IDT MIPS Microprocessor Family Software Developer's Guide

September 2000

6024 Silver Creek Valley Road, San Jose, California 95138
Telephone: (800) 345-7015 • (408) 284-8200 • FAX: (408) 284-2775
Printed in U.S.A.

DISCLAMER

Integrated Device Technology, Inc. reserves the right to make changes to its products or specifications at any time, without notice, in order to improve design or performance and to supply the best possible product. IDT does not assume any responsibility for use of any circuitry described other than the circuitry embodied in an IDT product. The Company makes no representations that circuitry described herein is free from patent infringement or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent, patent rights or other rights, of Integrated Device Technology, Inc.

LIFE SUPPORT POLICY

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

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

About This Manual

Notes

This software developer's guide provides an introduction and design overview as well as more detailed descriptions for the following IDT product families:

- *IDT79RC30xx family of 32-bit RISC controllers*
- *IDT79RC323xx family of 32-bit enhanced MIPS-2 embedded devices*
- *IDT79RC4xxx 64-BIT RISCONTROLLER family of high-performance 64-bit CPUs*
- *IDT79RC5000 family of MIPS-4 ISA compatible CPU devices*

The reference for all real hardware (non-synthetic) assembler instructions is provided in a separate book starting with the present revision of this Software Reference Manual.

Summary of Contents

Chapter 1, "Introduction," presents an overview of IDT's microprocessor families, including a discussion of the CPU Pipeline, and a comparison of MIPS ISA and CISC architecture.

Chapter 2, "MIPS Architecture," discusses the high-level architecture from the programmer's point of view, including comparisons of the basic address space of the RC30xx, RC4600/4700, and RC4650.

Chapter 3, "System Control Co-Processor Architecture," discusses the aspects of the MIPS architecture that must be managed by the operating system, including details about CPU Control and Co-Processor 0.

Chapter 4, "Exception Management," examines the software techniques used to manage exceptions, and includes several code examples.

Chapter 5, "Cache Management," discusses IDT's implementation of the on-chip caches for instructions (I-cache) and data (D-cache).

Chapter 6, "Memory Management," discusses memory management and the Translation Lookaside Buffer (TLB). Also included is a discussion of the RC4650's simple base-bounds mechanism, which it uses instead of a TLB.

Chapter 7, "Reset Initialization," reviews the CPU reset, compares it to an exception, and includes information on bootstrap sequences and starting up an application.

Chapter 8, "Floating Point Co-Processor," describes the operation of floating points, and compares the implementations in the various IDT MIPS microprocessors.

Chapter 9, "Assembler Language Programming," discusses the techniques and conventions of reading and writing MIPS assembler code, including a complete table of assembler instructions.

Chapter 10, "C Programming," provides an overview of the principles of designing an efficient C runtime environment, including a discussion of optimization.

Chapter 11, "Portability Considerations," discusses the main facets of designing for portability.

Chapter 12, "Writing Power-On Diagnostics," provides a pragmatic, hands-on look at producing usable diagnostics in the MIPS environment.

Chapter 13, "Instruction Timing and Optimization," discusses the scheduling implications in using MIPS instructions, and includes information about additional hazards.

Chapter 14, "Software Tools for Board Bring-Up," describes the software tools typically used by IDT when debugging a new board.

Chapter 15, "Software Design Examples," contains examples of C programs for applications and embedded systems.

Notes

Chapter 16, "Assembly Language Programming Tips," contains tips on optimizing your programming in a MIPS environment.

Chapter 17, "Assembly Language Syntax," contains the details of assembler directives and other assembler language programming syntax issues.

Revision History

December, 1998: Initial publication.

September 26, 2000: In Chapter 3, RC5000 has been added to the list of other devices in the heading for section Cache Error (CacheErr) Register. Also, the following sentence has been added to this section: "When a read response (cached or uncached) returns with bad parity, this exception is also taken."

Table of Contents

Notes

Notes

3 System Control Co-Processor Architecture

4 Exception Management

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

List of Tables

Notes

Notes

List of Figures

Notes

Notes

Introduction

Notes

Overview

IDT offers a variety of MIPS ISA-compatible CPUs targeted to embedded applications. The variety of price performance points enables system developers to design products around various family members quickly, reducing time-to-market and development cost.

In the applications segments these products typically serve, software development is increasingly the larger part of system development. This manual is intended to augment the various device interface manuals, and is targeted to the firmware developer using the IDT CPUs. The manual covers the MIPS architecture as seen by the programmer and attempts to address the most common issues facing developers.

This manual draws upon concepts embodied in various IDT software development products: most notably, IDT/c—a multi-host, multi-target C compiler for the IDT microprocessor family—and IDT/sim—the target resident ROM monitor/debugger for IDT-based systems.

Many of the IDT/MIPS architecture concepts discussed here are supported in a similar fashion by toolchains from other vendors. The ultimate choice of a toolchain is beyond the scope of this manual; it is not the purpose of this manual to guide developers toward one tool set over another. For more information, ask your local IDT sales representative about the "AdvantageIDT" program.

IDT's Microprocessor Families

IDT currently offers a wide variety of microprocessors. All of these devices are based on the MIPS architecture, so software developed for one processor should be easily portable to other family members. However, the MIPS architecture does allow kernel specific features to be varied by implementation; thus, minor changes in reset code, cache management code, or even exception code may need to occur when changing between certain family members.

In addition, the instruction set architecture undergoes "constant improvement" whereby later cores offer architectural features not found in earlier generations. Management of these features also affects portability. IDT currently offers 5 families of the MIPS architecture:

- *The RC30xx family of 32-bit RISC microcontrollers includes the RC3051, RC3052, RC3081 and RC3041 processors.*
- *The different members of the family offer different price/performance trade-offs by varying the presence of FPA and/or TLB, and by varying the cache sizes. All of these are based around the original MIPS-I ISA R3000A core.*
- *The R3C6100 integrated RISC microprocessor/microcontroller.*
- *This device features the MIPS-I R3000A core integrated with cache and with system functions such as communications channels, memory controllers, and DMA controllers/channels. In general, descriptions of R30xx operations also apply to this device.*
- *The RC4xxx 64-bit RISController family of high-performance 64-bit CPUs.*
- *These devices are realized around an IDT proprietary implementation of an RC4400 compatible CPU core. They use the MIPS-3 ISA. Some devices feature an ISA extension for DSP applications.*
- *The RC5xxx family of MIPS-4 ISA compatible CPU devices.*
- *In this family, the current device features multiple instruction issue, large caches, and high frequency operation. The descriptions of RC4xxx operations (particularly kernel operations) also apply to these devices.*
- *The RC32300 family based on the RISCore32300 IDT proprietary core.*
- *This family implements several features of the MIPS-2 and MIPS-4 ISA along with specialized*

Notes

extensions for DSP applications and software debugging. The RC32364 device - first member of the family - brings RC4xxx performance levels at lower costs and lower power consumption.

Although most programming occurs using a high-level language (usually “C”), and with little awareness of the underlying system or processor architecture, certain operations require the programmer to use assembly programming, and/or be aware of the underlying system or processor structure. This manual is designed to be consulted when addressing these types of issues.

IDT CPU	CPU Core	ISA Level	I-Cache Size	D-Cach Size	FPA	TLB	Comments
RC3041	R3000A	MIPS-1	2KB	512B	No	No	Variable Port Width Interface
RC3051	R3000A	MIPS-1	4KB	2KB	No	Optional	
RC3052	R3000A	MIPS-1	8KB	2KB	No	Optional	
RC3071	R3000A	MIPS-1	8KB/16KB	4KB	No	Optional	Half-frequency bus option
RC3081	R3000A	MIPS-1	8KB/16KB	4KB	Yes	Optional	Half-frequency bus option
RC36100	R3000A	MIPS-1	4KB	1KB	No	No	Integrated system controller and peripherals
RC32364	RisCore32300	MIPS-2 + DSP + MIPS-4 extensions	8KB	2KB	No	Yes	32 bit, DSP, low power consumption, enhanced JTAG, MIPS-4 extensions
RC4600	Proprietary 64-bit RIS Controller	MIPS-3	16KB	16KB	Yes	Yes	
RC4700	Enhanced 64-bit RIS Controller	MIPS-3	16KB	16KB	Yes	Yes	Enhanced multiply performance
RC4650	64-bit RIS Controller + DSP	MIPS-3 + DSP	8KB	8KB	Single-precision	Base-Bounds	Cost reduced 64-bit RIS-Controller + DSP
RC4640	64-bit RIS Controller + DSP	MIPS-3 + DSP	8KB	8KB	Single-precision	Base-bounds	32-bit bus width
RC5000	R5000	MIPS-4	32KB	32KB	Yes	Yes	Multi-issue execution core

CPU Pipeline

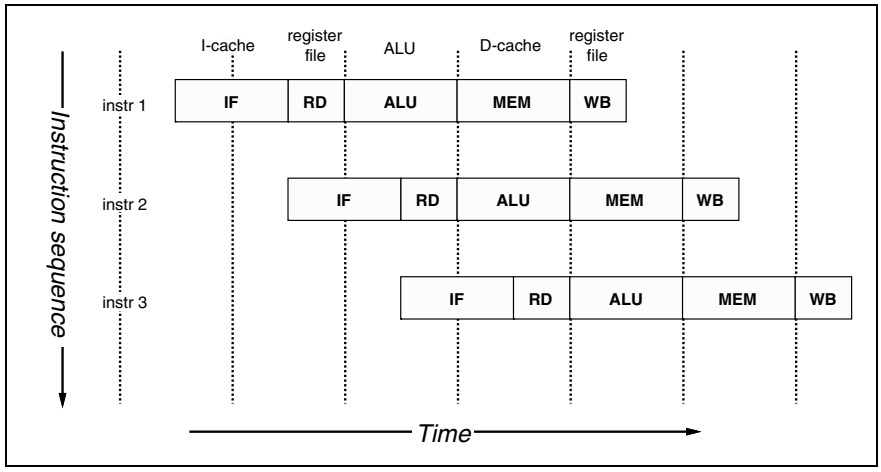


Figure 1.1 MIPS 5-stage pipeline

Notes

Pipelined processors operate by breaking instruction execution into multiple small independent "stages"; since the stages are independent, multiple instructions can be in varying states of completion at any one cycle. Also, this organization tends to facilitate higher frequencies of operation, since very complex activities can be broken down into "bite-sized" chunks. The result is that multiple instructions are executing at any one time, and that instructions are initiated (and completed) at very high frequency.

Pipelining success depends on the use of *caches*, which reduce the amount of time spent waiting for memory. The current IDT offerings use separate instruction and data caches, so the CPU can fetch an instruction and read or write a memory variable in the same clock phase. By combining high-frequency operation with high memory-bandwidth, very high-performance is achieved. The CPU normally runs from cache and a cache miss (where data or instructions have to be fetched from memory) is seen as an infrequent event.

Figure 1.1 shows a typical pipeline of an IDT CPU. This model assumes that instruction fetches and data accesses can be satisfied from the processor caches at the processor operation frequency. All instructions are rigidly defined to follow the same sequence of pipestages, even where the instruction does nothing at some stage. The net result is that, so long as it keeps hitting the cache, the CPU starts an instruction every clock. The pipeline stages are:

- ***Instruction fetch*** *(IF): gets the next instruction from the instruction cache (I-cache).*
- ***Read registers*** *(RD): decodes the instruction and fetches the contents of any CPU registers it uses.*
- ***Arithmetic/logic unit*** *(ALU): performs an arithmetic or logical operation in one clock (floating point math and integer multiply/divide can't be done in one clock and are handled differently; this is described later).*
- ***MEM****: the instruction can read/write memory variables in the data cache (D-cache). For typical programs, three out of four instructions do nothing in this stage, but allocating the stage to each instruction ensures that the processor never has two instructions wanting the data cache at the same time.*
- ***Write back*** *(WB): store the value obtained from an operation back to the register file.*

A pipeline limits the kinds of things instructions can do. For example:

- ***Instruction length****: ALL instructions are 32 bits (exactly one machine "word") long, so that they can be fetched in a constant time. This itself discourages complexity; there are not enough bits in the instruction to encode really complicated addressing modes, for example.*
- ***No arithmetic on memory variables****: data from cache or memory is obtained only in stage 4, which is much too late to be available to the ALU. Memory accesses occur only as simple load or store instructions which move the data to or from registers (this is described as a "load/store architecture").*

MIPS CPUs have 32 general-purpose registers, 3-operand arithmetical/logical instructions, and avoid complex and special-purpose instructions that compilers usually cannot generate. This makes the CPU an easy target for efficient optimizing compilers.

32-bit vs. 64-bit CPUs

IDT offers both 32-bit and 64-bit CPUs; the MIPS architecture defines 64-bit CPUs in such a way that they can cleanly run 32-bit applications. 32-bit and 64-bit processors operate the same, with respect to 8-bit or 16-bit data, as described later in this manual.

In the MIPS architecture, 64-bit CPUs implicitly sign-extend most 32-bit values, so that the value is interpreted the same when it is used as either a 32-bit value or as a 64-bit value. Additional instructions are provided when the size of the data is important--for example, when performing loads/stores or bit operations, or when testing for arithmetic carry of 32-bit values. The resulting architecture allows either 32-bit applications or 64-bit applications to be run on 64-bit processors.

Notes

In the reprogrammable computing world, the need for a 64-bit architecture is largely driven by needs to support large programs and large address spaces. In the embedded applications typically served by the IDT families, 64-bit addressing is rarely necessary. However, the ability to directly load, store, and manipulate 64-bit datums improves the performance of applications such as internetworking equipment and image decompression, which operate on large, but volatile, data streams.

Since 64-bit addressing is rarely needed, but 64-bit data sometimes are, most of the compiler tool chains allow the programmer to implement either an “A32D32”or “A32D64” model: that is, 32-bit addresses and 32-bit data, or 32-bit addresses with 32- or 64-bit datums. Control over these widths is typically achieved by a combination of variable declarations (“long long” or “double”) and/or compiler switches.

MIPS Architecture Levels

There are multiple generations of the MIPS architecture. The most commonly discussed are the MIPS-1, MIPS-2, MIPS-3, and MIPS-4 architectures. Successive generations implement all of the features of the previous generation, along with new instructions designed to solve key problems or enhance performance. Note that these ISA levels do not necessarily imply a particular structure for the MMU, caches, exception model, or other kernel specific resources. Thus, different implementations of ISA compatible chips may require different kernels. Figure 1.2 illustrates the relationship of the MIPS ISA levels.

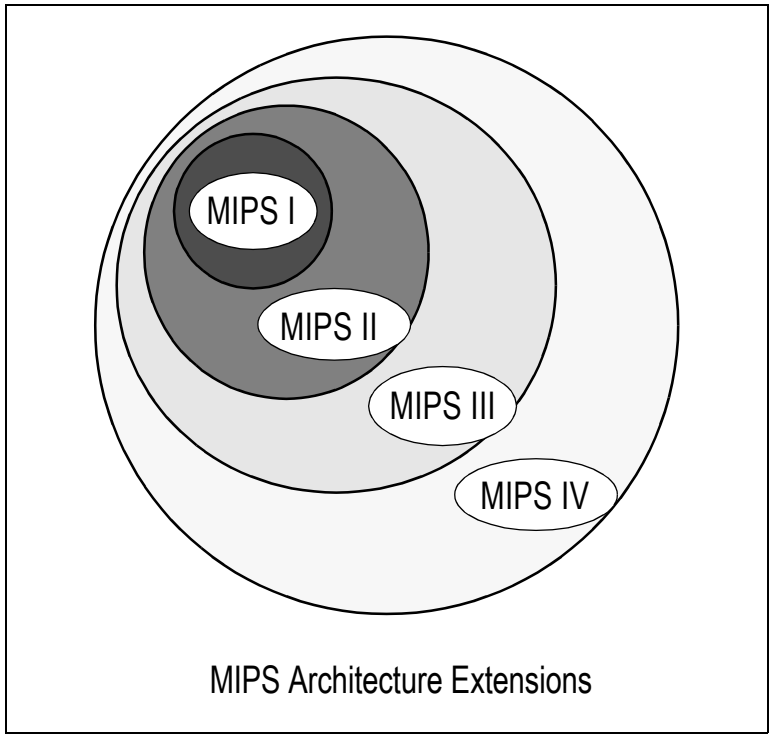


Figure 1.2 MIPS ISA Relationships

MIPS-1 is the ISA found in the R2000 and R3000 generation CPUs. It is a 32-bit ISA, and defines the basic instruction set. Any user application written with the MIPS-1 instruction set will operate correctly on all generations of the architecture.

The MIPS-2 ISA is also 32-bit. It adds some instructions to speed floating point data movement, eliminate software interlocks, add compiler driven branch-prediction, and other minor enhancements. This was first implemented in the MIPS R6000 ECL microprocessor.

The MIPS-3 ISA is a 64-bit ISA. In addition to supporting all MIPS-1 and MIPS-2 instructions, the MIPS-3 ISA contains 64-bit equivalents of certain earlier instructions that are sensitive to operand size (e.g. load double and load word are both supported), including doubleword (64-bit) data movement and arithmetic. This ISA was first implemented in the R4000 as a clean transition from the existing 32-bit architecture.

Notes

The MIPS-4 ISA adds instructions to improve floating point performance, such as multiply-add, and conditional move instructions. This ISA was first found in the MIPS R8000, and is also present in the R10000 and R5000. It is a 64-bit ISA. In addition, IDT has implemented small extensions to the ISA, notably in the RC4650 and RC4640. Although they are not strictly "MIPS extensions," they were added in cooperation with MIPS for the allocation of opcodes.

Similar additions were also made in the RISCore32300 core (RC32364 part) which, while being a MIPS-2 core, implements some features of MIPS-4 that do not involve 64-bit-ness, and also adds new instructions altogether.

MIPS ISA vs. CISC Architectures

Although the MIPS architecture is fairly straight-forward, there are a few features, visible only to assembly programmers, that may appear surprising at first. In addition, operations familiar to CISC architectures are irrelevant to the MIPS architecture. For example, the MIPS architecture does not mandate a stack pointer or stack usage; thus, programmers may be surprised to find that push/pop instructions do not exist directly.

Instruction Encoding Features

- *All instructions are 32-bits long: as mentioned above. This means, for example, that it is impossible to incorporate a 32-bit constant into a single instruction. A "load immediate" instruction is limited to a 16-bit value; a special "load upper immediate" must be followed by an "or immediate" to put a 32-bit constant value into a register. Note that this is true even for 64-bit instructions. That is, the opcodes remain encoded in 32-bits, even though the data operated upon is 64-bit.*
- *Instruction actions must fit the pipeline: actions can only be carried out in the designated pipeline phase, and must be complete in one clock. For example, the register writeback phase provides for just one value to be stored in the register file, so instructions can only change one register.*
- *3-operand instructions: arithmetic/logical operations don't have to specify memory locations, so there are plenty of instruction bits to define two independent source and one destination register. Compilers love 3-operand instructions, which give optimizers more scope to improve the code which handles complex expressions.*
- *32 registers: compilers like a large (but not necessarily too large) number of registers, but there is a cost in context-saving and in encoding the registers to be used by an instruction. Register $0 always returns zero, to give a compact encoding of that useful constant.*
- *No condition codes: the MIPS architecture does not provide condition code flags implicitly set by arithmetical operations. The motivation is to make sure that execution state is stored in one place – the register file. Conditional branches (in MIPS) test a single register for sign/zero, or a pair of registers for equality/inequality.*

Addressing and Memory Accesses

Memory references are always register loads or stores: arithmetic on memory variables complicates, and therefore, slows down the pipeline. Memory references only occur by explicit load or store instructions. The large register file allows a useful working set of data to be in registers.

Only one data addressing mode[1]: all loads and stores define the memory location with a single base register value modified by a 16-bit signed displacement. Note that the assembler and compiler tools can use the $0 register, along with the immediate value, to synthesize additional addressing modes from this one directly supported mode.

Byte-addressing: the instruction set includes load/store operations for 8- and 16-bit variables (referred to as *byte* and *halfword*). Partial-word load instructions come in two flavors – sign-extend and zero-extend.

Loads/stores must be address-aligned : memory word operations can only load or store data from a single 4-byte aligned word; halfword operations must be aligned on half-word addresses. Techniques to handle unaligned data efficiently will be explained later.

1. The MIPS-4 ISA does allow register+register addressing for floating-point operands.

Notes

Jump instructions: The op-code field in a MIPS instruction is 6 bits; leaving 26 bits to define the target of a jump. Since all instructions are 4-byte aligned in memory the two least-significant address bits need not be stored, allowing an address range of 2^{28} = 256Mbytes. Rather than make this branch PC-relative, this is interpreted as an absolute address within a 256Mbyte "segment". In theory, this could impose a limit on the size of a single program; in reality, it hasn't been a problem.

Branches out of segment can be achieved by using a *jr* instruction, using the contents of a register as the target. Conditional branches have a 16-bit displacement field (2^{18} byte range since instructions are 4-byte aligned) which is interpreted as a signed PC-relative displacement. Compilers can only code a simple conditional branch instruction, if they know that the target will be within 128Kbytes of the instruction following the branch.

Operations Not Directly Supported

- *No byte or halfword arithmetic: All arithmetical and logical operations are performed on 32-bit (or 64-bit) quantities. Byte and/or halfword arithmetic would require significant extra resources, many more op-codes. Where a program explicitly does arithmetic as short or char, the compiler must insert extra code to ensure that wraparound and overflows have the appropriate effect.*
- *No special stack support: conventional MIPS assembler usage does define a sp register, but the hardware treats sp just like any other register. There is a recommended format for the stack frame layout of subroutines, so that programs can mix modules from different languages and compilers. It is recommended that programmers stick to these software conventions, but there are no hardware requirements.*
- *Minimal subroutine overhead: There is one special feature; jump instructions have a "jump and link" option which stores the return address into a register. $31 is the default, so for convenience, and by convention, $31 becomes the "return address" register.*
- *Minimal interrupt overhead: The MIPS architecture makes very few presumptions about system exception handling, allowing fast response and a wide variety of software models. In the RC30xx family, the CPU stashes away the restart location in the special register EPC, and modifies the machine state just enough to signal why the trap happened, and to disallow further interrupts; then it jumps to a single predefined location. Everything else is determined by software.*

  Note: On an interrupt or trap, a MIPS CPU *does not* store anything on a stack, or write memory, or preserve any registers by itself.

By convention, two registers ($k0, $k1) are reserved so that interrupt/trap routines can "bootstrap" themselves—it is impossible to do anything on a MIPS CPU without using some registers. For a program running in any system which takes interrupts or traps, the values of these registers may change at any time, and thus should not be used.

Multiply and Divide Operations

The MIPS CPU does have an asynchronous integer multiply/divide unit. With its own special output registers, the multiply unit is relatively independent of the rest of the CPU.

Programmer-Visible Pipeline Effects

Programmers of MIPS CPUs must also be aware of certain MIPS pipeline effects. Specifically, the results of certain operations may not be available in the next instruction; the programmer needs to be explicitly aware of such cases.

Notes

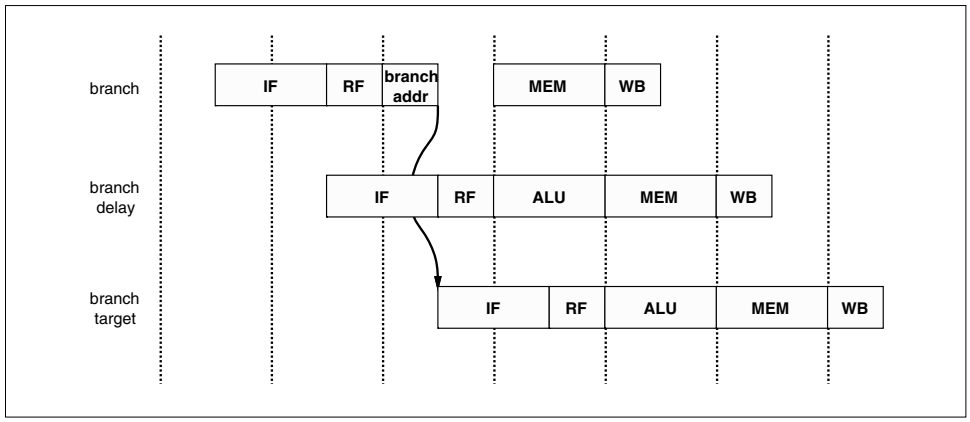


Figure 1.3 The pipeline and branch delays

- *Delayed branches: the pipeline structure of the MIPS CPU (see The pipeline and branch delays) means that when a jump instruction reaches the "execute" phase and a new program counter is generated, the instruction after the jump will already have been decoded. Rather than discard this potentially useful work, the architecture rules state that the instruction after a branch is always executed before the instruction at the target of the branch.*

For the "branch likely" instructions introduced in the MIPS-2 ISA, the delay slot is "nullified" if a conditional branch is not taken.

The pipeline and branch delays show that a special path is provided through the ALU to make the branch address available a half-clock early, ensuring that there is only a one cycle delay before the outcome of the branch is determined and the appropriate instruction flow (branch taken or not taken) is initiated.

It is the responsibility of the compiler system or the assembler-programmer to allow for, and use frequently, the instruction which would otherwise have been placed before the branch can be moved into the delay slot. Where nothing useful can be done, the delay slot is filled with a "nop" (no-op, or no-operation) instruction. Many MIPS assemblers will hide this feature from the programmer unless explicitly told not to, as described later.

- *Load data not available to next instruction: another consequence of the pipeline is that a load instruction's data arrives from the cache/memory system AFTER the next instruction's ALU phase starts – so it is not possible to use the data from a load in the following instruction. See Figure 1.4 for the pipeline and load delays sequence. On the MIPS-1 architecture, the programmer must insure that this rule is not violated.*

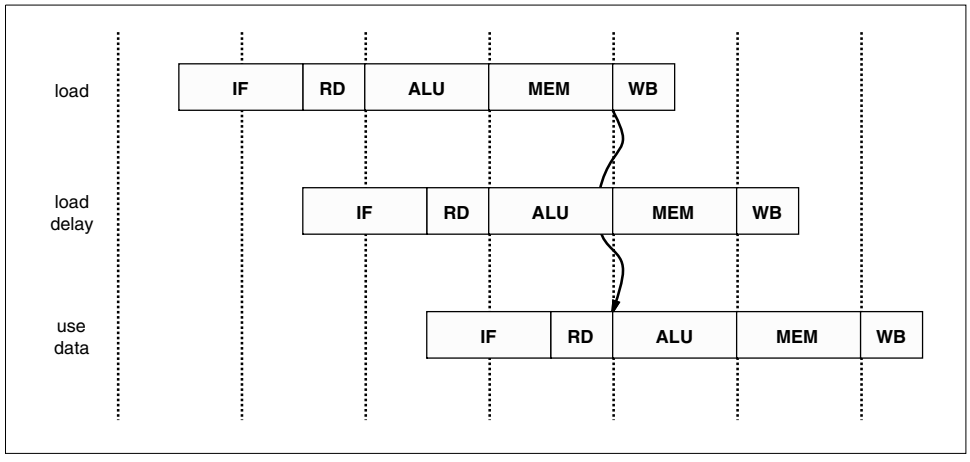


Figure 1.4 The pipeline and load delays

Notes

Again, most assemblers will hide this if they can. Frequently, the assembler can move an instruction which is independent of the load into the load delay slot; in the worst case, it can insert a NOP to insure proper program execution. The MIPS-2 ISA does not require a NOP to be placed in unfilled load delay slots.

Notes on Machine and Assembler Language

To simplify assembly level programming, the MIPS Corp's assembler (and many other MIPS assemblers) provides a set of "synthetic" instructions. A synthetic instruction is a common assembly level operation that the assembler will map into one or more ISA operating instruction. This mapping can be more intelligent than a mere macro expansion. For example, an immediate load may map into one instruction if the datum is small enough, or multiple instructions if the datum is larger. These instructions can dramatically simplify assembly level programming and assembly code readability.

This is obviously useful, but can be confusing. This manual will try to use synthetic instructions sparingly, and indicate when it happens. Moreover, the instruction tables below will consistently distinguish between synthetic and machine instructions. These features help human programmers; most compilers generate instructions which correspond one-for-one with machine code. However, some compilers will generate synthetic instructions. These are some of the helpful operations that the assembler can perform:

- *32-bit load immediates: The programmer can code a load with any value (including a memory location which will be computed at link time), and the assembler will break it down into two instructions to load the high and low half of the value.*
- *Load from memory location: The programmer can code a load from a memory-resident variable. The assembler will normally replace this by loading a temporary register with the high-order half of the variable's address, followed by a load whose displacement is the low-order half of the address.*

  Of course, this does not apply to variables defined inside C functions, which are implemented either in registers or on the stack.
- *Efficient access to memory variables: some C programs contain many references to static or extern variables, and a two-instruction sequence to load/store any of them is expensive. Some compilation systems, with run-time support, get around this.*

  Certain variables are selected at compile/assemble time (by default MIPS Corp's assembler selects variables which occupy 8 or less bytes of storage) and kept together in a single section of memory which must be smaller than 64Kbytes. The run-time system then initializes one register ($28 or gp (global pointer) by convention) to point to the middle of this section.

  Loads and stores to these variables can now be coded as a gp relative load or store.
- *More types of branch condition: the assembler synthesizes a full set of branches conditional on an arithmetic test between two registers.*
- *Simple or different forms of instructions: unary operations such as not and neg are produced as a nor or sub with the zero-valued register $0.*

  Two-operand forms of 3-operand instructions can be written; the assembler will put the result back into the first-specified register.
- *Hiding the branch delay slot: in normal coding most assemblers will not allow access to the branch delay slot, and may re-organize the instruction sequence substantially in search of something useful to do in the delay slot. An assembler directive, .set noreorder, is available where this must not happen.*
- *Hiding the load delay: many assemblers will detect an attempt to use the result of a load in the next instruction, and will either move code around or insert a* ***nop*** *(for MIPS-1).*
- *Unaligned transfers: the "unaligned" load/store instructions will fetch halfword and word quantities correctly, even if the target address turns out to be unaligned.*
- *Other pipeline corrections: some instructions (such as those which use the integer multiply unit) have additional constraints that are implementation specific (see the Appendix on hazards). Many assemblers will just "handle" these cases automatically, or at least warn the programmer about possible hazards violations.*
- *Other optimizations: some MIPS instructions (particularly floating point) take multiple clocks to pro-*

Notes

duce results. However, the hardware is "interlocked", so the programmer does not need to be aware of these delays to write correct programs. But the MIPS Corporation's assembler is particularly aggressive in these circumstances and will perform substantial code movement to try to make it run faster. This may need to be considered when debugging.

In general, it is best to use a dis-assembler utility to disassemble a resulting binary during debug. This will show the system designers the true code sequence being executed and "uncover" the modifications made by the assembler.

Notes

MIPS Architecture

Notes

Programmer's View of the Processor Architecture

This chapter describes the assembly programmer's view of the CPU architecture, in terms of registers, instructions, and computational resources. This viewpoint corresponds to an assembly programmer writing user applications.

Information about kernel software development (such as handling interrupts, traps, and cache and memory management) are described in later chapters.

Registers

There are 32 general purpose registers: $0 to $31. These are 32 bits wide in the RC30xx, and 64 bits wide in the RC4xxx and the RC5000. Two, and only two, are special to the hardware:

- *$0 always returns zero, writes are ignored.*
- *$31 is used by the normal subroutine-calling instructions (jal, bgezal, and bttzal) for the return address. Note that the call-by-register version (jalr) can use any register for the return address, though commonly it also uses $31.*

In all other respects, all registers are identical and can be used in any instruction. There is no programmer visible program counter. The subroutine transfer instructions store in a link register, which can be used to return from a subroutine. Also, there are no condition codes or status bits needed by the user-level programmer.

There are two registers associated with the integer multiplier. These registers—referred to as "HI" and "LO"—contain the product result of a multiply operation or the quotient and remainder of a divide. The result of multiplication may be up to 128-bits in case of the RC4xxx or up to 64-bits in case of the RC3xxx. HI/LO also function as accumulators in the "multiply-accumulate" instructions *mad/madu* in the RC4650 and the RC32364. The RC4650 and the RC32364 also have a true 3 operand multiply instruction which does not use HI/LO registers at all.

The floating point math co-processor (called *FPA* for floating point accelerator, also some times referred to as FPU in this manual), if available, adds 32 floating point registers[1]; in simple assembler language they are just called $0 to $31 again – the fact that these are floating point registers is implicitly defined by the instruction. Actually, in case of RC30xx, only the 16 even-numbered registers are usable for math; but they can be used for either single-precision (32 bit) or double-precision (64-bit) numbers. When performing double-precision arithmetic, the higher odd numbered register holds the low-order bits of the even numbered register specified in the instruction. Only moves between integer and FPA, or FPA load/store instructions, will refer to odd-numbered registers.

The RC4600/4700/RC5000 offers full 64-bit operations and its floating point unit can be configured in one of the following two ways:

- *When the FR bit in the CPU Status register equals 0, the floating point unit is configured for sixteen 64-bit registers for double-precision values or thirty-two 32-bit registers for single-precision values.*
- *When the FR bit in the CPU Status register equals 1, the floating point unit is configured for thirty-two 64-bit registers. Each register can hold single- or double-precision values.*

The RC4650 supports single precision floating point math only. Its floating point unit can be configured in one of the following two ways:

- *When the FR bit in the CPU Status register equals 0, the floating point unit is configured for sixteen 32-bit single-precision registers.*

1. The FPA also has a different set of registers called "co-processor 1 registers" for control purposes. These are typically used to manage the actions/state of the FPA, and should not be confused with the FPA data registers.

Notes

- *When the FR bit in the CPU Status register equals 1, the floating point unit is configured for thirty-two 32-bit single-precision registers.*

Some processors also support an MMU (RC30xx E, RC32364, RC4600, RC4700, RC5000). The RC4650 only supports base-bounds translation. There are dedicated registers to handle memory and address translation.

Conventional Names and Uses of General-Purpose Registers

Although the hardware makes few rules about the use of registers, their practical use is governed by a number of conventions. These conventions allow inter-changeability of tools and operating systems as well as library modules and are the compiler calling conventions that must be strictly followed.

With the conventional uses of the registers, go a set of conventional names. Given the need to fit in with the conventions, use of the conventional names is pretty much mandatory. The common names are described in Table 2.1.

Reg No	Name	Used for
0	zero	Always returns 0, writes are ignored.
1	at	(assembler temporary) Used by assembler (for synthetic instruction expansion)
2-3	v0-v1	Values (except FP) returned by subroutine
4-7	a0-a3	(arguments) First four parameters for a subroutine
8-15	t0-t7	(temporaries) subroutines may use without saving
24-25	t8-t9	
16-23	s0-s7	Subroutine “register variables”; a subroutine, which will change one of these, must save the old value and restore it before it exits, so the *calling* routine sees their values preserved.
26-27	k0-k1	Reserved for use by interrupt/trap handler.
28	gp	global pointer - some runtime systems maintain this to give easy access to “static” or “extern” variables.
29	sp	stack pointer
30	s8/fp	9th register variable. Subroutines which need one can use this as a ‘‘frame pointer’’.
31	ra	Return address for subroutine

Table 2.1 Conventional Register Names

Notes on Conventional Register Names

- *at: this register is often used inside the synthetic instructions generated by the assembler. If the programmer must use it explicitly, the directive.set noat stops the assembler from using it (there are some synthetic instructions that cause the assembler to issue warnings).*
- *v0-v1: used when returning non-floating-point values from a subroutine. To return anything bigger than registers, memory must be used (described in a later chapter).*
- *a0-a3: used to pass the first four integer parameters to a subroutine, may be different for mixture of integer and floating point parameters. The actual convention is fully described in a later chapter.*
- *t0-t9: by convention, subroutines may use these values without preserving them. This makes them easy to use as “temporaries” when evaluating expressions – but a caller must assume that they will be destroyed by a subroutine call.*
- *s0-s8: by convention, subroutines must guarantee that the values of these registers on exit are the same as they were on entry – either by not using them, or by saving them on the stack and restoring before exit.*

Notes

- *k0-k1: reserved for use by the trap/interrupt routines, which will not restore their original value; so they are of little use to anyone else.*
- *gp: (global pointer). Not all compilation systems or OS loaders support gp. If supported, it will point to a load-time-determined location in the midst of your static data. This means that loads and stores to data lying within 32Kbytes either side of the gp value can be performed in a single instruction using gp as the base register.*

  Without the global pointer, loading data from a static memory area takes two instructions: one to load the most significant bits of the 32-bit constant address computed by the compiler and loader, and one to do the data load.
  To use gp, a compiler must know at compile time that a datum will end up linked within a 64Kbyte range of memory locations. In practice it can only guess. The usual practice is to put "small" global data items in the area pointed to by gp, and to get the linker to fail if it gets too big. The definition of what is "small" can typically be specified with a compiler switch (most compilers use "-G"). The most common default size is 8 bytes or less.
- *sp: (stack pointer). Since it takes explicit instructions to raise and lower the stack pointer, it is generally done only on subroutine entry and exit; and it is the responsibility of the subroutine being called to do this. sp is normally adjusted, on entry, to the lowest point that the stack will need to reach at any point in the subroutine. Now the compiler can access stack variables by a constant offset from sp. Stack usage conventions are explained in a later chapter.*
- *fp: (also known as s8). A subroutine will use a "frame pointer" to keep track of the stack if it extends the stack by run-time. Some languages may do this explicitly (for many toolchains); C programs, which use the "alloca" library routine, will do so.*

  In this case, it is not possible to access stack variables from sp, so fp is initialized by the function prologue to a constant position relative to the function's stack frame. Note that a "frame pointer" subroutine may call or be called by subroutines that do not use the frame pointer; so the subroutine must preserve the value of fp.
- *ra: (return address). On entry to any subroutine, ra holds the address to which control should be returned – so a subroutine typically ends with the instruction "jr ra".*

  Subroutines, which themselves call subroutines, must first save *ra,* usually on the stack.

Integer Multiply Unit and Registers

The multiply unit consumes a small amount of die area but dramatically improves performance (and cache performance) over "multiply step" operations. It's basic operation is to multiply two 32-bit values together to produce a 64-bit result, which is stored in two 32-bit registers (called "hi" and "lo") which are private to the multiply unit. Instructions *mfhi*, *mflo* are defined to copy the result out into general registers.

In the RC4xxx, two 64-bit values may be multiplied to produce a 128-bit result. However, in the case of the RC4xxx, if the operands are 32-bits long only, they must be valid sign-extended values. For high level language programming this is not an issue, as the compiler will take care of the sign extension requirements; but it should be checked when porting assembler-level code from RC30xx to RC4xxx.

Unlike results for integer operations, the multiply result registers are *interlocked*. An attempt to read out the results before the multiplication is complete results in the CPU being stopped until the operation completes.

The integer multiply unit will also perform an integer division between values in two general-purpose registers; in this case the ''lo'' register stores the quotient, and the ''hi'' register the remainder.

In the RC30xx family, multiply operations take 12 clocks and division takes 35.

Instruction cycle timing for multiply and double multiply (64-bit) as well as divide and double divide for members of the RC4xxx family and RC32364 is listed in Table 2.2 The 3-operand multiply (MUL) and multiply-add (MAD) are available in RC4650 and RC32364 only. Multiply-subtract (MSUB) is available in RC32364 only.

Notes

Instruction	RC4600	RC4650	RC4700	RC3000	RC5000	RC32C364
MULT/U	10	4	8	12	4	4
DIV/U	42	36	42	35	36	36
DMULT/U	12	6	10	N/A	8	N/A
DDIV/U	74	68	74	N/A	68	N/A
MAD/U	N/A	4 or 3	N/A	N/A	N/A	4
MUL	N/A	4	N/A	N/A	N/A	4
MSUB/U	N/A	N/A	N/A	N/A	N/A	4

Table 2.2 Multiply and Divide Instruction Cycle Timing

The assembler has a synthetic multiply operation which starts the multiply and then retrieves the result into an ordinary register. Note that an assembler may even substitute a series of shifts and adds for multiplication by a constant, to improve execution speed.

Multiply/divide results are written into “hi” and “lo” as soon as they are available; the effect is not deferred until the writeback pipeline stage, as with writes to general purpose (GP) registers. If a *mfhi* or *mflo* instruction is interrupted by some kind of exception before it reaches the writeback stage of the pipeline, it will be aborted with the intention of restarting it. However, a subsequent multiply instruction which has passed the ALU stage will continue (in parallel with exception processing) and would overwrite the “hi” and “lo” register values, so that the re-execution of the *mfhi* would get wrong (i.e. new) data. For this reason it is recommended that a multiply should not be started within two instructions of an *mfhi/mflo*. The assembler will avoid doing this when possible.

