-up time and hold requirements of the reset configuration mode feature inputs with respect to the rising edge of $\overline{\text{Reset}}$ are met.
- That the minimum $\overline{\text{Reset}}$ pulse width is satisfied.

RESET CONFIGURATION MODE FEATURES

The R3041 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 reset mode feature bus at the negating edge of $\overline{\text{Reset}}$. The encoding of the mode selectable features on the reset mode feature bus is described in Table 11.1. Note that the R3041 uses both input pins and output pins which are tri-stated during $\overline{\text{Reset}}$ as inputs for the reset configuration mode features. Thus external state machines should not depend on the value of these pins until after $\overline{\text{Reset}}$ is negated.

Pin	Mode Feature
$\overline{\text{SInt(0)}}$	BigEndian
$\overline{\text{SInt(1)}}$	Reserved
$\overline{\text{SInt(2)}}$	Reserved
$\overline{\text{Int(3)}}$	$\overline{\text{AddrDisplayAndForceCacheMiss}}$
$\overline{\text{Int(4)}}$	Reserved
$\overline{\text{Int(5)}}$	Reserved
Addr(0)	$\overline{\text{ExtAddrHold}}$
Addr(1)	ReservedHigh
Addr(2)	$\overline{\text{BootProm8}}$
Addr(3)	$\overline{\text{BootProm16}}$
$\overline{\text{BE16(0)}}$	ReservedHigh
$\overline{\text{BE16(1)}}$	ReservedHigh

Table 11.1. R3041 Reset Configuration Mode Features

Internal Reset Pull-ups

The R3041 contains internal pull-up resistors on the following pins:

- reset configuration mode inputs: Addr(1:0), $\overline{\text{BE16(1:0)}}$
- tri-state input: $\overline{\text{TriState}}$

Addr(1:0), $\overline{\text{BE16(1:0)}}$, and $\overline{\text{TriState}}$ are designated as the no-connect Reserved pins in the R30xx family. Thus if left un-connected on the R3041, these pins have internal pull-ups to set them to their default values during reset. When using the internal pull-up resistors, warm resets require the same amount of reset time as power-up resets. If these pins are connected to an external device, then external pull-up / pull-down resistors or a tri-stateable device are required to initialize the reset configuration modes.

The other reset configuration inputs including $\overline{\text{SInt(0)}}$, $\overline{\text{Int(3)}}$ and Addr(3:2) do not have internal pull-up resistors and must pull-up or down these inputs externally.

A special case occurs when one of the Addr(3:0) or $\overline{\text{BE16(1:0)}}$ pins is pulled-down and is connected to a bipolar TTL input. Since $\overline{\text{BE16(1:0)}}$ are always pulled high, they will be excluded from the remainder of this section. In such a case, the external pull-down value would have to very low in order to supply the bipolar input enough current which conflicts with the CPU's ability to drive the signal high during normal operation after reset. This is in accordance with the following equations (where R is the pull-down resistance, V_{OH} and I_{OH} are relative to the CPU and I_{IL} and I_{IH} are relative to the chip being driven):

$$R_{PULLDOWN} \geq V_{OH} / (I_{OH} - I_{IH}) \text{ where } I_{OH} \geq I_{IH}$$
$$R_{PULLDOWN} \leq V_{IL} / I_{IL}$$

Using CMOS interfaces and/or memories will typically allow pull-up or pull-down values in the 3K to 10KΩ range. However, if bipolar interfaces and/or memories are used then assuming that the Addr(3:0) lines are attached to inputs which are on a bipolar buffer chip, solutions include:

Using a transceiver that is enabled to drive the Addr(3:0) pins during reset instead of using a buffer. External pull-downs (or pull-ups) are placed on the other side of the transceiver, since transceivers usually have a very large I_{OL} output current capability.

Using a transceiver instead of a buffer, since bipolar I/O pins typically have lower I_{IL} than dedicated bipolar input pins. The Addr(3:0) side of the transceiver is always disabled and external resistors are placed on the Addr(3:0) lines.

Choosing a buffer chip with a relatively low I_{IL} (of less than 600uA) and using external pull-down (or pull-up) resistors.

Reset Configuration Mode Pin Descriptions

Reserved

Reserved mode bits should be driven high if future compatibility is to be maintained with the R3041 family. Note that it is not mandatory that these pins be driven high.

BigEndian

Use **Big Endian Addressing:** 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 (inactive low), the processor will operate as a little-endian machine, and the RE bit will allow big-endian tasks to operate on a little-endian machine.

$\overline{\text{AddrDisplayAndForceCacheMiss}}$

If asserted (active low), two diagnostic functions are enabled:

Address Trace Display Mode: this mode (active low) will put the internally latched cached address out onto the A/D bus during unused bus cycles.

Force Cache Miss Mode: this mode (active low) causes all cacheable instruction and data references to do external bus accesses as if a cache miss occurred.

$\overline{\text{ExtAddrHold}}$

Extended Address Hold Time Mode: if asserted (active low) the address is held for an additional half clock past ALE de-asserting. $\overline{\text{DataEn}}$ is also delayed by one half clock. When not asserted (inactive high), the address is held only until ALE is de-asserted.

ReservedHigh

ReservedHigh mode bits are reserved for internal testing and must be driven high or if the pin is internally pulled-up, left un-connected.

$\overline{\text{BootProm8}}$

8-bit Boot PROM Mode. If asserted (active low), this mode will cause the port size mapping register to initialize all memory sub-regions to 8-bit ports instead of 32-bit ports. Thus an 8-bit boot PROM can be used to initialize the R3041. This mode can only be asserted if $\overline{\text{BootProm16}}$ is de-asserted.

$\overline{\text{BootProm16}}$

16-bit Boot PROM Mode: if asserted (active low), this mode will cause the port size mapping register to initialize all memory sub-regions to 16-bit ports instead of 32-bit ports. Thus a 16-bit boot PROM can be used to initialize the R3041. This mode can only be asserted if $\overline{\text{BootProm8}}$ is de-asserted.