Compilers will often generate code to trap on errors, particularly on divide by zero. Frequently, this instruction sequence is placed after the divide is initiated, to allow it to execute concurrently with the divide (and avoid a performance loss).

Instructions *mthi*, *mtlo* are defined to setup the internal registers from general-purpose registers. They are essential to restore the values of ‘‘hi’’ and ‘‘lo’’ when returning from an exception, but probably not for anything else.

The RC4650 and the RC32364 provide a couple of multiplication instructions that set it apart from the other members of its family. The *mad* (multiply and accumulate) instruction and its unsigned counterpart *madu* use the “hi” and “lo” registers as accumulators. In addition to these, another new instruction *mul* offers true 3 operand multiplication and eliminates the extra step of moving the result from the “lo” register to a general purpose register.

The RC32364 provides one more multiplication instruction. The MSUB (multiply and subtract) instruction and its unsigned counterpart MSUBU. These are similar to MAD and MADU except that they subtract from the accumulator instead of adding to it.

Instruction Types

A full list of RC30xx family integer instructions is presented in Appendix A. Floating point instructions are listed in Appendix B of this manual. The integer and floating point instructions are listed in appendixes at the end of this manual.

The MIPS ISA uses three instruction encoding formats. For the most part, instructions are in numerical order. Occasionally, to simplify reading, the list is re-ordered for clarity.

Instruction Terminology

The instruction encodings have been chosen to facilitate the design of a high-frequency CPU. Specifically:

- *The instruction encodings do reveal portions of the internal CPU design. Although there are vari-*

Notes

able encodings, those fields which are required very early in the pipeline are encoded in a very regular way:

- *Source registers are always in the same place: so that the CPU can fetch two instructions from the integer register file without any conditional decoding. Some instructions may not need both registers – but since the register file is designed to provide two source values on every clock nothing has been lost.*
- *16-bit constant is always in the same place: permitting the appropriate instruction bits to be fed directly into the ALU's input multiplexer, without conditional shifts.*

Throughout this manual, the description of various instructions will also refer to various subfields of the instruction, as follows:

op	The basic op-code, 6 bits long. Instructions with large sub-fields (for example, large immediate values, such as required for the "long" *j/jal* instructions, or arithmetic with a 16-bit constant) have a unique "op" field. Other instructions are classified in groups sharing an "op" value, distinguished by other fields ("op2" etc.).
rs, rs1, rs2	One or two fields identifying source registers.
rd	The register to be written by this instruction.
sa	Shift-amount: How far to shift, used in shift-by-constant instructions.
op2	Sub-code field used for the 3-register arithmetic/logical group of instructions (*op* value of zero).
offset	16-bit signed *word* offset defining the destination of a "PC-relative" branch. The branch target will be the instruction offset words away from the delay slot instruction; so a branch-to-self has an offset of -1.
target	26-bit *word* address to be jumped to (it corresponds to a 28-bit byte address, which is always word-aligned). The high-order 4 bits of the target address can't be specified by this instruction, and are taken from the address of the jump instruction. This means that these instructions can reach anywhere in the 256Mbyte region around the instructions' location. To jump further use a *jr* (jump register) instruction.
constant	16-bit integer constant for "immediate" arithmetic or logic operations. Arithmetic may or logical may not be sign extended (such as add sign-xtnd or zero-xtnd).
mf	Yet another extended opcode field, this time used by ''co-processor'' type instructions.
rg	Field which may hold a source or destination register.
crg	Field to hold the number of a CPU control register (different from the integer register file). Called ''crs''/''crd'' in contexts where it must be a source/destination respectively.

Loading and Storing: Addressing Modes

As mentioned above, there is only one basic addressing mode. Any load or store machine instruction can be written as:

operation dest-reg, offset(src-reg)

e.g.:lw $1, offset($2); sw $3, offset($4)

Any of the integer registers can be used for the destination and source. The offset is a sign extended integer, 16-bit number (so can be anywhere between -32768 and 32767); the program address used for the load is the sum of *dest-reg* and the *offset*. This address mode is normally enough to select a particular member of a C structure ("offset" being the distance between the start of the structure and the member required); or an array indexed by a constant; it is also enough to reference function variables from the stack or frame pointer; to provide a reasonable sized global area around the *gp* value for static and extern variables.

The assembler synthesizes simple direct addressing mode, to load the values of memory variables whose address can be computed at link time. More complex modes such as double-register or scaled index must be implemented with two or more instructions.

Notes

Data Types in Memory and Registers

The RC30xx family CPUs can load or store between 1 and 4 bytes in a single operation. Naming conventions are used in the documentation and to build instruction mnemonics:

"C" name	MIPS name	Size(bytes)	Assembler mnemonic
long long	doubleword	8	"d" as in *ld*†
int	word	4 (8‡)	''w'' as in *lw*
long	word	4 (8‡)	''w'' as in *lw*
short	halfword	2	''h'' as in *lh*
char	byte	1	''b'' as in *lb*

Notes:
†MIPS-III instruction; for RC4xxx and RC5000 only.
‡Some "C" compilers for RC4xxx will allow efficient 64-bit integer math with a special compile-time switch (e.g. -mint64 switch in IDT/C), where integer size is 8 bytes and assembler instruction "ld/sd" are used to load/store 8 bytes at a time.

Table 2.3 Naming Conventions

Integer Data Types

Byte and halfword loads come in two flavors:

- *Sign-extend: lb and lh load the value into the least significant bits of the 32/64-bit register, but fill the high order bits by copying the "sign bit" (bit 7 of a byte, bit 16 of a half-word). This correctly converts a signed value to a 32/64-bit signed integer.*
- *Zero-extend: instructions lbu and lhu load the value into the least significant bits of a 32/64-bit register, with the high order bits filled with zero. This correctly converts an unsigned value in memory to the corresponding 32/64-bit unsigned integer value; so byte value 254 becomes 32/64-bit value 254.*

For example, if the value 0xFE (-2, or 254 if interpreted as unsigned), then:

```
lb      t2, 0(t1)
lbu     t3, 0(t1)
```

will leave *t2* holding the value 0xFFFF FFFE (-2 as signed 32-bit) and *t3* holding the value 0x0000 00FE (254 as signed or unsigned 32-bit).

Subtle differences in the way shorter integers are extended to longer ones are a historical cause of C portability problems, and the modern C standard has elaborate rules. On machines like the MIPS, which does not support 8- or 16-bit precision arithmetic directly, expressions involving *short* or *char* variables are less efficient than word operations.

Unaligned Loads and Stores Using Assembler

Loads and stores in the MIPS architecture must be aligned. Half-words must be loaded from 2-byte boundaries, words from 4-byte boundaries; in the RC4xxx family, double words must be loaded from 8-byte boundaries. A load instruction with an unaligned address will cause a trap. If needed, software can provide a trap handler which will emulate the desired load operation and hide this feature from the application, at substantial performance cost.

The MIPS architecture provides a hardware mechanism to access unaligned data. The machine instructions are *lwl* (load word left), *lwr* (load word right), *swl* (store word left) and *swr* (store word right). For the RC4600/4700/5000, the equivalent 64-bit instructions are *ldl* (load double left), *ldr* (load double right), *sdl* (store double left) and *sdr* (store double right) which deal with up to 8 bytes as opposed to 4 as described in this section.

Notes

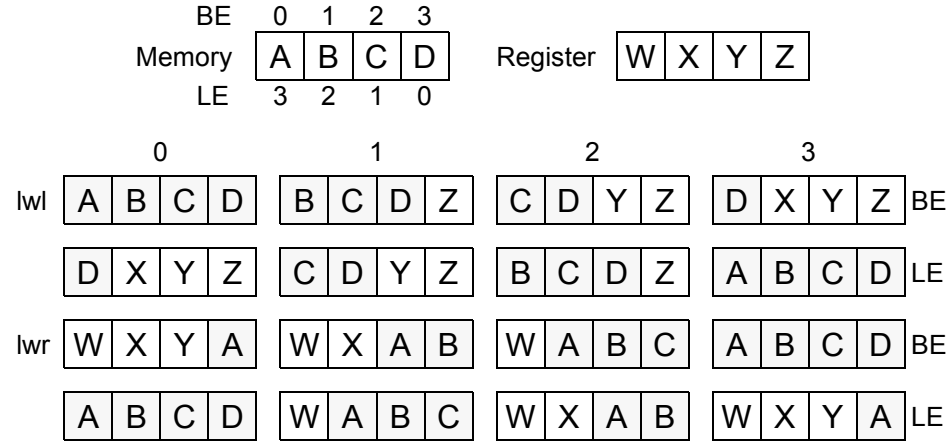
lwl loads one to four bytes from the least significant portion of a word starting from the specified address to the high (left) portion of the destination register; *lwr* loads from one to four bytes from the most significant portion of a word starting from the specified address to the low (right) portion of the register. To load a word into register *v0* from an arbitrary address in register *a0*, use the sequence

```
lwl     v0, 0(a0)
lwr     v0, 3(a0)
```

on a big endian machine and the sequence

```
lwr     v0, 0(a0)
lwl     v0, 3(a0)
```

on a little endian machine (see diagram below). This sequence is generated by the macro-instruction



ulw (unaligned load word). A macro-instruction *ulh* (unaligned load half) is also provided, synthesized by two loads and a shift. Note that the CPU allows the instruction pairs to use the same destination register without an intervening instruction; however, at least one instruction must be executed between the instruction pair and using the value of the destination register.

swl stores one to four bytes from the high (left) portion of the source register to the least significant portion of a word starting from the specified address; *swr* stores from one to four bytes from the low (right) portion of the register to the most significant portion of a word starting from the specified address. To store a word from register *v0* to an arbitrary address in register *a0*, use the sequence

```
swl     v0, 0(a0)
swr     v0, 3(a0)
```

on a big endian machine and the sequence

```
swr     v0, 0(a0)
swl     v0, 3(a0)
```

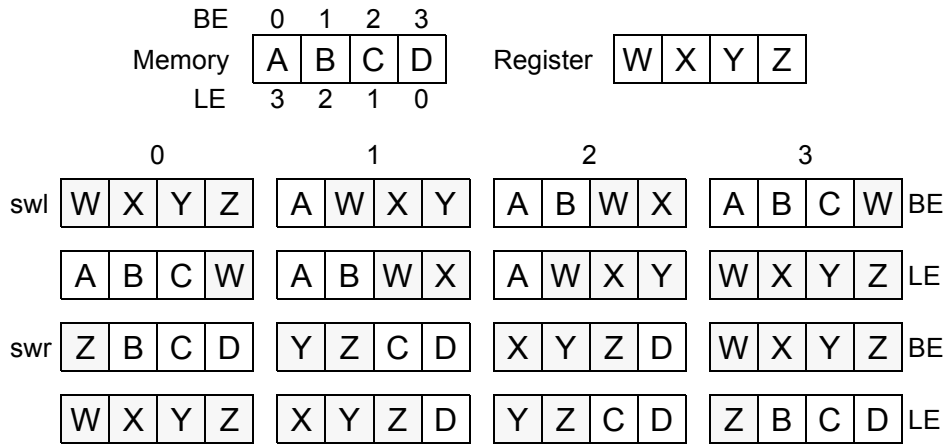
on a little endian machine (see diagram below). Note that the CPU uses hardware control to effect the partial word writes; *swl* and *swr* will ***not*** work if the destination device does not honor the byte enables, whereas *lwl* and *lwr* will work with any word-wide device.

Unaligned Loads and Stores Using "C"

All data items declared by "C" code will be correctly aligned by default. In certain embedded applications such as intelligent networking and datacom, the data structures may be forced to have unaligned data if the data structures are packed - no bytes between data structures or between fields within a structure to force alignment - to minimize memory usage. In such cases, a "C" programmer may be required to descend to assembler coding to deal with unaligned data accesses. Some "C" compilers, such as the IDT/C compiler, provide a mechanism to achieve unaligned data accesses through "C" itself.

The keyword *__attribute__* allows the programmer to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses.

Notes



The attribute of interest for achieving unaligned data accesses is “packed”. The “packed” attribute forces a one byte alignment on fields in a data structure. The compiler uses *lwl/lwr* for loading and *swl/swr* for storing unaligned data.

The following “C” code does *not* use the “packed” attribute. Study the assembler code generated after compiling:

```
/* Begin C code */
struct foo
{
char a ;
int x[2] ;
} foo;
```

Here is the “C” and generated assembler code when “packed” is used:

```
/* Begin C code */
struct foo
{
char a ;
int x[2] __attribute__ ((packed)) ;
} foo ;

main()
{
        foo.a = 'A' ;
        foo.x[0] = 18;
        foo.x[1] = 37;

}
/* End C code; begin partial listing of assembler code generated from above C
code*/
800201c8 <main+18> li $v1,65
800201cc <main+1c> sb $v1,0($v0)
800201d0 <main+20> li $v1,18
800201d4 <main+24> swl $v1,1($v0) ..... note the offset of 1 byte
800201d8 <main+28> swr $v1,4($v0)
800201dc <main+2c> li $v1,37
800201e0 <main+30> swl $v1,5($v0)
800201e4 <main+34> swr $v1,8($v0)
/* End assembler code */
```

The IDT/C compiler is efficient enough to recognize that if a field is larger than a certain number of bytes, it is better to not use the *lwl/lwr* and *swl/swr* pairs for the *entire* data transfer, and that it is smarter to use the pairs only to the point of reaching a word alignment beyond which regular *lw* or *sw* instructions can prove to be more efficient until the point where less than 4 bytes remain to be transferred using *lwl/lwr* or *swl/swr* again.

Notes

Note that the “packed” attribute works only on structures and not on simple variables such as int or char. To achieve packing of a simple variable, put it inside a structure with that variable as its only element.

Floating Point Data in Memory

This allows a programmer to load single-precision values by a load into an even-numbered floating point register; but the programmer can also load a double-precision value by a macro instruction, so that:

```
ldc1    $f2, 24(t1)
```

is expanded to two loads to consecutive registers:

```
lwc1    $f2, 24(t1)
lwc1    $f3, 28(t1)
```

The C compiler aligns 8-byte long double-precision floating point variables to 8-byte boundaries. RC30xx family hardware does not require this alignment; it is done to avoid compatibility problems with implementations of MIPS-2 or MIPS-3 CPUs such as the IDT RC4600 (64-bit RISController), where the *ldc1* instruction is a machine instruction and the alignment is necessary.

Basic Address Space of RC3xxx

The way in which MIPS processors use and handle addresses is subtly different from that of traditional CISC CPUs, and may appear confusing. Read the first part of this section carefully. Here are some guidelines:

- *The addresses put into programs are rarely the same as the physical addresses which come out of the chip (sometimes they’re close, but not the same). This manual will refer to them as program addresses and physical addresses respectively. A more common name for program addresses is “virtual addresses”; note that the use of the term “virtual address” does not necessarily imply that an operating system must perform virtual memory management (e.g. demand paging from disks...), but rather that the address undergoes some transformation before being presented to physical memory. Although virtual address is a proper term, this manual will typically use the term “program address” to avoid confusing virtual addresses with virtual memory management requirements. However, it should be remembered that the CPU always uses virtual (program) addresses, which are translated to physical addresses.*
- *A typical CPU has two operating modes: user and kernel. In user mode, any address above 2Gbytes (most-significant bit of the address set) is illegal and causes a trap. Also, some instructions cause a trap in user mode.*
- *The 32-bit program address space is divided into four big areas with traditional names; and different things happen according to the area an address lies in:*

  kuseg 0000 0000 – 7FFF FFFF (low 2Gbytes): these are the addresses permitted in user mode. In machines with an MMU, they will always be translated (more about the MMU in a later chapter). Software should not attempt to use these addresses unless the MMU is set up.

  For RC30xx CPUs without an MMU, the kuseg “program address” is transformed to a physical address by adding a 1GB offset; the address transformations for “base versions” of the RC30xx family are described later in this chapter. Note, however, that many embedded applications do not use this address segment (those applications which do not require that the kernel and its resources be protected from user tasks).

  kseg0 0x8000 0000 – 9FFF FFFF (512 Mbytes): these addresses are ‘‘translated’’ into physical addresses by merely stripping off the top bit, mapping them contiguously into the low 512 Mbytes of physical memory. This transformation operates the same for both “base” and “E” family members. This segment is referred to as “unmapped” because “E” version devices cannot redirect this translation to a different area of physical memory.

  Addresses in this region are always accessed through the cache, so may not be used until the caches are properly initialized. They will be used for most programs and data in systems using “base” family members; and will be used for the OS kernel for systems which do use the MMU (“E” version devices).

Notes

kseg1 0xA000 0000 – BFFF FFFF (512 Mbytes): these addresses are mapped into physical addresses by stripping off the leading three bits, giving a duplicate mapping of the low 512 Mbytes of physical memory. However, kseg1 program address accesses will not use the cache.

The *kseg1* region is the only chunk of the memory map which is guaranteed to behave properly from system reset; that's why the after-reset starting point (0xBFC0 0000, commonly called the "reset exception vector") lies within it. The *physical* address of the starting point is 0x1FC0 0000 – which means that the hardware should place the boot ROM at this physical address.

Software will therefore use this region for the initial program ROM, and most systems also use it for I/O registers. In general, IO devices should always be mapped to addresses that are accessible from Kseg1, and system ROM is always mapped to contain the reset exception vector. Note that code in the ROM can then be accessed uncacheably (during boot up) using kseg1 program addresses, and also can be accessed cacheably (for normal operation) using kseg0 program addresses.

kseg2 0xC000 0000 – FFFF FFFF (1 Gbyte): this area is only accessible in kernel mode. As for kuseg, in "E" devices program addresses are translated by the MMU into physical addresses; thus, these addresses must not be referenced prior to MMU initialization. For "base versions", physical addresses are generated to be the same as program addresses for kseg2.

Note that many systems will not need this region. In "E" versions, it frequently contains OS structures such as page tables; simpler OS'es probably will have little need for kseg2.

In case of the RC32364, kseg2 is actually from 0xC000 0000 to 0xFEFF FFFF (1008 Mbytes). The 16 Mbytes space beyond that is reserved for memory mapped on-chip registers and for ICE (in-circuit emulator).

Summary of RC3xxx System Addressing

MIPS program addresses are rarely simply the same as physical addresses, but simple embedded software will probably use addresses in kseg0 and kseg1, where the program address is related in an obvious and unchangeable way to physical addresses.

Physical memory locations from 0x2000 0000 (512Mbyte) upward may be difficult to access. In "E" versions of the RC30xx family, the only way to reach these addresses is through the MMU. In "base" family members, certain of these physical addresses can be reached using kseg2 or kuseg addresses: the address transformations for base RC30xx family members is described later in this chapter.

Kernel vs. User Mode

In kernel mode (the CPU resets into this state), all program addresses are accessible. In user mode:

- *Program addresses above 2Gbytes (top bit set) are illegal and will cause a trap.*

  Note that if the CPU has an MMU, this means all valid user mode addresses must be translated by the MMU; thus, User mode for "E" devices and RC32364 typically requires the use of a memory-mapped OS.

  For "base" CPUs, kuseg addresses are mapped to a distinct area of physical memory. Thus, kernel memory resources (including IO devices) can be made inaccessible to User mode software, without requiring a memory-mapping function from the OS. Alternately, the hardware can choose to "ignore" high-order address bits when performing address decoding, thus "condensing" kuseg, kseg2, kseg1, and kseg0 into the same physical memory.
- *Instructions beyond the standard user set become illegal. Specifically, the kernel can prevent User mode software from accessing the on-chip CP0 (system control coprocessor, which controls exception and machine state and performs the memory management functions of the CPU).*

Thus, the primary differences between User and Kernel modes are:

- *User mode tasks can be inhibited from accessing kernel memory resources, including OS data structures and IO devices. This also means that various user tasks can be protected from each other.*

Notes

- *User mode tasks can be inhibited from modifying the basic machine state, by prohibiting accesses to CP0.*

Note that the kernel/user mode bit does not change the interpretation of anything – just some things cease to be allowed in user mode. In kernel mode the CPU can access low addresses just as if it was in user mode, and they will be translated in the same way.

Memory Map for CPUs without MMU hardware

The treatment of kseg0 and kseg1 addresses is the same for all IDT RC30xx CPUs. If the system can be implemented using only physical addresses in the low 512Mbytes, and system software can be written to use only kseg0 and kseg1, then the choice of "base" vs. "E" versions of the RC30xx family is not relevant.

For versions without the MMU ("base versions"), addresses in kuseg and kseg2 will undergo a fixed address translation, and provide the system designer the option to provide additional memory.

The base members of the RC30xx family provide the following address translations for kuseg and kseg2 program addresses:

- *kuseg: this region (the low 2Gbytes of program addresses) is translated to a contiguous 2Gbyte physical region between 1-3Gbytes. In effect, a 1GB offset is added to each kuseg program address.*

  In hex:

Program Address		Physical Address
0x0000 0000 - 0x7FFF FFFF	→	0x4000 0000 - 0xBFFF FFFF

- *kseg2: these program addresses are genuinely untranslated. So program addresses from 0xC000 0000 – 0xFFFF FFFF emerge as identical physical addresses.*

This means that "base" versions can generate most physical addresses (without the use of an MMU), except for a gap between 512Mbyte and 1Gbyte (0x2000 0000 through 0x3FFF FFFF). As noted above, many systems may ignore high-order address bits when performing address decoding, thus condensing all physical memory into the lowest 512MB addresses.

Subsegments in the RC3041 and RC32364 – Memory Width Configuration

The RC3041 and the RC32364 CPUs can be configured to access different regions of memory as either 32-, 16- or 8-bits wide. Where the program requests a 32-bit operation to a narrow memory (either with an uncached access, or a cache miss, or a store), the CPU may break a transaction into multiple data phases, to match the datum size to the memory port width.

The width configuration is applied independently to subsegments of the normal kseg regions, as follows:

- *kseg0 and kseg1: as usual, these are both mapped onto the low 512Mbytes. This common region is split into 8 subsegments (64Mbytes each), each of which can be programmed as 8-, 16- or 32-bits wide. The width assignment affects both kseg0 and kseg1 accesses (that is, one can view these as subsegments of the corresponding "physical" addresses).*
- *kuseg: is divided into four 512Mbyte subsegments, each independently programmable for width. Thus, kuseg can be broken into multiple portions, which may have varying widths. An example of this may be a 32-bit main memory with some 16-bit PCMCIA font cards and an 8-bit NVRAM.*
- *kseg2: is divided into two 512Mbyte subsegments, independently programmable for width. Again, this means that kseg2 can support multiple memory subsystems, of varying port width.*

Note that once the various memory port widths have been configured (typically at boot time), software does not have to be aware of the actual width of any memory system. It can choose to treat all memory as 32-bit wide, and the CPU will automatically adjust when an access is made to a narrower memory region. This simplifies software development, and also facilitates porting to various system implementations (which may or may not choose the same memory port widths).

Notes

Kernel Mode Virtual Addressing in the 36100

When the 36100 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 remapped at context switch, user page tables, and for some dynamically allocated data areas.

Base versions of the RC30xx family (including the RC36100) 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 Processor Revision Identifier (PRId) register can be used to distinguish the RC36100 from other members of the RC30xx family.

RC36100 Address Translation

Processors that only implement the base versions of memory management perform direct segment mapping of virtual-to-physical addresses, as illustrated in Figure 2.1. 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.*
- *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.*
- *Virtual addresses in kseg0 and kseg1 are both translated identically to the same physical address region.*

Notes

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 special areas of the virtual address space shown in Figure 2.1 are translated to physical addresses identically with the remainder of their virtual address segment. In the RC30xx family, these address areas were indicated as “reserved” for compatibility with future devices.

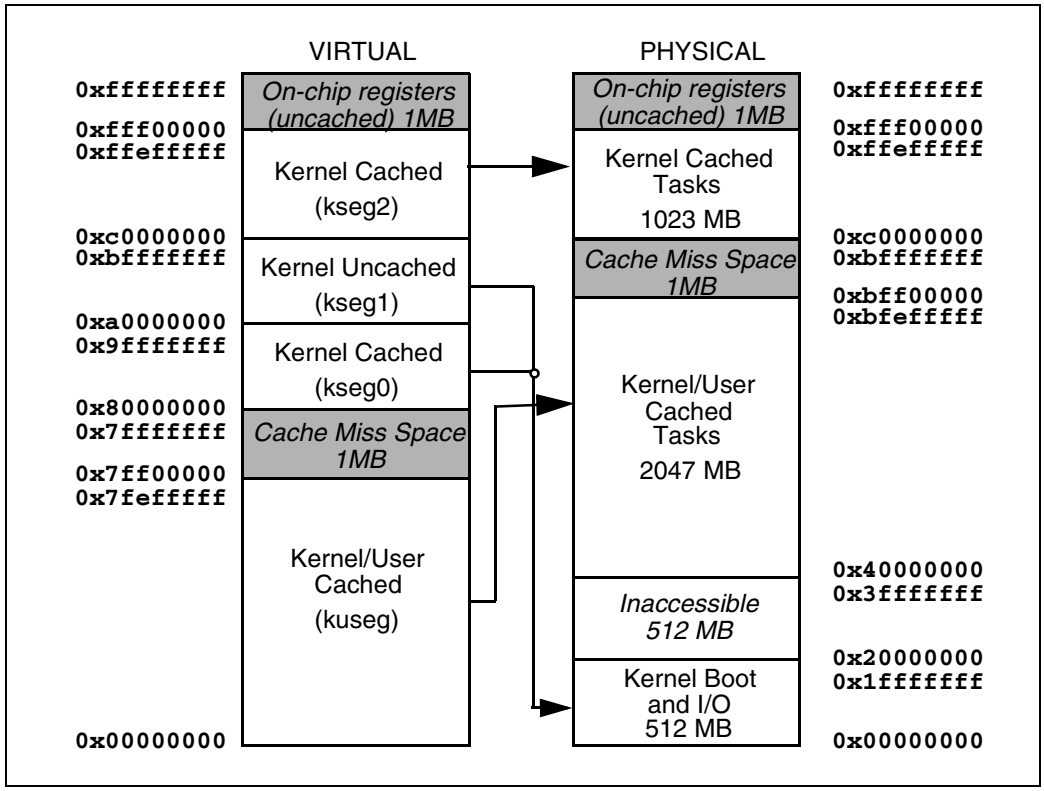


Figure 2.1 Virtual-to-Physical Address Translation in RC36100

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 2.4. 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.

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

Table 2.4 Virtual and Physical Address Relationships in Base Versions

Notes

Basic Address Space of RC4600/RC4700

Readers interested in the RC4x00 who may have skipped the preceding two sections because the sections pertain to RC30xx, are advised to review those sections before proceeding. Some of the general comments regarding the MIPS architecture in those sections are relevant even for the RC4xxx processors.

Unlike the RC30xx family, RC4xxx family does not have "base versions." All RC4600/RC4700 processors have memory management units (MMU). The RC4600/RC4700 uses an on-chip Translation Lookaside Buffer (TLB) to translate program addresses to physical addresses.

- *The RC4600/RC4700 has 3 modes of operation: User, Supervisor and Kernel.*
- *In the RC4600/RC4700, the program address space can be either 32-bits or 64-bits wide depending on the mode of operation and the setting of the corresponding extended address bit in the Status Register (UX, SX, KX); if the bit is 0, the addresses are 32-bits wide, and if the bit is set to 1, they are 64-bits wide.*
- *With a 36-bit Physical Address, a total of 64 Gigabytes of physical address space is available.*
- *Depending up on the mode of operation of the processor, different program address spaces become available as follows:*

User — In User mode, a single, contiguous program address space called u is available. Its size is 2 Gbytes (2^{31}) in 32-bit mode and it is called *useg*. In 64-bit mode the size is 1Tbyte (2^{40}) and the space label is *xuseg*.

Legal 32-bit addresses are 0x0000 0000 - 0x7FFF FFFF, and the 64-bit addresses are 0x0000 0000 0000 0000 - 0x0000 00FF FFFF FFFF. Presenting any addresses outside of these ranges while the processor is set up to be in User mode results in an Address Error exception. Cache accessibility is controlled by bit settings in the TLB entries.

Super — The Supervisor mode is designed for layered operating systems in which a true kernel runs in the Kernel mode described later, and the rest of the o/s runs in Supervisor mode.

In 32-bit Supervisor mode, two spaces named User Space and Supervisor Space can be addressed. Their labels are *suseg* and *sseg* respectively. The 2 Gbytes of *suseg* lie between 0x0000 0000 - 0x7FFF FFFF. The *sseg* is 512 Mbytes, from 0xC000 0000 to 0xDFFF FFFF.

In 64-bit Supervisor mode, three spaces named User Space (*xsuseg*), Current Supervisor Space (*xsseg*) and Separate Supervisor Space (*csseg*) are available. The 1 Tbyte *xsuseg* is from 0x0000 0000 0000 0000 to 0x0000 00FF FFFF FFFF. The *xsseg* goes from 0x4000 0000 0000 0000 till 0x4000 00FF FFFF FFFF, and is also 1 Tbytes long. Addressing of the *csseg* is compatible with addressing of the *sseg* in 32-bit mode; begins at 0xFFFF FFFF C000 0000 and ends at 0xFFFF FFFF DFFF FFFF, covering 512 Mbytes.

Kernel — The processor enters Kernel mode when:
ERL is set, or
EXL is set, or
KU Mode = Kernel

On exceptions, either ERL or EXL will be set. The processor remains in exception mode until an instruction to return from exception (eret) is executed, at which point the mode existing prior to detection of the exception is restored. Kernel-mode program address space is shown in Figure 2.2.

References to *kseg0 and kseg1* are not mapped through the TLB. The physical address is defined by the low-ordered 29 bits of the program address in kseg0 and kseg1. The cacheability and coherency for *kseg0* are determined by the settings in the Config register while kseg1 is never cacheable.

The 64-bit *xkuseg* offers a special feature for the ECC handler. If the ERL bit of the Status register is set, the segment becomes unmapped, uncached space allowing the ECC exception code to operate uncached using r0 as a base register.

The segment *xkphys* is a set of 8 physical spaces, each 2^{36} bytes long. References to these spaces do not go through the TLB; the physical address is taken from bits 35:0. bits 61:59 of the program address determine the cacheability and coherency as shown in Table 2.5

The regions *ckseg**x*** are compatible with their 32-bit counterparts *kseg**x***.

Notes

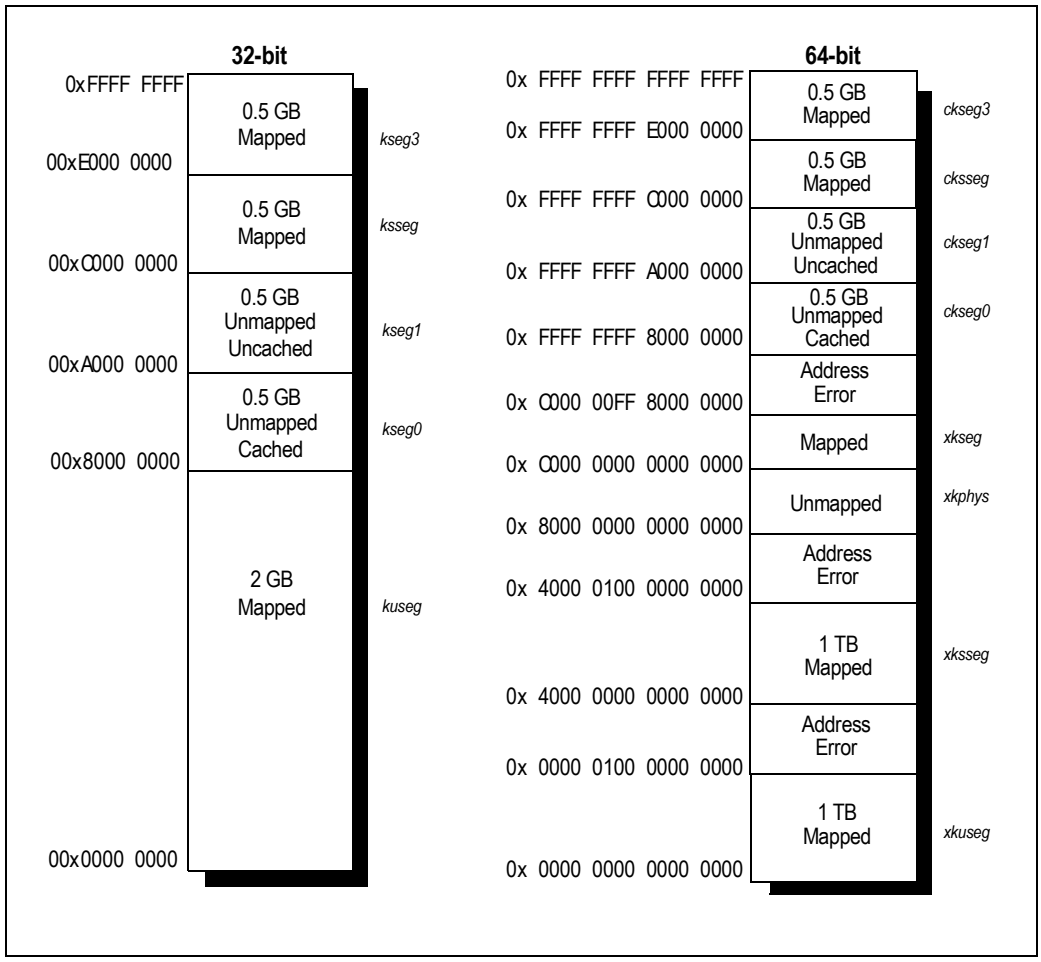


Figure 2.2 Kernel Mode Address Space

Value (61:59)	Cacheability and Coherency Attributes	Starting Address
0	Cacheable, noncoherent, write-through, no write allocate	0x8000 0000 0000 0000
1	Cacheable, noncoherent, write-through, write allocate	0x8800 0000 0000 0000
2	Uncached	0x9000 0000 0000 0000
3	Cacheable, noncoherent	0x9800 0000 0000 0000
4:7	Reserved	0xA000 0000 0000 0000

Table 2.5 Cacheability and Coherency

Basic Address Space of RC4650

Readers interested in the RC4650 who may have skipped sections regarding RC30xx addressing a few pages back, are advised to review those sections before proceeding. Some of the general comments regarding the MIPS architecture in those sections are relevant even for RC4650.

The RC4650 employs a simple mechanism to support the mapping of program addresses up on physical addresses. The TLB found in the RC4600/RC4700 is replaced by a “base-bounds” mechanism. When a program address is translated, its page number is first compared against the Bounds register. If the address is “in range,” the base register is added to the program address to form the physical address. There is a set

Notes

of base-bound registers for instruction addresses (IBase and IBounds registers) and another set for data (DBase and DBounds). In addition to these registers, a new Cache Algorithm (CAlg) register in CP0 allows a mix of cache attributes in a single system.

- *The processor program addresses are 32-bits wide; upper 32-bits of 64-bit registers are ignored. Physical address space is 4 Gbytes.*
- *The RC4650 has two operating modes, User mode and Kernel mode. The address spaces are defined as follows:*

useg The address space from 0x0000 0000 to 0x7FFF FFFF (2 Gbytes) is labelled as useg in the User mode. This is the only space available in User mode. The same address space is available from Kernel mode as well, where its label is kuseg.

kseg0 The 512 Mbyte address space 0x8000 0000 through 0x9FFF FFFF is defined as kseg0 aund is accessible in Kernel mode only. Addresses in kseg0 are not mapped using base-bounds mechanism; their physical addresses are calculated by setting the upper 3 bits of the program addresses to zero. The CAlg register controls cacheability of this segment. At reset kseg0 is cacheable.

kseg1 The 512 Mbyte address space 0xA000 0000 through 0xBFFF FFFF is defined as kseg1 and is accessible in Kernel mode only. Addresses in kseg1 are not mapped using base-bounds mechanism; their physical addresses are calculated by setting the upper 3 bits of the program addresses to zero. The CAlg register controls cacheability of this segment. At reset caches are disabled for kseg1 address space, but this can be changed later using CAlg register.

kseg2 The 1 Gbyte address space 0xC000 0000 through 0xFFFF FFFF is defined as kseg2 and is accessible in Kernel mode only. Addresses in kseg2 are not mapped using base-bounds mechanism; their physical addresses are calculated by setting the upper 3 bits of the program addresses to zero. The CAlg register controls cacheability of this segment.

Figure 2.3 shows the kernel mode address space.

Notes

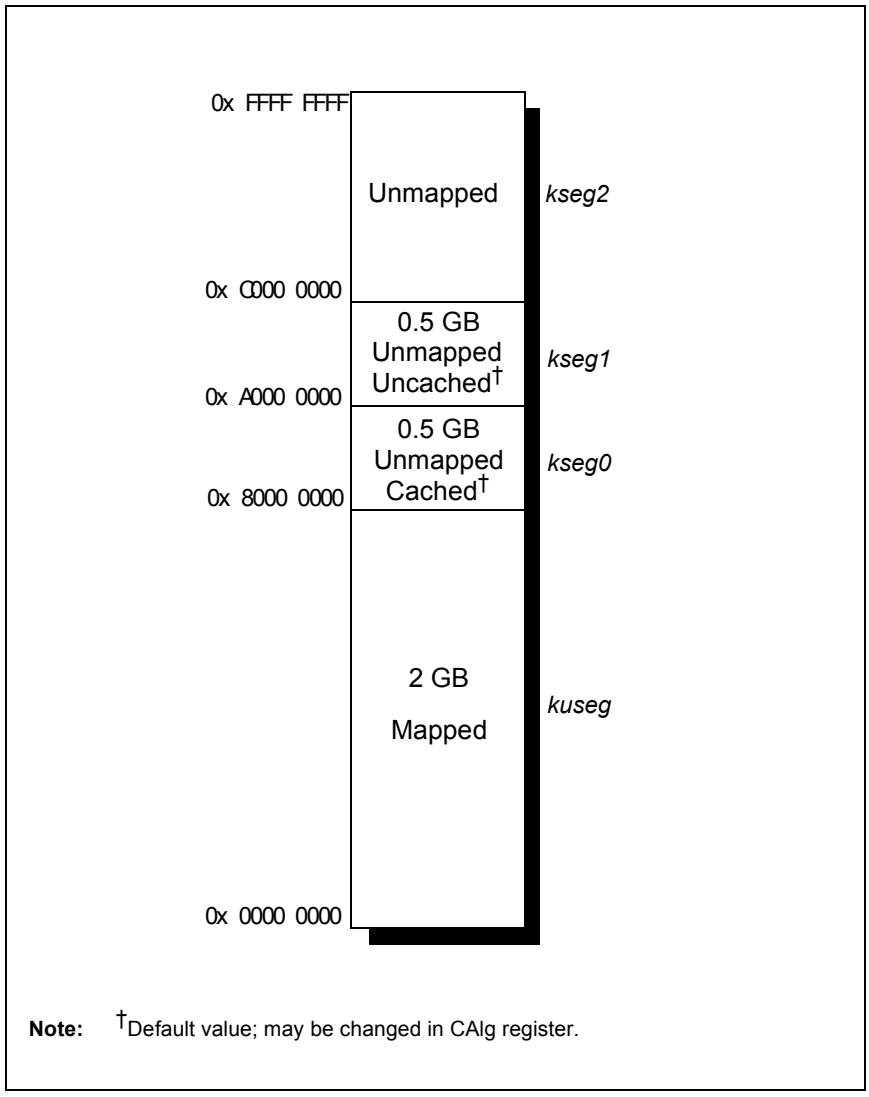


Figure 2.3 Kernel Mode Address Space

- *The address translation from program to physical address takes place using same algorithm for data as well as instructions although different base-bounds registers are used in each case. If addresses above 0x7FFF FFFF are generated in User mode, an address error exception is generated. For addresses in useg, bits 31:12 are compared to Bound register bits 30:12.*

  If the program address is bigger than the bounds address, a Bound Exception occurs. Otherwise, the physical address equals (program address bits 31:12 + Base register bits 31:12) concatenated with program address bits 11:0.
- *Program address bits 31:29 are used to select the appropriate CAlg fields to determine cacheability where applicable as described earlier.*

Notes

System Control Co-Processor Architecture

Notes

This chapter describes aspects of the MIPS architecture that must be managed by the operating system. Most of these features are transparent to the application programmer; however, most embedded systems programmers will have a view of the underlying CPU and system architecture and will find this material important.