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 as selected in the CP0 Cache Configuration register.
- Reverse Endianness capability enabled.
- Instruction Streaming enabled.
- Partial Word Stores enabled.

Other modes of the R3000A primarily pertain to its cache interface, which is incorporated within the R3041 and thus transparent to users of this processor.

RESET BEHAVIOR

While Reset is asserted, the processor maintains its interface in a state which allows the rest of the system to also be reset. Specifically:

- SysClk operates at one-half the ClkIn frequency.
- A/D is tri-stated
- ALE is driven negated (low).
- DataEn, Burst/WrNear,, BusGnt, Rd, and Wr are driven negated (high).
- MemStrobe, Last, and TC are driven negated (high).
- Diag is driven (value undefined).
- Addr(3:0), and BE16(1:0) are tri-stated.
- SBrCond(3:2) are configured as inputs and therefore tri-stated, i.e., ExtDataEn and IOStrobe are tri-stated.

The R3041 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 Exception Address 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 8.

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 at physical address 0x1FC0_0000. This address resides in uncached (non-burst), un-mapped memory, and thus does not require that the caches be initialized for the processor to execute boot code.

The processor needs to perform the following activities during boot:

- **Initialize the CP0 Status Register**
  The processor must be assured of having the kernel enabled to perform the boot sequence. Typically, a 'mtc0 rx, CO_SR' instruction is one of the first few instructions in the boot sequence. Specifically, co-processor usable bits, and cache control bits, must be set to the desired value before any data references (cached or uncached), diagnostics or initialization occur.

- **Initialize the CP0 Configuration Registers**
  The software should decide on the Cache Configuration, Port Sizes, and Bus Control during initialization.

- **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.

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

  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 to enter the user mode.

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 R3041 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.

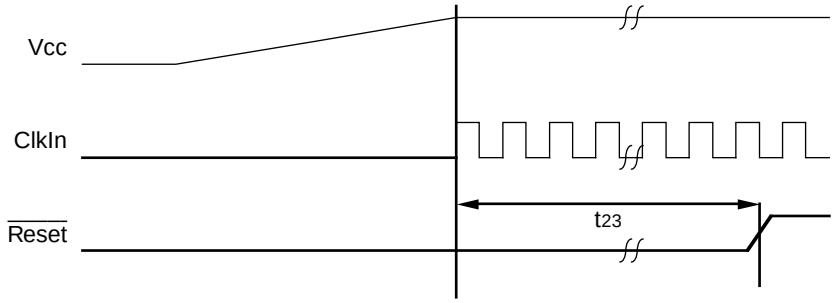


Figure 11.1. Cold Start

Figure 11.1 illustrates the power on reset requirements of the R30xx family.

Figure 11.2 illustrates the warm reset requirements of the processor when the reset configuration mode bits are driven.

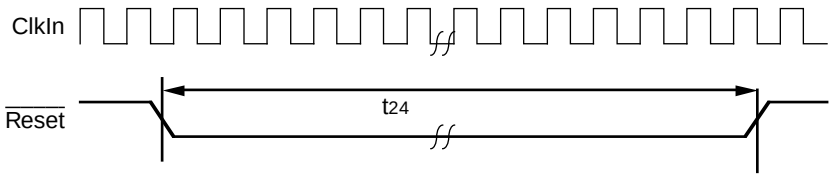


Figure 11.2. Warm Reset

Figure 11.3 illustrates the warm reset requirements of the processor when the reset configuration mode bits use the internal pull-ups.

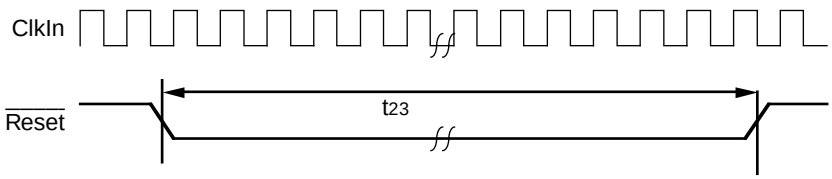


Figure 11.3. Warm Reset when using Internal Pull-Ups

Mode Initialization Timing Requirements

The mode initialization vectors are sampled by an internal 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 11.4.

Thus, the mode vectors have a set-up and hold time with respect to the rising

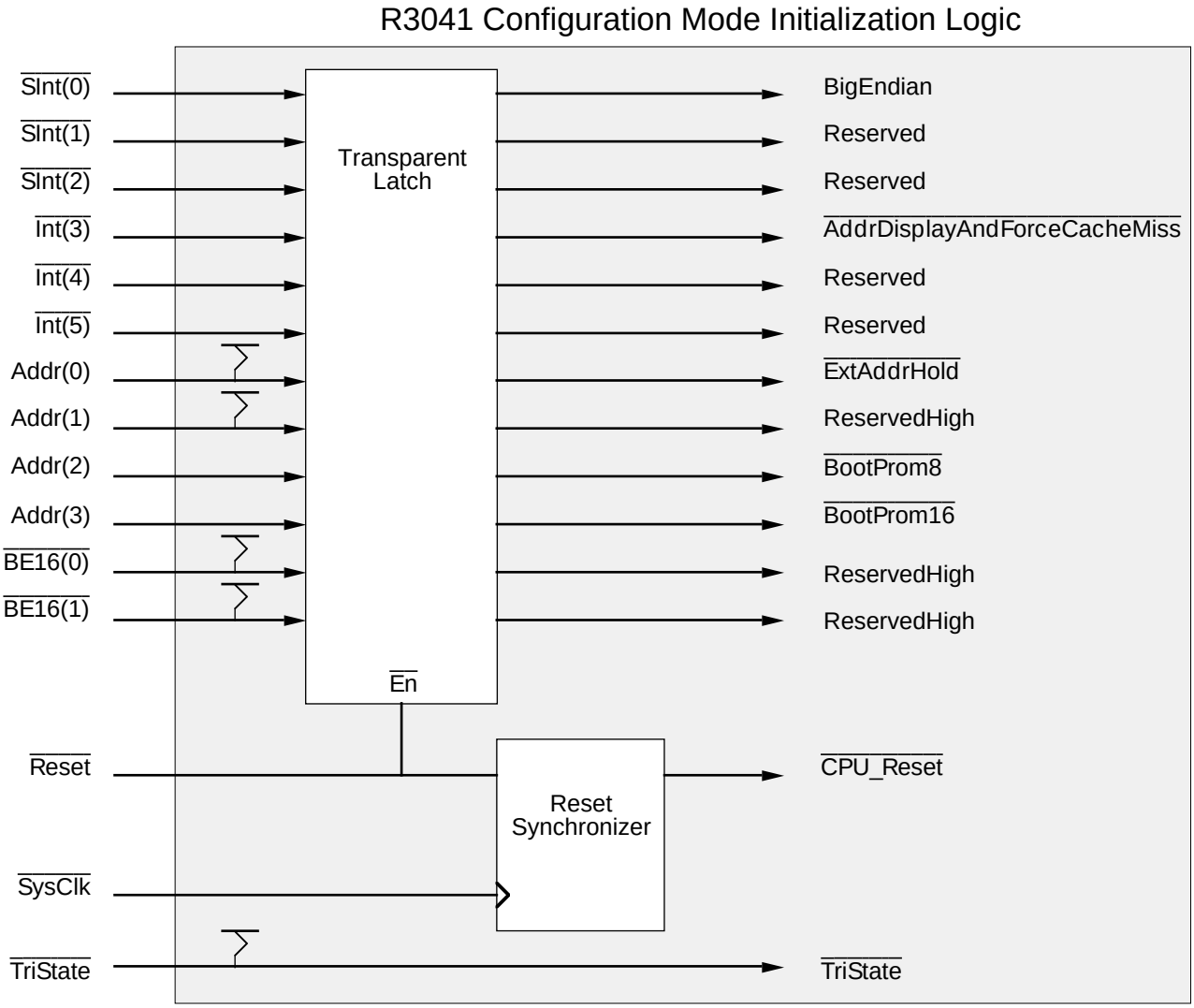


Figure 11.4. Configuration Mode Initialization Logic

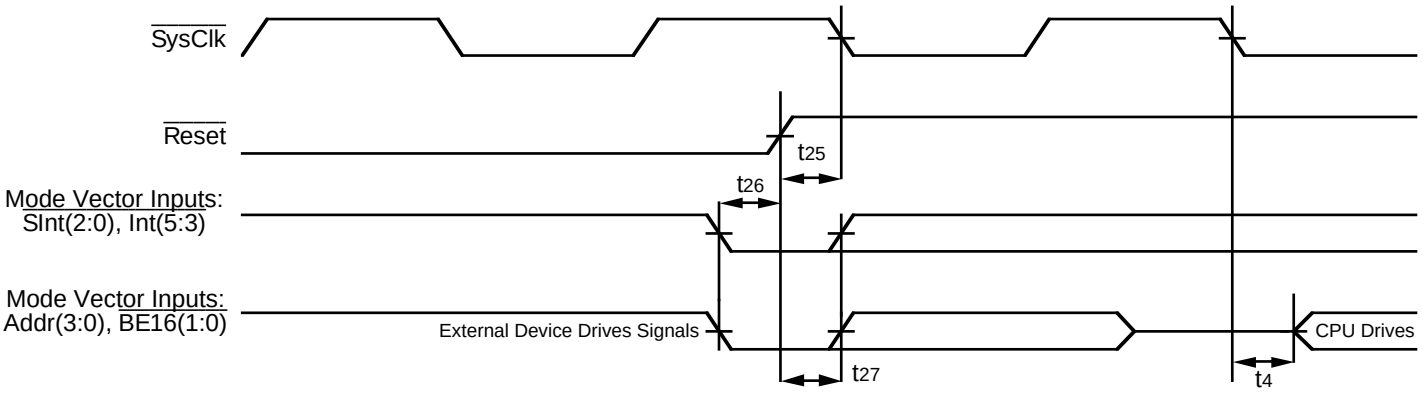


Figure 11.5. Mode Vector Timing

edge of $\overline{\text{Reset}}$, as illustrated in Figure 11.5.

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 $\overline{\text{Reset}}$ 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. Such a system should buffer the RC circuit such that a sufficiently fast monotonic rise time is generated which is capable of synchronously resetting any external state machines and logic at the same time as of resetting the CPU.

The $\overline{\text{Reset}}$ set-up time parameter can then be thought of as the amount of time $\overline{\text{Reset}}$ must be negated before the rising edge of $\overline{\text{SysClk}}$ 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.

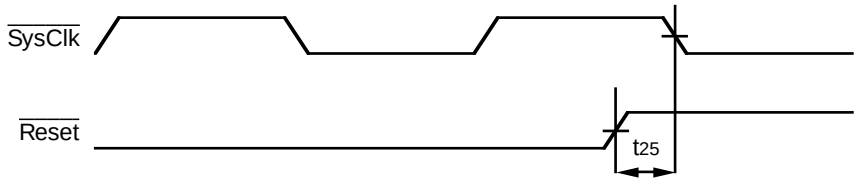


Figure 11.6. Reset Timing

Figure 11.6 illustrates the set-up time parameter of the R3041.

ClkIn Requirements

The input clock timing requirements are illustrated in Figure 11.7. The system designer does not need to be explicitly aware of the timing relationship between ClkIn and $\overline{\text{SysClk}}$. Note that $\overline{\text{SysClk}}$ is driven even during the $\overline{\text{Reset}}$ period as long as ClkIn is provided.