Co-processors
Opcodes are reserved and instruction fields defined for up to four “co-processors”. Architecturally, the co-processors can be tightly coupled to the base integer CPU; for example, the ISA defines instructions to move data directly between memory and the coprocessor, rather than requiring it to be moved into the integer processor first. MIPS uses the term “co-processor” in both a traditional and non-traditional sense. The FPA device is a traditional microprocessor co-processor: it is an optional part of the architecture, with its own particular instruction set. MIPS also uses the term “co-processor” for the functions required to manage the CPU environment, including exception management, cache control, and memory management. This segmentation insures that the chip architecture can be varied (e.g. cache architecture, interrupt controller, etc.), without impacting user mode software compatibility. These functions are grouped by MIPS into the on-chip “co-processor 0”, or “system control co-processor” - and these instructions implement the whole CPU control system. Note that co-processor 0 has no independent existence, and is certainly not optional. It provides a standard way of encoding the instructions which access the CPU status register; so that, although the definition of the status register changes among implementations, programmers can use the same assembler for both CPUs. Similarly, the exception and memory management strategies can be varied among implementations, and these effects isolated to particular portions of the OS kernel.

CPU Control Summary

This chapter, coupled with chapters on cache management, memory management, and exception processing, provide details on managing the machine and OS state. The areas of interest include:

- *CPU control and co-processor: how privileged instructions are organized, with shortform descriptions. There are relatively few privileged instructions; most of the low-level control over the CPU is exercised by reading and writing bit-fields within special registers.*
- *Exceptions: external interrupts, invalid operations, arithmetic errors – all result in “exceptions”, where control is transferred to an exception handler routine.*

  MIPS exceptions are extremely simple – the hardware does the absolute minimum, allowing the programmer to tailor the exception mechanism to the needs of the particular system. A later chapter describes MIPS exceptions, why they are precise, exception vectors, and conventions about how to code exception handling routines.

  Special problems can arise with nested exceptions: exceptions occurring while the CPU is still handling an earlier exception. Hardware interrupts have their own style and rules. The Exception Management chapter includes an annotated example of a moderately-complicated exception handler.
- *Caches and cache management: all current RC30xx and RC4xxx implementations have dual caches (the I-cache for instructions, the D-cache for data). On-chip hardware is provided to manage the caches, and the programmer working with I/O devices, particularly with DMA devices, may need to explicitly manage the caches in particular situations.*

  To manipulate the caches, the RC30xx CPU allows software to isolate them, inhibiting cache/memory traffic and allowing the processor to access cache as if it were simple memory; and the RC30xx CPU can swap the roles of the I-cache and D-cache (the only way to make the I-cache writable). The RC4xxx provides direct access to both primary caches through its cache instruction. The RC32364 also implements the cache instruction.

Notes

Caches must sometimes be cleared of stale or invalid/uninitialized data. Even following power-up, caches are in a random state and must be cleaned up before they can be used. A later chapter will discuss the techniques used by software to manage the on-chip cache resources.

In addition, techniques to determine the on-chip cache sizes will be shown (greatest flexibility is achieved if software can be written to be independent of cache sizes). For the diagnostics programmer, techniques to test the cache memory and probe for particular entries will be discussed.

On some CPU implementations the system designer may make configuration choices about the cache (e.g. the RC3081 and RC3071 allow the cache organization to be selected between 16kB of I-cache/4kB of D-cache and 8kB each of I- and D- cache). The cache management chapter will also discuss some of the considerations to apply to make a proper selection.

- *Write buffer: in RC30xx family CPUs the D-cache is always write through; all writes go to main memory as well as the cache. This simplifies the caches, but main memory won't be able to accept data as fast as the CPU can write it. Much of the performance loss can be made up by using a FIFO to buffer write cycles (both address and data). In the RC30xx family, this FIFO, called the write buffer, is integrated on-chip. In the RC32364 and theRC4xxx, the D-cache can be set up to be either write-back or write-through. The FIFO store described above also exists in the RC32364 and the RC4xxx.*

  System programmers may need to know that writes happen later than the code sequence suggests. The chapter on cache management discusses this.

- *CPU Reset: at reset almost nothing is defined, so the software must configure the CPU carefully. In MIPS CPUs, reset is implemented in almost exactly the same way as the exceptions. A later chapter on reset initialization discusses ways of finding out which CPU is executing the software, and how to get a ROM program to run. An example of a C runtime environment, attending to the stack and special registers, is provided.*
- *Memory management and the TLB/Base-Bounds: A later chapter will discuss address translation and managing the translation hardware (base-bounds mechanism in RC4650 or the TLB in others). This section is mostly for OS programmers.*
- *Power management: The RC4xxx and the RC323xx processors can be put into a mode called "standby" mode with the use of the WAIT instruction. In this mode the internal core of the CPU operates at considerably reduced power. For more information about this topic, refer to the RISC Microprocessor Application Guide.*

CPU Control and "Co-processor 0"

CPU Control Instructions

Control functions are implemented with registers (most of which consist of multiple bitfields). There are several CPU control instructions used in the memory management implementation, which are described later in this manual. Aside from the MMU, CPU control in the RC30xx defines just one instruction beyond the necessary move to and from the control registers.

mtc0 rs, <nn>–Move to co-processor zero

Loads "co-processor 0" register number nn from CPU general register rs. It is unusual, and not good practice, to refer to CPU control registers by their number in assembler sources; normal practice is to use the names listed in Table 3.2. In some tool-chains the names are defined by a C-style "include" file, and the C pre-processor run as a front-end to the assembler; the assembler manual should provide guidance on how to do this. This is the only way of setting bits in a CPU control register.

mfc0 rd, <nn –Move from co-processor zero

General register rd is loaded with the values from CPU control register number nn. Once again, it is common to use a symbolic name and a macro-processor to save remembering the numbers. This is the only way of inspecting bits in a control register.

rfe *–Restore from exception (RC30xx)*

Notes

This instruction is available in RC30xx only. Note that this is not “return from exception”. This instruction restores the status register to go back to the state prior to the trap. To understand what it does, refer to the status register *SR* defined later in this chapter. The only secure way of returning to user mode from an exception is to return with a *jr* instruction which has the *rfe* in its delay slot.

eret ***–Exception return (RC4xxx)***

This is a RC4xxx instruction which actually returns from an exception, interrupt or error trap. Unlike a branch or jump instruction, eret does not execute the next instruction. This instruction is also available in the RC32364.

The RC4xxx has some additional instructions for CPU control. Doubleword counterparts of the *mtc0/ mfc0* instructions are also available as ***dmtc0/dmfc0*** which allow 64-bit transfers. The ***wait*** instruction puts the CPU in low-power standby mode. For more information about standby mode, refer to the *IDT79RC4600 & IDT79RC4700 64-bit RISController Processor Hardware User’s Manual.*

Standard CPU Control Registers

Register Mnemonic	Description	CP0 Register Number
PRId	CP0 type and rev level.	15
SR	(status register) CPU mode flags.	12
Cause	Describes the most recently recognized exception.	13
EPC	Exception return address.	14
BadVaddr	Contains the last invalid program address which caused a trap. It is set by address errors of all kinds, even if there is no MMU.	8
Config	CPU configuration (RC3071, RC3081, RC3041, RC4xxx, RC32364 only).	3/16
BusCtrl	(RC3041 only) configure bus interface signals. Needs to be setup to match the hardware implementation.	2
PortSize	(RC3041 only) used to flag some program address regions as 8- or 16-bits wide. Must be programmed to match the hardware implementation.	10
Count	(RC3041/RC4xxx/RC32364, read/write) a 24-bit counter incrementing with the CPU clock. (32-bit in RC4x00 and RC32364).	9
Compare	(RC3041/RC4xxx/RC32364, read/write) a 24-bit value used to wraparound the *Count* value and set an output signal. (32-bit in RC4xxx and RC32364).	11
Context	(RC4600/RC4700 only) pointer to kernel virtual page table entry (PTE) for 32-bit address spaces.	4
XContext	(RC4600/RC4700 only) pointer to kernel virtual page table entry (PTE) for 64-bit address spaces.	20
ECC	(RC4600/RC4700/RC4650/RC32364 only) secondary-cache error checking and correcting (ECC) and Primary parity.	26
CacheErr	(RC4600/RC4700/RC4650/RC32364 only) Cache Error and Status register.	27
ErrorEPC	(RC4600/RC4700/RC4650 only) Error Exception Program Counter.	30
IWatch	(RC4650/RC32364 only, read/write) specifies a instruction program address that causes a Watch exception.	18
DWatch	(RC4650/RC32364 only, read/write) specifies a data program address that causes a Watch exception.	19

Table 3.1 Standard CPU Control Registers (Part 1 of 2)

Notes

Register Mnemonic	Description	CP0 Register Number
IEPC	(RC32364 only)Imprecise Exception Program Counter	22
DEPC	(RC32364 only) Debug Exception Program Counter	23
Debug	(RC32364 only) Debug Control / Status Register	24
TagLo	(RC32364 only) Cache Tag Register	28

Table 3.1 Standard CPU Control Registers (Part 2 of 2)

Control Register Formats

A note about reserved fields: many unused control register fields are marked "0." Bits in such fields are guaranteed to read zero and should be written as zero. Other reserved fields are marked reserved or ¥; software must write them as zero and should not assume that it will get back zero, or any other particular value.

Figure 3.1 shows the layout and fields of the *PRId* register, a read-only register. The Imp field should be related to the CPU control register set.

PRId Register

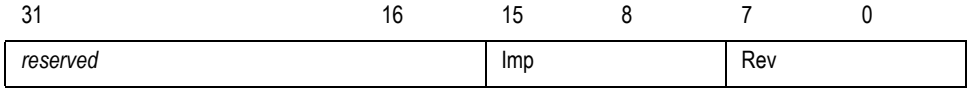


Figure 3.1 PRId Register Format

The encoding of Imp is described in Table 3.2:

CPU type	"Imp" value	"Rev" value
RC3000A (including RC3051, RC3052, RC3071, and RC3081)	3	undefined
IDT unique (RC3041)	7	0
RC36100	7	10
RC32364	0x26	0
RC4600	0x20	undefined
RC4700	0x21	undefined
RC4650	0x22	undefined

Table 3.2 "Imp" and "Rev" bit values

Note that when the Imp field indicates IDT unique, the revision number can be used to distinguish among various CP0 implementations. Refer to the RC3041 User's manual for the revision level appropriate for that device. Since the RC3051, 52, 71, and 81 are kernel compatible with the RC3000A, they share the same Imp value.

When printing the value of this register, it is conventional to print them out as "x.y" where "x" and "y" are the decimal values of Imp and Rev, respectively. Do not use this register and the CPU manuals to size things or establish the presence or absence of any particular features. The software will be more portable and robust if it is designed to include code sequences that probe for the existence of individual features. This manual will provide examples to determine cache sizes, presence or absence of TLB, FPA, etc.

Notes

Status Register (RC3xxx)

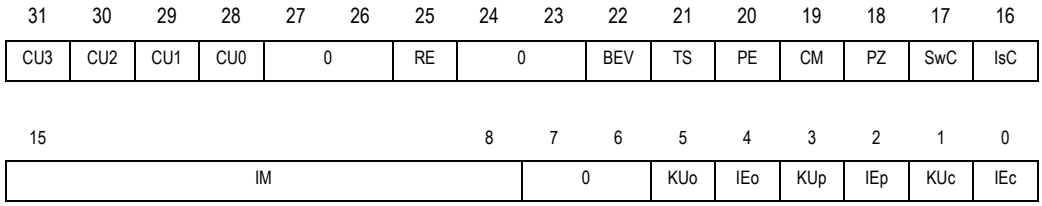


Figure 3.2 Status Register Format (RC3xxx)

Note that there are no modes such as non-translated or non-cached in MIPS CPUs; all translation and caching decisions are made on the basis of the program address. Fields are:

CU3, CU2 Bits (31:30) control the usability of "co-processors" 3 and 2, respectively. In the RC30xx family, these might be enabled if software wishes to use the BrCond(3:2) input pins for polling, or to speed exception decoding.

CU1 Co-processor 1 usable: set 1 to use FPA if present, 0 to disable. When 0, all FPA instructions cause an interrupt exception, even for the kernel. It can be useful to turn off an FPA even when one is available; it may also be enabled in devices which do not include an FPA, if the intent is to use the BrCond(1) pin as a polled input.

CU0 Co-processor 0 usable: set 1 to use some nominally-privileged instructions in user mode (this is rarely, if ever, done). Co-processor 0 instructions are always usable in kernel mode, regardless of the setting of this bit.

RE Reverse endianness in user mode. The MIPS processors can be configured, at reset time, with either "endianness" (byte ordering convention, discussed in the various CPU's User's Manuals and later in this manual). The RE bit allows applications with one byte ordering convention to be run on systems with the opposite convention, presuming OS software provided the necessary support.

When RE is active, user mode software runs as if the CPU had been configured with the opposite endianness.

BEV Boot exception vectors: when BEV == 1, the CPU uses the ROM (kseg1) space exception entry point (described in a later chapter). BEV is usually set to zero in running systems; this relocates the exception vectors. to RAM addresses, speeding accesses and allowing the use of "user supplied" exception service routines.

TS TLB shutdown: In devices that implement the full RC3000A MMU, TS is set if a program address simultaneously matches two TLB entries. Prolonged operation in this state, in some implementations, could cause internal contention and damage to the chip. TLB shutdown is terminal, and can be cleared only by a hardware reset.

In RC30xx base family members, which do not include the TLB, this bit is set by reset; software can rely on this to determine the presence or absence of TLB.

PE Parity Error: set if a cache parity error has occurred. No exception is generated by this condition, which is really only useful for diagnostics. The MIPS architecture has cache diagnostic facilities because earlier versions of the CPU used external caches, and this provided a way to verify the timing of a particular system. For those implementations the cache parity error bit was an essential design debug tool.

For CPUs with on-chip caches this feature is rarely needed; only the RC3071 and RC3081 implement parity over the on-chip caches.

CM Cache Miss: set if a data cache miss occurred while the cache was isolated.

PZ Parity Zero: when set, cache parity bits are written as zero and not checked. This was useful in old RC3000A systems which required external cache RAMs, but is of little relevance to the RC30xx family.

SwC/IsC Swap caches and Isolate (data) cache. Cache mode bits for cache management and diagnostics. For more details, see the chapter on cache management. These bits are undefined on reset. The system software should set these to known values before proceeding.

Notes

- *IsC set 1: makes all loads and stores access only the data cache. In this mode, a partial-word store invalidates the cache entry. Note that when this bit is set, even uncached data accesses will not be seen on the bus; further, this bit is not initialized by reset. Boot-up software must insure this bit is properly initialized before relying on external data references.*
- *SwC set: reverses the roles of the I-cache and D-cache, so that software can access and invalidate I-cache entries.*

IM — Interrupt mask: an 8 bit field defining which interrupt sources, when active, will be allowed to cause an exception. Six of the interrupt sources are external pins (one may be used by the FPA, which although it lives on the same chip is logically external); the other two are the software-writable interrupt bits in the *Cause* register.

No interrupt prioritizing is provided by the CPU. This is described in greater detail in the chapter dealing with exceptions.

KUc/IEc — The two basic CPU protection bits.

KUc is set 1 when running with kernel privileges, 0 for user mode. In kernel mode, software can get at the whole program address space, and use privileged ("co-processor 0") instructions. User mode restricts software to program addresses between 0x0000 0000 and 0x7FFF FFFF, and can be denied permission to run privileged instructions; attempts to break the rules result in an exception.

IEc is set 0 to prevent the CPU taking any interrupt, 1 to enable.

KUp/IEp — KU previous, IE previous: on an exception, the hardware takes the values of KUc and IEc and saves them here; at the same time as changing the values of KUc, IEc to [1, 0] (kernel mode, interrupts disabled). The instruction *rfe* can be used to copy KUp, IEp back into KUc, IEc.

KUo/IEo — KU old, IE old:

on an exception the KUp, IEp bits are saved here. Effectively, the six KU/IE bits are operated as a 3-deep, 2-bit wide stack which is pushed on an exception and popped by an *rfe*.

This provides an opportunity to cleanly recover from an exception occurring so early in an exception handling routine that the first exception has not yet saved *SR*. It is particularly useful to allow the user TLB refill code to be made shorter, as described in the memory management chapter.

Status Register (RC32364)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Figure 3.3 shows the format of the entire register and Table 3.3 explains the fields. The following list provides details of the more important *Status* register fields:

- *The 8-bit Interrupt Mask (IM) field controls the individual enabling of eight interrupt conditions. Interrupts must be generally enabled before they can cause the exception (IE set), and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending (IP) field of the Cause register (for more information, refer to the Interrupt Pending (IP) field of the Cause register).IM[1:0] are the masks for the two software interrupts and IM[7:2] correspond to Int[5:0].*
- *The 4-bit Coprocessor Usability (CU) field controls the usability of 4 possible coprocessors. Regardless of the CU0 bit setting, CP0 is always usable in Kernel mode. For all other cases, an instruction for or access to an unusable coprocessor causes an exception.*
- *The 9-bit Diagnostic Status (DS) field (Status[24:16]) is used for self-testing and checks the cache and virtual memory system.*
- *The Reverse-Endian (RE) bit, bit 25, reverses the endianness of the machine. At system reset, the processor can be configured as either little-endian or big-endian. This selection is always used in Kernel and Supervisor modes, and also in User mode when the RE bit is 0. Setting the RE bit to 1 inverts the User mode endianness.*

Notes

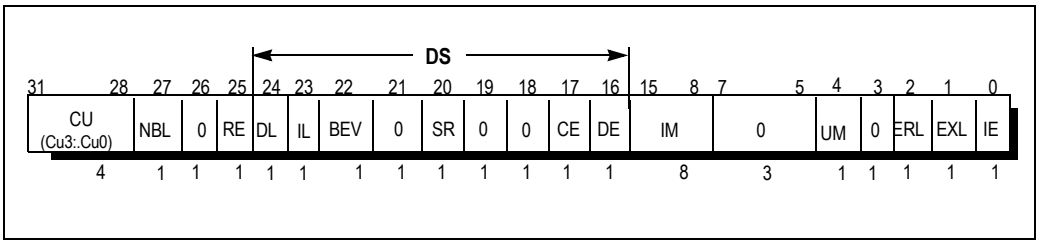


Figure 3.3 Status Register (RC32364)

Field	Description
CU	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. 1 → usable0 → unusable
NBL	Enables or Disables Non-Blocking Load 1 → Enable0 → Disable **Note**: This bit will be cleared whenever RC32364 takes an imprecise exception caused by a non-blocking load instruction. It is the responsibility of the exception handler to turn this bit on again.
RE	*Reverse-Endian* bit, valid in User mode.
DL	Data Cache Lock enable. This bit enables the data cache lock function. If this bit is set during Data cache fill, the cache line at that particular set will be locked. Please refer to the "Cache Operation" section for more detail 0 → disable Data cache locking 1 → enable Data cache locking
IL	Instruction Cache Lock enable. This bit enables the instruction cache lock function. If this bit is set during Instruction cache fill, the cache line at that particular set will be locked. Please refer to the "Cache Operation" section for more detail 0 → disable Instruction cache locking 1 → enable Instruction cache locking
BEV	Controls the location of TLB refill and general exception vectors. 0 → normal1→ bootstrap
SR	1→ Indicates a soft reset or NMI has occurred.
CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
DE	Specifies that cache parity errors cannot cause exceptions. 0 → parity remains enabled 1 → disables parity
0	Reserved. Must be written as zeroes and return zeroes when read.
IM	Interrupt Mask: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the Interrupt Mask field of the Status register and the Interrupt Pending field of the Cause register. IM[7:2] correspond to interrupts Int[5:0] and IM[1:0] to the software interrupts. 0 → disabled1 → enabled
UM	User Mode Bits 1 → User 0 → Kernel

Table 3.3 Status Register Fields (RC32364) (Part 1 of 2)

Notes

Field	Description
ERL	Error Level 0 → normal1 → error
EXL	Exception Level 0 → normal1 → exception **Note:** When going from 0 to 1, IE should be disabled (0) first. This would be done when preparing to return from the exception handler, such as before executing the ERET instruction.
IE	Interrupt Enable 0 → disable interrupts1 → enables interrupts

Table 3.3 Status Register Fields (RC32364) (Part 2 of 2)

Status Register (RC4600/RC4700)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields; Figure 3.4 shows the status register format and field names.

Status Register Format (RC4600/RC4700)

Figure 3.4 shows the format of the *Status* register. Table 3.4, which follows the figure, describes the *Status* register fields.

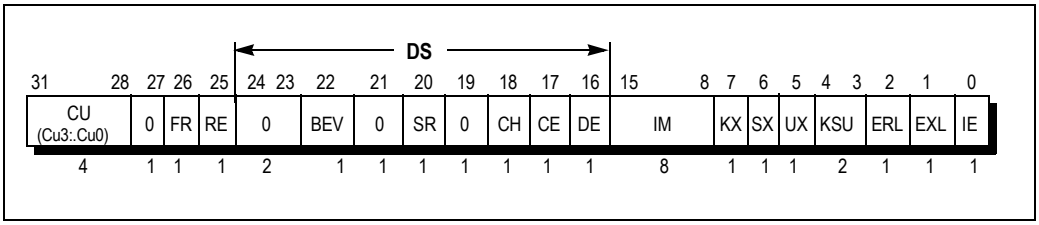


Figure 3.4 Status Register (4600/4700)

Field	Description
CU	Controls the usability of each of the four coprocessor unit numbers. CP0 is always usable when in Kernel mode, regardless of the setting of the CU_0 bit. 1 → usable0 → unusable
FR	Enables additional floating-point registers 0 → 16 registers1 → 32 registers
RE	*Reverse-Endian* bit, valid in User mode.
BEV	Controls the location of TLB refill and general exception vectors. 0 → normal1→ bootstrap
SR	1→ Indicates a soft reset or NMI has occurred.
CH	Hit (tag match and valid state) or miss indication for last CACHE Hit Invalidate, Hit Write Back Invalidate, Hit Write Back, or Hit Set Virtual for a primary cache. 0 → miss1 → hit
CE	Contents of the ECC register set or modify the check bits of the caches when CE = 1; see description of the *ECC* register.
DE	Specifies that cache parity errors cannot cause exceptions. 0 → parity remains 1 → disables parity enabled

Table 3.4 Status Register Fields (4600/4700) (Part 1 of 2)

Notes

Field	Description
0	Reserved. Must be written as zeroes and return zeroes when read.
IM	*Interrupt Mask*: controls the enabling of each of the external, internal, and software interrupts. An interrupt is taken if interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. IM[7:2] correspond to interrupts Int[5:0] and IM[1:0] to the software interrupts. 0 → disabled1→ enabled
KX	KX controls whether the TLB Refill Vector or the XTLB Refill Vector address is used for TLB misses on kernel addresses 0 → TLB Refill Vector1 → XTLB Refill Vector
SX	Enables 64-bit virtual addressing and operations in Supervisor mode. The extended-addressing TLB refill exception is used for TLB misses on supervisor addresses. 0 → 32–bit1 → 64–bit
UX	Enables 64-bit virtual addressing and operations in User mode. The extended-addressing TLB refill exception is used for TLB misses on user addresses. 0 → 32–bit1 → 64–bit
KSU	Mode bits 10_2 → User01_2 → Supervisor00_2 → Kernel
ERL	Error Level 0 → normal1 → error
EXL	Exception Level 0 → normal1 → exception **Note:** When going from 0 to 1, IE should be disabled (0) first. This would be done when preparing to return from the exception handler, such as before executing the ERET instruction.
IE	Interrupt Enable 0 → disable interrupts1 → enables interrupts

Table 3.4 Status Register Fields (4600/4700) (Part 2 of 2)

Status Register Modes and Access States

Fields of the *Status* register set the modes and access states described in the sections that follow.

Interrupt Enable: Interrupts are enabled when all of the following conditions are true:

- *IE = 1*
- *EXL = 0*
- *ERL = 0*

If these conditions are met, the settings of the *IM* bits identify the interrupt.

> **Note:** Setting the IE bit may be delayed by up to 3 cycles. If performing nested interrupts, re-enable the IE bit first.

Operating Modes: The following CPU *Status* register bit settings are required for User, Kernel, and Supervisor modes (see Chapter 4 for more information about operating modes).

- *The processor is in User mode when KSU = 10_2, EXL = 0, and ERL = 0.*
- *The processor is in Supervisor mode when KSU = 01_2, EXL = 0, and ERL = 0.*
- *The processor is in Kernel mode when KSU = 00_2, or EXL = 1, or ERL = 1.*

32- and 64-bit Virtual Addressing: The following CPU *Status* register bit settings select 32- or 64-bit virtual addressing for User and Supervisor operating modes. Enabling 64-bit virtual addressing permits the execution of 64-bit opcodes and translation of 64-bit virtual addresses. 64-bit virtual addressing for User and Supervisor modes can be set independently but is always used for Kernel mode.

- *The KX field controls whether the TLB Refill Vector or the XTLB Refill Vector address is used for TLB misses on Kernel addresses. 64-bit opcodes are always valid in Kernel mode.*

Notes

- *64-bit addressing and operations are enabled for Supervisor mode when SX = 1.*
- *64-bit addressing and operations are enabled for User mode when UX = 1.*

Status Register Reset

The contents of the *Status* register are undefined at reset, except for the following bits — *ERL* and *BEV* = 1.

The *SR* bit distinguishes between Reset and Soft Reset (Nonmaskable Interrupt [NMI]).

Status Register (RC4650)

The *Status* register (SR) in the RC4650 is similar to that in the RC4600 for the most part. Please refer to the previous section for details. Figure 3.5 shows the format of the entire register in the RC4650. Following the figure is a description of the fields that are unique to the RC4650.

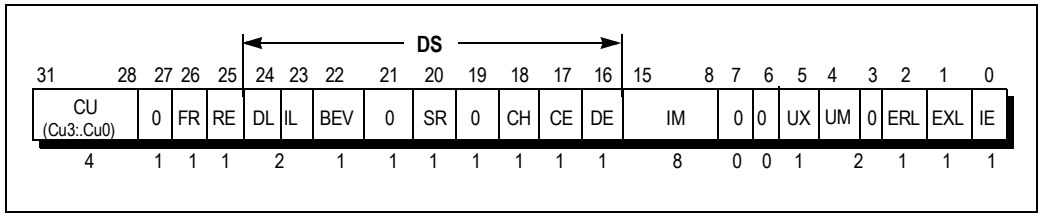


Figure 3.5 Status Register (4650)

Bits 24, 23, 7, 6, 4, and 3 are different in the RC4650, as compared to the RC4600. In the RC4650, because it does not have a TLB, does not support 64-bit program addressing, and has only two operating modes, bits 7, 6 and 3 are reserved. As noted in Table 3.5, bits 24 (DL) and 23 (IL) are used for cache locking.

DL	Data cache lock, a new bit in RC4650. Does not prevent refills into set A when set A is invalid. Does not inhibit update of the D-cache on store operations. 0 → normal operation 1→ refill into set A disabled
IL	Instruction cache lock, a new bit in RC4650. Does not prevent refills into set A when set A is invalid. 0 → normal operation 1→ refill into set A disabled
UM	User Mode bit, a new bit in RC4650. 1 → User 0 → Kernel (Simplification of KSU, remains subject to EXL and ERL, as on RC4xxx.

Table 3.5 DL and IL Bits in 4650 Status Register

Table 3.6 shows the fields in the *Cause* register, which are consulted to determine the kind of exception that happened and will be used to decide which exception routine to call.

Cause Register (RC3xxx and RC4600/RC4700)

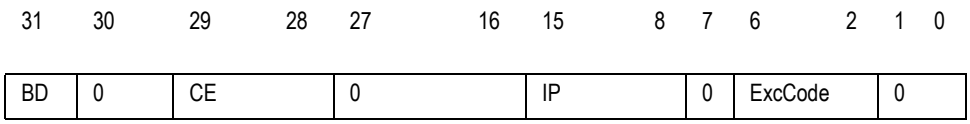


Table 3.6 Cause Register Fields (RC3xxx and RC4600/RC4700)

BD Branch Delay: if set, this bit indicates that the EPC does not point to the actual "exception" instruction, but rather to the branch instruction which immediately precedes it.

When the exception restart point is an instruction which is in the "delay slot" following a branch, *EPC* has to point to the branch instruction; it is harmless to re-execute the branch, but if the CPU returned from the exception to the branch delay instruction itself the branch would not be taken and the exception would have broken the interrupted program.

Notes

The only time software might be sensitive to this bit is if it must analyze the "offending" instruction (if BD == 1 then the instruction is at *EPC* + 4). This would occur if the instruction needs to be emulated (e.g. a floating point instruction in a device with no hardware FPA; or a breakpoint placed in a branch delay slot).

CE — Co-processor error: if the exception is taken because a "co-processor" format instruction was for a "co-processor" which is not enabled by the CUx bit in *SR,* then this field has the co-processor number from that instruction.

IP — Interrupt Pending: shows the interrupts which are currently asserted (but may be "masked" from actually signalling an exception). These bits follow the CPU inputs for the six hardware levels. Bits 9 and 8 are read/writable, and contain the value last written to them. However, any of the 8 bits active when enabled by the appropriate IM bit and the global interrupt enable flag IEc in *SR,* will cause an interrupt.

IP is subtly different from the rest of the *Cause* register fields; it doesn't indicate what happened when the exception took place, but rather shows what is happening now.

ExcCode — A 5-bit code which indicates what kind of exception happened, as detailed in Table 3.7.

ExcCode Value	Mnemonic	Description
0	Int	Interrupt
1	Mod	TLB modification
2	TLBL	TLB load/TLB store
3	TLBS	
4	AdEL	Address error (on load/I-fetch or store respectively). Either an attempt to access outside kuseg when in user mode, or an attempt to read a word or half-word at a misaligned address.
5	AdES	
6	IBE	Bus error (instruction fetch or data load, respectively). External hardware has signalled an error of some kind; proper exception handling is system-dependent. The RC30xx family CPUs can't take a bus error on a store; the write buffer would make such an exception "imprecise".
7	DBE	
8	Syscall	Generated unconditionally by a *syscall* instruction.
9	Bp	Breakpoint - a *break* instruction.
10	RI	reserved instruction
11	CpU	Co-Processor unusable
12	Ov	arithmetic overflow. Note that unsigned versions of instructions (e.g. *addu*) never cause this exception.
13	Tr	Trap Exception in RC4600/RC4700/RC32364; reserved in RC3xxx
14	-	Reserved
15	FPE	Floating-Point exception; reserved in parts with no FPA
16-31	-	Reserved.

Table 3.7 ExcCode Values: R3xxx/R4600/R4700 Exception differences

Cause Register (RC4650)

The *Cause* register fields (shown in Figure 3.6) are similar to those in the RC4600, as described in the previous section. Notable differences between the RC4650 and the RC4600 cause registers are described in Figure 3.6.

Notes

Cause Register (RC32364)

The *Cause* register fields (shown in Figure 3.6) are identical to those in the RC4650, with just one exception. Bit 26 is zero in case of a RC4650. but in the case of RC32364 it is called IPE as described in the last entry of Table 3.8.

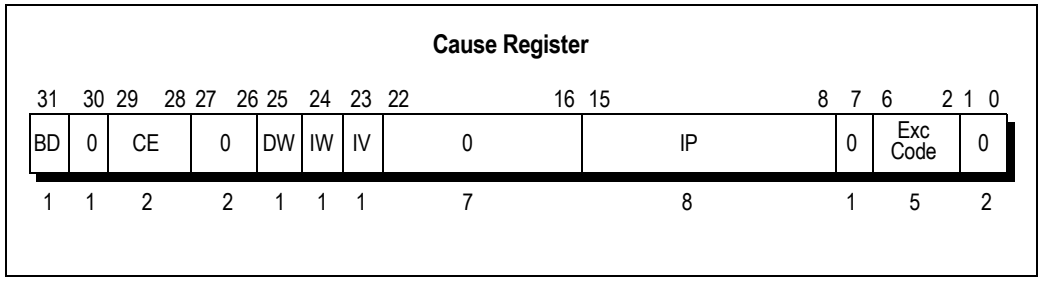


Figure 3.6 Cause Register Format (RC4650)

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal
0	Reserved. Currently read as 0 and must be written as '0'.
CE	Coprocessor unit number referenced when a Coprocessor Unusable exception is taken.
DW	On a Watch exception, indicates that the DWatch register matched. On other exceptions this field is undefined.
IW	On a Watch exception, indicates that the IWatch register matched. On other exceptions this field is undefined.
IV	Enables the new dedicated interrupt vector. 1 → interrupts use new exception vector (200) 0 → interrupts use common exception vector (180)
IP	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt
Exc-Code	Exception code field (see Table 3.7)
IPE	RC32364 only Indicates if last exception is imprecise. This occurs when an exception is taken on a NonBlocking Load 1 → Imprecise 0 → Precise

Table 3.8 Cause Register Field Descriptions

EPC Register

This is a 32-bit read/write register containing the 32-bit address of the return point for this exception (64-bits for RC4600/RC4700). The instruction causing the exception is at EPC, unless BD is set in *Cause,* in which case EPC points to the previous (branch) instruction. The RC4600/RC4700/RC32364 will not write to EPC if *EXL* bit in *SR* is set. Also, the RC4600, RC4700, RC32364 and RC4650 use ErrorPC on cache errors and NMI soft reset.

Notes

BadVaddr Register (RC3xxx)

A 32-bit register containing the address whose reference led to an exception; set on any MMU-related exception, on an attempt by a user program to access addresses outside kuseg, or if an address is wrongly aligned for the datum size referenced.

After any other exception this register is undefined. Note in particular that it is not set after a bus error.

BadVaddr Register (RC4xxx/RC4650/RC32364)

The 64-bit (32 bits on RC3xxx and RC4650) Bad Virtual Address register (*BadVAddr*) is a read-only register that displays the most recent virtual address that caused one of the following exceptions: Address Error (e.g., unaligned access), TLB Invalid, TLB Modified, TLB Refill, Virtual Coherency Data Access, or Virtual Coherency Instruction Fetch. In the RC4650, bounds exception is recognized in place of TLB exceptions because a TLB does not exist.

The processor does not write to the *BadVAddr* register when the *EXL* bit in the *Status* register is set to a 1. The *BadVAddr* register does not save any information for bus errors, since bus errors are not addressing errors.

Processor-Specific Registers

Count and Compare Registers (RC3041 only)

Only present in the RC3041, these provide a simple 24-bit counter/timer running at CPU cycle rate. *Count* counts up, and then wraps around to zero once it has reached the value in the *Compare* register. As it wraps around the **Tc*** CPU output is asserted. According to CPU configuration (bit TC of the *BusCtrl* register), **Tc*** will either remain active until reset by software (re-write *Compare*), or will pulse. In either case the counter just keeps counting. To generate an interrupt **Tc*** must be connected to one of the interrupt inputs.

From reset *Compare* is setup to its maximum value (0xFF FFFF), so the counter runs up to 2^{24}-1 before wrapping around.

Count and Compare Registers (RC4xxx & RC32364 only)

The 32-bit *Count* register acts as a timer, incrementing at a constant rate—half the maximum instruction issue rate—whether or not an instruction is executed, retired, or any forward progress is made through the pipeline.

This register can be read or written. It can be written for diagnostic purposes or system initialization; for example, to synchronize processors.

The 32-bit *Compare* register acts as a timer; it maintains a stable value that does not change on its own. When the value of the *Count* register equals the value of the *Compare* register, interrupt bit *IP(7)* in the *Cause* register is set. This causes an interrupt as soon as the interrupt is enabled.

Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use however, the *Compare* register is write-only.

Config Register (RC3071 and RC3081)

31	30	29	28 26	25	24	23	22 0
Lock	Slow Bus	DB Refill	FPInt	Halt	RF	AC	reserved

- *Lock: set this bit to write to the register for the last time; all future writes to Config will be ignored.*
- *Slow Bus: hardware may require that this bit be set. It only matters when the CPU performs a store while running from a cached location. The system hardware design determines the proper setting for this bit; setting it to '1' should be permissible for any system, but loses some performance in*

Notes

memory systems able to support more aggressive bus performance.

If set 1, an idle bus cycle is guaranteed between any read and write transfer. This enables additional time for bus tri-stating, control logic generation, etc.

- *DB: "data cache block refill", set 1 to reload 4 words into the data cache on any miss, set 0 to reload just one word. Can be initialized either way on the RC3081, by a reset-time hardware input.*
- *FPInt: controls the CPU interrupt level on which FPA interrupts are reported. On original RC3000 CPUs the FPA was external and this was determined by wiring; but the RC3081's FPA is on the chip and it would be inefficient (and jeopardize pin-compatibility) to send the interrupt off chip and on again.*

  *Set FPInt to the binary value of the CPU interrupt pin number which is dedicated to FPA interrupts. By default the field is initialized to "011" to select the pin **Int3**[1]; MIPS convention put the FPA on external interrupt pin 3. For whichever pin is dedicated to the FPA, the CPU will then ignore the value on the external pin; the IP field of the cause register will simply follow the FPA.*

  On the RC3071, this field is "reserved" and must be written as "000".
- *Halt: set to bring the CPU to a standstill. It will start again as soon as any interrupt input is asserted (regardless of the state of the interrupt mask). This is useful for power reduction, and can also be used to emulate old MC68000 "Halt" operation.*
- *RF: slows the CPU to 1/16th of the normal clock rate, to reduce power consumption. Illegal unless the CPU is running at 33Mhz or higher. Note that the CPUs output clock (which is normally used to synchronize all the interface logic) slows down too; the hardware design should also accommodate this feature if software desires to use it.*
- *Alternate cache (AC): 0 for 16K I-cache/4K D-cache, but set 1 for 8K I-cache/8K D-cache.*
- *Reserved: must only be written as zero. It will probably read as zero, but software should not rely on this.*

Config Register (RC3041)

31	30	29	28 20	19	18 0
Lock	**1**	DBR	0	FDM	0

- *Lock: set 1 to finally configure register (additional writes will not have any effect until the CPU is reset).*
- *1 and 0: set fields to exactly the value shown.*
- *DBlockRefill (DBR): set 1 to read 4 words into the cache on a miss, 0 to refill just the word missed on. The proper setting for a given system is dependent on a number of factors, and may best be determined by measuring performance in each mode and selecting the best one. Note that it is possible for software to dynamically reconfigure the refill algorithm depending on the current code executing, presuming the register has not been "locked".*
- *Force D-Cache Miss (FDM): set 1 for an RC3041-specific cache mode, where all loads result in data being fetched from memory (missing in the data cache), but the incoming data is still used to refill the cache. Stores continue to write the cache. This is useful when software desires to obtain the high-bandwidth of the cache and cache refills, but the corresponding main memory is "volatile" (e.g. a FIFO, or updated by DMA).*

Config Register (RC32364)

The Config register specifies various configuration options selected on the RC32364 processor.

1. **Note:** The external pin *Int3* corresponds to the bit numbered "5" in IP of the Cause register or IM of the SR register. That's because both the Cause and SR fields support two "software interrupts" numbered as bits 0 and 1.

Notes

Some configuration options, as defined by Config bits 31:3, are set by the hardware during reset and are included in the Config register as read only status bits for software access. The K0 field is the only read/ write field (as indicated by Config register bits 2:0) and is controlled by software. On reset, these fields are undefined.

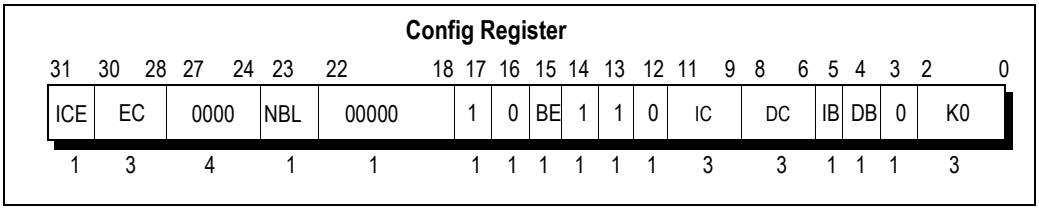


Figure 3.7 Config Register Format (RC32364)

Field	Description
ICE	In-Circuit Emulator existence: 0 → No ICE hardware connected to the CPU 1 → ICE hardware connected to the CPU These states are determined through an EJTAG Control Register bit.
EC	External Clock: Indicates the relationship of the execution core pipeline clock to the input system clock, as determined at reset: 0 → processor clock frequency multiplied by 2 1 → processor clock frequency multiplied by 3 2 → processor clock frequency multiplied by 4 3 → processor clock frequency multiplied by 5 4 → processor clock frequency multiplied by 6 5 → processor clock frequency multiplied by 7 6 → processor clock frequency multiplied by 8 7 Reserved
NBL	Non-Blocking Load pending. This read-only bit indicates there is a non-blocking load pending at the time this Config register is read. This bit is used for diagnostic purpose only. 0 → No non-blocking load pending 1 → There is a non-blocking load pending
BE	BigEndianMemory. The endianness is determined at reset. 0 → Little endian1 → Big endian
IC	Primary I-cache Size (I-cache size = 2^{9+IC} bytes). In RC32364 processor, this is set to 8 Kbytes (IC = 4)
DC	Primary D-cache Size (D-cache size = 2^{9+DC} bytes). In RC32364 processor, this is set to 2 Kbytes (DC = 2)
IB	Primary I-cache line size0 → 16 bytes (4 Words)
DB	Primary D-cache line size0 → 16 bytes (4 Words)
K0	*kseg0* coherency algorithm (uses same encodings as *EntryLo0* and *EntryLo1* registers)
Others	Reserved. Returns indicated values when read. Should be written with indicated values.

Table 3.9 Config Register Fields (RC32364)

Config Register (RC4600/RC4700)

The *Config* register specifies various configuration options selected on RC4600/RC4700 processors; Figure 3.8 lists these options.

Some configuration options, as defined by *Config* bits 31:3, are set by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access. The K0 field is the only read/write field (as indicated by *Config* register bits 2:0) and controlled by software; on reset these fields are undefined.

Notes

Figure 3.8 shows the format of the *Config* register; Table 3.10, which follows the figure, describes the *Config* register fields.

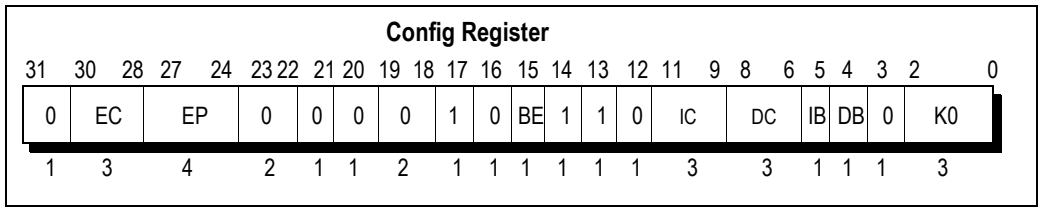


Figure 3.8 Config Register Format (RC4600/RC4700)

Field	Description
EC	System clock ratio: 0 → processor clock frequency divided by 2 1 → processor clock frequency divided by 3 2 → processor clock frequency divided by 4 3 → processor clock frequency divided by 5 4 → processor clock frequency divided by 6 5 → processor clock frequency divided by 7 6 → processor clock frequency divided by 8 7 Reserved
EP	Writeback data rate: 0 → DDDD Doubleword every cycle 1 → DDxDDx2 Doublewords every 3 cycles 2 → DDxxDDxx2 Doublewords every 4 cycles 3 → DxDxDxDx2 Doublewords every 4 cycles 4 → DDxxxDDxxx2 Doublewords every 5 cycles 5 → DDxxxxDDxxxx2 Doublewords every 6 cycles 6 → DxxDxxDxxDxx2 Doublewords every 6 cycles 7 → DDxxxxxDDxxxxx2 Doublewords every 7 cycles 8 → DxxxDxxxDxxxDxxx2 Doublewords every 8 cycles 9 - 15Reserved
BE	BigEndianMem 0 → Little endian1 → Big endian
IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes). In the RC4600/RC4700 processor, this is set to 16 Kbytes (IC = 010)
DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes). In the RC4600/RC4700 processor, this is set to 16 Kbytes (DC = 010)
IB	Primary I-cache line size1 → 32 bytes (8 Words)
DB	Primary D-cache line size1 → 32 bytes (8 Words)
K0	*kseg0* coherency algorithm (see *EntryLo0* and *EntryLo1* registers)
Others	Reserved. Returns indicated values when read.

Table 3.10 Config Register Fields (RC4600/RC4700)

Config Register (RC4650)

The *Config* register specifies various configuration options selected on RC4650 processors. Some configuration options, as defined by *Config* bits 31:3, are set by the hardware during reset and are included in the *Config* register as read-only status bits for the software to access.

Figure 3.9 shows the format of the *Config* register; Table 3.11, which follows the figure, describes the *Config* register fields.

Notes

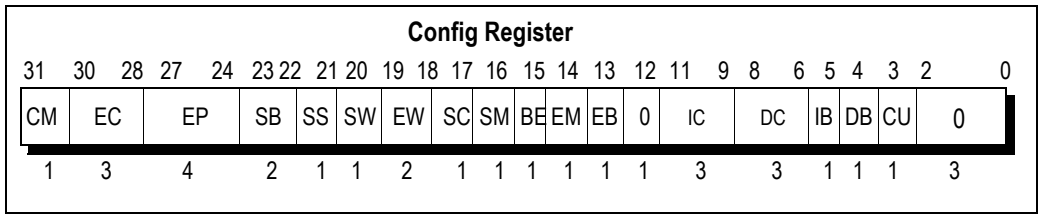


Figure 3.9 Config Register Format (RC4650)

Field	Description
EC	Pipeline clock ratio: 0 —>processor input clock frequency multiplied by 2 1 —> processor input clock frequency multiplied by 3 2 —> processor input clock frequency multiplied by 4 3 —> processor input clock frequency multiplied by 5 4 —> processor input clock frequency multiplied by 6 5 —> processor input clock frequency multiplied by 7 6 —> processor input clock frequency multiplied by 8 7Reserved
EP (EW=1)	Write-back data rate: 0 —> WWWWWWWW 1 word every cycle 1 —> WWxWWxWWxWW2 words every 3 cycles 2 —> WWxxWWxxWWxxWWxx2 words every 4 cycles 3 —> WxWxWxWxWxWxWxWx2 words every 4 cycles 4 —> WWxxxWWxxxWWxxxWWxxx2 words every 5 cycles 5 —> WWxxxxWWxxxxWWxxxxWWxxxx2 words every 6 cycles 6 —> WxxWxxWxxWxxWxxWxxWxxWxx2 words every 6 cycles 7 —> WWxxxxxWWxxxxxWWxxxxxWWxxxx2 words every 7 cycles 8 —> WxxxWxxxWxxxWxxxWxxxWxxxWxxxWxxx2 words every 8 cycles
EP (EW=0)	Write-back data rate: 0 —> DDDD 1 double word every cycle 1 —> DDxDDx2 double words every 3 cycles 2 —> DDxxDDxx2 double words every 4 cycles 3 —> DxDxDxDx2 double words every 4 cycles 4 —> DDxxxDDxxx2 double words every 5 cycles 5 —> DDxxxxDDxxxx2 double words every 6 cycles 6 —> DxxDxxDxxDxx2 double words every 6 cycles 7 —> DDxxxxxDDxxxx2 double words every 7 cycles 8 —> DxxxDxxxDxxxDxxx2 double words every 8 cycles
EW	SysAD bus size; 0 —> 64 bits, 1 —> 32 bits (from serial mode bits)
BE	BigEndianMem 0 —> Little Endian1 —> Big Endian
IC	Primary I-cache Size (I-cache size = 2^{12+IC} bytes). In the RC4650 processor this is set to 8KB (IC = 001).
DC	Primary D-cache Size (D-cache size = 2^{12+DC} bytes). In the RC4650 processor this is set to 8KB (DC = 001).
IB	Primary I-cache line size1 —> 32 bytes (8 Words)
DB	Primary D-cache line size1 —> 32 bytes (8 Words)
Others	Reserved. Returns indicated values when read.

Table 3.11 Config Register Format (RC4650)

Notes

BusCtrl Register (RC3041 only)

The RC3041 CPU has many hardware interface options not available on other members of the RC30xx family, which are intended to allow the use of simpler and cheaper interface and memory components. The *BusCtrl* register does most of the configuration work. It needs to be set strictly in accordance with the needs of the hardware implementation. Note also that its default settings (from reset) leave the interface compatible with other RC30xx family members.

Figure 3.10 shows the layout of the fields, and their uses are described in the list that follows the figure.

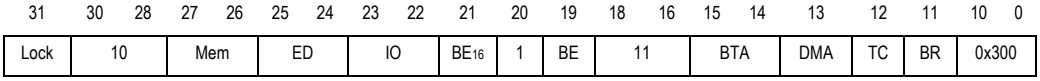


Figure 3.10 Fields in the R3041 Bus Control (BusCtrl) Register

- *Lock: when software has initialized BusCtrl to its desired state it may write this bit to prevent its contents being changed again until the system is reset.*
- *10 and other numbers: write exactly the specified bit pattern to this field (hex used for big ones, but others are given as binary). Improper values may cause test modes and other unexpected side effects.*
- *Mem: **MemStrobe*** control. Set this field to xy binary, where x set means the strobe activates on reads, and y set makes it active on writes.*
- *ED: **ExtDataEn*** control. Encoded as for "Mem". Note that the BR bit must be zero for this pin to function as an output.*
- *IO: **IOStrobe*** control. Encoded as for "Mem". Note that the BR bit must be zero for this pin to function as an output.*
- *BE16: "BE16**(1:0)*** read control" – 0 to make these pins active on write cycles only.*
- *BE: **BE(3:0)*** read control" – 0 to make these pins active on write cycles only.*
- *BTA: Bus turn around time. Program with a binary number between 0 and 3, for 0-3 cycles of guaranteed delay between the end of a read cycle and the start of the address phase of the next cycle. This field enables the use of devices with slow tri-state time, and enables the system designer to save cost by omitting data transceivers.*
- *DMA: DMA Protocol Control, enables "DMA pulse protocol". When set, the CPU uses its DMA control pins to communicate its desire for the bus even while a DMA is in progress.*
- *TC: **TC*** negation control. **TC*** is the output pin which is activated when the internal timer register Count reaches the value stored in Compare. Set TC zero to make the **TC*** pin just pulse for a couple of clock periods; leave TC as 1, and **TC*** will be asserted on a compare and remain asserted until software explicitly clears it (by re-writing Compare with any value).*

  *If **TC*** is used to generate a timer interrupt, then use the default (TC == 0). The pulse is more useful when the output is being used by external logic (e.g. to signal a DRAM refresh).*
- *BR: **SBrCond(3:2)** control. Set zero to recycle the **SBrCond(3:2)** pins as **IOStrobe** and **ExtDataEn** respectively.*

PortSize Register (RC3041 only)

The *PortSize* register is used to flag different parts of the program address space for accesses to 8-, 16- or 32-bit wide memory.

Hardware design requirements determine the settings of this register. See "IDT79RC3041 Hardware User's Manual" for details.

Context Register (RC4600/RC4700 only)

The *Context* register is a read/write register containing the pointer to an entry in the page table entry (PTE) array; this array is an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the CPU loads the TLB with the missing translation from the PTE array.

Notes

Normally, the operating system uses the *Context* register to address the current page map which resides in the kernel-mapped segment, *kseg3*. The *Context* register duplicates some of the information provided in the *BadVAddr* register, but the information is arranged in a form that is more useful for a software TLB exception handler. Figure 3.11 shows the format of the *Context* register; Table 3.12, which follows the figure, describes the *Context* register fields.

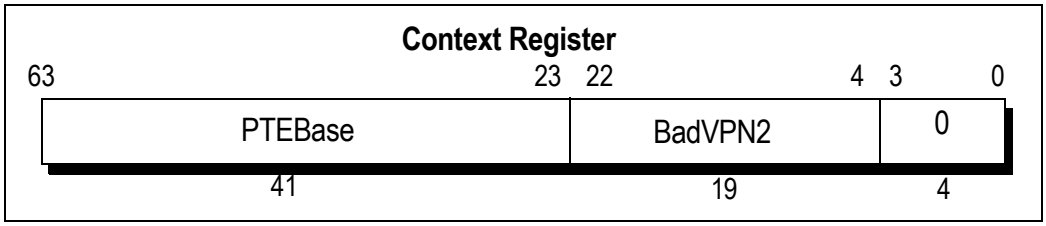


Figure 3.11 Context Register Format

Field	Description
BadVPN2	This field is written by hardware on a miss. It contains the virtual page number (VPN) of the most recent virtual address that did not have a valid translation.
PTEBase	This field is a read/write field for use by the operating system. It is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

Table 3.12 Context Register Fields

The 19-bit *BadVPN2* field contains bits 31:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format can directly address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

XContext Register (RC4600/RC4700 only)

The read/write *XContext* register contains a pointer to an entry in the page table entry (PTE) array, an operating system data structure that stores virtual-to-physical address translations. When there is a TLB miss, the operating system software loads the TLB with the missing translation from the PTE array. The *XContext* register duplicates some of the information provided in the *BadVAddr* register, and puts it in a form useful for a software TLB exception handler.

The *XContext* register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the *XContext* register to address the current page map, which resides in the kernel-mapped segment *kseg3*.

Figure 3.12 shows the format of the *XContext* register; Table 3.13, which follows the figure, describes the *XContext* register fields.

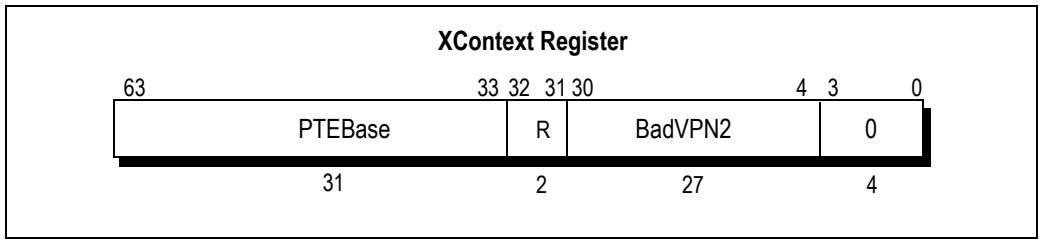


Figure 3.12 XContext Register Format

The 27-bit *BadVPN2* field has bits 39:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format may be used directly to address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Notes

Field	Description
BadVPN2	The *Bad Virtual Page Number*/2 field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address.
R	The *Region* field contains bits 63:62 of the virtual address. 00_2 = user 01_2 = supervisor 11_2 = kernel.
PTEBase	The *Page Table Entry Base* read/write field is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

Table 3.13 XContext Register Fields

Error Checking and Correcting (ECC) Register (RC4600/RC4700/ RC4650/RC32364 only)

The 8-bit *Error Checking and Correcting* (*ECC*) register reads or writes primary-cache data parity bits for cache initialization, cache diagnostics, or cache error processing. (Tag parity is loaded from and stored to the *TagLo* register.)

The *ECC* register is loaded by the Index Load Tag CACHE operation. Content of the ECC register is:

- *written into the primary data cache on store instructions (instead of the computed parity) when the CE bit of the Status register is set*
- *substituted for the computed instruction parity for the CACHE operation Fill*

To force a cache parity value, use the *Status CE* bit and the ECC register.

Figure 3.13 shows the format of the *ECC* register. Table 3.14, which follows the figure, describes the register fields.

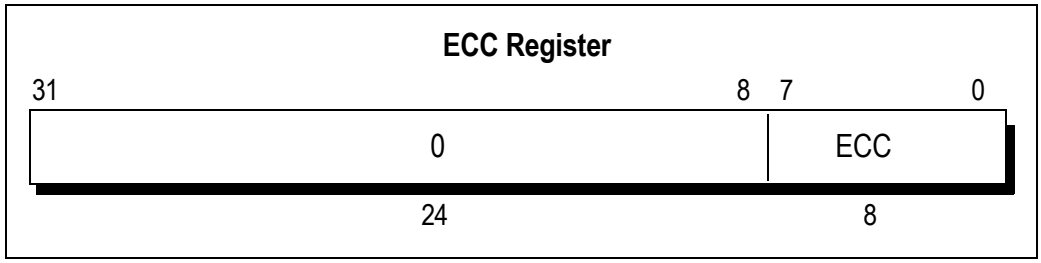


Figure 3.13 ECC Register Format

Field	Description
ECC	An 8-bit field specifying the parity bits read from or written to a primary cache.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 3.14 ECC Register Fields

Cache Error (CacheErr) Register (RC4600/RC4700/RC4650/RC5000/ RC32364 only)

The 32-bit read-only *CacheErr* register processes parity errors in the primary cache. Parity errors cannot be corrected.

The *CacheErr* register holds cache index and status bits that indicate the source and nature of the error; it is loaded when a Cache Error exception is taken. When a read response (cached or uncached) returns with bad parity, this exception is also taken.

Figure 3.14 shows the format of the *CacheErr* register; Table 3.15, which follows the figure, describes the *CacheErr* register fields.

Notes

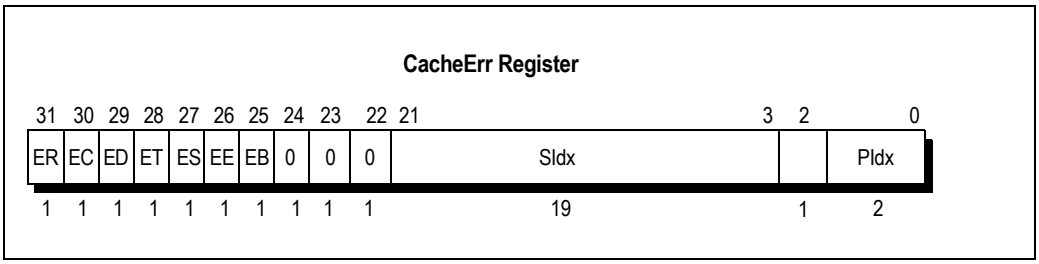


Figure 3.14 CacheErr Register Format

Field	Description
ER	Type of reference 0 → instruction 1 → data
EC	Cache level of the error 0 → primary 1 → reserved
ED	Indicates if a data field error occurred 0 → no error 1 → error
ET	Indicates if a tag field error occurred 0 → no error 1 → error
ES	Indicates the error occurred accessing processor-managed resources, in response to an external request. 0 → internal reference 1 → external reference Since the RC4600/RC4700 doesn't have any external events that would look in a cache (which is the only processor-managed resource), this bit would not be set under normal operating conditions.
EE	Set if the error occurred on the SysAD bus. Taking a cache error exception sets/clears this bit.
EB	Set if a data error occurred in addition to the instruction error (indicated by the remainder of the bits). If so, this requires flushing the data cache after fixing the instruction error.
SIdx	Physical address 21:3 of the reference that encountered the error. The address may not be the same as the address of the double word in error, but it is sufficient to locate that double word in the secondary cache.
PIdx	Virtual address 13:12 of the double word in error. To be used with SIdx to construct a virtual index for the primary caches. Only the lower two bits (bits 1 and 0) are vAddr; the high bit (bit 2) is zero.
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 3.15 CacheErr Register Fields

Error Exception Program Counter (Error EPC) Register (RC4600/ RC4700/RC4650/RC32364 only)

The *ErrorEPC* register is similar to the *EPC* register, except that *ErrorEPC* is set on Reset, Soft Reset, NMI, and CACHE errors. It is also used to store the program counter (PC) on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions. Note that there is no branch delay slot indication for the *ErrorEPC* register.

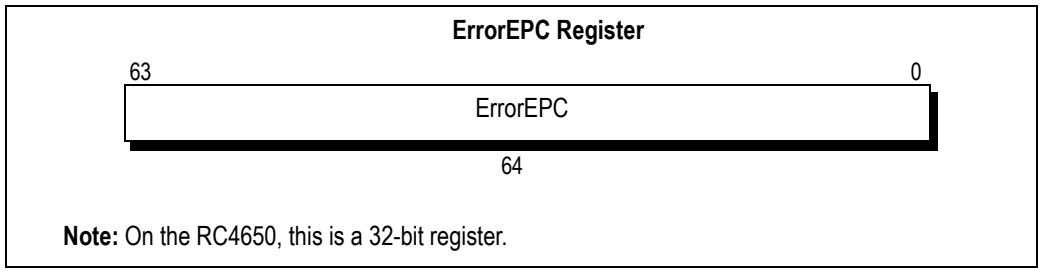
The read/write *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- *the virtual address of the instruction that caused the exception*

Notes

- *the virtual address of the immediately preceding branch or jump instruction, when this address is in a branch delay slot.*

Figure 3.15 shows the format of the *ErrorEPC* register.



Note: On the RC4650, this is a 32-bit register.

Figure 3.15 ErrorEPC Register Format

IWatch Register (RC4650/RC32364 only)

The *IWatch* register is a read/write register that specifies an Instruction virtual address that causes a Watch exception. When $VADDRC3_{1..2}$ of an instruction fetch matches IVAddr of this register, and the I bit is set, a Watch exception is taken. Matches that occur when EXL = 1 or ERL = 1 do not take the exception immediately, but are instead postponed until both EXL and ERL are cleared. The priority of IWatch exceptions is just below Instruction Address Error exceptions. Figure 3.16 shows the format of the *IWatch* register; Table 3.16, which follows the figure, describes the *IWatch* register fields.

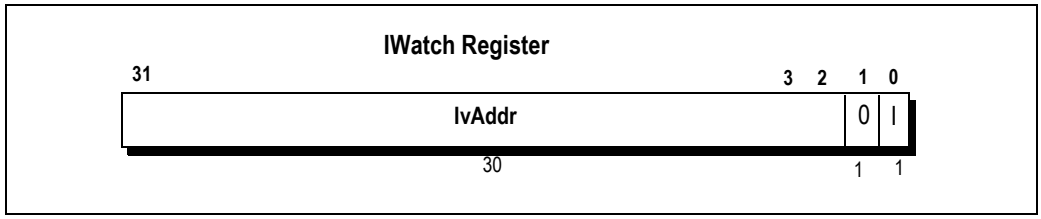


Figure 3.16 IWatch Register Format

Field	Description
IvAddr	Instruction virtual address that causes a watch exception (bits 31:2).
I	0 **--->** IWatch disabled, 1 **--->** IWatch enabled.
0	reserved for future use.
Note: IWatch.I is cleared on Reset.	

Table 3.16 /Watch Register Fields

DWatch Register (RC4650/RC32364 only)

DWatch is a read/write register that specifies a Data virtual address that causes a Watch exception. Data Watch exception is taken when $VAddr_{31..3}$ of a load matches DVAddr of this register and the R bit is set, or when $VAddr_{31..3}$ of a store matches DvAddr of this register and the W bit is set. Matches that occur when EXL = 1 or ERL = 1 do not take the exception immediately, but are instead postponed until both EXL and ERL are cleared. The priority of DWatch exceptions is just below Data Address Error exceptions. DWatch exceptions do not occur on CACHE ops.

Figure 3.17 shows the format of the *DWatch* register. Table 3.17, which follows the figure, describes the *DWatch* register fields.

Notes

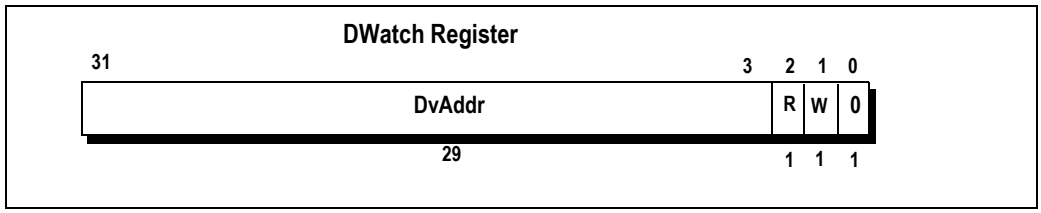


Figure 3.17 DWatch Register Format

Field	Description
DvAddr	Data virtual address that causes a watch exception.
R	0 **--->** DWatch disabled for loads, 1 **--->** DWatch enabled for loads.
W	0 **--->** DWatch disabled for stores, 1**--->** DWatch enabled for stores.
0	reserved for future use.
Note: DWatch.R and DWatch.W are cleared on Reset.	

Table 3.17 DWatch Register Fields

TagLo Register (RC4650/RC32364 only)

The *TagLo* register is a 32-bit read/write register that holds the primary cache tag and parity during cache initialization, cache diagnostics, or cache error processing. The *TagLo* register is written by the CACHE and MTC0 instructions. The *P* field of this register is ignored on Index Store Tag operations. Parity is computed by the store operation.

Figure 3.18 shows the format of the TagLo register, for primary cache operations, and Table 3.18 explains the fields.

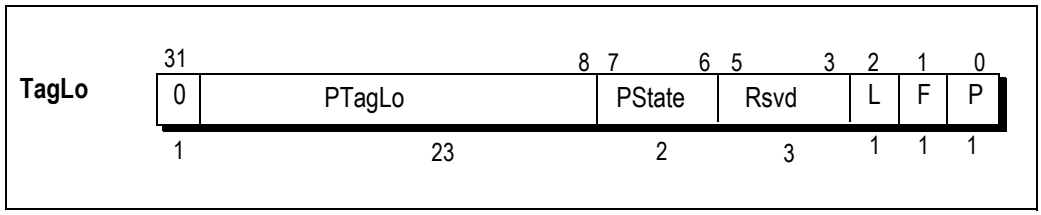


Figure 3.18 TagLo Register Format

Field	Description
PTagLo	**In the case of Data Cache**, the PTagLo field specifies the physical address bits 31:9. **In the case of Instruction Cache** (8kbytes), the PTagLo field specifies the physical address bits 31:11. The 2 least significant bits are undefined.
PState	Specifies the primary cache state.
P	Specifies the primary tag even parity bit.
F	The FIFO bit used to implement FIFO refill of the cache. For software, there is no particular use of this bit.
Rsvd	Reserved. Must be written as zeroes.
L	Lock bit used to implement cache line lock function.

Table 3.18 TagLo Register Field Descriptions

Notes

Value	Cache State Attribute
0	Invalid
1	Shared
2	Clean Exclusive
3	Dirty Exclusive

Table 3.19 Primary Cache State Values

CP0 Registers and System Operation Support

The various CP0 registers and their fields provide support at specific times during system operation.

- *After hardware reset: software must initialize SR to get the CPU into the right state to bootstrap itself.*
- *Hardware configuration at start-up: an RC3041, RC3071, or RC3081 require initialization of Config, BusCtrl, and/or PortSize at start-up. The system hardware implementation will dictate the proper configuration of these registers. Note that most of the bits of the Config register in a RC4xxx are initialized from the boot-time mode stream which software programmers have no control over other than to read them.*
- *After any exception: any MIPS exception (apart from one particular MMU event) invokes a single common "general exception handler" routine, at a fixed address.*

  On entry, no program registers are saved, only the return address in EPC/ErrorEPC. The MIPS hardware knows nothing about stacks.

  Exception software will need to use at least one of k0 and k1 to point to some "safe" (exception-proof) memory space. Key information can be saved, using the other k0 or k1 register to stage data from control registers where necessary.

  Consult the Cause register to find out what kind of exception it was and dispatch accordingly.
- *Returning from exception: control must eventually be returned to the value stored in EPC on entry.*

  Whatever kind of exception it was, software will have to adjust SR back upon return from exception. In the RC30xx, the special instruction rfe does the job; but note that it does not transfer control. To make the jump back software must load the original EPC value back into a general-purpose register and use a jr operation. In the RC4xxx, the special instruction is eret and unlike rfe, it actually does transfer the control and a jr is not needed after it.
- *Interrupts: SR is used to adjust the interrupt masks, to determine which (if any) interrupts will be allowed "higher priority" than the current one. The hardware offers no interrupt prioritizing, but the software can do whatever it likes.*
- *Instructions that always cause exceptions: are often used (for system calls, breakpoints, and to emulate some kinds of instruction). These sometimes require partial decoding of the offending instruction, which can usually be found at the location EPC. But there is a complication; suppose that an exception occurs just after a branch but in time to prevent the branch delay slot instruction from running. Then EPC will point to the branch instruction (resuming execution starting at the delay slot would cause the branch to be ignored), and the BD bit will be set.*

  This Cause register bit flags this event; to find the instruction at which the exception occurred, add 4 to the EPC value when the BD bit is set.
- *Cache management routines: In the RC3xxx, SR contains bits defining special modes for cache management. In particular, they allow software to isolate the data cache, and to swap the roles of the instruction and data caches.*

The subsequent chapters will describe appropriate treatment of these registers, and provide software examples of their use.

Exception Management

Notes

This chapter describes the software techniques used to recognize and decode exceptions, save state, dispatch exception service routines, and return from exception. Various code examples are provided.

Exceptions

In the MIPS architecture interrupts, traps, system calls or any event that disrupts the normal flow of execution are called "exceptions" and are handled by a single mechanism. Possible events that disrupt normal flow include:

- *External events: For example, interrupts, or a bus error on a read. Note that for the RC30xx floating point exceptions are reported as interrupts, since when the RC3000A was originally implemented the FPA was indeed external.*
- *Interrupts are the only exception conditions which can be disabled under software control.*
- *Program errors and unusual conditions: For example, illegal instructions (including "co-processor" instructions executed with the appropriate SR disabled), integer overflow, address alignment errors, accesses outside kuseg in user mode.*
- *Memory translation exceptions: For example, using an invalid translation, or a write to a write-protected page; and access to a page for which there is no translation in the TLB or the XTLB (in the RC4x00 only); accessing addresses beyond values set for IBound or Dbound or hitting the IWatch or DWatch address values in the RC4650 only.*
- *System calls and traps: For example, exceptions deliberately generated by software to access kernel facilities in a secure way (syscalls, conditional traps planted by careful code, and breakpoints).*

Some events do not cause exceptions, although other CPU architectures may handle them as such. Software must use other mechanisms to detect:

- *bus errors on write cycles (MIPS CPUs don't detect these as exceptions at all; the use of a write buffer would make such an exception "imprecise", in that the instruction which generated the store data is not guaranteed to be the one which recognizes the exception).*
- *parity errors detected in the cache (the PE bit in SR is set, but no exception is signalled).*

Precise Exceptions

IDT devices implement *precise exceptions*:

- *Unambiguous cause: after an exception caused by any internal error, the EPC points to the instruction which caused the error (it might point to the preceding branch for an instruction which is in a branch delay slot, but will signal occurrence of this using the BD bit).*
- *Exceptions are seen in instruction sequence: exceptions can arise at several different stages of execution, creating a potential hazard. For example, if a load instruction suffers a TLB miss the exception won't be signalled until the "MEM" pipestage; if the next instruction suffers an instruction TLB miss (at the "IF" pipestage) the logically second exception would be signalled first (since the IF occurs earlier in the pipe than MEM).*
- *To avoid this problem, early-detected exceptions are not activated until it is known that all previous instructions will complete successfully; in this case, the instruction TLB miss is suppressed and the exception caused by the earlier instruction handled. The second exception may or may not happen again upon return from handling the data fault.*
- *Subsequent instructions nullified: because of the pipelining, instructions lying in sequence after the EPC may well have been started. But the architecture guarantees that no effects produced by these instructions will be visible in the registers or CPU state; and no effect at all will occur which will prevent execution being restarted at the EPC.*

Notes

Note that this isn't quite true of, for example, the result registers in the integer multiply unit (logically, the architecture considers these changed by the initiation of a multiply or divide). But provided that the instruction arrangement rules required by the assembler are followed, no problems will arise.

The implementation of precise exceptions requires a number of clever techniques. For example, the FPA cannot update the register file until it knows that the operation will not generate an exception. However, the CPU contains logic to allow multi-cycle FPA operations to occur concurrently with integer operations, yet maintain precise exceptions.

Exception Timing

The architecture determines that an exception seems to have happened just before the execution of the instruction which caused it. The first fetch from the exception routine will be made within 1 clock of the time when the faulting instruction would have finished; in practice it is often faster.

On an interrupt, for most devices, the last instruction to be completed before interrupt processing starts will be the one which has just finished its MEM stage when the interrupt is detected. The *EPC* target will be the one which has just finished its ALU stage.

Some of the interrupt inputs to RC30xx family CPUs are resynchronised internally (to support interrupt signalling from asynchronous sources) and the interrupt will be detected only on the rising edge of the second clock after the interrupt becomes active.

Exception Vectors

In the RC30xx only one exception is handled differently; a TLB miss on an address in *kuseg*. Although the architecture uses software to handle this condition (which occurs very frequently in a heavily-used multi-tasking, virtual memory OS), there is significant architectural support for a "preferred" scheme for TLB refill. The preferred refill scheme can be completed in about 13 clocks in the RC30xx.

In the RC4600/RC4700 two other exceptions are handled differently in addition to the TLB miss described above. These are the XTLB miss and the cache error exceptions.

In the RC4650, the TLB does not exist, but the cache error is handled in a manner similar to that in RC4600/RC4700. In addition, the RC4650 has an extra feature for handling "interrupts" only (*dedicated* interrupt vector). The "IV" bit in the *Cause* register can be set by the user. If the "IV" bit is set, the program control jumps to a unique location up on receiving and interrupt type exception, where the user can have his interrupt handlers. This allows for faster interrupt handling and simplified code development.

It is also useful to have two sets of entry points. It is essential for high performance to locate the vectors in cached memory for OS use, but this is highly undesirable at start-up; the need for a robust and self-diagnosing start-up sequence mandates the use of uncached read-only memory for vectors.

So the exception system adds some more "magic" addresses (exception vectors) to the one used for system start-up. The number of "magic" addresses is 4 in case of RC30xx, 6 in RC4650, 10 in RC32364 and 8 in RC4x00. The reset mechanism on the MIPS CPU is remarkably like the exception mechanism, and is sometimes referred to as the *reset exception*. The complete list of exception vector addresses is shown in Table 4.1

	Vector Virtual Address		
Description	**RC30xx**	**RC4600/RC4700/RC5000**	**RC4650**
Reset exception	0xbfc0 0000	0xffff ffff bfc0 0000	0xbfc0 0000
TLB miss (BEV=0)	0x8000 0000	0xffff ffff 8000 0000	N/A
TLB miss (BEV=1)	0xbfc0 0100	0xffff ffff bfc0 0200	N/A
XTLB miss (BEV=0)	N/A	0xffff ffff 8000 0080	N/A

Table 4.1 Exception Vector Addresses (Part 1 of 2)

Notes

	Vector Virtual Address		
Description	**RC30xx**	**RC4600/RC4700/RC5000**	**RC4650**
XTLB miss (BEV=1)	N/A	0xffff ffff bfc0 0280	N/A
cache error (BEV=0)	N/A	0xffff ffff a000 0100	0xa000 0100
cache error (BEV=1)	N/A	0xffff ffff bfc0 0300	0xbfc0 0300
new dedicated interrupt in RC4650 (BEV=0)	N/A	N/A	0x8000 0200
new dedicated interrupt in RC4650 (BEV=1)	N/A	N/A	0xbfc0 0400
All other exceptions (BEV=0)	0x8000 0080	0xffff ffff 8000 0180	0x8000 0180
All other exceptions (BEV=1)	0xbfc0 0180	0xffff ffff bfc0 0380	0xbfc0 0380

Table 4.1 Exception Vector Addresses (Part 2 of 2)

Exception	BEV	EXL	IV	ICE	RC32364 Processor Vector
Reset, Soft Reset, NMI	X	X	X	X	0xBFC0 0000
Debug (ICE)	X	X	X	1	0xFF20 0200
Debug (ICE)	X	X	X	0	0xBFC0 0480
TLB refill	1	0	X	X	0xBFC0 0200
TLB refill	1	1	X	X	0xBFC0 0380
TLB refill	0	0	X	X	0x8000 0000
TLB refill	0	1	X	X	0x8000 0180
Cache Error	1	X	X	X	0xBFC0 0300
Cache Error	0	X	X	X	0xA000 0100
Interrupt	1	X	1	X	0xBFC0 0400
Interrupt	1	X	0	X	0xBFC0 0380
Interrupt	0	X	1	X	0x8000 0200
Interrupt	0	X	0	X	0x8000 0180
Others	1	X	X	X	0xBFC0 0380
Others	0	X	X	X	0x8000 0180
Note: X means don't care					

Table 4.2 RC32364 Exception Vectors

On an exception, the CPU hardware:

1. Saves current PC to the ErrorPC, on Cache errors, NMI, reset, or soft reset. All other exceptions are saved to the EPC.
2. Saves the pre-existing user-mode and interrupt-enable flags in the status register (SR) by pushing the 3-entry stack inside *SR,* changing to kernel mode with interrupts disabled.
3. Sets the *Cause* register, which identifies exception type and pending external interrupts. For addressing exceptions, the *BadVaddr* register is set. And for Memory management system exceptions, MMU registers are set (for more details on memory management, refer to Chapter 6).
4. Transfers control to the exception vector address.

Exception Handling – Basics

Any MIPS exception handler must go through the same stages:

Notes

- *Bootstrapping: on entry to the exception handler, because the state of the interrupted program was not saved, the first job is to provide room to preserve relevant state information.*

  This is done by using the k0 and k1 registers (which are reserved for "kernel mode" use, and therefore should contain no application program state), to reference a piece of memory which can be used for other register saves.
- *Dispatching different exceptions: consult the Cause register. The initial decision is likely to be made on the "Exclude" field, which is thoughtfully aligned so that its code value (between 0 and 31) can be used to index an array of words without a shift. The code will be something like this:*

```
mfc0    t1, C0_CAUSE
nop
and     t2, t1, 0x3f
lw      t2, tablebase(t2)
nop
jr      t2
```

- *Constructing the exception processing environment: complex exception handling routines may be written in a high level language; in addition, software may wish to be able to use standard library routines. To do this, software will have to switch to a suitable stack, and save the values of all registers which "called subroutines" may use.*
- *Processing the exception: this is system and cause dependent.*
- *Returning from an exception: The return address is contained in the EPC register on exception entry; the value must be placed into a general purpose register for return from exception (note that the software may have placed the EPC value on the stack at exception entry).*
  - *In the RC30xx, returning control is now done with a jr instruction, and the change of state back from kernel to the previous mode is done by an rfe instruction after the jr, in the delay slot.*
  - *In the RC4xxx, the returning mechanism is different. The instruction eret does everything including returning. There are no delay slots of eret and eret itself also must not be placed in a delay slot of some other instruction. eret picks up the return address either from the ErrorEPC register or the EPC register depending up on whether an error trap was being serviced or not. It also clears the ERL or EXL bit appropriately.*

Nesting Exceptions

In many cases, the system may permit, or will be unable to avoid, nested exceptions: exceptions occurring within the exception processing routine-*nested* exceptions.

If improperly handled, this could cause chaos; vital state for the interrupted program is held in *EPC* (or *ErrPC* for RC4xxx) and *SR,* and another exception would overwrite them. To permit nested exceptions, these values must be saved elsewhere. Moreover, once exceptions are re-enabled, software can no longer rely on the values of *k0* and *k1, since a subsequent (nested) exception may alter their values.*

The normal approach to this is to define an *exception frame*; a memory-resident data structure with fields to store incoming register values, so that they can be retrieved on return. Exception frames are usually arranged logically as a stack.

Stack resources are consumed by each exception, so arbitrarily nested exceptions cannot be tolerated. Most systems sort exceptions into a priority order, and arrange that while an exception is being processed only higher-priority exceptions are permitted. Such systems need have only as many exception frames as there are priority levels.

Software can inhibit certain exceptions, as follows:

- *Interrupts: can be individually masked by software to conform to system priority rules;*
- *Privilege Violations: can't happen in kernel mode; virtually all exception service routines will execute in kernel mode;*
- *Addressing errors, TLB misses, Bound violations: software must be written to ensure that these never happen when processing higher priority exceptions.*

Notes

Typical system priorities are (lowest first): non-exception code, interrupt (lowest)... interrupt (highest), Data Watch (RC4650), data Cache error (RC4xxx), Data TLB errors (non-RC4650), Data address errors, Data Bounds error (RC4650), illegal instructions and traps, bus errors, Instruction TLB errors (non-RC4650), instruction Cache error, Instruction Watch (RC4650), Instruction address errors, Instruction Bounds error (RC4650), soft reset.

Exception Routines

The following are a set of exception related routines from IDT/sim.

The routine "_exception" receives exceptions, saves all state, and calls the appropriate service routine. Code used to install the exception handler in memory is also shown.

Majority of the code is common to both RC30xx as well as RC4xxx. Places where code is unique to either RC30xx or RC4x00 or RC4650 or RC32364 have been enclosed within **#ifdef CPU_R3000** or **#ifdef CPU_R4000 or CPU_R4650 or CPU_RC32364** respectively, along with comments at the top of each routine.