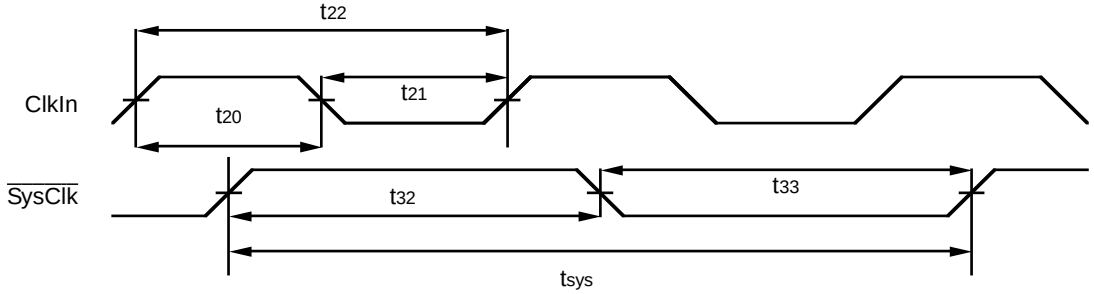


Figure 11.7. R3041 Clocking

INTRODUCTION

This chapter discusses particular features of the R3041 included to facilitate debugging of R3041-based systems. These features are intended to be used by an in-circuit emulator, in-circuit tester, board level tester, logic analyzer, hardware modeler, 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 and data cache misses forced, thus allowing all internal cache accesses to be displayed on the bus interface.
- The ability to tri-state all output pins including $\overline{\text{SysClk}}$, thus allowing an in-circuit emulator or tester to drive and control the output pins directly.
- The ability to deterministically set the phase relationship of the $\overline{\text{SysClk}}$ output relative to the ClkIn input. This feature allows board level testers and hardware modelers to control the $\overline{\text{SysClk}}$ output.
- The ability to distinguish data and instruction accesses via the Diag pin, allowing logic analyzers to do instruction disassembly (see Chapter 6).
- A software breakpoint instruction.

Note that the features described in this chapter are intended for initial debug or production testing rather than for functional use in a fielded end-user system.

ADDRESS DISPLAY

Activating the $\overline{\text{AddrDisplay}}$ mode with its reset configuration mode forces the CPU to display Instruction stream addresses on its A/D bus during idle bus cycles. Note that activating the $\overline{\text{AddrDisplay}}$ mode also activates the $\overline{\text{ForceCacheMiss}}$ mode described below. Refer to Figure 12.1 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(31:9) displays TAG(31:9)
- A/D(8:4) displays AddrLo(8:4)
- A/D(3:2) displays AddrLo(10:9)
- 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.

Note that the two cycles immediately before a main memory read may contain erroneous addresses. Specifically, if the memory read is due to an instruction cache miss, the address displayed two cycles before the assertion of $\overline{Rd}$ will be that of the cache contents, rather than the current program counter.

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.

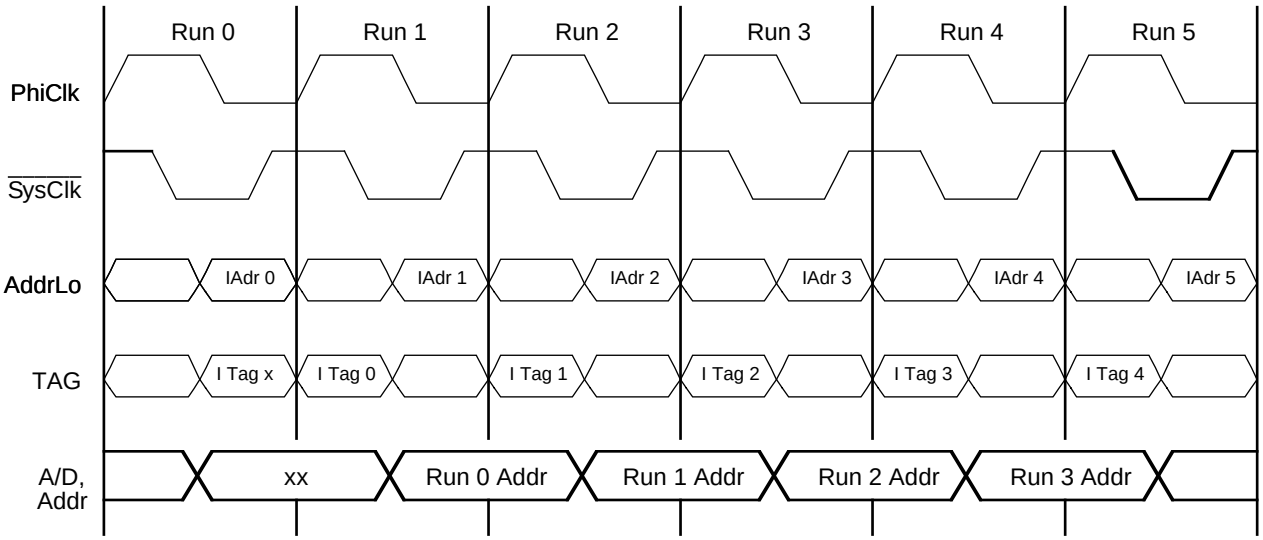


Figure 12.1. R3041 Debug Mode Instruction Address Display

FORCING INSTRUCTION AND DATA CACHE MISSES

Another feature for debugging is the ability to force an instruction and data cache miss. As with the $\overline{AddrDisplay}$ mode, this mode is not intended for use in a fielded production system.

The $\overline{ForceCacheMiss}$ mode is invoked with the same reset configuration mode bit as the $\overline{AddrDisplay}$ mode. Activating $\overline{ForceCacheMiss}$ forces all instruction and data cache accesses to be treated like cache misses. Thus cache accesses will be put onto the external A/D bus. Note that instruction cache misses and 4-word data block refills are still done in burst mode.

Tri-Stating All Outputs

The R3041 has a dedicated $\overline{TriState}$ input pin, which when asserted, disables all its outputs. This mode is useful for in-circuit emulators and testers which can then drive those pins to simulate the functions of the chip. Exiting this mode requires that a $\overline{Reset}$ be given before normal operation can take place. The pin description is as follows:

$\overline{TriState}$ **O**

Tri-State All Outputs: An active low input to the device which

requests that the processor tri-state all of its outputs. In addition to the outputs which are tri-stated during a DMA operation, SysClk, TC, and BusGnt are also tri-stated. TriState can be used for in-circuit testing and emulation during board production manufacture.