In general, the differences between RC30xx and RC4xxx exception issues are:

- *RC4xxx has additional exception vectors (cacheerror, xtlb)*
- *As compared to RC30xx, RC4xxx has additional as well as different CP0 registers which become part of saved context prior to handling an interrupt*
- *returning from handler with rfe and jump in RC30xx as opposed to eret in RC4xxx and RC32364*

```
/*
** exception.s - contains functions for setting up and handling exceptions
** Copyright 1989-1998 Integrated Device Technology, Inc. All Rights Reserved
*/

#include "iregdef.h"
#include "idtcpu.h"
#include "idtmon.h"
#include "setjmp.h"
#include "excepthd.h"
#include "under.h"
/*
** move_exc_code() - moves the exception code to the
** utlb and gen exception vectors in RC30xx
**                  OR
** tlb, xtlb, cacheerror, and gen exception vectors in RC4xxx
**
*/
FRAME(move_exc_code,sp,0,ra)
#if defined(DEBUGSIM) || defined(NO_COPY_EXCEPTION_VECTOR)
        j       ra
#endif
#if defined(CPU_R3000)
        .set    noreorder
        la      t1,exc_utlb_code
        la      t2,exc_norm_code
        li      t3,UT_VEC
        li      t4,E_VEC
        li      t5,VEC_CODE_LENGTH
1:
        lw      t6,0(t1)
        lw      t7,0(t2)
        sw      t6,0(t3)
        sw      t7,0(t4)
        addiu   t1,4
        addiu   t3,4
        addiu   t4,4
        subu    t5,4
        bne     t5,zero,1b
        addiu   t2,4
```

Notes

```
        move    t5,ra           # assumes clear_cache doesnt use t5
        li      a0,UT_VEC
        jal     clear_cache
        li      a1,VEC_CODE_LENGTH
        nop
        li      a0,E_VEC
        jal     clear_cache
        li      a1,VEC_CODE_LENGTH
        move    ra,t5           # restore ra
        j       ra
        nop
        .set    reorder
#endif
#if defined(CPU_R4000) || defined(CPU_RC32364)
        move    t5,ra           # assumes clear_cache doesnt use t5
/*
** Check to see if RC4650.  If yes, don't have to copy tlb-related vectors
*/
        lw      a0,cputype
        and     a0,R4650
        bnez    a0,r4650_vec

        /* TLB exception vector */
        la      t1,exc_tlb_code
        li      t2,T_VEC
        li      t3,VEC_CODE_LENGTH
1:
        lw      t6,0(t1)
        addiu   t1,4
        subu    t3,4
        sw      t6,0(t2)
        addiu   t2,4
        bne     t3,zero,1b
    nop
    nop
        li      a0,T_VEC
        li      a1,VEC_CODE_LENGTH
        jal     clear_cache
    nop
    nop
#if defined(CPU_R4000)
        la      t1,exc_xtlb_code
        li      t2,X_VEC
        li      t3,VEC_CODE_LENGTH
1:
        lw      t6,0(t1)
        addiu   t1,4
        subu    t3,4
        sw      t6,0(t2)
        addiu   t2,4
        bne     t3,zero,1b


        /* extended TLB exception vector */
        li      a0,X_VEC
        li      a1,VEC_CODE_LENGTH
        jal     clear_cache
#endif
r4650_vec:
        /* cache error exception vector */
        la      t1,exc_cache_code
        li      t2,C_VEC
        li      t3,VEC_CODE_LENGTH
1:
        lw      t6,0(t1)
        addiu   t1,4
```

Notes

```
        subu    t3,4
        sw      t6,0(t2)
        addiu   t2,4
        bne     t3,zero,1b
      nop
        li      a0,C_VEC
        li      a1,VEC_CODE_LENGTH
        jal     clear_cache
    nop
        /* normal exception vector */
        la      t1,exc_norm_code
        li      t2,E_VEC
        li      t3,VEC_CODE_LENGTH
1:
        lw      t6,0(t1)
        addiu   t1,4
        subu    t3,4
        sw      t6,0(t2)
        addiu   t2,4
        bne     t3,zero,1b
    nop
        li      a0,E_VEC
        li      a1,VEC_CODE_LENGTH
        jal     clear_cache
    nop
/*
**      The following is the interrupt handling vector for RC4650.
*/
        lw      a0,cputype
        and     a0,R4650
        beqz    a0,not4650
        nop
       /* Is the CPU RC4650? If not, skip over */

        /* interrupt vector specific to RC4650 */
    la    t1,exc_norm_code
    li    t2,I_VEC
    li    t3,VEC_CODE_LENGTH
1:
    lw    t6,0(t1)
    addiu  t1,4
    subu   t3,4
    sw    t6,0(t2)
    addiu  t2,4
    bne   t3,zero,1b
    nop
    li    a0,I_VEC
    li    a1,VEC_CODE_LENGTH
    jal   clear_cache
    nop
not4650:
        move    ra,t5               # restore ra
        j       ra
    nop
#endif
ENDFRAME(move_exc_code)

** enable_int(mask) - enables interrupts - mask is positoned so it only
** needs to be or'ed into the status reg. This also does some other things !!!!
** caution should be used if invoking this while in the middle of a debugging
** session where the client may have nested interrupts.
**
** This code will work for both RC30xx as well as RC4xxx
**
*/
FRAME(enable_int,sp,0,ra)
```

Notes

```
#if defined(CPU_R3000)
        .set    noreorder
        la      t0,client_regs
        lreg    t1,R_SR*R_SZ(t0)
        nop
        or      t1,SR_IEP
        or      t1,a0
        sreg    t1,R_SR*R_SZ(t0)
        mfc0    t0,C0_SR
        or      a0,SR_IEC
        or      t0,a0
        mtc0    t0,C0_SR
        j       ra
        nop
        .set    reorder
#endif
#if defined(CPU_R4000)|| defined(CPU_RC32364)
        la      t0,client_regs
        lreg    t1,R_SR*R_SZ(t0)
        or      a0,SR_IE
        or      t1,a0
        sreg    t1,R_SR*R_SZ(t0)

        mfc0    t1,C0_SR
        or      t1,a0
        mtc0    t1,C0_SR

        j       ra
    nop
#endif
ENDFRAME(enable_int)

/*
** disable_int(mask) - disable the interrupt - mask is the compliment of
** the bits to be cleared - i.e. to clear ext int 5 the mask would be - 0xffff7fff
**
** This code will work for both RC30xx as well as RC4xxx
**
*/
FRAME(disable_int,sp,0,ra)
#if defined(CPU_R3000)
        .set    noreorder
        la      t0,client_regs
        lreg    t1,R_SR*R_SZ(t0)
        nop
        and     t1,a0
        sreg    t1,R_SR*R_SZ(t0)
        mfc0    t0,C0_SR
        nop
        and     t0,a0
        mtc0    t0,C0_SR
        j       ra
        nop
        .set    reorder
#endif
#if defined(CPU_R4000)||defined(CPU_RC32364)
        la      t0,client_regs
        lreg    t1,R_SR*R_SZ(t0)
        and     t1,a0
        sreg    t1,R_SR*R_SZ(t0)
        mfc0    t1,C0_SR
        and     t1,a0
        mtc0    t1,C0_SR
        j       ra
    nop
#endif
```

Notes

```
ENDFRAME(disable_int)


/*
** the following sections of code are copied to the vector area
**      at location 0x80000000 (utlb miss) and location 0x80000080
**      (general exception).
**
*/

        .set    noreorder
        .set    noat            # must be set so la does not use at

#if defined(CPU_R3000)
FRAME(exc_utlb_code,sp,0,ra)
        la      k0,except_regs
        sreg    AT,R_AT*R_SZ(k0)
        sreg    gp,R_GP*R_SZ(k0)
        sreg    v0,R_V0*R_SZ(k0)
        li      v0,UTLB_EXCEPT
        la      AT,_exception
        j       AT
        nop
        ENDFRAME(exc_utlb_code)
FRAME(exc_norm_code,sp,0,ra)
        la      k0,except_regs
        sreg    AT,R_AT*R_SZ(k0)
        sreg    gp,R_GP*R_SZ(k0)
        sreg    v0,R_V0*R_SZ(k0)
        li      v0,NORM_EXCEPT
        la      AT,_exception
        j       AT
        nop
        ENDFRAME(exc_norm_code)
#endif
#if defined(CPU_R4000)||defined(CPU_RC32364)
FRAME(exc_tlb_code,sp,0,ra)
        la      k0,except_regs
        sreg    AT,R_AT*R_SZ(k0)
        sreg    gp,R_GP*R_SZ(k0)
        sreg    v0,R_V0*R_SZ(k0)
        li      v0,TLB_EXCEPT
        la      AT,_exception
        j       AT
        nop
        ENDFRAME(exc_tlb_code)
#if !defined(CPU_RC32364)
FRAME(exc_xtlb_code,sp,0,ra)
        la      k0,except_regs
        sreg    AT,R_AT*R_SZ(k0)
        sreg    gp,R_GP*R_SZ(k0)
        sreg    v0,R_V0*R_SZ(k0)
        li      v0,XTLB_EXCEPT
        la      AT,_exception
        j       AT
        nop
        ENDFRAME(exc_xtlb_code)
#endif
FRAME(exc_cache_code,sp,0,ra)
        la      k0,except_regs
        sreg    AT,R_AT*R_SZ(k0)
        sreg    gp,R_GP*R_SZ(k0)
        sreg    v0,R_V0*R_SZ(k0)
        li      v0,CACHE_EXCEPT
        la      AT,_exception  /* FIXME: this can't be handled this way */
        j       AT
```

Notes

```
        nop
        ENDFRAME(exc_cache_code)
FRAME(exc_norm_code,sp,0,ra)
        la      k0,except_regs
        sreg    AT,R_AT*R_SZ(k0)
        sreg    gp,R_GP*R_SZ(k0)
        sreg    v0,R_V0*R_SZ(k0)
        li      v0,NORM_EXCEPT
        la      AT,_exception
        j       AT
        nop
        ENDFRAME(exc_norm_code)
#endif
        .set    reorder

/*
** common exception handling code
** Save various registers so we can print informative messages
** for faults (whether in monitor or client mode)
**      Reg.(k0) points to the exception register save area.
**      If we are in client mode then some of these values will
**      have to be copied to the client register save area.
*/
FRAME(_exception,sp,0,ra)
        .set    noreorder
        sreg    v0,R_EXCTYPE*R_SZ(k0)    # save exception type (gen or utlb)
        sreg    v1,R_V1*R_SZ(k0)
        mfc0    v0,C0_EPC
        mfc0    v1,C0_SR
        nop
        sreg    v0,R_EPC*R_SZ(k0)   # save the pc at the time of the exception
        sreg    v1,R_SR*R_SZ(k0)
        mfc0    v0,C0_IWATCH
        mfc0    v1,C0_ECC
        nop
        sreg    v0,R_IWATCH*R_SZ(k0)
        sreg    v1,R_ECC*R_SZ(k0)
        mfc0    v0,C0_TLBLO0
        mfc0    v1,C0_TLBLO1
        nop
        sreg    v0,R_TLBLO0*R_SZ(k0)
        sreg    v1,R_TLBLO1*R_SZ(k0)
        mfc0    v0,C0_INX
        mfc0    v1,C0_RAND
        nop
        sreg    v0,R_INX*R_SZ(k0)
        sreg    v1,R_RAND*R_SZ(k0)
        mfc0    v0,C0_WIRED
        mfc0    v1,C0_CTXT
        nop
        sreg    v0,R_WIRED*R_SZ(k0)
        sreg    v1,R_CTXT*R_SZ(k0)
        mfc0    v0,C0_PAGEMASK
        mfc0    v1,C0_COUNT
        nop
        sreg    v0,R_PAGEMASK*R_SZ(k0)
        sreg    v1,R_COUNT*R_SZ(k0)
        mfc0    v0,C0_PRID
        mfc0    v1,C0_DWATCH
        nop
        sreg    v0,R_PRID*R_SZ(k0)
        sreg    v1,R_DWATCH*R_SZ(k0)
        mfc0    v0,C0_ERRPC
        mfc0    v1,C0_CAUSE
        nop
        sreg    v0,R_ERRPC*R_SZ(k0)
```

Notes

```
        sreg    v1,R_CAUSE*R_SZ(k0)
        mfc0    v0,C0_CONFIG
        mfc0    v1,C0_CACHEERR
        nop
        sreg    v0,R_CONFIG*R_SZ(k0)
        sreg    v1,R_CACHEERR*R_SZ(k0)
        mfc0    v0,C0_TAGLO

        nop
        sreg    v0,R_TAGLO*R_SZ(k0)
        mfc0    v1,C0_BADVADDR
        nop
        sreg    v1,R_BADVADDR*R_SZ(k0)
        .set    noat
        la      AT,client_regs   # get address of client reg save area
        sreg    sp,R_SP*R_SZ(k0)
        lw      v0,user_int_fast #see if a client wants a shot at it
        sreg    a0,R_A0*R_SZ(k0)
        sreg    ra,R_RA*R_SZ(k0)
        lw      sp,fault_stack   # use "fault" stack
        beq     v0,zero,1f       # skip the following if no client
        nop
        move    AT,k0            # pass frame pointer in AT, a0 & k0!!
        move    a0,k0
        jal     v0
        nop
        la      k0,except_regs
        la      AT,client_regs
        beq     v0,zero,1f       # returns false if user did not handle
        nop
        la      v1,except_regs
        lreg    ra,R_RA*R_SZ(v1)
        lreg    AT,R_AT*R_SZ(v1)
        lreg    gp,R_GP*R_SZ(v1)
        lreg    k0,R_EPC*R_SZ(v1)
        lreg    v0,R_V0*R_SZ(v1)
        lreg    sp,R_SP*R_SZ(v1)
#if defined(CPU_R4000)||defined(CPU_RC32364)
        rmtc0   k0,C0_EPC
#endif
        lreg    a0,R_A0*R_SZ(v1)
        lreg    v1,R_V1*R_SZ(v1)
#if defined(CPU_R3000)
        j       k0
        rfe
#endif
#if defined(CPU_R4000)||defined(CPU_RC32364)
        eret
        nop
#endif

/*
** Save registers if in client mode
** then change mode to prom mode currently k0 is pointing
** exception reg. save area - v0, v1, AT, gp, sp regs were saved
** epc, sr, badvaddr and cause were also saved.
*/
1:
        lreg    v0,R_MODE*R_SZ(AT)# get the current op. mode
        lreg    v1,R_EXCTYPE*R_SZ(k0)
        sreg    v0,R_MODE*R_SZ(k0)# save the current prom mode
        sreg    v1,R_EXCTYPE*R_SZ(AT)
#if defined(R4650_DIAG)
        .set noreorder
        lw      v1,IsInDiag
        nop
```

Notes

```
        beqz    v1,1f
        nop
        lw      v1,user_int_normal
        nop
        beqz    v1,1f
        nop
        nop
        nop
        j       2f
        nop
        .set reorder
    1:
#endif

        li      v1,MODE_MONITOR     # see if it was

        beq     v0,v1,nosave        # was in prom mode
        nop
#if defined(R4650_DIAG)
    2:
        la      k0,except_regs
        la      AT,client_regs
#endif
        lreg    v0,R_A0*R_SZ(k0)
        sreg    v1,R_MODE*R_SZ(AT)# now in prom mode
        sreg    v0,R_A0*R_SZ(AT)
        lreg    v0,R_BADVADDR*R_SZ(k0)
        lreg    v1,R_CAUSE*R_SZ(k0)
        sreg    v0,R_BADVADDR*R_SZ(AT)
        sreg    v1,R_CAUSE*R_SZ(AT)
        lreg    v0,R_GP*R_SZ(k0)
        lreg    v1,R_EPC*R_SZ(k0)
        sreg    v0,R_GP*R_SZ(AT)
        sreg    v1,R_EPC*R_SZ(AT)
        lreg    v0,R_SR*R_SZ(k0)
        lreg    v1,R_AT*R_SZ(k0)
        sreg    v0,R_SR*R_SZ(AT)
        sreg    v1,R_AT*R_SZ(AT)
        lreg    v0,R_V0*R_SZ(k0)
        lreg    v1,R_V1*R_SZ(k0)
        sreg    v0,R_V0*R_SZ(AT)
        sreg    v1,R_V1*R_SZ(AT)
        sreg    a1,R_A1*R_SZ(AT)
        sreg    a2,R_A2*R_SZ(AT)
        sreg    a3,R_A3*R_SZ(AT)
        sreg    t0,R_T0*R_SZ(AT)
        sreg    t1,R_T1*R_SZ(AT)
        sreg    t2,R_T2*R_SZ(AT)
        sreg    t3,R_T3*R_SZ(AT)
        sreg    t4,R_T4*R_SZ(AT)
        sreg    t5,R_T5*R_SZ(AT)
        sreg    t6,R_T6*R_SZ(AT)
        sreg    t7,R_T7*R_SZ(AT)
        sreg    s0,R_S0*R_SZ(AT)
        sreg    s1,R_S1*R_SZ(AT)
        sreg    s2,R_S2*R_SZ(AT)
        sreg    s3,R_S3*R_SZ(AT)
        sreg    s4,R_S4*R_SZ(AT)
        sreg    s5,R_S5*R_SZ(AT)
        sreg    s6,R_S6*R_SZ(AT)
        sreg    s7,R_S7*R_SZ(AT)
        sreg    t8,R_T8*R_SZ(AT)
        li      v0,0xbababadd  #This reg (k0) is invalid
        sreg    t9,R_T9*R_SZ(AT)
        sreg    v0,R_K0*R_SZ(AT)# should be obvious
        sreg    k1,R_K1*R_SZ(AT)
```

Notes

```
                sreg    fp,R_FP*R_SZ(AT)
                lreg    sp,R_SP*R_SZ(k0)
                lreg    ra,R_RA*R_SZ(k0)
                sreg    sp,R_SP*R_SZ(AT)
                sreg    ra,R_RA*R_SZ(AT)

                # iff ((status_base & client_regs.sr) & SR_CU1) savefpregs();
                lw      v0,status_base
                lreg    v1,R_SR*R_SZ(AT)
                li      t0,SR_CU1
                and     t1,v1,v0
                and     t1,t0
                beqz    t1,1f       # skip fpu regs if SR_CU1 not set
                nop

                cfc1    t0,$30
                cfc1    t1,$31
                sreg    t0,R_FEIR*R_SZ(AT)
                sreg    t1,R_FCSR*R_SZ(AT)
                ctc1    zero,$31            # clear exceptions
#if !defined(CPU_RC32364)
#if __mips >= 3
                /* always save the even registers */
                /*
                ** if __mips >= 3, then check for cputype
                ** if cpu is RC4650 then do single word store
                ** else do double word store
                */
                .set noreorder
                lw      a0,cputype
                andi    a0,a0,R4650
                bnez    a0,2f
                nop
                .set reorder

                sdc1    fp0,R_F0*R_SZ(AT)
                sdc1    fp2,R_F2*R_SZ(AT)
                sdc1    fp4,R_F4*R_SZ(AT)
                sdc1    fp6,R_F6*R_SZ(AT)
                sdc1    fp8,R_F8*R_SZ(AT)
                sdc1    fp10,R_F10*R_SZ(AT)
                sdc1    fp12,R_F12*R_SZ(AT)
                sdc1    fp14,R_F14*R_SZ(AT)
                sdc1    fp16,R_F16*R_SZ(AT)
                sdc1    fp18,R_F18*R_SZ(AT)
                sdc1    fp20,R_F20*R_SZ(AT)
                sdc1    fp22,R_F22*R_SZ(AT)
                sdc1    fp24,R_F24*R_SZ(AT)
                sdc1    fp26,R_F26*R_SZ(AT)
                sdc1    fp28,R_F28*R_SZ(AT)
                sdc1    fp30,R_F30*R_SZ(AT)

                li      t0,SR_FR
                and     t0,v1                       # is client_regs[sr].SR_FR set?
                beqz    t0,1f
                nop

                /* SR_FR was set: save the odd 16 registers too */
                sdc1    fp1,R_F1*R_SZ(AT)
                sdc1    fp3,R_F3*R_SZ(AT)
                sdc1    fp5,R_F5*R_SZ(AT)
                sdc1    fp7,R_F7*R_SZ(AT)
                sdc1    fp9,R_F9*R_SZ(AT)
                sdc1    fp11,R_F11*R_SZ(AT)
                sdc1    fp13,R_F13*R_SZ(AT)
                sdc1    fp15,R_F15*R_SZ(AT)
```

Notes

```
                sdc1    fp17,R_F17*R_SZ(AT)
                sdc1    fp19,R_F19*R_SZ(AT)
                sdc1    fp21,R_F21*R_SZ(AT)
                sdc1    fp23,R_F23*R_SZ(AT)
                sdc1    fp25,R_F25*R_SZ(AT)
                sdc1    fp27,R_F27*R_SZ(AT)
                sdc1    fp29,R_F29*R_SZ(AT)
                sdc1    fp31,R_F31*R_SZ(AT)
            j   1f
                nop
            2:
                /* always save the even registers */
                swc1    fp0,((R_F0*R_SZ)+4)(AT)
                swc1    fp2,((R_F2*R_SZ)+4)(AT)
                swc1    fp4,((R_F4*R_SZ)+4)(AT)
                swc1    fp6,((R_F6*R_SZ)+4)(AT)
                swc1    fp8,((R_F8*R_SZ)+4)(AT)
                swc1    fp10,((R_F10*R_SZ)+4)(AT)
                swc1    fp12,((R_F12*R_SZ)+4)(AT)
                swc1    fp14,((R_F14*R_SZ)+4)(AT)
                swc1    fp16,((R_F16*R_SZ)+4)(AT)
                swc1    fp18,((R_F18*R_SZ)+4)(AT)
                swc1    fp20,((R_F20*R_SZ)+4)(AT)
                swc1    fp22,((R_F22*R_SZ)+4)(AT)
                swc1    fp24,((R_F24*R_SZ)+4)(AT)
                swc1    fp26,((R_F26*R_SZ)+4)(AT)
                swc1    fp28,((R_F28*R_SZ)+4)(AT)
                swc1    fp30,((R_F30*R_SZ)+4)(AT)

                li      t0,SR_FR
                and     t0,v1                   # is client_regs[sr].SR_FR set?
                beqz    t0,1f
                nop

                /* SR_FR was set: save the odd 16 registers too */
                swc1    fp1,((R_F1*R_SZ)+4)(AT)
                swc1    fp3,((R_F3*R_SZ)+4)(AT)
                swc1    fp5,((R_F5*R_SZ)+4)(AT)
                swc1    fp7,((R_F7*R_SZ)+4)(AT)
                swc1    fp9,((R_F9*R_SZ)+4)(AT)
                swc1    fp11,((R_F11*R_SZ)+4)(AT)
                swc1    fp13,((R_F13*R_SZ)+4)(AT)
                swc1    fp15,((R_F15*R_SZ)+4)(AT)
                swc1    fp17,((R_F17*R_SZ)+4)(AT)
                swc1    fp19,((R_F19*R_SZ)+4)(AT)
                swc1    fp21,((R_F21*R_SZ)+4)(AT)
                swc1    fp23,((R_F23*R_SZ)+4)(AT)
                swc1    fp25,((R_F25*R_SZ)+4)(AT)
                swc1    fp27,((R_F27*R_SZ)+4)(AT)
                swc1    fp29,((R_F29*R_SZ)+4)(AT)
                swc1    fp31,((R_F31*R_SZ)+4)(AT)
        /* end */
        #else
                /* always save the even registers */
                swc1    fp0,R_F0*R_SZ(AT)
                swc1    fp2,R_F2*R_SZ(AT)
                swc1    fp4,R_F4*R_SZ(AT)
                swc1    fp6,R_F6*R_SZ(AT)
                swc1    fp8,R_F8*R_SZ(AT)
                swc1    fp10,R_F10*R_SZ(AT)
                swc1    fp12,R_F12*R_SZ(AT)
                swc1    fp14,R_F14*R_SZ(AT)
                swc1    fp16,R_F16*R_SZ(AT)
                swc1    fp18,R_F18*R_SZ(AT)
                swc1    fp20,R_F20*R_SZ(AT)
                swc1    fp22,R_F22*R_SZ(AT)
```

Notes

```
                swc1    fp24,R_F24*R_SZ(AT)
                swc1    fp26,R_F26*R_SZ(AT)
                swc1    fp28,R_F28*R_SZ(AT)
                swc1    fp30,R_F30*R_SZ(AT)


                lui t0,(SR_FR & 0xffff0000)
                ori t0,$zero,(SR_FR & 0x0000ffff)

                and     t0,v1                   # is client_regs[sr].SR_FR set?
                beqz    t0,1f
                nop

                /* SR_FR was set: save the odd 16 registers too */
                swc1    fp1,R_F1*R_SZ(AT)
                swc1    fp3,R_F3*R_SZ(AT)
                swc1    fp5,R_F5*R_SZ(AT)
                swc1    fp7,R_F7*R_SZ(AT)
                swc1    fp9,R_F9*R_SZ(AT)
                swc1    fp11,R_F11*R_SZ(AT)
                swc1    fp13,R_F13*R_SZ(AT)
                swc1    fp15,R_F15*R_SZ(AT)
                swc1    fp17,R_F17*R_SZ(AT)
                swc1    fp19,R_F19*R_SZ(AT)
                swc1    fp21,R_F21*R_SZ(AT)
                swc1    fp23,R_F23*R_SZ(AT)
                swc1    fp25,R_F25*R_SZ(AT)
                swc1    fp27,R_F27*R_SZ(AT)
                swc1    fp29,R_F29*R_SZ(AT)
                swc1    fp31,R_F31*R_SZ(AT)
        #endif

        #endif
        1:      mflo    v0
                mfhi    v1
                sreg    v0,R_MDLO*R_SZ(AT)
                sreg    v1,R_MDHI*R_SZ(AT)

                mfc0    v0,C0_INX
                mfc0    v1,C0_RAND
                mfc0    t0,C0_PRID
                sreg    v0,R_INX*R_SZ(AT)
                sreg    v1,R_RAND*R_SZ(AT)
                sreg    t0,R_PRID*R_SZ(AT)
        #if defined(CPU_R3000)
                mfc0    v0,C0_TLBLO
                mfc0    v1,C0_TLBHI
                sreg    v0,R_TLBLO*R_SZ(AT)
                mfc0    v0,C0_CTXT
                sreg    v1,R_TLBHI*R_SZ(AT)
                sreg    v0,R_CTXT*R_SZ(AT)
        #endif
        #if defined(CPU_R4000) || defined(CPU_RC32364)
                mfc0    v0,C0_TLBLO0
                mfc0    v1,C0_TLBLO1
                rmfc0   t0,C0_TLBHI
                sreg    v1,R_TLBLO1*R_SZ(AT)
                sreg    v0,R_TLBLO0*R_SZ(AT)
                sreg    t0,R_TLBHI*R_SZ(AT)

                rmfc0   v0,C0_CTXT
                mfc0    v1,C0_PAGEMASK
                mfc0    t0,C0_WIRED
                sreg    v0,R_CTXT*R_SZ(AT)
                sreg    v1,R_PAGEMASK*R_SZ(AT)
                sreg    t0,R_WIRED*R_SZ(AT)
```

Notes

```
        mfc0    v0,C0_COUNT
        mfc0    v1,C0_COMPARE
        mfc0    t0,C0_CONFIG
        sreg    v0,R_COUNT*R_SZ(AT)
        sreg    v1,R_COMPARE*R_SZ(AT)
        sreg    t0,R_CONFIG*R_SZ(AT)

#if !defined(CPU_RC32364)
        mfc0    v0,C0_WATCHLO
        mfc0    v1,C0_WATCHHI
        mfc0    t0,C0_LLADDR
        sreg    v0,R_WATCHLO*R_SZ(AT)
        sreg    v1,R_WATCHHI*R_SZ(AT)
        sreg    t0,R_LLADDR*R_SZ(AT)
#endif
#if defined(CPU_RC32364)
        mfc0    v0,C0_IWATCH
        mfc0    v1,C0_DWATCH
        sreg    v0,R_IWATCH*R_SZ(AT)
        sreg    v1,R_DWATCH*R_SZ(AT)
#endif
        mfc0    v0,C0_ECC
        mfc0    v1,C0_CACHEERR
        rmfc0   t0,C0_ERRPC
        sreg    v0,R_ECC*R_SZ(AT)
        sreg    v1,R_CACHEERR*R_SZ(AT)
        sreg    t0,R_ERRPC*R_SZ(AT)

        mfc0    v0,C0_TAGLO
#if !defined(CPU_RC32364)
        mfc0    v1,C0_TAGHI
        rmfc0   t0,C0_XCTXT
#endif
        sreg    v0,R_TAGLO*R_SZ(AT)
#if !defined(CPU_RC32364)
        sreg    v1,R_TAGHI*R_SZ(AT)
        sreg    t0,R_XCTXT*R_SZ(AT)
#endif
#endif

nosave:
        # set monitors status reg value
        lw      v0,status_base
        nop
        mtc0    v0,C0_SR
        nop

        .set    at
        .set    reorder
#if defined(R4650_DIAG)
        .set    noreorder
/*
** check for any additional exception handlers
**  used in diagnostics for RC4650
*/
        lw      a0,IsInDiag
        beqz    a0,1f
        nop
        lw      v0,user_int_normal
        nop
        beqz    v0,1f
        nop
        jalr    v0
        nop
        jal     resume
```

Notes

```
        nop
        jal     promexit
        nop
1:
        .set    reorder
#endif
        j       exception_handler

        ENDFRAME(_exception)



/*
** resume -- resume execution of client code
**
** Note that some portions of this code are specific to RC4xxx and some to
** RC30xx. Please follow the #ifdef and #else carefully!!!
*/
FRAME(resume,sp,0,ra)
        jal     install_sticky
        jal     clr_extern_brk
        jal     clear_remote_int
        .set    noat
        .set    noreorder
        la      AT,client_regs

        lw      v0,status_base
        lreg    v1,R_SR*R_SZ(AT)
#if defined (CPU_R4000)
        li      t1,SR_FR        # add SR_FR to allowable bits
        li      t2,NOT_SR_IEC   # but disable SR_IE (int enable)
        or      v0,t1
        and     v0,t2
#else
        li  t2,NOT_SR_IEC       # disable SR_IEC (int enable)
        and     v0,t2
#endif
        and     v0,v1
        mtc0    v0,C0_SR        # set $status to computed value
        sll     v1,v0,2                 # XXX shift SR_CU1 to top bit
        bgez    v1,1f                   # skip fpu regs if clear
        nop

        /* When setting the f.p status we cancel any pending f.p.
           exceptions, to avoid a nested trap.  If the user
           has dealt with the failing instruction, then we
           don't want to see them again.  If they haven't
           then the instruction will fail again anyway. */
#if !defined( CPU_RC32364)
        lreg    v1,R_FCSR*R_SZ(AT)# get saved f.p status
        li      t0,~0x3f000     # remove exceptions
        and     v1,t0
        ctc1    v1,$31                  # set f.p status

#if (__mips >= 3)
        /* always save the even registers */
        /*
        ** if __mips >= 3, then check for cputype
        ** if cpu is RC4650 then do single word store
        ** else do double word store
        */
        .set noreorder
        lw      a0,cputype
        andi    a0,a0,R4650
        bnez    a0,2f
```

Notes

```
        nop
        .set reorder

        /* always restore even registers */
        ldc1    fp0,R_F0*R_SZ(AT)
        ldc1    fp2,R_F2*R_SZ(AT)
        ldc1    fp4,R_F4*R_SZ(AT)
        ldc1    fp6,R_F6*R_SZ(AT)
        ldc1    fp8,R_F8*R_SZ(AT)
        ldc1    fp10,R_F10*R_SZ(AT)
        ldc1    fp12,R_F12*R_SZ(AT)
        ldc1    fp14,R_F14*R_SZ(AT)
        ldc1    fp16,R_F16*R_SZ(AT)
        ldc1    fp18,R_F18*R_SZ(AT)
        ldc1    fp20,R_F20*R_SZ(AT)
        ldc1    fp22,R_F22*R_SZ(AT)
        ldc1    fp24,R_F24*R_SZ(AT)
        ldc1    fp26,R_F26*R_SZ(AT)
        ldc1    fp28,R_F28*R_SZ(AT)
        ldc1    fp30,R_F30*R_SZ(AT)

        and     t1,v0                   # is SR_FR set?
        beqz    t1,1f
        nop

        /* SR_FR is set: restore the odd 16 registers too */
        ldc1    fp1,R_F1*R_SZ(AT)
        ldc1    fp3,R_F3*R_SZ(AT)
        ldc1    fp5,R_F5*R_SZ(AT)
        ldc1    fp7,R_F7*R_SZ(AT)
        ldc1    fp9,R_F9*R_SZ(AT)
        ldc1    fp11,R_F11*R_SZ(AT)
        ldc1    fp13,R_F13*R_SZ(AT)
        ldc1    fp15,R_F15*R_SZ(AT)
        ldc1    fp17,R_F17*R_SZ(AT)
        ldc1    fp19,R_F19*R_SZ(AT)
        ldc1    fp21,R_F21*R_SZ(AT)
        ldc1    fp23,R_F23*R_SZ(AT)
        ldc1    fp25,R_F25*R_SZ(AT)
        ldc1    fp27,R_F27*R_SZ(AT)
        ldc1    fp29,R_F29*R_SZ(AT)
        ldc1    fp31,R_F31*R_SZ(AT)

        j       1f
        nop
2:
        /* always restore even registers */
        lwc1    fp0,((R_F0*R_SZ)+4)(AT)
        lwc1    fp2,((R_F2*R_SZ)+4)(AT)
        lwc1    fp4,((R_F4*R_SZ)+4)(AT)
        lwc1    fp6,((R_F6*R_SZ)+4)(AT)
        lwc1    fp8,((R_F8*R_SZ)+4)(AT)
        lwc1    fp10,((R_F10*R_SZ)+4)(AT)
        lwc1    fp12,((R_F12*R_SZ)+4)(AT)
        lwc1    fp14,((R_F14*R_SZ)+4)(AT)
        lwc1    fp16,((R_F16*R_SZ)+4)(AT)
        lwc1    fp18,((R_F18*R_SZ)+4)(AT)
        lwc1    fp20,((R_F20*R_SZ)+4)(AT)
        lwc1    fp22,((R_F22*R_SZ)+4)(AT)
        lwc1    fp24,((R_F24*R_SZ)+4)(AT)
        lwc1    fp26,((R_F26*R_SZ)+4)(AT)
        lwc1    fp28,((R_F28*R_SZ)+4)(AT)
        lwc1    fp30,((R_F30*R_SZ)+4)(AT)

        and     t1,v0                   # is SR_FR set?
        beqz    t1,1f
```

Notes

```
        nop

        /* SR_FR is set: restore the odd 16 registers too */
        lwc1    fp1,((R_F1*R_SZ)+4)(AT)
        lwc1    fp3,((R_F3*R_SZ)+4)(AT)
        lwc1    fp5,((R_F5*R_SZ)+4)(AT)
        lwc1    fp7,((R_F7*R_SZ)+4)(AT)
        lwc1    fp9,((R_F9*R_SZ)+4)(AT)
        lwc1    fp11,((R_F11*R_SZ)+4)(AT)
        lwc1    fp13,((R_F13*R_SZ)+4)(AT)
        lwc1    fp15,((R_F15*R_SZ)+4)(AT)
        lwc1    fp17,((R_F17*R_SZ)+4)(AT)
        lwc1    fp19,((R_F19*R_SZ)+4)(AT)
        lwc1    fp21,((R_F21*R_SZ)+4)(AT)
        lwc1    fp23,((R_F23*R_SZ)+4)(AT)
        lwc1    fp25,((R_F25*R_SZ)+4)(AT)
        lwc1    fp27,((R_F27*R_SZ)+4)(AT)
        lwc1    fp29,((R_F29*R_SZ)+4)(AT)
        lwc1    fp31,((R_F31*R_SZ)+4)(AT)
#else
        /* always restore even registers */
        lwc1    fp0,R_F0*R_SZ(AT)
        lwc1    fp2,R_F2*R_SZ(AT)
        lwc1    fp4,R_F4*R_SZ(AT)
        lwc1    fp6,R_F6*R_SZ(AT)
        lwc1    fp8,R_F8*R_SZ(AT)
        lwc1    fp10,R_F10*R_SZ(AT)
        lwc1    fp12,R_F12*R_SZ(AT)
        lwc1    fp14,R_F14*R_SZ(AT)
        lwc1    fp16,R_F16*R_SZ(AT)
        lwc1    fp18,R_F18*R_SZ(AT)
        lwc1    fp20,R_F20*R_SZ(AT)
        lwc1    fp22,R_F22*R_SZ(AT)
        lwc1    fp24,R_F24*R_SZ(AT)
        lwc1    fp26,R_F26*R_SZ(AT)
        lwc1    fp28,R_F28*R_SZ(AT)
        lwc1    fp30,R_F30*R_SZ(AT)

        and     t1,v0                   # is SR_FR set?
        beqz    t1,1f
        nop

        /* SR_FR is set: restore the odd 16 registers too */
        lwc1    fp1,R_F1*R_SZ(AT)
        lwc1    fp3,R_F3*R_SZ(AT)
        lwc1    fp5,R_F5*R_SZ(AT)
        lwc1    fp7,R_F7*R_SZ(AT)
        lwc1    fp9,R_F9*R_SZ(AT)
        lwc1    fp11,R_F11*R_SZ(AT)
        lwc1    fp13,R_F13*R_SZ(AT)
        lwc1    fp15,R_F15*R_SZ(AT)
        lwc1    fp17,R_F17*R_SZ(AT)
        lwc1    fp19,R_F19*R_SZ(AT)
        lwc1    fp21,R_F21*R_SZ(AT)
        lwc1    fp23,R_F23*R_SZ(AT)
        lwc1    fp25,R_F25*R_SZ(AT)
        lwc1    fp27,R_F27*R_SZ(AT)
        lwc1    fp29,R_F29*R_SZ(AT)
        lwc1    fp31,R_F31*R_SZ(AT)
#endif

#endif /* !defined( CPU_RC32364)*/
1:
        lreg    a0,R_A0*R_SZ(AT)
        lreg    a1,R_A1*R_SZ(AT)
        lreg    a2,R_A2*R_SZ(AT)
```

Notes

```
        lreg    a3,R_A3*R_SZ(AT)
        lreg    t0,R_T0*R_SZ(AT)
        lreg    t1,R_T1*R_SZ(AT)
        lreg    t2,R_T2*R_SZ(AT)
        lreg    t3,R_T3*R_SZ(AT)
        lreg    t4,R_T4*R_SZ(AT)
        lreg    t5,R_T5*R_SZ(AT)
        lreg    t6,R_T6*R_SZ(AT)
        lreg    t7,R_T7*R_SZ(AT)
        lreg    s0,R_S0*R_SZ(AT)
        lreg    s1,R_S1*R_SZ(AT)
        lreg    s2,R_S2*R_SZ(AT)
        lreg    s3,R_S3*R_SZ(AT)
        lreg    s4,R_S4*R_SZ(AT)
        lreg    s5,R_S5*R_SZ(AT)
        lreg    s6,R_S6*R_SZ(AT)
        lreg    s7,R_S7*R_SZ(AT)
        lreg    t8,R_T8*R_SZ(AT)
        lreg    t9,R_T9*R_SZ(AT)
        lreg    k1,R_K1*R_SZ(AT)
        lreg    gp,R_GP*R_SZ(AT)
        lreg    fp,R_FP*R_SZ(AT)
        lreg    ra,R_RA*R_SZ(AT)
        lreg    v0,R_MDLO*R_SZ(AT)
        lreg    v1,R_MDHI*R_SZ(AT)
        mtlo    v0
        mthi    v1

#if defined(CPU_R3000)
        lreg    v0,R_INX*R_SZ(AT)
        lreg    v1,R_TLBLO*R_SZ(AT)
        mtc0    v0,C0_INX
#ifdef CPU_RC32364
    nop     /* give some pre-recovery time */
    # nop
#endif
        mtc0    v1,C0_TLBLO
#ifdef CPU_RC32364
    nop     /* give some recovery time */
    nop
#endif
        lreg    v0,R_TLBHI*R_SZ(AT)
        lreg    v1,R_CTXT*R_SZ(AT)
#ifdef CPU_RC32364
    nop     /* give some pre-recovery time */
    nop
#endif
        rmtc0   v0,C0_TLBHI
#ifdef CPU_RC32364
    nop     /* give some recovery time */
    # nop
#endif
        rmtc0   v1,C0_CTXT
#ifdef CPU_RC32364
    lw      v0,R_CONFIG*4(AT)
    lw      v1,R_COUNT*4(AT)
    mtc0    v0,C0_CONFIG
    mtc0    v1,C0_COUNT
    lw      v0,R_COMPARE*4(AT)
    nop
    mtc0    v0,C0_COMPARE
#else
    lw      v0,R_CONFIG*4(AT)
    nop
    mtc0    v0,C0_CONFIG
    nop
```

Notes

```
#endif
#endif
#if defined(CPU_R4000)||defined( CPU_RC32364)
        lreg    v0,R_TLBLO0*R_SZ(AT)
        lreg    v1,R_TLBLO1*R_SZ(AT)
        mtc0    v0,C0_TLBLO0
        mtc0    v1,C0_TLBLO1

        lreg    v0,R_CTXT*R_SZ(AT)
        lreg    v1,R_TLBHI*R_SZ(AT)
        rmtc0   v0,C0_CTXT
        rmtc0   v1,C0_TLBHI

        lreg    v0,R_PAGEMASK*R_SZ(AT)
        lreg    v1,R_WIRED*R_SZ(AT)
        mtc0    v0,C0_PAGEMASK
        mtc0    v1,C0_WIRED

        lreg    v0,R_INX*R_SZ(AT)
#if !defined( CPU_RC32364)
        lreg    v1,R_LLADDR*R_SZ(AT)
#endif
        mtc0    v0,C0_INX
#if !defined( CPU_RC32364)
        mtc0    v1,C0_LLADDR
#endif

#if !defined( CPU_RC32364)
        lreg    v0,R_WATCHLO*R_SZ(AT)
        lreg    v1,R_WATCHHI*R_SZ(AT)
        mtc0    v0,C0_WATCHLO
        mtc0    v1,C0_WATCHHI
#else
        lreg    v0,R_IWATCH*R_SZ(AT)
        lreg    v1,R_DWATCH*R_SZ(AT)
    mtc0   v0,C0_IWATCH
    mtc0   v1,C0_DWATCH
#endif

        lreg    v0,R_ECC*R_SZ(AT)
        lreg    v1,R_CACHEERR*R_SZ(AT)
        mtc0    v0,C0_ECC
        mtc0    v1,C0_CACHEERR

        lreg    v0,R_TAGLO*R_SZ(AT)
#if !defined( CPU_RC32364)
        lreg    v1,R_TAGHI*R_SZ(AT)
#endif
        mtc0    v0,C0_TAGLO
#if !defined( CPU_RC32364)
        mtc0    v1,C0_TAGHI
#endif
        lreg    v0,R_ERRPC*R_SZ(AT)
        nop
        rmtc0   v0,C0_ERRPC
#endif /* CPU_RC4000 */

        lreg    v0,R_CAUSE*R_SZ(AT)
        lreg    v1,R_SR*R_SZ(AT)
        mtc0    v0,C0_CAUSE             /* only sw0 and 1 writable  */
#if defined(CPU_R3000)
        li      v0,~(SR_KUC|SR_IEC|SR_PE)/* make sure we aren't intr */
        and     v1,v0
#endif
#if defined(CPU_R4000)||defined( CPU_RC32364)
        or      v1,SR_EXL               /* make sure exception bit is set */
```

Notes

```
#endif
        mtc0    v1,C0_SR

        li      k0,MODE_USER
        sreg    k0,R_MODE*R_SZ(AT)    /* reset  mode */
        lreg    k0,R_EPC*R_SZ(AT)
        lreg    v1,R_V1*R_SZ(AT)
        lreg    sp,R_SP*R_SZ(AT)
#if defined(CPU_R4000)||defined( CPU_RC32364)
        rmtc0   k0,C0_EPC
#endif
        lreg    v0,R_V0*R_SZ(AT)
        lreg    AT,R_AT*R_SZ(AT)
#if defined(CPU_R3000)
        j       k0
        rfe
#endif
#if defined(CPU_R4000)||defined( CPU_RC32364)
        eret
#endif
        .set    reorder
        .set    at
        ENDFRAME(resume)

/*
** clear_stat() -- clear status register
** returns current sr
*/
FRAME(clear_stat,sp,0,ra)
        .set    noreorder
        lw      v1,status_base
        mfc0    v0,C0_SR
        mtc0    v1,C0_SR
#if defined (CPU_R4000) || defined( CPU_RC32364)
        nop
#endif
        j       ra
        nop
        ENDFRAME(clear_stat)

        .set    reorder

/*
** setjmp(jmp_buf) -- save current context for non-local goto's
** return 0
*/
FRAME(setjmp32,sp,0,ra)
#if (__mips < 3)
.globl setjmp
setjmp:
#endif
        sw      ra,JB_PC*4(a0)
        sw      sp,JB_SP*4(a0)
        sw      fp,JB_FP*4(a0)
        sw      s0,JB_S0*4(a0)
        sw      s1,JB_S1*4(a0)
        sw      s2,JB_S2*4(a0)
        sw      s3,JB_S3*4(a0)
        sw      s4,JB_S4*4(a0)
        sw      s5,JB_S5*4(a0)
        sw      s6,JB_S6*4(a0)
        sw      s7,JB_S7*4(a0)
        move    v0,zero
        j       ra
        ENDFRAME(setjmp32)
```

Notes

```
/*
** longjmp(jmp_buf, rval)
*/
FRAME(longjmp32,sp,0,ra)
#if (__mips < 3)
.globl longjmp
longjmp:
#endif
        lw      ra,JB_PC*4(a0)
        lw      sp,JB_SP*4(a0)
        lw      fp,JB_FP*4(a0)
        lw      s0,JB_S0*4(a0)
        lw      s1,JB_S1*4(a0)
        lw      s2,JB_S2*4(a0)
        lw      s3,JB_S3*4(a0)
        lw      s4,JB_S4*4(a0)
        lw      s5,JB_S5*4(a0)
        lw      s6,JB_S6*4(a0)
        lw      s7,JB_S7*4(a0)
        move    v0,a1
        j       ra
        ENDFRAME(longjmp32)

#if (__mips >= 3)
FRAME(setjmp64,sp,0,ra)
.globl  setjmp
setjmp:
        sd      ra,JB_PC*8(a0)
        sd      sp,JB_SP*8(a0)
        sd      fp,JB_FP*8(a0)
        sd      s0,JB_S0*8(a0)
        sd      s1,JB_S1*8(a0)
        sd      s2,JB_S2*8(a0)
        sd      s3,JB_S3*8(a0)
        sd      s4,JB_S4*8(a0)
        sd      s5,JB_S5*8(a0)
        sd      s6,JB_S6*8(a0)
        sd      s7,JB_S7*8(a0)
        move    v0,zero
        j       ra
        ENDFRAME(setjmp64)

FRAME(longjmp64,sp,0,ra)
.globl  longjmp
longjmp:
        ld      ra,JB_PC*8(a0)
        ld      sp,JB_SP*8(a0)
        ld      fp,JB_FP*8(a0)
        ld      s0,JB_S0*8(a0)
        ld      s1,JB_S1*8(a0)
        ld      s2,JB_S2*8(a0)
        ld      s3,JB_S3*8(a0)
        ld      s4,JB_S4*8(a0)
        ld      s5,JB_S5*8(a0)
        ld      s6,JB_S6*8(a0)
        ld      s7,JB_S7*8(a0)
        move    v0,a1
        j       ra
        ENDFRAME(longjmp64)
#endif
```

Notes

Interrupts

The MIPS CPUs are provided with 6 individual hardware interrupt bits, activated by CPU input pins. In the case of the RC3081, one pin is used internally by the FPA; in the RC4xxx, one pin is used internally by timer interrupt and 2 internal software-controlled interrupt bits. An active level is sensed in each cycle and will cause an exception if enabled.

The interrupt enable occurs in the following two parts:

- *The global interrupt enable bit in the status register – when set to zero, no interrupt exception will occur. The global interrupt enable is usually switched back on by an rfe instruction at the end of an exception routine in the RC30xx (different from RC4xxx/RC32364 behavior); this means that the interrupt cannot take effect until the CPU has returned from the exception and finished with the EPC register, avoiding undesirable recursion in the interrupt routine. In the RC4xxx, while returning from an exception routine (eret instruction), the appropriate ERL/EXL bit in the Status register will be cleared by the processor, depending on whether the exception was an error trap (load PC from ErrorPC) or not (load PC from EPC). The global interrupt enable bit (IE) in the Status register of the RC4xxx/RC32364 must be managed by software (disable just before using eret, etc.), especially if handling nested exceptions or interrupts.*
- *The individual interrupt mask bits, IM, in the status register, one for each interrupt. Setting the bit to 1enables the corresponding interrupt. These are software manipulated to allow system appropriate interrupts. Changes to the individual bits are usually made "under cover," with the global interrupt enable off.*

Software Interrupts

This is used as a mechanism for high-priority interrupt routines to flag actions which will be performed by lower-priority interrupt routines, once the system has dealt with all high priority business. As the high-priority processing completes, the software will open up the interrupt mask, and the pending software interrupt will occur.

Pin	SR/Cause Bit Number	Notes
—	8	Software interrupt
—	9	Software interrupt
Int0*	10	
Int1*	11	
Int2*	12	
Int3*	13	Usual choice for FPA in the RC30xx. The pin for the interrupt selected for FPA interrupts on an RC3081 is effectively a no-connect.
Int4*	14	
Int5*	15	Used for timer in RC4xxx. See *Compare* register in Chapter 3.

Table 4.3 Interrupt Bitfields and Interrupt Pins

Interrupt processing proper begins after an exception is received and the Type field in *Cause* signals that it was caused by an interrupt. Interrupt Bitfields and Interrupt Pins describes the relationship between *Cause* bits and input pins.

Once the interrupt exception is "recognized" by the CPU, the stages are:

- *Consult the Cause register IP field, logically-"and" it with the current interrupt masks in the SR IM field to obtain a bit-map of active, enabled interrupt requests. There may be more than one, and any of them would have caused the interrupt.*
- *Select one active, enabled interrupt for attention. The selection can be done simply by using fixed priorities; however, software is free to implement whatever priority mechanism is appropriate for the system.*

Notes

- *Software needs to save the old interrupt mask bits of the SR register, but it is quite likely that the whole SR register was saved in the main exception routine.*
- *Change IM in SR to ensure that the current interrupt and all interrupts of equal or lesser priority are inhibited.*
- *If not already performed by the main exception routine, save the state required for nested exception processing.*
- *Set the global interrupt enable bit IEc (RC30xx) or IE (RC4xxx/RC32364) in SR to allow higher-priority interrupts to be processed.*
- *Call the particular interrupt service routine for the selected, current interrupt.*
- *On return, disable interrupts again by clearing IEc/IE in SR, before returning to the normal exception stream.*

Notes

Cache Management

Notes

Caches and Cache Management

IDT CPUs implement separate on-chip caches for instructions (I-cache) and data (D-cache). In general, hardware functions are provided only for normal operation of the caches; software routines must initialize the cache following system start-up, and to invalidate cache data when required[1].

The RC4xxx/RC5000/RC32364 provide a *cache* instruction which allows direct operations on cache blocks. As a result, the cache management techniques employed for RC30xx tend to be significantly different from those for RC4xxx/RC5000/RC32364. The RC32364 and RC4650 also provide a cache-locking feature; the RC36100 facilitates cache locking, but uses a different mechanism. Low level software, to be portable across RC30xx and RC4xxx/RC5000/RC32364, must be written with great care and by making use of conditional compiles or CPU identification and differentiation techniques.

Cache holds a copy of memory data that has recently been read or written, so it can be returned quickly to the CPU. In the MIPS architecture accesses in the cache take just one clock, and an I-cache and a D-cache access can occur in parallel.

When a cacheable location is read (a data load):

- *It will be returned from the D-cache, if the cache contains the corresponding physical address and the cache line is valid there (called a cache "hit").*
- *If the data is not found in the D-cache (cache miss), the data is read from external memory. According to the CPU type and how it is set up, it may read one or more words from memory. The data is loaded into the cache and normal operation resumes.*

  In normal operation, cache miss processing will cause the targeted cache line to "invalidate" the valid data already present in the cache. In the RC30xx caches, cache data is never more up-to-date than memory (because the cache is *write-through*, described below), so the previously cached data can be discarded without any trouble.

  However, in the RC4xxx and RC32364, two different modes of data cache writing are available, according to the configuration selected for the virtual address. One of the modes, the write-through mode, is similar to that of the RC30xx. The other, called the *write-back* mode, does not write the data from data cache to memory until the cache line is replaced. The fact that data in a particular cache line is valid but not written back to the memory is indicated by a bit (w-bit) in the cache line.

When data is loaded from an uncacheable location, it is always obtained from external memory. On an uncacheable load, cache data is neither used nor modified.

When software writes a cached location:

- *If the CPU is doing a 32-bit store in a RC3xxx or a 64-bit store in a RC4xxx, the cache is always updated (possibly discarding data from a previously cached location).*
- *For partial stores, the cache will only be updated if the reference hits in the cache; then data will be extracted from the cache, merged with the store data, and written back.*
- *If the partial-word store misses in the cache, then the cache is left alone.*
- *In all cases in the RC3xxx and the RC4xxx write-through mode, the write is also made to main memory. In the RC4xxx/RC5000/RC32364 write-back mode, of course, the write to main memory is postponed until the time of replacement of the cache line during a cache read, or until the user program forces a write using the cache instruction.*

When the store target is an uncached location the cache is not consulted or modified.

---

1. Note that the RC3071 and RC3081 do implement a DMA protocol that allows automatic, hardware-based data cache invalidation.

Notes

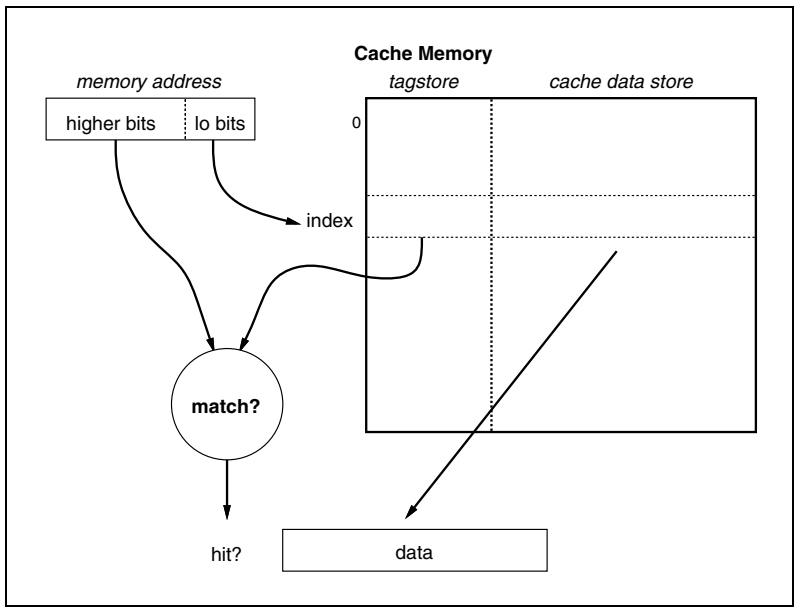


Figure 1.1 Direct Mapped Cache

RC30xx Cache Characteristics

Both caches in the RC30xx are:

- *Physically indexed, physically tagged: the RC30xx CPUs program address (virtual address) is translated to a physical address, just as is used to address real memory, before being used for the cache lookup. The TAG comparison (checking for a hit) is also based on physical addresses.*
- *Direct mapped: Each physical address has only one location in each RC30xx cache where it may reside. At each cache index there is only one data item stored – this will be just one word in the D-cache but is usually a 4-word line for the I-cache (see Figure 1.1). The tag is kept next to the data, which stores the memory address for which this data is a copy.*

  If the tag matches the high-order (higher number) address bits then the cache line contains the data the CPU is looking for; the data is returned and execution continues.

  This is a *direct mapped* cache because there is only one tag/data pair at each cache index. More complex caches may have more than one tag field, and compare them simultaneously with the physical address.

  A direct-mapped cache is simple, but can suffer from cache thrashing; the CPU will run slowly if a program loop is regularly accessing a pair of locations whose low-order addresses are equal. To avoid this situation, the RC30xx family implements relatively large caches and minimizes the probability of reasonable program loops causing thrashing.
- *Cache lines: the line size is the number of data elements stored with each tag. For RC30xx family CPUs the I-cache implements a 4-word line size; the D-cache always has 1-word lines.*

  When a cache miss occurs the whole line must be filled from memory. But it is quite possible to fetch more than a line's worth of data; and RC30xx family CPUs can be configured to fetch 4 words of data on a D-cache miss, refilling 4 1-word "lines".
- *Write through: the D-cache is write-through, meaning that all store operations result in a store to main memory. This means that all data in the cache is duplicated in main memory, and can therefore be discarded at any time. In particular, when data is being read following a cache miss it can always be stored in the cache without regard for the data which was previously stored at the same index.*
- *Partial word write implementations: when the CPU writes only part of a word, it is essential that any valid cache data should still end up as a duplicate of main memory. One simple approach is to invalidate the cache line and to write only to main memory (the main memory must be byte-address-*

Notes

able). But the RC30xx family uses a more efficient strategy:

a) if the location being written is present in the cache (cache hit) the cache data is read into the CPU, the partial-word data merged with it, the whole word written back to the cache, and the partial-word written to memory.

b) where the write misses in the cache the partial-word write is performed to memory only, and the cache left alone.

Note that this takes an extra clock, so a partial-word write (regardless of whether it hits in the cache) is slower than a whole-word write.

Cache Locking

A portion of a cache is said to be locked when a particular piece of code or data is loaded into a cache location that will not be selected later for refill by other data.

When To Use Cache Locking

Cache locking is useful in the following cases:

- *a portion of code must reside in cache permanently (for example, time-critical exception vectors) for real-time performance*
- *a given section of code is executed frequently and can fit inside a portion of the instruction cache*
- *a given section of data is accessed frequently and can fit inside the data cache (for example, tables containing routing information in an embedded network application)*

Cache Locking in RC32364

The locking feature of the RC32364 is on a per-line basis; that is, the kernel may set status register control bits that allow individual cache lines to be locked in the cache. Locked cache lines can be changed by any of the following operations or conditions:

- *cache operations*
- *store operations to cached virtual address*
- *if they become valid*

In the RC32364, both the Instruction and Data cache are two-way set associative, with set A and set B. By setting the DL or IL bit in the Status register of CP0, a refilled cache line of a selected set, at that time, can be locked in the appropriate cache; therefore, a future fill into this cache line will always use the other set. Furthermore, if one set of a cache line has already been locked, the second attempt to lock this cache line will be ignored.

A Data store operation to locked data will update the D-cache contents; locking merely prevents the cache line contents from being replaced by the contents of a different physical location. The locked cache line can be unlocked by using a Cache operation to invalidate that line. Anytime the valid bit of a cache line is cleared, the lock bit is cleared simultaneously. The basic algorithm consists of the following three steps.

1. Set the appropriate cache-lock enable bit(s)
2. Load the critical code/data into the cache(s)
3. Clear the appropriate cache lock enable bit(s)

Example: Data Cache Locking in RC32364

Notes

For this example, assume an application in which a table must be kept in cache. After completing the initialization of data structures, etc., in the start-up code, the DL bit in the Status register can be set to enable the cache line locking, perform reads through cached addresses to load the data into the data cache, and then—to prevent further cache locking—clear the DL bit. A sample code fragment for the Data Cache Locking operation follows:

```
.set noreorder
jal flush_cache /* Flush the cache */
mfc0 a0, C0_SR /* Get old SR value */
li a1, SR_SET_DL /* SR_SET_DL = 0x00100000 */
or a0, a0, a1
mtc0 a0, CO_SR /* Set the Lock bit for data cache */
nop
nop
nop /* 3 nops: safety against CP0 hazard */
la t0, critical_table /* This table should always be in cache */
li t1, table_size /* Size of table in bytes */
li t2, 0 /* Number of bytes read into cache */
1: lw a0, 0(t0)
addiu t2, 4
bneq t2, t1, 1b /* Loop back till done */
addiu t0, 4 /* bump read address */
mfc0 a0, C0_SR /* Get old SR value */
li a1, SR_CLR_DL /* SR_CLR_DL = 0xffefffff */
and a0, a0, a1
mtc0 a0, C0_SR /* Clear the Lock bit for data cache */
nop
nop
nop /* 3 nops: safety against CP0 hazard */
```

Example: Instruction Cache Locking in RC32364

For this example, assume an application in which a critical function must be kept in cache. Also assume that the size of the function is known. (If not known, the size can be determined by generating a disassembly of the object file.)

After completing the initialization of data structures, etc., in the start-up code, the IL bit of the Status register can be set to enable cache line locking, perform the FILL operation in the CACHE instruction that will fill the instruction cache with the critical function, and then—to prevent further cache locking—clear the IL bit. A sample code fragment for the Instruction Cache Locking operation follows:

```
.set noreorder
jal flush_cache /* Flush the cache */
la t0, 1f /* Get address of label '1' */
li t1, 0xA0000000
or t0, t0, t1
jr t0 /* Uncached execution from now onwards */
nop
1: la t0, func_start_addr /* Start address of critical code */
li t1, func_size /* Critical code size */
li t2, 0 /* Number of words read into cache */
mfc0 a0, C0_SR /* Get old SR value */
li a1, SR_SET_IL /* SR_SET_IL = 0x00080000 */
or a0, a0, a1
mtc0 a0, C0_SR /* Set Lock bit for instruction cache */
nop
nop
nop
2: cache Fill_I, 0(t0) /* Fill Operation */
addiu t2, 4
bneq t2, t1, 2b /* Loop back till done */
addiu t0, 4 /* bump read address */
mfc0 a0, C0_SR /* Get old SR value */
li a1, SR_CLR_IL /* SR_CLR_IL = 0xfff7ffff */
and a0, a0, a1
mtc0 a0, C0_SR /* Clear Lock bit for instruction cache */
nop
```

Notes

```
nop
nop
nop
nop /* 5 nops: safety against CP0 hazard */
la v0, 3f
jr v0
nop
3: /* Resume execution in mode as linked */
```

Cache Locking in RC36100

The RC36100 allows the data and or instruction caches to be partitioned into 2 or 4 portions, each servicing different regions of the address space. The software designer can then make data and instruction references in such a way as to force them to be placed in different partitions of the cache, effectively "locking" them into cache.

Caches can be partitioned by writing to the Cache Configuration register. When the cache is partitioned, different visual addresses map to the same physical memory locations, but are placed in different parts of the cache.

For instance, when the data cache is split in two, virtual addresses 0x8000_0000 through 0x8fff_ffff and 0x9000_0000 through 0x9fff_ffff both map to the physical address 0x0000_0000 through 0x0fff_ffff, but map to different partitions in the cache. Figure 1.2, which follows, illustrates this point.

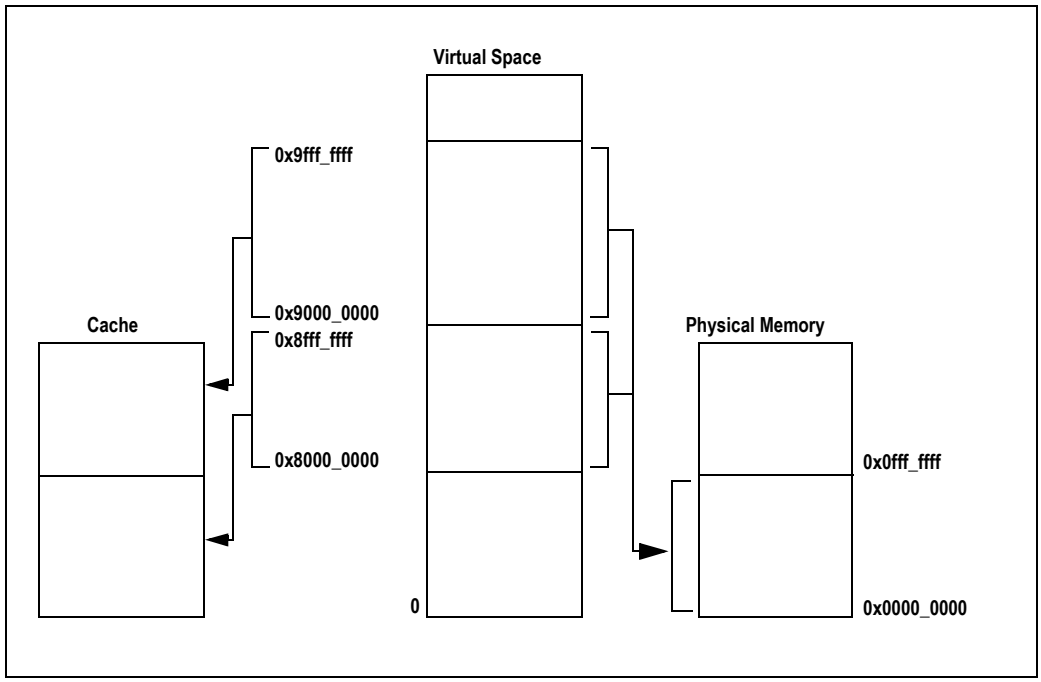


Figure 1.2 Cache partitioning example (RC36100)

For example, by using the address 0x9xxx_xxxx to refer to "critical" data, it force s that data into always being mapped to one partition of the cache. In this example, if all other data accesses were made to 0x8xxx_xxx, only the critical data would stay in the data cache. The instruction cache behavior and handling can be inferred analogously.

Cache Isolation and Swapping in RC30xx

In the RC30xx, no special instructions are provided to explicitly access the caches; everything is done with load and store instructions.

Notes

To distinguish operations for cache management from regular memory references, without having to dedicate a special address region for this purpose, the RC30xx architecture provides bits in the *SR* to support cache management:

- *The SR bit "IsC" will isolate the D-cache. In this mode, loads and stores affect only the cache and loads also "hit" regardless of whether the tag matches. With the D-cache isolated a partial-word write will invalidate the appropriate cache line.*

  Caution: when the D-cache is isolated, not even loads/stores marked by their address or TLB entry as ''uncached'' will operate normally. One consequence of this is that the cache management routines must not make any data accesses; they are typically written in assembler, using only register variables.

- *The CPU provides a mode where the caches are swapped (SR "SwC" bit), to allow the I-Cache to be targeted by store instructions; then the D-cache acts as an I-cache, and the I-cache acts as the D-cache. Once the caches are swapped and isolated I-cache entries may be read, written and invalidated (invalidation uses the same partial word write mechanism described above).*

  Note that cache isolation does not stop uncached instruction fetches from referencing main memory.

  The D-cache behaves like an I-cache (provided it was sufficiently initialized to work as a D-cache); however, the I-cache does not behave like a D-cache. It is unlikely that it will ever be useful to have the caches swapped but not isolated.

  If software does use a swapped I-cache for word stores (a partial-word store invalidates the line, as before) it must make sure those locations are invalidated before returning to normal operation.

Rc32364/RC4600/RC4700/RC4650/RC5000 Cache Characteristics

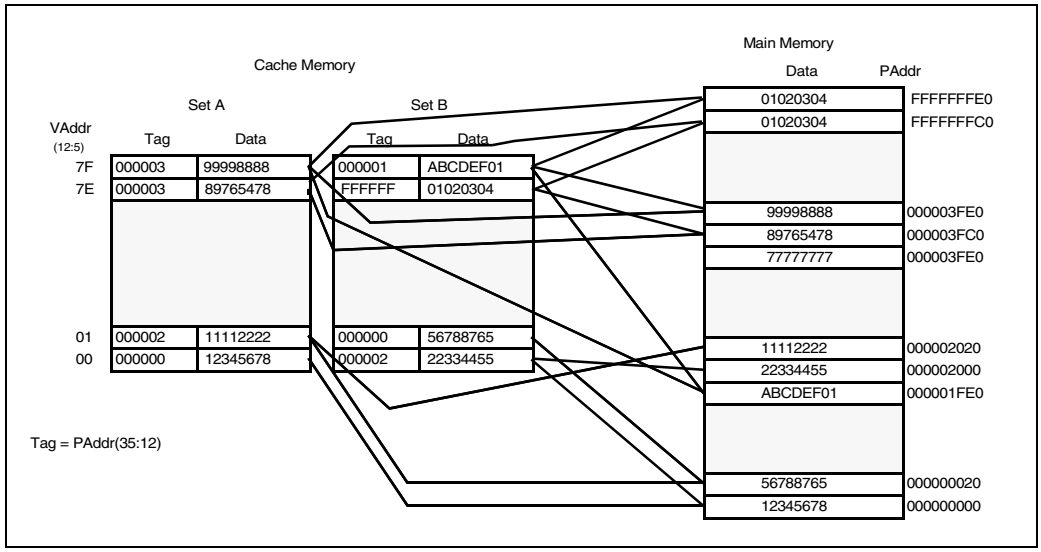


Figure 1.3 Two-way Set-associative Cache

Both caches in the RC4xxx/RC32364 are:

- *Two-way set associative: A direct mapped cache with multiple sets of entries (2 in case of RC4xxx - hence the term two-way) is called a set associative cache.*
- *Indexed with virtual address, checked with physical tag: Unlike the RC30xx, the RC4xxx/RC32364 caches are indexed with low-order bits of the program (virtual) address. In the RC4650, since the size of one set of primary caches is 4KB, the virtual index is coincidently the same as the physical index. Having indexed to a particular cache line entry, a hit or a miss is determined by matching the block number with the value in the tag field for that particular entry (line). The block number is formed by the highest bits of physical address.*
- *Cache lines: The line size is the number of data elements stored with each tag. For RC4xxx family CPUs both caches the implement a 8-word (32-byte) line size. For RC32364 the line size is 4-word (16-byte).*

Notes

- *Write-through or Write-back: The I-cache is of course write-through. This means that all store operations result in a store to main memory. The D-cache, however, can be configured to be write-through or write-back on a per page basis during cache initialization. In the write-back mode, the cache line is written back to memory only when forced by the user's code (cache instruction) or when it needs to be replaced to make space for data being read from memory. A bit in the tag (W bit) indicates whether or not there is a need to write back (whether cache line has more updated data than corresponding memory location). After writing back in this mode, the W bit is cleared. The W bit is unaffected in write through mode.*
- *Cache locking: (**RC4650 only**) This feature allows 4KB of data and/or 4KB of instructions to reside in the appropriate cache without being disturbed by cache refill algorithms. A program can move time critical code or data to set A of the appropriate cache and lock the set. Set A is the only one that can be locked.*

  At reset, both caches are unlocked. When both sets are invalid, the CPU always selects set A to fill first. So, to lock code into a particular cache, invalidate caches, set appropriate cache lock bits (DL or IL) in the CP0 *status* register, and immediately load time-critical code or data into the caches. Entries will be filled even in the locked cache as long as they are marked invalid. Once filled and marked valid, as long as the set is locked, they will not be refilled. The program is free to unlock caches at any time by clearing the appropriate lock bit in the CP0 *status* register. Code sample for locking is presented towards the end of this chapter.

  Cache locking for the **RC32364** is explained a few pages back.
- *Partial line write implementations: When performing a store, the on-chip cache controller must insure that the single tag field continues to describe all 32 bytes of the line. Although the rules used by the on-chip cache controller may seem complex, there is no action required of the software engineer.*

  When the cache is programmed to be in write-through mode, performing a store to a cached address will:

  – *For a cache hit: update that part of the cache line and also main memory*
  – *For a cache miss: 1) write-allocate cache—the CPU performs a read-modify-write cycle, as follows: does a block read, which fills the cache, and updates the relevant parts of cache line and main memory. 2) cache with no write-allocate—the CPU writes only to the main memory.*

  When the cache is programmed to be in write-back mode, a store operation that hits in cache updates only the relevant part of the cache line. A miss results in the whole line being brought in and the relevant part of the cache line being updated. In either case, main memory will not be updated until it is time to replace the cache line.
- *Coherency: Multi-master systems must have a mechanism to maintain data consistency throughout the system. This mechanism is called a cache coherency protocol and is handled with software. The RC32364/RC4600/RC4700/RC4650 do not provide any hardware cache coherency.*

  In the IDT RC4xxx/RC32364, these attributes merely control the cacheability and write rules for the virtual addresses. Bits in the TLB control coherency on a per page basis in the RC32364/RC4600/RC4700. In the RC4650, since there is no TLB, a new CP0 register (number 17) called *CAlg* register is handed the task. *CAlg* defines the cache algorithm for each 512 Mb region of the program address space.
- *Processor Synchronization: Although RC32364/RC4600/RC4700/RC4650 do not support symmetric multi-processing, synchronization operation to guarantee orderly access to shared memory is important for multi-master and heterogeneous multi-processor systems. Synchronization can be achieved in many ways through software. The most common two techniques are test-and-set and use of a counter. The RC4xxx/RC32364 instructions Load Linked (LL) and Store Conditional (SC) provide simple and efficient support for processor synchronization using many techniques including the two mentioned above.*

Notes

Initializing and Sizing the Caches

At machine start-up, the caches are in a random state, so the result of a cached read is unpredictable. In addition, following a reset the RC30xx status register SwC and IsC bits are also in a random state, so start-up software should set them to a known state before attempting any load or store (even uncached). In the RC4xxx/RC32364, care must be taken to ensure that the initialization code does what it does for both sets (2-*way* set-associative caches).

Different members of the RC30xx family, RC32364, RC5000 and the RC4xxx family have different cache sizes. Software will be more portable if it dynamically determines the size of the I-cache and D-cache at initialization time, rather than hard-wiring a particular value.

RC30xx Cache Sizing Code Sample:

A number of algorithms are possible. Shown below is the code contained in IDT/sim for cache sizing in a RC30xx. The basic algorithm works as follows:

- *isolate the D-cache;*
- *swap the caches when sizing the I-cache;*
- *Write a marker into the initial cache entry.*
- *Start with the smallest permissible cache size.*
- *Read memory at the location for the current cache size. If it contains the marker, that is the correct size. Otherwise, double the size to try and repeat this step until the marker is found.*

```
/*
** Config_cache() -- determine sizes of i and d caches
** Sizes stored in globals dcache_size and icache_size
*/ This code is RC30xx specific only

#define CONFIGFRM ((4*4)+4+4)
FRAME(config_cache,sp, CONFIGFRM, ra)
        .set    noreorder
        subu    sp,CONFIGFRM
        sw      ra,CONFIGFRM-4(sp)# save return address
        sw      s0,4*4(sp)      # save s0 in first regsave slot
        mfc0    s0,C0_SR        # save SR
        mtc0    zero,C0_SR      # disable interrupts
        .set    reorder
        jal     _size_cache
        sw      v0,dcache_size
        li      v0,SR_SWC       # swap caches
        .set    noreorder
        mtc0    v0,C0_SR
        jal     _size_cache
        nop
        sw      v0,icache_size
        mtc0    zero,C0_SR      # swap back caches
        and     s0,~SR_PE       # do not inadvertantly clear PE
        mtc0    s0,C0_SR        # restore SR
        .set    reorder
        lw      s0,4*4(sp)      # restore s0
        lw      ra,CONFIGFRM-4(sp)# restore ra
        addu    sp,CONFIGFRM    # pop stack
        j       ra
        ENDFRAME(config_cache)

/*
** _size_cache()
** return size of current data cache
*/
FRAME(_size_cache,sp,0,ra)
        .set    noreorder
        mfc0    t0,C0_SR        # save current sr
        nop
```

Notes

```
        and     t0,~SR_PE       # do not inadvertently clear PE
        or      v0,t0,SR_ISC    # isolate cache
        mtc0    v0,C0_SR
        /*
         * First check if there is a cache there at all
         */
        move    v0,zero
        li      v1,0xa5a5a5a5   # distinctive pattern
        sw      v1,K0BASE       # try to write into cache
        lw      t1,K0BASE       # try to read from cache
        nop
        mfc0    t2,C0_SR
        nop
        .set    reorder
        and     t2,SR_CM
        bne     t2,zero,3f      # cache miss, must be no cache
        bne     v1,t1,3f        # data not equal -> no cache
        /*
         * Clear cache size boundries to known state.
         */
        li      v0,MINCACHE
1:
        sw      zero,K0BASE(v0)
        sll     v0,1
        ble     v0,MAXCACHE,1b

        li      v0,-1
        sw      v0,K0BASE(zero) # store marker in cache
        li      v0,MINCACHE     # MIN cache size

2:      lw      v1,K0BASE(v0)   # Look for marker
        bne     v1,zero,3f      # found marker
        sll     v0,1                    # cache size * 2
        ble     v0,MAXCACHE,2b  # keep looking
        move    v0,zero                 # must be no cache
        .set    noreorder
3:      mtc0    t0,C0_SR        # restore sr
        j       ra
        nop
        ENDFRAME(_size_cache)
        .set    reorder
```

RC32364/RC4xxx/RC5000 Cache Sizing Code Sample:

The RC32364/RC4xxx/RC5000 coding is much simpler. Here, the cache sizes can be read from the *config* register, as shown in the following code sample. The RC5000 config register also specifies the secondary cache size.

```
#define LEAF(label)FRAME(label,sp,0,ra) /*macro used ahead*/

/*
 * void config_cache()
 *
 * Work out size of I, D & S caches, assuming they
        are already initialised.
 */
LEAF(config_cache)
        lw      t0,icache_size
        bgtz    t0,8f                   # already known?
        move    v0,ra
        bal     _size_cache
        move    ra,v0

        sw      t2,icache_size
        sw      t3,dcache_size
        sw      t6,scache_size
```

Notes

```
        sw      t4,icache_linesize
        sw      t5,dcache_linesize
        sw      t7,scache_linesize
8:      j       ra
END(config_cache)


/*
 * static void _size_cache()
 *
 * routine to determine cache sizes by looking at RC4xxx config
 * register.  Sizes are returned in registers, as follows:
 *      t2      icache size
 *      t3      dcache size
 *      t6      scache size
 *      t4      icache line size
 *      t5      dcache line size
 *      t7      scache line size
 */
LEAF(_size_cache)
        mfc0    t0,C0_CONFIG

        and     t1,t0,CFG_ICMASK
        srl     t1,CFG_ICSHIFT
        li      t2,0x1000
        sll     t2,t1

        and     t1,t0,CFG_DCMASK
        srl     t1,CFG_DCSHIFT
        li      t3,0x1000
        sll     t3,t1

        li      t4,32
        and     t1,t0,CFG_IB
        bnez    t1,1f
        li      t4,16
1:

        li      t5,32
        and     t1,t0,CFG_DB
        bnez    t1,1f
        li      t5,16
1:

        move    t6,zero                 # default to no scache
        move    t7,zero                 #

        and     t1,t0,CFG_C_UNCACHED# test config register
        bnez    t1,1f   # no scache if uncached/non-coherent

        li      t6,0x100000     # assume 1Mb scache <<-NOTE
        and     t1,t0,CFG_SBMASK
        srl     t1,CFG_SBSHIFT
        li      t7,16
        sll     t7,t1
1:      j       ra
END(_size_cache)


/* RC4000 configuration register definitions used above*/
#define CFG_SBMASK0x00c00000/* Secondary cache block size */
#define CFG_SBSHIFT22
#define CFG_BE0x00008000/* Big Endian */
#define CFG_ICMASK0x00000e00/* Instruction cache size */
#define CFG_ICSHIFT9
#define CFG_DCMASK0x000001c0/* Data cache size */
#define CFG_DCSHIFT6
#define CFG_IB0x00000020/* Instruction cache block size */
```

Notes

```
#define CFG_DB0x00000010/* Data cache block size */
#define CFG_K0MASK0x00000007/* KSEG0 coherency algorithm */

/*
 * RC4000 primary cache mode
 */
#define CFG_C_UNCACHED2
#define CFG_C_NONCOHERENT3
#define CFG_C_COHERENTXCL4
#define CFG_C_COHERENTXCLW5
#define CFG_C_COHERENTUPD6

/*
 * RC4000 cache operations
 */
#define Index_Invalidate_I          0x0     /* 0    0 */
#define Index_Writeback_Inv_D                                   0x1 /* 0    1 */
#define Index_Invalidate_SI         0x2     /* 0    2 */
#define Index_Writeback_Inv_SD                                  0x3 /* 0    3 */
#define Index_Load_Tag_I            0x4     /* 1    0 */
#define Index_Load_Tag_D            0x5     /* 1    1 */
#define Index_Load_Tag_SI           0x6     /* 1    2 */
#define Index_Load_Tag_SD           0x7     /* 1    3 */
#define Index_Store_Tag_I           0x8     /* 2    0 */
#define Index_Store_Tag_D           0x9     /* 2    1 */
#define Index_Store_Tag_SI          0xA     /* 2    2 */
#define Index_Store_Tag_SD          0xB     /* 2    3 */
#define Create_Dirty_Exc_D          0xD     /* 3    1 */
#define Create_Dirty_Exc_SD         0xF     /* 3    3 */
#define Hit_Invalidate_I            0x10    /* 4    0 */
#define Hit_Invalidate_D            0x11    /* 4    1 */
#define Hit_Invalidate_SI           0x12    /* 4    2 */
#define Hit_Invalidate_SD           0x13    /* 4    3 */
#define Hit_Writeback_Inv_D         0x15    /* 5    1 */
#define Hit_Writeback_Inv_SD                    0x17            /* 5    3 */
#define Fill_I              0x14    /* 5    0 */
#define Hit_Writeback_D             0x19    /* 6    1 */
#define Hit_Writeback_SD            0x1B    /* 6    3 */
#define Hit_Writeback_I             0x18    /* 6    0 */
#define Hit_Set_Virtual_SI          0x1E    /* 7    2 */
#define Hit_Set_Virtual_SD          0x1F    /* 7    3 */
```

Initializing RC30xx Cache

In a properly initialized cache, every cache entry is either invalid or correctly corresponds to a memory location, and also contains correct parity. Again, the sample code shown is from IDT/sim for **RC30xx**. The code works as follows:

- *Check that SR bit PZ is cleared to zero (1 disables parity; the RC3071 and RC3081 contain parity bits, and thus PZ=1 could cause the caches to be initialized improperly).*
- *Isolate the D-cache, swap to access the I-cache.*
- *For each word of the cache: first write a word value (writing correct tag, data and parity), then write a byte (invalidating the line).*

  Note that for an I-cache with 4 words per line this is inefficient; it would be enough to write just one byte in the line to invalidate the entry. Unless the system uses the invalidate routine often it doesn't seem worth the trouble.