The R3041 TriState input pin is designated in the R3051 as the no-connect Rsvd(4) pin.

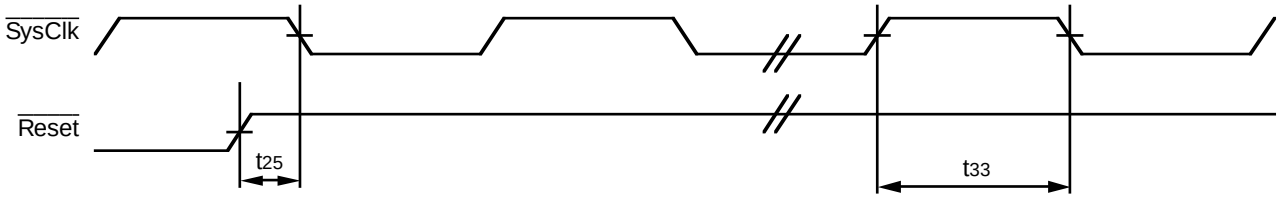


Figure 12.2. R3041 SysClk Phase Initialization Case A

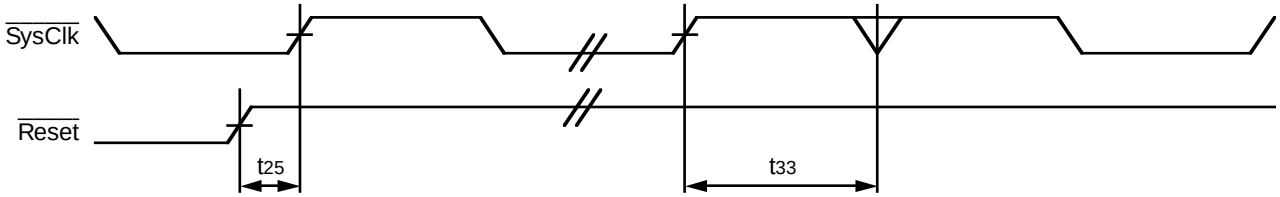


Figure 12.3. R3041 SysClk Phase Initialization Case B

Initializing SysClk for Test

Another feature for board level testing is the ability to initialize the phase of SysClk to its high phase. A low to high transition on Reset will cause the internally synchronized (delay of less than or equal to 2 clocks) version of Reset to always set SysClk high during its next phase. Thus the state of SysClk can be deterministically controlled within a known number of ClkIn transitions. The two cases are shown in Figures 12.2 and 12.3.

Using Diag for Instruction Disassembly

The R3041 provides a Diagnosis pin which during its data phase outputs whether a read transaction is the result of an instruction fetch or the result of a data fetch. This information is independent of the information given during the address phase of whether or not the read was a result of a cached or uncached read. Note that this pin is undefined on writes, however, all writes by necessity must be data writes. The pin description is as follows:

Diag O

Diagnostic Pin: During the address phase of the read transaction, this output indicates whether the read is a result of a cache miss (high) or an uncacheable reference (low).

During the remainder of the transfer, this output indicates whether the read is an instruction (high) or a data reference (low).

The Diag pin is undefined during write transactions.

This pin is useful in the initial debug of R3041 based systems.

The R3041 Diag output pin is designated in the R3051 family as the Diag(1) output pin.

Breakpoint Instruction

The R3051 family defines as described in Chapter 2, the breakpoint instruction, BREAK, that invokes an exception when executed. Thus debug kernel software can set breakpoints and single step through RAM based software.

EMULATION ISSUES

If the system designer wishes to allow an emulator to attach to the R3041, it is strongly recommended that the designer consider the following:

1.) For the CPU, either:
   A.) For a soldered on CPU, bring the "tri-state" pin to a test point that gets asserted when the emulator is attached (if the emulator is attached to the connector).
   B.) Use a socket for the CPU. PLCC-to-PLCC sockets are available from various manufacturers.
2.) In the memory controller state machine, either:
   A.) Reset the state machine whenever $\overline{Rd}$ or $\overline{Wr}$ de-assert.
   B.) Add an extra pulled-up input which, when asserted by the optional emulator pin, will reset the memory controller (about the current access). This allows the emulator to use overlay memory, with greater flexibility (e.g.: wait-states), faster or slower than board memory.

IDT recommends that the designer contact the 3rd party emulator vendor for additional considerations.

Integrated Device Technology, Inc.

COMPATIBILITY AMONG R30XX FAMILY DEVICES

APPENDIX A

INTRODUCTION

One of the unique advantages of the IDT R30xx family is the high level of pin, socket, and software compatibility across a very wide price-performance range. Although some devices do offer features not found in other family members, in general it is very straightforward to design a single system and set of software capable of using either the R3041, R3051, R3052, R3071, or R3081; the decision as to which processor to use can be made at board manufacturing time (as opposed to at design time) or as a program of field upgrades.

This chapter discusses compatibility issues among the various R30xx family members. The goal of this chapter is to provide the system designer with the understanding necessary to be able to interchange various R30xx family members in a single design environment, and with a single set of software tools.

SOFTWARE CONSIDERATIONS

In general, software considerations among the various family members can be summarized into the following areas:

- Cache Size differences. One of the obvious differences among the devices is the amount of instruction and data cache integrated on chip. Although the cache size is typically transparent to the applications software, the kernel must typically know how much cache to flush, etc. during system boot up. This manual presents an algorithm for determining the amount of cache on the executing processor; to insure compatibility, software should be written to dynamically determine the amount of cache on-chip.
- Differences in CP0 registers. Another area where the various family members differ slightly is in their implementation of CP0 registers. Table A.1 summarizes the CP0 registers of the various family members.
  In general, these differences are only relevant at system startup. The startup code should determine which device is running, and branch to a CPU specific CP0 initialization routine. Determining which CPU is executing is straightforward, and can be accomplished by reading the PrID register (to determine the presence of an R3041) and other simple tests. IDT/sim version 5.0 contains a module which can accomplish this identification.