RC30xx Cache Initialization Code:

```
FRAME(flush_cache,sp,0,ra)
        lw      t1,icache_size
        lw      t2,dcache_size
```

Notes

```
                .set    noreorder
                mfc0    t3,C0_SR        # save SR
                nop
                and     t3,~SR_PE       # dont inadvertently clear PE
                beq     t1,zero,_check_dcache # if no i-cache check
                                                        d-cache
                nop
                li      v0,SR_ISC|SR_SWC# disable intr, isolate
                                                        and swap
                mtc0    v0,C0_SR
                li      t0,K0BASE
                .set    reorder
                or      t1,t0,t1

1:              sb      zero,0(t0)
                sb      zero,4(t0)
                sb      zero,8(t0)
                sb      zero,12(t0)
                sb      zero,16(t0)
                sb      zero,20(t0)
                sb      zero,24(t0)
                addu    t0,32
                sb      zero,-4(t0)
                bne     t0,t1,1b
                /*
                 * flush data cache
                 */
_check_dcache:
                li      v0,SR_ISC       # isolate and swap back caches
                .set    noreorder
                mtc0    v0,C0_SR
                nop
                beq     t2,zero,_flush_done
                .set    reorder
                li      t0,K0BASE
                or      t1,t0,t2

1:              sb      zero,0(t0)
                sb      zero,4(t0)
                sb      zero,8(t0)
                sb      zero,12(t0)
                sb      zero,16(t0)
                sb      zero,20(t0)
                sb      zero,24(t0)
                addu    t0,32
                sb      zero,-4(t0)
                bne     t0,t1,1b

                .set    noreorder
_flush_done:
                mtc0    t3,C0_SR        # un-isolate, enable interrupts
                .set    reorder
                j       ra
                ENDFRAME(flush_cache)
```

Initializing RC4xxx/RC32364/RC5000 Cache

Here, again, because of the *cache* instruction in the RC4xxx/RC32364/RC5000, the initialization process is straight forward. All the programmer needs to worry about is picking the correct operation for the *cache* instruction and setting up the proper loops, a task nicely mechanized by the macros provided below.

RC4xxx/RC32364/RC5000 Specific Cache Initialization Code:

```
/*
 * void flush_cache (void)
 *
```

Notes

```
 * Flush and invalidate all caches
 */
LEAF(flush_cache)
        /* secondary cacheops do all the work if present */
        lw      a2,scache_size
        blez    a2,1f
        lw      a3,scache_linesize
        li      a0,PHYS_TO_K0(0)
        move    a1,a2
        icacheop(a0,a1,a2,a3,Index_Writeback_Inv_SD)
        b       2f

1:      lw      a2,icache_size
        blez    a2,2f
        lw      a3,icache_linesize
        li      a0,PHYS_TO_K0(0)
        move    a1,a2
        icacheop(a0,a1,a2,a3,Index_Invalidate_I)

        lw      a2,dcache_size
        lw      a3,dcache_linesize
        li      a0,PHYS_TO_K0(0)
        move    a1,a2
        icacheop(a0,a1,a2,a3,Index_Writeback_Inv_D)

2:      j       ra
END(flush_cache)



/* cache operation macros use in code above*/
/*
* first some helpers...
 */
#define _mincache(size, maxsize) \
        bltu    size,maxsize,8f ;\
        move    size,maxsize ;      \
8:

#define _align(tmp, minaddr, maxaddr, linesize) \
        subu    tmp,linesize,1 ;\
        not     tmp ;                       \
        and     minaddr,tmp ;       \
        addu    maxaddr,-1 ;        \
        and     maxaddr,tmp

/* This is a bit of a hack really because it relies on minaddr=a0 */
#define _doop1(op1) \
        cache   op1,0(a0)
#define _doop2(op1, op2) \
        cache   op1,0(a0) ;         \
        cache   op2,0(a0)

/* specials for cache initialisation */
#define _doop1lw1(op1) \
        cache   op1,0(a0) ;         \
        lw      zero,0(a0) ;        \
        cache   op1,0(a0)
#define _doop121(op1,op2) \
        cache   op1,0(a0) ;         \
        nop;                                \
        cache   op2,0(a0) ;         \
        nop;                                \
        cache   op1,0(a0)

#define _oploopn(minaddr, maxaddr, linesize, tag, ops) \
        .set    noreorder ;         \
7:      _doop##tag##ops ;  \
```

Notes

```
        bne     minaddr,maxaddr,7b ;\
        addu    minaddr,linesize ;\
        .set    reorder

/* Now the actual cachop macros */

#define icacheopn(kva, n, cache_size, cache_linesize, tag, ops) \
        _mincache(n, cache_size);\
        blez    n,9f ;                          \
        addu    n,kva ;                         \
        _align(t1, kva, n, cache_linesize) ; \
        _oploopn(kva, n, cache_linesize, tag, ops) ; \
9:

#define vcacheopn(kva, n, cache_size, cache_linesize, tag, ops) \
        blez    n,9f ;                          \
        addu    n,kva ;                         \
        _align(t1, kva, n, cache_linesize) ; \
        _oploopn(kva, n, cache_linesize, tag, ops) ; \
9:

#define icacheop(kva, n, cache_size, cache_linesize, op) \
        icacheopn(kva, n, cache_size, cache_linesize, 1, (op))

#define vcacheop(kva, n, cache_size, cache_linesize, op) \
        vcacheopn(kva, n, cache_size, cache_linesize, 1, (op))
```

Invalidation

Invalidation marks specified cache line(s) as containing no valid references to main memory. Software needs to invalidate:

- *the D-cache, when memory contents have been changed by something other than store operations from the CPU. Typically this is performed when some DMA device is reading into memory.*
- *the I-cache, when instructions have been either written by the CPU or obtained by DMA. The hardware does not prevent the same locations from being used in the I- and D-cache. An update by the processor will not change the I-cache contents.*

Note that the system could be constructed to use unmapped accesses to those variables shared with a DMA device; the only difference is in performance. In general, small areas where DMA is frequently compared to CPU activity should be mapped uncached; larger areas where CPU activity predominates should be invalidated by the driver at appropriate points. Keep in mind that invalidating a word of data in the cache is faster (probably 4-7 times) than an uncached load.

To invalidate the cache in the **RC30xx**:

- *Figure out the address range to invalidate, up to the cache size.*
- *Isolate the RC30xx D-cache. Once it is isolated, the system must insure at all costs against an exception (since the memory interface will be temporarily disabled). Disable interrupts and ensure that software which follows cannot cause a memory access exception;*
- *to work on the I-cache, swap the caches in the RC30xx;*
- *write a byte value to each cache line in the range;*
- *(unswap and) unisolate (RC30xx only).*

The invalidate routine is normally executed with its instructions cacheable. This sounds like a lot of trouble; but in fact shouldn't require any extra steps to run cached. An invalidation routine in uncached space will run 4-10 times slower.

Two code samples follow. First for the RC30xxx and the next one for RC4xxx/RC32364/RC5000.

Again, the example code fragments shown are taken from IDT/sim:

```
/*
** clear_cache(base_addr, byte_count)
```

Notes

```
** flush portion of cache
** RC30xx specific code
*/
FRAME(clear_cache,sp,0,ra)

        /*
         * flush instruction cache
         */
        lw      t1,icache_size
        lw      t2,dcache_size
        .set    noreorder
        mfc0    t3,C0_SR            # save SR
        nop
        and     t3,~SR_PE           # dont inadvertently clear PE
        nop
        nop
        li      v0,SR_ISC|SR_SWC# disable intr, isolate and swap
        mtc0    v0,C0_SR
        .set    reorder
        bltu    t1,a1,1f            # cache is smaller than region
        move    t1,a1
1:      addu    t1,a0                       # ending address + 1
        move    t0,a0

        sb      zero,0(t0)
        sb      zero,4(t0)
        sb      zero,8(t0)
        sb      zero,12(t0)
        sb      zero,16(t0)
        sb      zero,20(t0)
        sb      zero,24(t0)
        addu    t0,32
        sb      zero,-4(t0)
        bltu    t0,t1,1b

        /*
         * flush data cache
         */

        .set    noreorder
        nop
        li      v0,SR_ISC           # isolate and swap back caches
        mtc0    v0,C0_SR
        nop
        .set    reorder
        bltu    t2,a1,1f            # cache is smaller than region
        move    t2,a1
1:      addu    t2,a0                       # ending address + 1
        move    t0,a0

1:      sb      zero,0(t0)
        sb      zero,4(t0)
        sb      zero,8(t0)
        sb      zero,12(t0)
        sb      zero,16(t0)
        sb      zero,20(t0)
        sb      zero,24(t0)
        addu    t0,32
        sb      zero,-4(t0)
        bltu    t0,t2,1b

        .set    noreorder
        mtc0    t3,C0_SR            # un-isolate, enable interrupts
        .set    reorder
        j       ra
        ENDFRAME(clear_cache)
```

Notes

```
/* RC4xxx/RC5000/RC32364 code sample...

 * void clean_cache (unsigned kva, size_t n)
 *
 * Writeback and invalidate address range in all caches
 */
LEAF(clean_cache)

        /* secondary cacheops do all the work (if fitted) */
        lw      a2,scache_size
        blez    a2,1f
        lw      a3,scache_linesize
        vcacheop(a0,a1,a2,a3,Hit_Writeback_Inv_SD)
        b       2f

1:      lw      a2,icache_size
        blez    a2,2f
        lw      a3,icache_linesize
        /* save kva & n for subsequent loop */
        move    t8,a0
        move    t9,a1
        vcacheop(a0,a1,a2,a3,Hit_Invalidate_I)

        lw      a2,dcache_size
        lw      a3,dcache_linesize
        /* restore kva & n */
        move    a0,t8
        move    a1,t9
        vcacheop(a0,a1,a2,a3,Hit_Writeback_Inv_D)

2:      j       ra
END(clean_cache)
```

Locking Set A of RC4650 Caches

Example of RC32364 cache locking is provided within the first few pages of this chapter.

For RC4650, as described earlier, one can lock set A (size 4Kb) of D-cache and/or I-cache to store time-critical instructions or data elements for fast access.

An example of locking a data table in D-cache follows:

In the startup code, after initialization of data structures, flushing of caches etc. is done, the user can perform reads through cached addresses to load the data into the data cache, and then set the DL bit in the *status* register to lock set A of the data cache.

Here is a sample code fragment:

```
        .set noreorder
        jal     flush_cache                     /* invalidate caches */
        nop
        la      t0, critical_table/* table to lock */
        li      t1, table_size                  /* Size of table in bytes */
        li      t2, 0                           /* read bytes to cache */
1:      lw      a0, 0(t0)
        addiu   t2, 4
        bneq    t2, t1, 1b                      /* Loop back till done */
        addiu   t0, 4                           /* bump read address */

        mfc0    a0, C0_SR                       /* Get old SR value */
        li      a1, SR_DL                       /* SR_DL = 0x00100000 */
        or      a0, a0, a1
        mtc0    a0, C0_SR                       /* Set Lock bit-Dcache */
        nop
```

Notes

```
        nop
        nop                                  /* 3 nops:CP0 hazard */
```

Example: Instruction Cache Locking

An example of an I-cache locking function follows. For the purposes of this example, assume that the function size is known (when not known, the function size can easily be found by generating a disassembly of the object file).

In the startup code, after initialization of data structures and cache flushing caches is done, the user can perform the *fill* operation in the *cache* instruction to fill the instruction cache with the critical function. Then set the IL bit in the *status* register to lock set A of the instruction cache.

Here is a sample code fragment:

```
        .set noreorder
        la     t0, 1f                        /* Get address of labelÕ */
        li     t1, 0xA0000000
        or     t0, t0, t0
        jr     t0                            /* Uncached from now on*/
        nop
1:      jal    flush_cache
        nop
        la     t0, func_start_addr /* addr of code to lock*/
        li     t1, func_size                 /* Critical code size */
        li     t2, 0                         /* words to cache counter*/
2:      cache  Fill_I, 0(t0)                 /* Fill Operation */
        addiu  t2, 4
        bneq   t2, t1, 1b                    /* Loop back till done */
        addiu  t0, 4                         /* bump read address */

        mfc0   a0, C0_SR                     /* Get old SR value */
        li     a1, SR_IL                     /* SR_IL = 0x00080000 */
        or     a0, a0, a1
        mtc0   a0, C0_SR                     /* Set Lock bit-Icache */
        nop
        nop
        nop
        nop
        nop             /* 5 nops: CP0 hazard */
la     v0, 3f
jr     v0
nop

3:  /* Resume execution in mode as linked */
```

Testing and Probing

During test, debug or when profiling, it may be useful to build a picture of the cache contents.

In the RC4xxx/RC5000/RC32364, use the *cache* instruction to read the tag at any index and in any cache (*Index Load Tag* opcode). The value will be read into the *TagLo* register. Use the *mfc0* instruction to move the tag value to a general register to properly mask off the 8 low order bits to isolate the tag only. Do this on both sets to create a full picture of the cache.

In the RC30xx, software cannot read the tag value directly, but, for a valid line, it can determine the tag value by searching as follows:

- *isolate the cache;*
- *load from the cache line at each possible line start address (low-order bits fixed, high-order bits*

Notes

ranging over physical memory which exists in the system). After each load, consult the CM bit in SR, which will be "0" only when the tag value matches.

Configuration (RC3041/71/81 only)

The RC3041, RC3071, and RC3081 processors allow the programmer to make choices about the cache by setting fields in the *Config* register:

- *Cache refill burst size (R3041/71/81): by default the RC3041 refills only 1 word in the D-cache on a cache miss; but software can program it to use 4-word burst reads instead, by setting the Config DBR bit. The bit can be changed at any time, without needing to invalidate the cache.*

  The refill of RC3071 and RC3081 processors can be configured by hardware at reset-time, but software can override that choice.

  This support is provided in the hope of enhancing performance. The proper selection for a given system will depend on both the hardware and the application. Some systems may find an advantage in "toggling" the bit for various portions of the software. In general, the proper burst size selection can be determined as follows:

  Burst reads make most sense when the memory is capable of returning a burst of data significantly faster than it can return 4 individual words. Many DRAM systems are like this; most ROM and static RAM memories are not. Similarly, data accessed from narrow memory ports should rarely be configured for a multi-word burst.

  If programs tend to access memory sequentially (working up or down a large array, for example) then the burst refill will offer a very useful degree of data prefetch, and performance will be enhanced. If cache access is more random, the burst refill may actually reduce performance (since it involves overwriting cached data with memory data the program may never use).

  As a general rule, the bigger the D-cache, the smaller the penalty for burst refills.

- *Bigger I-cache in exchange for smaller D-cache (R3071/81): the R3081 cache can be organized either with both I-cache and D-cache 8Kbytes in size, or with a 16Kbyte I-cache and 4Kbyte D-cache. The configuration is programmed using the AC bit in the Config register. After changing the cache configuration both caches should be re-initialized, while running uncached. This means that most systems will not dynamically reconfigure the caches.*

  For a given system, the best configuration depends on the application. Cache effects are extremely hard to predict, so it is recommended that both configurations be tried and measured, while running as much of the real system as possible.

  As a general rule, with large applications, the big I-cache will probably be best. But, if the system spends most of its time manipulating lots of data from tight program loops, the big D-cache may be better.

Write Buffer

The write-through cache (RC30xx CPUs) can be a big performance bottleneck. In the average C program, only about 10% of the instructions are stores, but these accesses tend to come in bursts such as when a function prologue saves a few registers.

DRAM memory frequently has the characteristic that the first write of a group takes a long time (typically, 5-10 clocks on these CPUs), and subsequent ones are relatively fast as long as they follow quickly and within the same DRAM page.

If the CPU simply waits for all writes to complete, the performance hit will be significant. So the RC30xx provides a *write buffer*, a FIFO store which keeps a number of entries each containing both data to be written, and the address at which to write it. The 4-entry queue provided by RC30xx family CPUs is efficient for well-tuned DRAM.

In general, the operation of the write buffer is completely transparent to software. Occasionally, the programmer needs to be aware of what is happening:

- *Timing relations for I/O register accesses: When software performs a store to write an I/O register,*

Notes

the store reaches memory after a small, but indeterminate, delay. Some consequences are:

- *other communication with the I/O system (e.g. interrupts) may happen more quickly – for example, the CPU may get an interrupt from a device "after" it has been programmed to generate no interrupts.*
- *if the I/O device needs some time to recover after a write the program must ensure that the write buffer FIFO is empty before counting out that time period.*
- *at the end of interrupt service, when writing to an I/O device to clear the interrupt it is asserting, software must insure that the command is actually written to the device, and that it has had to respond, before re-enabling that interrupt; otherwise, spurious interrupts may be signalled.*

In these cases, the programmer must ensure that the CPU waits while the write buffer empties. It is good practice to define a subroutine which does this job; it is traditionally called *wbflush()*. The following subsection provides hints on implementing this function.

Implementing *wbflush()*

IDT CPUs enforce strict write priority (all pending writes retired to memory before main memory is read). Thus, implementing *wbflush()* is as simple as implementing an uncached load (e.g. from the boot PROM vector). This will stall the CPU until the writes have finished, and the load finished too. Alternately, the overhead can be minimized by performing an uncached load from the fastest memory available in the system.

The code fragment below shows an implementation of wbflush taken from IDT/sim:

```
/*
** wbflush() flush the write buffer - this is specific for each hardware
**                                   configuration.
*/
FRAME(wbflush,sp,0,ra)
        .set noreorder

        lw      t0,wbflush#read an uncached memory location
        j       ra
        nop
        .set reorder
        ENDFRAME(wbflush)
```

Notes

Memory Management

Notes

Translation Lookaside Buffer (TLB)

Many IDT processors have on-chip memory management hardware. This provides a mechanism for dynamically translating program addresses in the mapped regions to physical addresses. For most processors, a key piece of hardware is the TLB. The RC4650, which does not have a TLB, handles the task of address translation using a simple base-bounds mechanism, which will be described later in this chapter.

Memory is managed on a page basis. In the RC30xx, the page size is fixed at 4Kbytes. The low-order 12 bits of the program address are used directly as the low order bits of the physical address.

In the RC4600/RC4700/RC32364/RC5000, the page size is variable. The page size is defined by the setting of a CP0 register called the *PageMask* register. Valid page sizes are: 4 Kb, 16 Kb, 64 Kb, 256 Kb, 1 Mb, 4 Mb, 16 Mb.

The TLB is *associative memory with* 64-entries in the RC30xx, 16 odd/even page entries (total of 32 pages) in the RC32364 and 48 odd/even page entries (total of 96 pages) in the RC4600/RC4700/RC5000. Each entry in an associative memory consists of a key field and a data field; when presented with a key, the memory returns the data of any entry where the key matches.

In the current IDT family, the TLB is referred to as “fully-associative”; this emphasizes that all keys are really compared with the input value in parallel.

The TLB’s key field contains these main sections:

- *Virtual page number: (VPN) this is just a program address with some number of the low bits cut off, since the low-order bits don’t participate in the translation process. In the RC30xx, with its fixed page size of 4 Kb, the VPN is simply program address with the low 12 bits cut off. In the RC4600/ RC4700, VPN is variable because the page sizes are variable.*

  For example, in the 32-bit mode, for the smallest page size of 4 Kb, the lowest 12 bits of program address are cut off resulting in a 20-bit VPN mapping 1M number of pages. At the other extreme, in the 32-bit mode, for 16 Mb page size, the lowest 24 bits of program address are cut off resulting in a 8-bit VPN mapping 256 pages. In the 64-bit mode, for smallest (4 Kb) sized pages, the VPN is 28-bits long mapping 256 M pages; for largest (16 Mb) sized pages, the VPN is 16-bits and maps 64 K pages.

  Note that the actual VPN field in the TLB entries is only 27-bits long and the number stored there is actually VPN divided by 2. This follows the odd / even page concept and is aided in translation by the two CP0 registers, EntryLo0 and EntryLo1, as described later.

- *Address Space Identifier. (ASID): this is a magic number used to stamp translations, and (optionally) is compared with an extended part of the key.*

  In multi-tasking systems, it is common to have all user-level tasks executing at the same virtual address (of course they are using different physical addresses); they are said to be using different address spaces. So translation records for different tasks will often share the same value of “VPN”. Without an ASID, when the OS switches from one task to another, it would have to find and invalidate all TLB translations relating to the old task’s address space, to prevent them from being erroneously used for the new one.

  Instead, the OS assigns a 6-bit (in RC30xx) or 8-bit (in RC4xxx/RC32364) unique code to each task’s distinct address space. During normal running this code is kept in the ASID field of the EntryHi register, and is used together with the program address to form the lookup key; so a translation with an ASID code which doesn’t match is quietly ignored.

  Since the ASID is only 6/8 bits long, OS software does have to lend a hand if there are ever more than 64/256 address spaces in concurrent use. In such a system, new tasks are assigned new ASIDs until all 64/256 are assigned; at that time, all tasks are flushed of their ASIDs “de-assigned”

Notes

and the TLB flushed. As each task is re-entered, a new ASID is given. Thus, ASID flushing is relatively infrequent.

The TLB data field includes:

- *Physical frame number (PFN): the physical address with the low address bits cut off. In an address translation, the VPN bits are replaced by the corresponding PFN bits to form the true physical address.*
- *Cache control bit (N) (RC30xx only): set to 1 to make the page uncacheable and set to 0 for CACHE.*
- *Coherency attribute bits (C) (RC4600/RC4700/RC32364/RC5000 only): a 3-bit field with following definition of bits:*

  000 = Cacheable, noncoherent, write-through, no write allocate

  001 = Cacheable, noncoherent, write-through, write allocate

  010 = Uncached

  011 = Cacheable, noncoherent, write-back

  1xx = Reserved.
- *Write control bit (D): set to 1 allows stores to execute. The "D" comes from this being called the "dirty bit." A typical use for these bits is discussed later in the "Simulating dirty bits" section of this chapter.*
- *Valid bit (V): set to 0 to make this entry usable. Access to an invalid page produces a different trap from a TLB refill exception, so making a page invalid means that unusual or complex conditions can be made to take a different trap, which does not have to be handled by the simple refill code.*
- *Global bit (G): set to disable the ASID-matching scheme, allowing an OS to map some program addresses to the same physical address for all tasks.*

Translating an address is now simple, as follows:

- *CPU generates a program address: either for an instruction fetch, a load or a store, in one of the translated address regions. The appropriate number of low bits are separated off, and the resulting VPN together with the current value of the ASID field in the EntryHi register used as the key to the TLB.*
- *TLB matches key: selecting the matching entry. The PFN is glued to the low-order bits of the program address to form a complete physical address.*
- *Valid?: the V and D bits are consulted. If it isn't valid, or a store is being attempted with D cleared, the CPU takes a trap. As with all translation traps, the BadVaddr register will be filled with the offending program address and TLB registers Context and EntryHi pre-filled with relevant information. The system software can use these registers to obtain data for exception service.*
- *Cached?: if the N bit (or proper combination of C bits in RC4600/RC4700/RC32364/RC5000) is set the CPU looks in the cache for a copy of the physical location's data; if it isn't there it will be fetched from memory and a copy left in the cache. Where the N bit is clear (or C bits = 2 in RC4600/RC4700/RC32364/RC5000) the CPU neither looks in nor refills the cache.*

There are only 64 entries in the RC30xx's TLB, which can hold translations for a maximum of 256 Kbytes of program addresses. This is far short of enough for most systems. The TLB is almost always going to be used as a software-maintained "cache" for a much larger set of translations in the RC30xx.

When a program address lookup in the TLB fails, a *TLB refill* trap is taken. System software has the job of:

- *figuring out whether there is a correct translation; if not the trap will be dispatched to the software which handles address errors.*
- *if there is a correct translation, constructing a TLB entry which will implement it;*
- *if the TLB is already full (and it almost always is full in running systems), selecting an entry which can be discarded;*
- *writing the new entry into the TLB.*

Notes

See below for how this can be tackled; but note here that although special CPU features help out with one particular class of implementations, the software can refill the TLB any way it likes

Memory Management and Base-bounds

In the RC4650, TLB is replaced with a base-bounds mechanism for program-to-physical address translation and with a new *CAlg* register for controlling cache attributes of areas of address space.

The base-bounds mechanism uses two pairs of 32-bit CP0 registers: one pair for instructions and the other for data. Each pair contains a Base register and a Bound register. These registers are called IBase/IBound, for instruction address spaces, and DBase/DBound, for data spaces.

When an address is translated, its page number is first compared against the appropriate Bounds register. If the address is recognized as valid, the base register is added to the program address to form the physical address.

Base-bounds can be used to execute multiple user tasks sharing same virtual addresses mapped to separate physical addresses. An OS can support task protection by writing appropriate values to these registers at context switch time.

A *mtc0* instruction can be used to change the contents of base/bound registers and must be executed in unmapped space. Mapped space cannot be entered for 5 instructions following a change to these registers.

MMU Registers

Table 6.1 lists the MMU CPU control registers, and includes a description of each.

Register Mnemonic	Description	CP0 Reg #
EntryHi	Together these registers hold a TLB entry. All reads and writes to the TLB must be staged through them. EntryHi also remembers the current ASID (**RC30xx** only)	10
EntryLo		2
EntryLo0	Same as EntryLo above; but holds PFN for ***even*** pages in **RC4600/RC4700/RC32364/RC5000** only	2
EntryLo1	Same as EntryLo above; but for ***odd*** pages in **RC4600/RC4700/RC32364/RC5000** only	3
Index	Determines which TLB entry will be read/written by appropriate instructions	0
Random	Pseudo-random value (actually a free-running counter) used by a *tlbwr* to write a new TLB entry into a “randomly” selected location.	1
PageMask	Holds comparison mask that sets the variable page size for each TLB entry in **RC4600/RC4700/RC32364/RC5000** only	5
Wired	Specifies cut-off number for nonreplaceable TLB entries in the **RC4600/RC4700/RC32364/RC5000** only	6
Context	Convenience register provided to speed up the processing of TLB refill traps. The high-order bits are read/write; the low-order 21 bits reflect the *BadVaddr* value. (The register is designed so that, if the system uses the “favored” arrangement of memory-held copies of memory translation records, it will be setup by a TLB refill trap to point to the memory location of the record needed to map the offending address. This speeds up the process of finding the current memory mapping, and arranging EntryHi/Lo properly).	4
Xcontext	Similar to Context above, used for 64-bit address space in **RC4600/RC4700/RC5000** only	20
IBase	User instruction space address base in **RC4650** only	0

Table 6.1 MU Registers (Part 1 of 2)

Notes

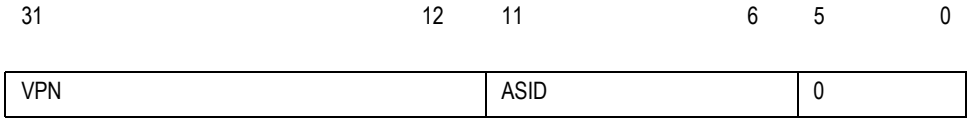
Register Mnemonic	Description	CP0 Reg #
IBound	Outer limit of User instruction space in **RC4650** only	1
DBase	User data space address base in **RC4650** only	2
DBound	Outer limit of User data space in **RC4650** only	3
CAlg	Cache algorithm for each 512 Mb region of space in **RC4650** only	

Table 6.1 MU Registers (Part 2 of 2)

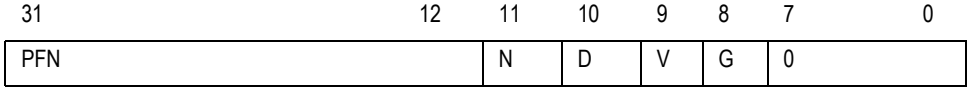
Description of MMU Registers

This subsection contains descriptions of the MMU registers and details about the register fields.

EntryHi, EntryLo (RC30xx)



EntryHi Register (TLB key fields)



EntryLo Register (TLB data fields)

These two registers represent a TLB entry, and are best considered as a pair. Fields in *EntryHi* are:

- *VPN: "virtual page number", the high-order bits of a program address. On a refill exception this field is set up automatically to match the program address which could not be translated. To write a different TLB entry, or attempt a TLB probe, software must set it up "manually".*
- *ASID: "address space identifier", normally left holding the OS' value for the current address space. This is not changed by exceptions. Most software systems will deliberately write this field only to setup the current address space.*

  However, software must be careful when using tlbr to inspect TLB entries; the operation overwrites the whole of EntryHi, so software needs to restore the correct current ASID value afterwards.

The fields in *EntryLo* are:

- *PFN: the high-order bits of the physical address to which values matching EntryHi's VPN will be translated.*
- *N: "noncacheable"; 0 to make the access cacheable, 1 for uncacheable.*
- *D: "dirty", but really a write-enable bit. 1 to allow writes, 0 and any store using this translation will be trapped.*
- *V: "valid", if 0 any address matching this entry will cause an exception.*
- *G: "global". When the G bit in a TLB entry is set, that TLB entry will match solely on the VPN field, regardless of whether the TLB entry's ASID field matches the value in EntryHi.*
- *Fields called "0": these fields always return zero; but unlike many reserved fields, they do not need to be written as zero (nothing happens regardless of the data written). This is important; it means that the memory-resident data which is used to generate EntryLo when refilling the TLB can contain some software-interpreted data in these fields, which the TLB hardware will ignore without the need to spend precious CPU cycles masking it.*

Notes

EntryHi, EntryLo0 EntryLo1(RC4600/RC4700/RC32364/RC5000)

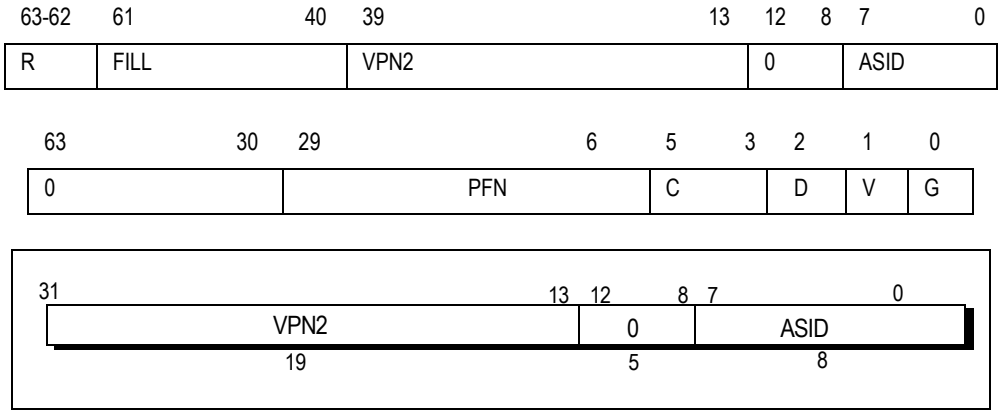


Figure 6.1 32-bit EntryHi Register Fields

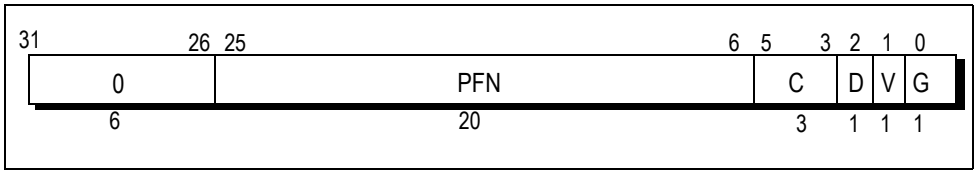


Figure 6.2 32-bit EntryLo0, EntryLo1 Register Fields in RC32364

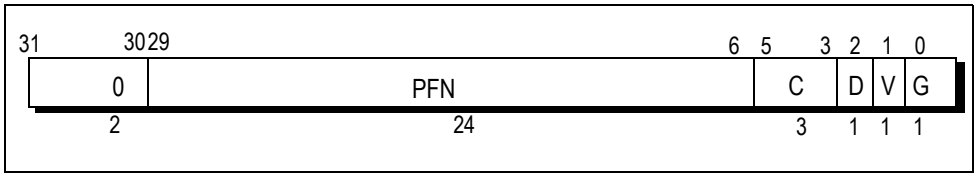


Figure 6.3 EntryLo0, EntryLo1 Register Fields in 32-bit Mode of RC5000

These three registers represent a TLB entry and are best considered as a pair. *EntryLo0* and *EntryLo1* have the same format only the first register refers to even pages and the second to odd pages.

The fields in *EntryHi* are:

- *VPN2: Virtual page number divided by two (maps to two pages).*
- *ASID: Address space ID field. An 8-bit field that lets multiple processes share the TLB; each process has a distinct mapping of otherwise identical virtual page numbers.*
- *R: Region. (00 → user, 01 → supervisor, 11 → kernel) used to match* $vAddr_{63...62}$
- *Fill: Reserved. Returns zero when read, ignored on writes.*
- *0: Reserved. Must be written as zeroes, and returns zeroes when read.*

The fields in *EntryLo0/1* are:

- *PFN: Page frame number; the upper bits of the physical address.*
- *C: Specifies the TLB page coherency attribute; see Table 6.2.*
- *D: Dirty. If this bit is set, the page is marked as dirty and, therefore, writable. This bit is actually a write-protect bit that software can use to prevent alteration of data.*
- *V: Valid. If this bit is set, it indicates that the TLB entry is valid; otherwise, a TLBL or TLBS miss occurs.*
- *G: Global. If this bit is set in both Lo0 and Lo1, then the processor ignores the ASID during TLB lookup.*
- *0: Reserved. Must be written as zeroes, and returns zeroes when read.*

Notes

The TLB page coherency attribute (*C*) bits specify whether references to the page should be cached; if cached, the algorithm selects between several coherency attributes. Table 6.2 shows the coherency attributes selected by the *C* bits.

C(5:3) Value	Page Coherency Attribute
0	Cacheable, noncoherent, write-through, no write allocate
1	Cacheable, noncoherent, write-through, write allocate
2	Uncached
3	Cacheable, noncoherent, write-back
4 - 7	Reserved

Table 6.2 TLB Page Coherency Attributes

Index Register (RC30xx)

31	30	14	13	8	7	0
P	×		Index		×	

The "P" field is set when a *tlbp* instruction (tlb probe, used to see if the TLB can translate a particular VPN) failed to find a valid translation; since it is the top bit it appears to make the 32-bit value negative, which is easy to test for.

Index Register (RC4600/RC4700/RC32364/RC5000)

The *Index* register is a 32-bit, read/write register containing six bits to index an entry in the TLB. The high-order bit of the register shows the success or failure of a TLB Probe (TLBP) instruction.

The *Index* register also specifies the TLB entry affected by TLB Read (TLBR) or TLB Write Index (TLBWI) instructions.

Figure 6.4 shows the format of the *Index* register; Table 6.3, which follows the figure, describes the *Index* register fields.

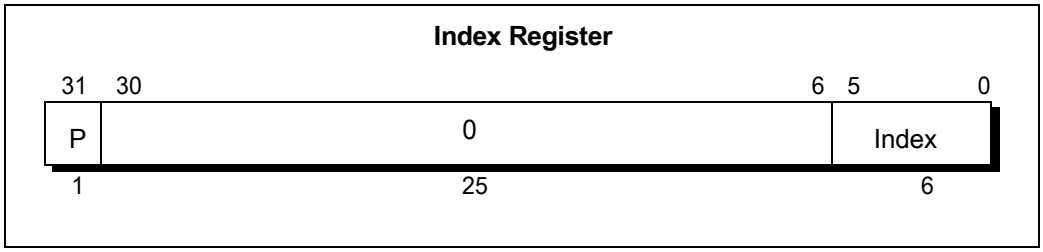


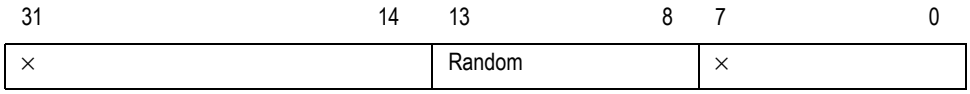
Figure 6.4 Index Register

Field	Description
P	Probe failure. Set to 1 when the previous TLBProbe (TLBP) instruction was unsuccessful.
Index	Index to the TLB entry affected by the TLBRead and TLBWrite instructions
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.3 Index Register Fields

Notes

Random Register (RC30xx)



Most systems never have to read or write the *Random* register, shown in Figure 6.5, in normal use; but it may be useful for diagnostics. The hardware initializes the *Random* field to its maximum value (63) on reset, and it decrements every clock period until it reaches 8, when it wraps back to 63 and starts again.

Figure 6.5 Random Register in RC32364

Random Register (RC4600/RC4700/RC32364/RC5000)

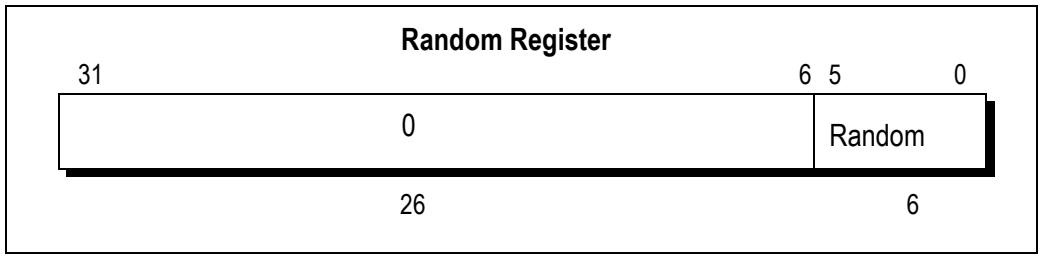


Figure 6.6 Random Register in RC4600/RC4700/RC5000

Field	Description
Random	TLB random index
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.4 Random Register Fields

The *Random* register is a read-only register of which six bits index an entry in the TLB. This register decrements as each instruction executes, and its values range between an upper and a lower bound, as follows:

- *A lower bound is set by the number of TLB entries reserved for exclusive use by the operating system (the contents of the Wired register).*
- *An upper bound is set by the total number of TLB entries. Thus the upper bound is 47 (The TLB entries are number from 0 to 47) in case of RC4600/RC4700/RC5000 and it is 15 in case of RC32364. Only valid instructions are counted.*

The *Random* register specifies the entry in the TLB that is affected by the TLB Write Random instruction. The register does not need to be read for this purpose; however, the register is readable for diagnostics.

To simplify testing, the *Random* register is set to the value of the upper bound upon system reset. This register is also set to the upper bound when the *Wired* register is written.

Figure 6.5 and Figure 6.6 show the format of the *Random* register; Table 6.4 describes the *Random* register fields.

Notes

PageMask Register (RC4600/RC4700/RC32364/RC5000 only)

The *PageMask* register is a read/write register used for reading from or writing to the TLB; it holds a comparison mask that sets the variable page size for each TLB entry, as shown in Table 6.5.

TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical address, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison.