Register	R3041	R3051/52	R3071/81
$0	rsvd	Index	Index
$1	rsvd	Random	Random
$2	BusCtrl	EntryLo	EntryLo
$3	CacheConfig	rsvd	Config
$4	rsvd	Context	Context
$5-$7	rsvd	rsvd	rsvd
$8	BadVA	BadVA	BadVA
$9	Count	rsvd	rsvd
$10	PortSize	EntryHi	EntryHi
$11	Compare	rsvd	rsvd
$12	Status	Status	Status
$13	Cause	Cause	Cause
$14	EPC	EPC	EPC
$15	PrID	PrID	PrID

Table A.1. CP0 Registers in the R30xx family

- "E" vs. "non-E" parts. In general, few applications will freely interchange devices with TLB's with those that do not. However, a given kernel source tree may be used across multiple applications; in this case, the startup code should examine the "TS" bit of the status register after reset to determine the presence of an on-chip TLB, and initialize the TLB if needed.
- Hardware vs. Software Floating Point. The R3081 offers a very high-performance floating point accelerator on-chip, while the R3041, R3051, R3052, and R3071 do not. In this case, it may be advantageous to generate two distinct binaries from the same source tree (one for hardware floating point and one for software). However, the R30xx architecture does support the ability to trap on floating point instructions (for later emulation), by negating the CP1 useable bit. Thus, initialization software may wish to determine the presence of an on-chip FPA, and initialize the CP1 useable bit accordingly.

HARDWARE CONSIDERATIONS

In general, the R3041, R3051/52, and R3071/R3081 offer the same system interface and pin-out, simplifying the interchange of the various family members. However, the R3041 and the R3071/R3081 offer some device specific features, which should be considered when designing a common board. The differences among the devices are summarized below.

R3041 Unique Features

The R3041 includes features targeting reduced system cost. Systems may wish to take full advantage of these features, in which case they may sacrifice the ability to readily interchange various CPUs in the design. Specifically, the R3041 can be interchanged with an R3051 or R3081 only in systems which implement a full 32-bit wide memory interface to the CPU, since the R3051 and R3081 do not offer the variable port width interface found in the R3041.

In general, the areas of differences between the R3041 and the R3051 are summarized below:

- The R3041 has a unique processor ID (PRId) of 0x0000_0700.
- The R3041 has the base address translation memory map only (w/o TLB).
- Different Instruction and Data Cache sizes.
- The R3041 software selects the DBlockRefill mode, rather than as a reset mode.
- The R3041 does not externally connect the BrCond(1:0) input pins.
- Diag(1:0) are not available on the R3041. Similar information is available with the Diag pin.
- The R3041 $\overline{\text{WrNear}}$ page size is decreased.
- The R3041 has additional/different reset modes.
- The R3041 includes new Co-processor 0 Config Registers.
- The R3041 can configure SBrCond(3:2) as outputs.
- The R3041 uses pins that are Reserved as no-connects on the R3051/R3081.
- The R3041 has an Extended Address Hold mode.
- The R3041 has a Slow Bus Turnaround mode with programmable bus wait timing.
- The R3041 has 8-bit and 16-bit ports with appropriately sized bus cycles. The R3041 can boot directly from an 8- or 16-bit wide PROM.
- The R3041 has additional outputs for $\overline{\text{BE16}}$(1:0), Last, $\overline{\text{MemStrobe}}$, $\overline{\text{ExtDataEn}}$, and $\overline{\text{IOStrobe}}$, and $\overline{\text{TC}}$.
- The R3041 has a read/write mask for $\overline{\text{BE}}$(3:0).
- The R3041 has an on-chip Timer with Count and Compare registers in CP0.
- The R3041 has a DMA protocol option.
- The R3041 is offered in a TQFP package, not available in other family

members.

R3071/R3081 Unique Features

The R3071/R3081 include features targeted to simplifying its use in high-frequency, high-performance systems. Systems may wish to take advantage of these features, in which case they may sacrifice some level of interchangeability with other CPUs. Key differences between the R3071/R3081 and the R3051 are summarized below:

- The R3081 includes an on-chip FPA.
- The R3071/R3081 features larger caches, which are configurable.
- The R3081on-chip FPA uses one of the six CPU interrupts; the corresponding input pin is logically not connected.
- The R3071/R3081 implements Half-frequency bus mode.
- The R3071/R3081 features Hardware cache coherency capability during DMA.
- The R3071/R3081 can use (or may require, for some speed grades) a 1x (rather than 2x) clock input.
- The R3071/R3081 $\overline{\text{WrNear}}$ page size is increased.
- The R3071/R3081 implement an additional CP0 Config register.
- The R3071/R3081 implements a power down (reduced frequency, halt) option.
- The R3071/R3081 features a dynamic data cache miss refill option.
- The R3071/R3081 BrCond(1) input is not available externally. It may be used as a “Run” output indicator in “debug” mode.
- The R3071/R3081 implement additional reset mode vectors.
- The R3071/R3081 differ slightly in their use of the reserved pins.

In general, the similarities in features allow the R3041 to use the same DRAM, I/O, and peripheral controllers that the R3051/52/71/81 use. It is possible by only using a subset of the interface features of the R3041 to also use the same system board socket as the R3051/52/71/81. However, many of these features, for instance the Extended Address Hold mode and the BootProm8 mode, allow inexpensive interface alternatives that often will justify a dedicated system board design.

Pin Description Differences

Table A.2 lists the significant R3051/52, R3071/81, and R3041 pin differences. These differences can easily be accommodated in a single board design, as described in this chapter.

R3051/52	R3071/81	R3041
Rsvd(0)	$\overline{\text{CohReq}}$	Addr(0)
Rsvd(1)	Rsvd(1)	Addr(1)
Rsvd(2)	Rsvd(2)	$\overline{\text{BE16(0)}}$
Rsvd(3)	Rsvd(3)	$\overline{\text{BE16(1)}}$
Rsvd(4)	Rsvd(4)	$\overline{\text{TriState}}$
BrCond(0)	BrCond(0)	$\overline{\text{MemStrobe}}$
BrCond(1)	unused/$\overline{\text{Run}}$	$\overline{\text{TC}}$
Diag(0)	Diag(0)	$\overline{\text{Last}}$
Diag(1)	Diag(1)	Diag

Table A.2. Pin Considerations Among R30xx Family Members

Reset Mode Selection

Table A.3 shows the various reset mode vectors available in the various family members. As can be seen from the table, there are differences in the mode vector options available in the different devices.

Designing a board which accommodates these differences is very straightforward:

- Use pull-up resistors on Addr(3:2). These pull-ups will have no effect on the R3051/52 or R3071/81; in the R3041, they will cause the device to boot from a 32-bit wide EPROM, which is compatible with the R3051/52 and R3071/81.
- Do not connect anything to the R3051 reserved pins. This will insure that the R3051/52 and R3071/81 function properly. In the R3041, this will negate the Extended Address Hold feature, causing the address to data transition of the processor A/D bus to be compatible with the R3051/52 and R3071/81.
- Use dip-switches with a MUX or 3-state buffer to select the reset initialization presented on the interrupt pins. Thus, selecting different reset mode vectors merely involves setting the dip switches.

Note that may systems may not need to do this either. For example, using pull-ups on the interrupt inputs will result in a BigEndian system for all devices, and in general disable the various device specific modes of the R3071/81 and R3041.

Pin	R3041	R3051/52	R3071/81
$\overline{\text{Int(5)}}$	Rsvd	Rsvd	$\overline{\text{CoherentDMA}}$
$\overline{\text{Int(4)}}$	Rsvd	Rsvd	$\overline{\text{1xClkEn}}$
$\overline{\text{Int(3)}}$	$\overline{\text{AddrDisplay}}$	Rsvd	$\overline{\text{1/2FreqBus}}$
$\overline{\text{SInt(2)}}$	Rsvd	DBlockRefill	DBlkRefill
$\overline{\text{SInt(1)}}$	Rsvd	$\overline{\text{Tri-State}}$	$\overline{\text{Tri-State}}$
$\overline{\text{SInt(0)}}$	BigEndian	BigEndian	BigEndian
Addr(3)	$\overline{\text{BootProm16}}$	N/A	N/A
Addr(2)	$\overline{\text{BootProm8}}$	N/A	N/
Rsvd(4)	$\overline{\text{Tri-State}}$	NC	NC
Rsvd(3)	Rsvd(*)	NC	NC
Rsvd(2)	Rsvd(*)	NC	NC
Rsvd(1)	Rsvd(*)	NC	NC
Rsvd(0)	$\overline{\text{ExtAddrHold}}$(*)	NC	NC

Table A.3. Reset Mode Vectors of R3041, R3051/52, and R3071/81

NOTES:
Rsvd: Must be driven high
N/A: Must not be driven
NC: Must not be connected
*: Contains an internal pull-up

Reserved No-Connect Pins

The R3051/52/71/81 contain not-to-be-connected reserved pins that R3041 systems may use. Table A.4 illustrates the different uses of the reserved pins.

To insure compatibility in systems using the same physical socket, various options exist:

- Use the internal pull-ups of the R3041 by extending the length of warm resets to be the same as that of power-up resets.
- Use external pull-ups which can be removed when an R3051/52/71/81 is used. This is so the R3051/52/71/81 Reserved pins have no chance of being driven.
- Use a tri-statable device to drive the reset configuration mode pins during reset and which then tri-state after reset when the R3041 is used, but which can be removed when the R3051/52/71/81 is used.

Of these options, the first is obviously the simplest; by not connecting the reserved pins, the R3051/52 and R3071/81 specifications will be met, and the extended features of the R3041 will not be accessed.

Pin	R3041	R3051/52	R3071/81
Rsvd(4)	$\overline{\text{Tri-State}}$	Rsvd	Rsvd
Rsvd(3)	$\overline{\text{BE16(1)}}$	Rsvd	Rsvd
Rsvd(2)	$\overline{\text{BE16(0)}}$	Rsvd	Rsvd
Rsvd1)	Addr(1)	Rsvd	Rsvd
Rsvd(0)	Addr(0)	Ravd	$\overline{\text{CohReq}}$

Table A.4. Rsvd Pins of R3041, R3051/52, and R3071/81

DIAG Pins

The R3051 features a pair of DIAG output pins which can be used during system debug. There are subtle differences in these pins in the various family members:

- The R3071/81 indicates the cacheability of data on writes, to simplify cache coherency. Since the R3041 and R3051/52 do not feature cache coherency, this feature would not be used in systems which wish to interchange the various family members.
- The R3041 uses a single DIAG pin (on the same physical pin as DIAG(1), to report the cacheability of an access. The other pin is used as the "Last" output of the R3041. Since the "Last" output is not available on the R3051/52 or R3071/81, systems designed to interchange CPUs will not use this output.

In general, the DIAG pins will only be used in system debug, rather than used to control some aspect of board operation. Thus, the differences in these pins will not impact the interchangeability of various CPUs.

BrCond(1:0), SBrCond(3:2)

There are also some differences among the devices in their treatment of the BrCond input pins. Specifically:

- The R3051 allows software to access all of BrCond(3:0).
- The R3071/81 reserves BrCond(1) for internal use by the FPA. Software can access the BrCond(3:2) and BrCond(0) inputs.
- The R3041 does not provide access to the BrCond(1:0) pins, which instead are used for other functions. Additionally, the R3041 defaults to using the SBrCond(3:2) pins as inputs on reset, although they can be used to provide other functions.

Thus, to insure CPU interchangeability, the system designer should provide pull-ups on BrCond(1:0), and only use BrCond(3:2). Of course, if these are also not used, pull-ups should be provided.

Slow Bus Turn Around Mode

Slow bus turn around on the R3041 allows extra cycles between changes in A/D bus direction. The R3071/81 also have a bus turn around feature, but the maximum number of extra cycles is fewer. Note that with the bus turnaround slowed, the R3041 continues to operate in a 100% compatible fashion with the R3051 (there is no R3051 transaction that "guarantees" a "quick" bus turnaround).

Note that there is a hardware solution to bus turnaround in the R3051, which will also work with the R3041 and R3071/81. This involves using the DMA arbiter to prevent the R3041/51/52/71/81 from issuing a bus cycle, and is explained in an applications note available from IDT.

Most systems that are using an R3041 and R3051 in the same socket may want to immediately reprogram the Bus Turn Around Control bits in the Bus Control CP0 register to '00' to match up exactly with the R3051 (and thus increase performance), instead of the default '11' which is used at reset, although it is not strictly necessary.

The R3081 FPA Interrupt

The on-chip FPA of the R3081 reports exceptions to the CPU using one of the general purpose interrupts. The corresponding input pin is ignored. Systems desiring to interchange an R3041/51/52/71 with an R3081 must reserve an interrupt pin for the FPA, and provide a pull-up for that signal. The R3081 Config register allows software to select any of the 6 interrupts; at reset, the default used is interrupt 3.

Half-Frequency Bus Mode

The R3071/81 allow the bus to operate at one-half the CPU frequency. When enabled, the bus will operate as for an R3041/51/52 operating at half the frequency of the R3071/81 CPU. Thus, this mode is entirely compatible with an R3041/51/52 at one-half the R3081 frequency.

In the R3071 and R3081, this feature is enabled as a reset option. Systems may choose to employ a jumper on this value, so that this feature may be selectively enabled when a R3071/R3081 is used, but the pin may be pulled-high or pulled-low when an R3041 is used.

Reduced Frequency/Halt Capability

This R3071/R3081 mode is incorporated to reduce power consumption when waiting for an interrupt or other external event. This mode is unavailable in an R3041/51/52.

Note that reduced frequency mode will appear to merely reduce the bus frequency of the R3071/R3081; most R3041/51 systems should operate correctly under this circumstance. However, the DRAM refresh timer, and other real-time timers, should either use a clock source other than the $\overline{\text{SysClk}}$ output, or reprogram their time constants, when this feature is used.

The R3041/51/52 does not offer the software stall capability of the R3071/R3081. Software executing on an R3041/51 which attempts to halt the processor will product no effect, and thus may result in erroneous software operation.

DMA Issues

Each of the CPUs can operate using R3051 compatible DMA. In these systems, the processor will attempt to continue execution out of on-chip cache during bus DMA; however, once the CPU core needs the bus, it will wait for the external master to relinquish the bus.

The R3071/R3081 allow hardware cache coherency during DMA writes. This capability may be disabled using the Coherent DMA Enable feature of the processor.

The R3041 implements a DMA Pulse Protocol, whereby the R3041 may negate BusGnt during an external DMA cycle to indicate that it wishes to regain bus mastership. This feature is not available on the other family members, and can be enabled or disabled via the R3041 CP0 registers.

To insure CPU compatibility, systems should disable both the R3071/R3081 cache coherency mode, and the R3041 Pulse Protocol, so that all devices will operate in R3051 compatible fashion.

Debug Features

Debug and in-circuit emulator features are not compatible between the R3041 and the R3051/52 and R3071/81. These debug features are intended for initial development and manufacturing tests and are not recommended for functional use on fielded end-user systems. These features include the Diag pin(s), Tri-State mode, AddrDisplay mode, and ForceCacheMiss mode.

$\overline{\text{WrNear}}$ Page Size

The various processors implement different choices for the size of the address compared for $\overline{\text{WrNear}}$ output assertion:

- The R3051/52 compare Address(31:10), compatible with 64kxn and deeper DRAMs.
- The R3071/81 compare Address(31:11), compatible with 256kxn and deeper DRAMs.
- The R3041 compares Address(31:8), compatible with 64kxn and deeper DRAMs in an 8-bit wide memory port.

To insure proper operation, the system designer can make one of two choices:

- Ignore the $\overline{\text{WrNear}}$ output, which simplifies system design but sacrifices performance.
- Always use 256kxn or deeper DRAMs.

Hardware Compatibility Summary

It is very simple to design a board capable of using any of the 5 CPUs described above. Table A.5 provides a summary of the design considerations to insure CPU interchangeability. In general, any board designed around the R3051 can easily be migrated up in performance to the R3071/R3081, or down in cost to the R3041.

Design Consideration	Compatible Solution
WrNear page size	Use 256kx4 or larger DRAM
Rsvd Pins	Leave disconnected
BrCond pins	Use only BrCond(3:2); Pullups on BrCond(1:0)
R3081 FPA Interrupt	Reserve one CPU interrupt for FPA; Use external Pull-up
DIAG pins	Use only for system debug; not a production function
Reset Logic	Pull-ups on Addr(3:2); no connects on reserved lines Dip switches and mux on Interrupt lines
DMA options	Use R3051 compatible DMA
Bus Turn-around	Meet R3051 timing or use DMA to add time

Table A.5. Summary of Hardware Design Considerations

SUMMARY

The R30xx family offers a unique level of compatibility among various CPUs, offering a wide range of price performance options for a single design. This capability extends not only to the signal interface, but to the actual footprint of the device itself. Using advanced packaging techniques, the 84-pin PLCC footprint is available across the entire family, including the entire frequency range of the family.

Although some systems will find it advantageous to use the features particular to a given CPU; others will find advantage in the ability to offer a single design, with real value added manufacturing and field upgrade capability. This choice is unique among high-performance embedded processors.