When the *Mask* field is not one of the values shown in Table 6.5, the operation of the TLB is undefined.

Page Size	Bit											
	24	23	22	21	20	19	18	17	16	15	14	13
4 Kbytes	0	0	0	0	0	0	0	0	0	0	0	0
16 Kbytes	0	0	0	0	0	0	0	0	0	0	1	1
64 Kbytes	0	0	0	0	0	0	0	0	1	1	1	1
256 Kbytes	0	0	0	0	0	0	1	1	1	1	1	1
1 Mbyte	0	0	0	0	1	1	1	1	1	1	1	1
4 Mbytes	0	0	1	1	1	1	1	1	1	1	1	1
16 Mbytes	1	1	1	1	1	1	1	1	1	1	1	1

Table 6.5 PageMask Register Fields

Wired Register (RC4600/RC4700/RC32364/RC5000 only)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB, as shown in Figure 6.7. Wired entries are nonreplaceable entries, which cannot be overwritten by a TLB write random operation. Random entries can be overwritten.

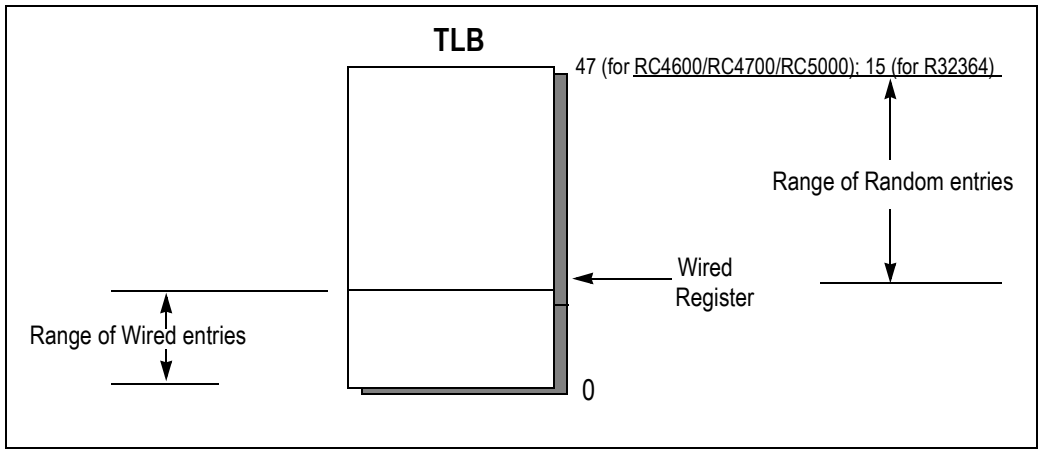


Figure 6.7 Wired Register Boundary

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to the value of its upper bound (see *Random* register, above). Figure 6.8 shows the format of the *Wired* register; Table 6.6, which follows the figure, describes the register fields.

Note that the RC32364 contains a 16 entry TLB and that the Wired register contains the capability of indicating up to 64 TLB entries. In programming, the value written to the Wired register must be within the valid range of the number of entries of the current device. For future versions of this device, the RC32364 implements additional bits.

Notes

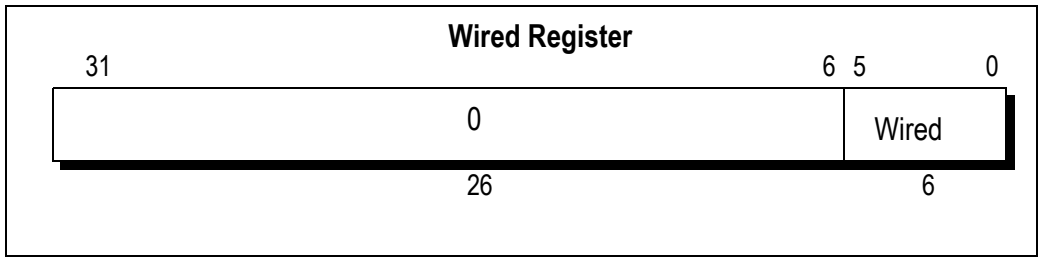
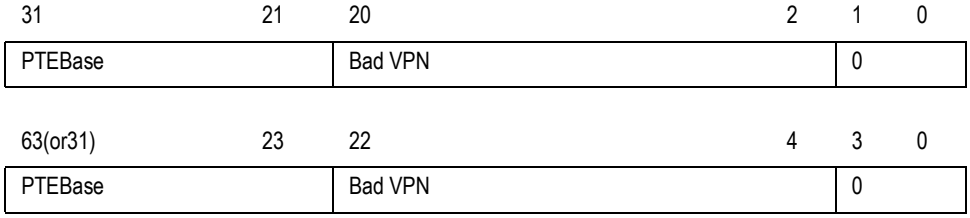


Figure 6.8 Wired Register

Field	Description
Wired	TLB Wired boundary (the number of wired TLB entries)
0	Reserved. Must be written as zeroes, and returns zeroes when read.

Table 6.6 Wired Register Fields

Context Register



- *PTEBase: a location which just stores what is put in it. In the "standard" refill handler, this will be the high-order bits of the starting address of a memory-resident page table.*
- *Bad VPN: following an addressing exception this holds the high-order bits of the address; exactly the same as the high-order bits of BadVaddr. However, if the system uses the "standard" TLB refill exception handling code the 32-bit value formed by Context is directly usable as a pointer to the memory-resident page table, considerably shortening the refill exception code.*
- *Fields marked 0: can be written with any value, but they will always read zero.*

XContext Register (RC4600/RC4700/RC5000 only)

The XContext register duplicates some of the information provided when 64-bit addressing is enabled in the BadVAddr register, and puts it in a form useful for a software TLB exception handler.

The XContext register is for use with the XTLB refill handler, which loads TLB entries for references to a 64-bit address space, and is included solely for operating system use. The operating system sets the PTE base field in the register, as needed. Normally, the operating system uses the XContext register to address the current page map, which resides in the kernel-mapped segment kseg3.

Figure 6.9 shows the format of the XContext register; Table 6.7, which follows the figure, describes the XContext register fields.

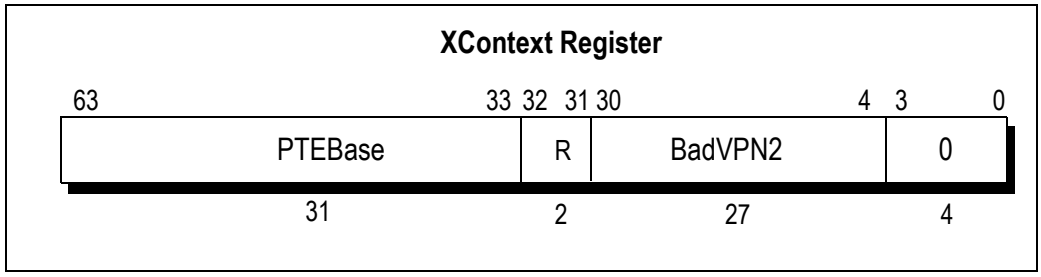


Figure 6.9 XContext Register Format

Notes

The 27-bit *BadVPN2* field has bits 39:13 of the virtual address that caused the TLB miss; bit 12 is excluded because a single TLB entry maps to an even-odd page pair. For a 4-Kbyte page size, this format may be used directly to address the pair-table of 8-byte PTEs. For other page and PTE sizes, shifting and masking this value produces the appropriate address.

Field	Description
BadVPN2	The *Bad Virtual Page Number*/2 field is written by hardware on a miss. It contains the VPN of the most recent invalidly translated virtual address.
R	The *Region* field contains bits 63:62 of the virtual address. 00_2 = user 01_2 = supervisor 11_2 = kernel.
PTEBase	The *Page Table Entry Base* read/write field is normally written with a value that allows the operating system to use the *Context* register as a pointer into the current PTE array in memory.

Table 6.7 XContext Register Fields

IBase Register (RC4650 only)

The *IBase* register provides the User Instruction address space Base address. Figure 6.10 shows the format of the *IBase* register; Table 6.8, which follows the figure, describes the *IBase* register fields.

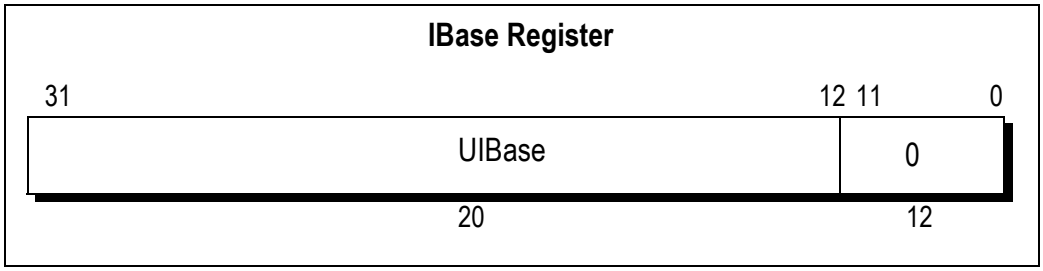


Figure 6.10 IBase Register

Field	Description
UIBase	Added to $vAddr_{31..12}$ for user space to get physical address
0	Reserved. Reads as 0, should be written as 0.

Table 6.8 IBase Register Fields

IBound Register (RC4650 only)

The *IBound* register provides the User Instruction address space Bound address. Virtual addresses greater than this value cause address error exceptions. Figure 6.11 shows the format of the *IBound* register; Table 6.9, which follows the figure, describes the *IBound* register fields.

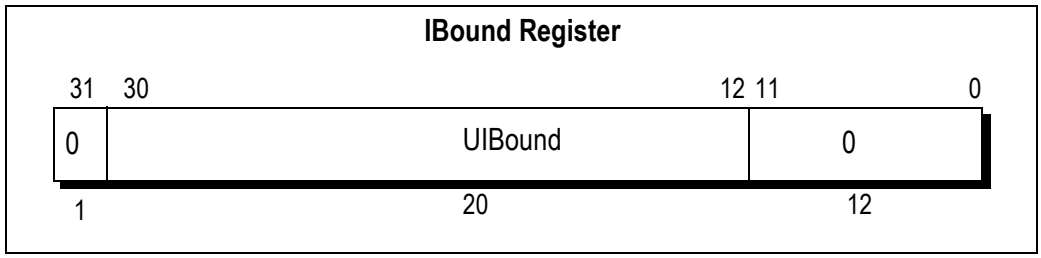


Figure 6.11 IBound Register

Notes

Field	Description
UIBound	Compared to $vAddr_{30..12}$ for user space to validate address
0	Reserved. Reads as 0, should be written as 0.

Table 6.9 IBound Register Fields

DBase Register (RC4650 only)

The *DBase* register provides the User Data address space Base address. Figure 6.12 shows the format of the *DBase* register; Table 6.10, which follows the figure, describes the *DBase* register fields.

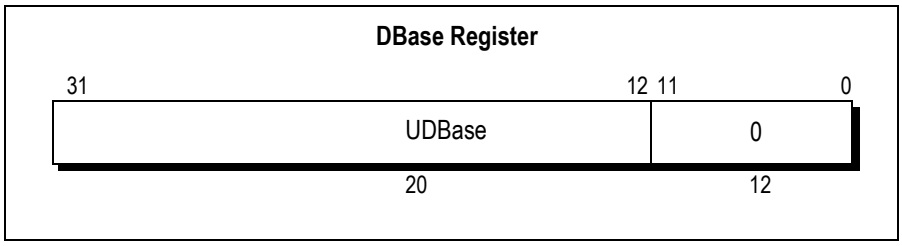


Figure 6.12 DBase Register

Field	Description
UDBase	Added to $vAddr_{31..12}$ for user space to get physical address
0	Reserved. Reads as 0, should be written as 0.

Table 6.10 DBase Register Fields

DBound Register (RC4650 only)

The *DBound* register provides the User Data address space Bound. Figure 6.13 shows the format of the *DBound* register; Table 6.11, which follows the figure, describes the *DBound* register fields.

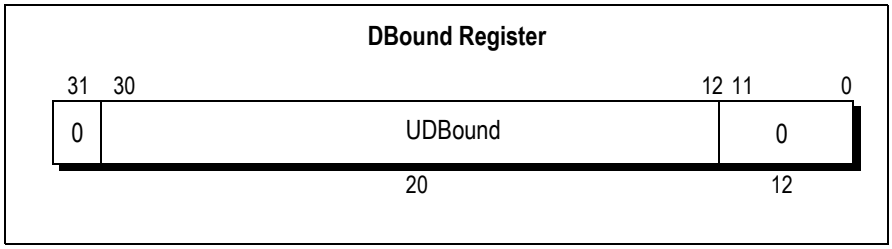


Figure 6.13 DBound Register

Field	Description
UDBound	Compared to $vAddr_{31..12}$ for user space to validate address
0	Reserved. Reads as 0, should be written as 0.

Table 6.11 DBound Register Fields

CAlg Register (RC4650 only)

The *CAlg* register is a read-write register that specifies the cache algorithm for each 512MB region of the virtual address space.

CAlg is initialized to 0x22233333 on Reset. Bits 31, 27, 23, 19, 15, 11, 7, and 3 are not implemented, and are reserved for future use. They read as zero and are ignored on write.

Notes

Figure 6.14 below shows the format of the *CAlg* register; Table 6.12, which follows the figure, describes the *CAlg* register fields.

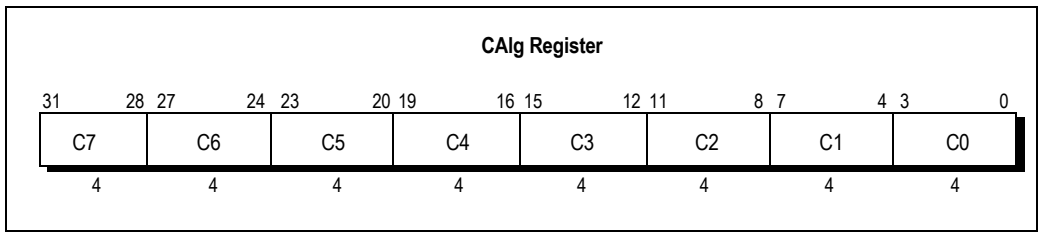


Figure 6.14 CAlg Register

The Cache algorithms are as follows:

0	Cached, non-coherent, write-through, no write-allocate
1	Cached, non-coherent, write-through, write-allocate
2	Uncached
3	Cached, non-coherent, write-back, write-allocate
4-15	Reserved

Field	Description
C0	Cache algorithm for 0x00000000 to 0x1FFFFFFF (part of useg/kuseg)
C1	Cache algorithm for 0x20000000 to 0x3FFFFFFF (part of useg/kuseg)
C2	Cache algorithm for 0x40000000 to 0x5FFFFFFF (part of useg/kuseg)
C3	Cache algorithm for 0x60000000 to 0x7FFFFFFF (part of useg/kuseg)
C4	Cache algorithm for 0x80000000 to 0x9FFFFFFF (k seg0)
C5	Cache algorithm for 0xA00000000 to 0xBFFFFFFF (k seg 1)
C6	Cache algorithm for 0xC0000000 to 0xDFFFFFFF (part of kseg2)
C7	Cache algorithm for 0xE0000000 to 0xFFFFFFFF (part of kseg2)

Table 6.12 CAlg Register Fields

TLB Control Instructions

The following instructions are not valid for the RC4650; however, they are not guaranteed to generate a trap in the RC4650 either.

tlbr Read TLB entry at index

tlbwi *Write TLB entry at index* -------The above two instructions move MMU data between the TLB entry selected by the *Index* register and the *EntryHi* and *EntryLo* registers.

tlbwr *Write TLB entry selected by Random* --------copies the contents of *EntryHi* & *EntryLo* into the TLB entry indexed by the *random* register. This saves time when using the recommended random replacement policy. In practice, *tlbwr* will be used to write a new TLB entry in a TLB refill exception handler; *tlbwi* will be used anywhere else.

Notes

tlbp *TLB lookup (probe)* --------- searches (probes) the TLB for an entry whose virtual page number and ASID matches those currently in *EntryHi*, and stores the index of that entry in the *index* register (*index* is set to a negative value if nothing matches). If more than one entry matches, anything might happen. Note: *tlbp* does not fetch data from the TLB, and the instruction following a *tlbp* must not be a load or store.

Programming to the TLB

TLB entries are set up by writing the required fields into *EntryHi* and *EntryLo* and using a *tlbwr* or *tlbwi* instruction to copy that entry into the TLB proper.

When handling a TLB refill exception, *EntryHi* has been set up automatically with the current ASID and the required VPN.

In the RC30xx, be careful not to create two entries that will match the same program address/ASID pair. If the TLB contains duplicate entries, an attempt to translate such an address, or probe for it, produces a fatal "TLB shutdown" condition (indicated by the TS bit in *SR* being set). This condition can only be cleared with a hardware reset. The RC4xxx/RC32364/RC5000 do not provide any detection or shutdown for multiple matches. The result is undefined for such a condition.

System software often won't need to read TLB entries at all. But if necessary, software can find the TLB entry matching some particular program address using *tlbp* to setup the *Index* register. Don't forget to save *EntryHi* and restore it afterwards because its ASID field is likely to be important.

Use a *tlbr* to read the TLB entry into *EntryHi* and *EntryLo.*

How Refills Occur

When a program makes an access to a mapped page for which no translation record is present, the CPU takes a TLB refill exception. The assumption is that system software is maintaining a large number of page translations and is using the TLB as a cache of recently-used translations; so the refill exception will normally be handled by finding a correct translation, installing it, and returning to user code.

To save time on user-program TLB refill exceptions:

- *refill exceptions on kuseg program addresses are vectored through a low-memory address used for no other exception; in the RC4600/RC4700/RC5000, there are two vectors—one for 32-bit space, the other for 64-bit space*
- *special exception rules permit the kuseg refill handler to risk a nested TLB refill exception on a kseg2 address.*

  The problem is that before an exception routine can itself suffer an exception it must first save the previous program state, represented by the EPC return address and some SR bits. This is helped out in the RC30xx by a hardware feature and a software convention:
  - *the KUo, IEo bits in the status register act as a third level of the processor-state stack, so that the CPU state already saved as a result of the kuseg refill exception can be preserved during the nested exception.*
  - *The kuseg refill handler copies EPC into the k1 register; the general exception code and kseg2 refill handler are then careful to preserve its value, enabling a clean return.*

Using ASIDs

By setting up TLB entries with a particular ASID setting and with the *EntryLo* G bit zero, those entries will only ever match a program address when the CPU's *ASID* register is set the same. This allows software to map up to 64 (RC30xx) or 256 (RC4xxx/RC32364/RC5000) different address spaces simultaneously, without requiring that the OS clear out the TLB on a context change.

In typical usage, new tasks are assigned an "un-initialized" ASID. The first time the task is invoked, it will presumably miss in the TLB, allowing the assignment of an ASID. If the system does run out of new ASIDs, it will flush the TLB and mark all tasks as "new". Thus, as each task is re-entered, it will be assigned a new ASID. This sequence is expected to happen infrequently if ever.

Notes

The Random Register and "Wired" Entries

The hardware offers no way of finding out which TLB entries have been used most recently. When the system needs to replace a mapping dynamically (using the TLB as a cache) the only practicable strategy is to replace an entry at random. The CPU makes this easy by maintaining the *Random* register, which counts (down) with every processor cycle.

However, it is often useful to have some TLB entries which are guaranteed to stay there unless explicitly removed. These may be useful to map pages which are known to be required very often; they are critical because they allow the system to map pages and *guarantee* that no refill exception will be generated on them.

The stable TLB entries are described as "wired" and on RC30xx family CPUs consist of TLB entries 0 through 7. There is nothing special about these entries; the magic is in the *Random* register, which never takes values 0-7 in the RC30xx; it cycles directly from 63 down to 8 before reloading with 63. So conventional random replacement leaves TLB entries 0 through 7 unaffected, and entries written there will stay until explicitly removed.

More flexibility is offered in the RC4600/RC4700/RC32364/RC5000 with the help of *Wired* register. Instead of the fixed number of entries (0 through 7) in the RC30xx, here you can specify the number of wired entries. If *wired* was set to "x", *random* will cycle through numbers 47 though "x", skipping numbers "x-1" through zero.

Memory Translation – Setup

The following code fragment initializes the TLB to ensure no match on any kuseg or kseg2 address. This is important, and is preferable to initializing with all "0"'s (which is a kuseg address, and which would cause multiple matches if referenced):

```
LEAF(mips_init_tlb)
        mfc0    t0,C0_ENTRYHI  # save asid
        mtc0    zero,C0_ENTRYLO# tlblo = !valid
        # do the above for both EntryLo regs in RC4600
        li      a1,NTLBID<<TLBIDX_SHIFT # index
        li      a0,KSEG1_BASE# tlbhi = impossible vpn

        .set noreorder
1:      subu    a1,1<<TLBIDX_SHIFT
        mtc0    a0,C0_ENTRYHI
        mtc0    a1,C0_INDEX
        bnez    a1,1b
        tlbwi                                   # BDSLOT
        .set    reorder

        mtc0    t0,C0_ENTRYHI  # restore asid
        j       ra
END(mips_init_tlb)
```

TLB Exception Sample Code

There are two examples provided. The first is written in C, and assumes that the OS provides a low-level handler which saves state, including copying the exception registers into an "xcpcontext'' structure, and dispatches through programmable tables to a C routine. Note that the actual bit-field locations inside a set of given registers of the same name in the RC30xx and RC4600/RC4700/RC32364/RC5000 are typically different. The following code is generic enough to work on both provided your #defines are correct for the processor of choice.

Basic Exception Handler

```
/* install C exception handler for TLB miss exception */
xcption (XCPTTLBMISS, tlbmiss);
```

Notes

```
...

#define VMPGSHIFT12/* convert address to page number */

tlbmiss (int code, struct xcptcontext *xcp)
{
 unsigned pfn = map_virt_to_phys (xcp->vaddr) >> VMPGSHIFT;
 unsigned vpn = xcp->vaddr >> VMPGSHIFT;
 unsigned asid = 0;

 /* write a random tlb (entryhi, entrylo) pair */
 /* mark it valid, global, uncached, and not writable/dirty */
 r3k_tlbwr ((vpn <<TLBHI_VPNSHIFT) | (asid <<TLBHI_PIDSHIFT),
          (pfn <<TLBLO_PFNSHIFT) | TLB_V | TLB_G | TLB_N);
 return 0;
}
```

The macro (or routine) *map_virt_to_phys*() which does the actual work, will be system dependent.

Fast kuseg Refill from Page Table

This routine implements the translation mechanism which the MIPS architects had in mind for user addresses in a Unix-like OS. It relies upon building a page table in memory, for each address space. The page table consists of a linear array of one-word entries, indexed by the VPN, whose format is matched to the bitfields of the *EntryLo* register.

Such a scheme is simple, but has one problem. Since each 4Kbytes of user address space takes 4 bytes of table space, the entire 2Gbyte user space needs a 2Mbyte table (in the RC30xx, for example), which is a large chunk of data. Moreover, most user address spaces are used at the bottom (for code and data) and at the top (for a downward growing stack) with a huge gap in between.

Inspired by Digital's VAX architecture, the solution that has been adopted is to locate the page table itself in virtual memory, in the kseg2 region, which neatly solves two problems at once:

- *saves physical memory, since the unused gap in the middle of the page table will never be referenced.*
- *provides an easy mechanism for remapping a new user page table when changing context, without having to find enough virtual addresses in the OS to map all the page tables at once.*

The MIPS architecture gives positive support to this mechanism in the form of the *Context* register. If the page table starts at a 1Mbyte boundary (since it is in virtual memory, any gap created won't use up physical memory space) and the *Context* PTEBase field is filled with the high-order bits of the page table starting address, then following a user refill exception the *Context* register will contain the address of the entry needed for the refill, with no further calculation.

The resulting routine looks like this:

```
        .set    noreorder
        .set    noat
xcpt_vecfastutlb:
        mfc0    k1,C0_CONTEXT
        mfc0    k0,C0_EPC       # mfc0 delay slot
        lw      k1,0(k1)        # may double fault (k0 = orig EPC)
        nop
        mtc0    k1,C0_ENTRYLO
        nop
        tlbwr
        jr      k0
        rfe
xcpt_endfastutlb:
        .set    at
        .set    reorder
```

Notes

Simulating Dirty Bits

An operating system providing a page for an application program to use often wants to keep track of whether that page has been modified since the OS last obtained it (perhaps from disc or network) or saved a copy of it. Non-modified pages are cheap to discard, since they can easily be replaced if required.

In OS parlance such modified pages are called “dirty” and the OS must take care of them until the application program exits, or the dirty page saved away to backing store.

To help out with this process it is common for CISC CPUs to maintain a bit in the memory-resident page table indicating that a write operation to the page has occurred.

The MIPS CPU does not directly implement this feature, even in the TLB entries. The “D” bit of the page table (found in the *EntryLo* register) is a write-enable, and is of course used to flag read-only pages.

To simulate “dirty” bits, the OS should mark new pages (in the page table) with D clear. Since the CPU will consider that page “write-protected”, a trap will result when the page is first modified; system software can identify this as a legitimate write but use the event to set a “modified” bit in the memory resident tables (it will also want to set the D bit in the TLB entry so that the write can proceed, but since TLB entries are randomly and unpredictably replaced this would be useless as a way of remembering the modified state).

Use of TLB in Debugging

In systems which do not require the TLB for normal execution, it still may prove useful during initial system debug. Although its use for this purpose will be system dependent, some general ideas follow:

- *To hit a “trap” when software “wanders into the weeds” (e.g. makes mysterious references or instruction fetches from strange memory locations), software can initialize the TLB with only those mappings which correspond to valid program addresses. Thus, a TLB trap will occur in the exact instruction which makes the reference, and full processor state will be visible.*
- *To identify which task or subroutine is modifying a particular memory location, that location can be “write-protected”, generating a trap on store.*

The TLB may have one additional consequence in debugging. In a virtual memory OS, the actual physical memory location of a task (or even of portions of the OS) can move around as memory is paged. This can make low-level debugging difficult, since one cannot set a logic analyzer to trap on the right physical address.

To resolve this situation, software can utilize a system specific “NOP” instruction. Recall that updates to the zero register $0 will be ignored; software can use this fact to generate a specific NOP instruction for the reference in question; the logic analyzer can then be used to search for this particular instruction fetch, correctly identifying the current virtual to physical mapping.

TLB Management Utilities

The following routines implement the most common TLB management functions. These code fragments are taken from IDT/sim.

```
/*  Functions dealing with the TLB.
**          Use resettlb() defined here and called from excepthand.c
**          to initialize tlb.
*/

/*
**          idttlb.s - fetch the registers associated with and the contents
**                     of the tlb.
**
*/
#include "iregdef.h"
#include "idtcpu.h"
#include "idtmon.h"
```

Notes

```
        .text

/*
** ret_tlblo -- returns the 'entrylo' contents for the TLB
**      'c' callable - as ret_tlblo(index) - where index is the
**      tlb entry to return the lo value for - if called from assembly
**      language then index should be in register a0.
*/
FRAME(ret_tlblo,sp,0,ra)
        .set    noreorder
        mfc0    t0,C0_SR        # save sr
        nop
        and     t0,~SR_PE       # dont inadvertantly clear PE
        mtc0    zero,C0_SR      # clear interrupts
        mfc0    t1,C0_TLBHI     # save pid
        sll     a0,TLBINX_INXSHIFT# position index
        mtc0    a0,C0_INX       # write to index register
        nop
        tlbr                            # put tlb entry in entrylo and hi
        nop
        mfc0    v0,C0_TLBLO     # get the requested entry lo
        mtc0    t1,C0_TLBHI     # restore pid
        mtc0    t0,C0_SR        # restore status register
        j       ra
        nop
ENDFRAME(ret_tlblo)

/*
** ret_tlbhi -- return the tlb entry high content for tlb entry
**                      index
*/
FRAME(ret_tlbhi,sp,0,ra)
        mfc0    t0,C0_SR        # save sr
        nop
        and     t0,~SR_PE
        mtc0    zero,C0_SR      # disable interrupts
        mfc0    t1,C0_TLBHI     # save current pid
        sll     a0,TLBINX_INXSHIFT# position index
        mtc0    a0,C0_INX       # drop it in C0 register
        nop
        tlbr                            # read entry to entry hi/lo
        nop
        mfc0    v0,C0_TLBHI     # to return value
        mtc0    t1,C0_TLBHI     # restore current pid
        mtc0    t0,C0_SR        # restore sr
        j       ra
        nop
ENDFRAME(ret_tlbhi)

/*
** ret_tlbpid() -- return tlb pid contained in the current entry hi
*/
FRAME(ret_tlbpid,sp,0,ra)
        mfc0    v0,C0_TLBHI     # fetch tlb high
        nop
        and     v0,TLBHI_PIDMASK# isolate and position
        srl     v0,TLBHI_PIDSHIFT
        j       ra
        nop
ENDFRAME(ret_tlbpid)

/*
** tlbprobe(address, pid) -- probe the tlb to see if address is currently
**                                      mapped
**      a0 = vpn  - virtual page numbers are 0=0 1=0x1000, 2=0x2000...
**                      virtual page numbers for the RC3000 are in
```

Notes

```
**                              entry hi bits 31-12
**      a1 = pid  - this is a process id ranging from 0 to 63
**                  this process id is shifted left 6 bits and or'ed into
**                  the entry hi register
**      returns an index value (0-63) if successful -1 -f not
*/
FRAME(tlbprobe,sp,0,ra)
        mfc0    t0,C0_SR        /* fetch status reg */
        and     a0,TLBHI_VPNMASK/* isolate just the vpn */
        and     t0,~SR_PE       /* don't inadvertantly clear pe */
        mtc0    zero,C0_SR
        mfc0    t1,C0_TLBHI
        sll     a1,TLBHI_PIDSHIFT/* possition the pid */
        and     a1,TLBHI_PIDMASK
        or      a0,a1                   /* build entry hi value */
        mtc0    a0,C0_TLBHI
        nop
        tlbp                            /* do the probe */
        nop
        mfc0    v1,C0_INX
        li      v0,-1
        bltz    v1,1f
        nop
        sra     v0,v1,TLBINX_INXSHIFT/* get index positioned for return */
1:
        mtc0    t1,C0_TLBHI     /* restore tlb hi */
        mtc0    t0,C0_SR        /* restore the status reg */
        j       ra
        nop
ENDFRAME(tlbprobe)

/*
** resettlb(index) Invalidate the  TLB entry specified by index
*/
FRAME(resettlb,sp,0,ra)
        mfc0    t0,C0_TLBHI     # fetch the current hi
        mfc0    v0,C0_SR        # fetch the status reg.
        li      t2,K0BASE&TLBHI_VPNMASK
        and     v0,~SR_PE       # dont inadvertantly clear PE
        mtc0    zero,C0_SR
        mtc0    t2,C0_TLBHI     # set up tlbhi
        mtc0    zero,C0_TLBLO
        sll     a0,TLBINX_INXSHIFT
        mtc0    a0,C0_INX
        nop
        tlbwi                           # do actual invalidate
        nop
        mtc0    t0,C0_TLBHI
        mtc0    v0,C0_SR
        j       ra
        nop
ENDFRAME(resettlb)

/*
** Setup TLB entry
**
** map_tlb(index, tlbhi, phypage)
**      a0  =  TLB entry index
**      a1  =  virtual page number and PID
**      a2  =  physical page
*/
FRAME(map_tlb,sp,0,ra)

        sll     a0,TLBINX_INXSHIFT
        mfc0    v0,C0_SR        # fetch the current status
        mfc0    a3,C0_TLBHI     # save the current hi
```

Notes

```
        and     v0,~SR_PE       # dont inadvertantly clear parity

        mtc0    zero,C0_SR
        mtc0    a1,C0_TLBHI     # set the hi entry
        mtc0    a2,C0_TLBLO     # set the lo entry
        mtc0    a0,C0_INX       # load the index
        nop
        tlbwi                           # put the hi/lo in tlb entry indexed
        nop
        mtc0    a3,C0_TLBHI     # put back the tlb hi reg
        mtc0    v0,C0_SR        # restore the status register
        j       ra
        nop
ENDFRAME(map_tlb)


/*
** Set current TLBPID. This assumes PID is positioned correctly in reg.
**                              a0.
*/
FRAME(set_tlbpid,sp,0,ra)

        sll     a0,TLBHI_PIDSHIFT
        mtc0    a0,C0_TLBHI
        j       ra
        nop
        .set    reorder
ENDFRAME(set_tlbpid)
```

Notes

Reset Initialization

Notes

Starting Up

A CPU reset is almost the same as an exception: *EPC* points to the instruction being executed when reset was detected, and most registers are unchanged. However, reset disrupts normal operation and a register being loaded or a cache location being stored to or refilled at the moment reset occurred may be undefined.

It is possible to use the preservation of state through reset to implement some useful post-mortem debugging, but the system hardware needs to help; the CPU cannot tell whether reset occurred to a running system or from power-up.

The RC3xxx CPU responds to reset with a jump to program location 0xBFC0 0000 and the RC4xxx/RC5000 to location 0xFFFF FFFF BFC0 0000. These correspond to the physical address 0x1FC0 0000 in the unmapped and uncached *kseg1* region (*ckseg1* in 64-bit mode).

Following reset, enough state is defined so that the CPU can execute uncached instructions. Virtually nothing else is defined:

- *Only a few state bits are guaranteed in SR; the CPU is in kernel mode (KUc = 0), interrupts are disabled (IEc = 0), exceptions will vector through the uncached entry points (BEV = 1); the TS bit is guaranteed in RC30xx family CPUs (it will be cleared to 0 if the CPU has MMU hardware ("E" versions), set to 1 for base versions). In the RC4xxx/RC5000/RC32364, the SR bit is cleared in case of reset and the SR bit is set to 1 in case of soft reset or NMI. The ERL bit is also set to 1 in the RC4xxx/RC5000/RC32364. In the RC30xx, the D-cache may or may not be isolated (IsC = 1), so software cannot rely on data loads and stores working, even to uncached space, without first initializing this field to '0'.*
- *In the RC4xxx/RC5000/RC32364, after a reset (and not a soft reset), the Random register is initialized to the value of its upper bound and the Wired register is initialized to 0. Some of the Config register bits are initialized from the boot-time mode stream.*
- *The cache may be in a random, undefined state; so a cached load might return uninitialized cache data without reading memory.*
- *For RC4600/RC4700/RC5000/RC32364 and RC30xx "E" versions, the TLB may be in a random state and must not be accessed or referenced until initialized (the hardware has only minimal protection against the possibility that there are duplicate matches in the TLB, and the result will be a TLB shutdown which can be amended only by a further reset).*

The traditional start-up sequence is:

- *Branch to the main ROM code. The branch represents a very simple test that the CPU is functioning and successfully reading instructions.*
  - *Test equipment which can track the addresses of CPU reads and writes will show the CPUs uncached instruction fetches from reset; if the CPU starts up and branches to the right place, then evidence is strong that the CPU is getting mostly-correct data from the ROM.*
- *Set the status register to some known state. Now software can load and store reliably in uncached space.*
- *Software will probably have to run without using memory until it has initialized and checked on the integrity of RAM. This will be slow (the CPU is still running uncached from ROM), so it may be desirable to constrain the initialization and check function to the data which the ROM program itself will use.*
- *The system will probably have to make some contact with the outside world (a console port or diagnostic register) so it can report any problem with the initialization process.*
- *Software can now assign some stack space and set up enough registers to be able to call a stan-*

Notes

dard C routine.

- *Now the caches can be initialized, and the CPU can be used for more strenuous tasks. Some systems can run code from ROM cached, and some can't; the CPU can only cache instructions from a memory which is capable of supplying data in 4-word bursts, and the ROM subsystem may or may not oblige.*

The following start-up code for the **RC30xx** is taken from IDT/sim:

```
/*
** Copyright 1989 Integrated Device Technology, Inc.
**          All Rights Reserved
**
** sample initialization (reset) code for the RC30xx
*/

#include "excepthdr.h"
#include "iregdef.h"
#include "idtcpu.h"
#include "idtmon.h"
#include "under.h"

/*-------------------------------------------------------------------
**          external declarations - defined in the module shown in
**                                  parenthesis
*-------------------------------------------------------------------*/
.extern   mem_start,4  /*start loc. for mem test */
.extern   mem_end,4  /*end loc. for mem test */
.extern parity_error,4  /* global parity error count (idtglobals.c) */
.extern status_base,4  /* contains value to be loaded into status */
                              /* register on startup */
.extern fp_int_line,4  /* fpu external interrupt line */
.extern fp_int_num,4  /* fpu external interrupt number */

          .text
FRAME(start,sp,0,ra)
          .set      noreorder
          li        v0,SR_PE|SR_CU1# enable coproc 1 clear parity error and set
          mtc0      v0,C0_SR          # state unknown on reset
          mtc0      zero,C0_CAUSE # clear software interrupts

     # check to see if RC3041
          mfc0      t0, CO_PRID
          nop
          li        t2, 0x00000700   # RC3041 has rev no 0x00000700
          bne       t0, t2,not41

          # RC3041 specific initialization code here

          # load appropriate values in busctrl and portsize registers.
          # disable coprocessor 1
          j         commcod
not41:
          # check to see if RC3081

          li        t3,0xaaaa5555
          mtc1      t3, $f0             #put 0xaaaa5555 in f0
          mtc1      zero, $f1# 0 in f1
          mfc1      t0, $f0
          mfc1      t4, $f1             # read registers back
          bne       t0, t3, its51# no FPA, must be 3051(52)
          bne       t4, zer0, its51 # no FPA, must be 3051(52)

          # RC3081 specific initialization code here
          j         commcod
```

Notes

```
its51:
        # RC3051 specific initialization here
        # disable coprocessor 1
commcod:
        # code common to all processors
        li      v0,K1BASE # verify that ram can be accessed
        li      t0,0xaaaa5555
        sw      t0,0(v0)
        sw      zero,4(v0)# put a different pattern on bus
        lw      t1,0(v0)
        nop
        beq     t1,t0,2f  # is memory accessable
/* memory not accessable, hang here, no point in proceeding */
1:      nop
        b       1b
        nop

2:      li      t0,-1
        sw      t0,8(v0)
        sw      zero,4(v0)
        lw      t1,8(v0)
        nop
        bne     t0,t1,1b
        nop
        .set    reorder
        sw      zero,parity_error# clear parity error count
        jal     initmem                 # initializes sp
        jal     initialize          # initialize memory and tlb
        j       yourcode
ENDFRAME(start)

/*
** initmem -- config and init cache, clear prom bss
** clears memory between PROM_STACK-0x2000 and PROM_STACK-4 inclusive
*/
#define INITMEMFRM ((4*4)+4)
FRAME(initmem,sp, INITMEMFRM, ra)
        la      v0,_fbss # clear bss
        la      v1,end          # end of bss


        .set    noreorder
1:      sw      zero,0(v0)/* clear bss */
        bltu    v0,v1,1b
        add     v0,4


/*
**      Initialize stack
*/
        add     v1,v0,P_STACKSIZE/* end of prom stack */
        sw      v1,mem_start
        sub     v1,v1,(4*4)
        sw      v1,fault_stack    /* top of fault stack */
        subu    sp,v1,P_STACKSIZE/4/* monitor stack top */
        subu    sp,INITMEMFRM
1:      sw      zero,0(v0)
        bltu    v0,v1,1b
        add     v0,4
        sw      ra,INITMEMFRM-4(sp)
        .set    reorder

        jal     config_cache    /* determine cache sizes */
        jal     flush_cache     /* flush cache */
        lw      ra,INITMEMFRM-4(sp)
        addu    sp,INITMEMFRM
```

Notes

```
        j       ra
ENDFRAME(initmem)

/*
** initialize -- initializes memory and tlb
*/
#define  INITFRM ((4*4)+4)
FRAME(initialize,sp, INITFRM,ra)
        subu    sp,INITFRM
        sw      ra,INITFRM-4(sp)
        jal     init_io                 /* initialize io */
        jal     init_memory     /* initialize memory and tlb */
        lw      ra,INITFRM-4(sp)
        addu    sp,INITFRM
        j       ra
ENDFRAME(initialize)
```

The following start-up code for the **RC4xxx/RC5000/RC32364** is from IDT/sim:

```
/*
** Copyright 1993, 1995 Integrated Device Technology, Inc.
**      All Rights Reserved
**
** sample initialization (reset) code for the RC4xxx
*/
#include "excepthdr.h"
#include "iregdef.h"
#include "idtcpu.h"
#include "idtmon.h"
#include "under.h"

/*-------------------------------------------------------
** external declarations - defined in the module shown in
** parenthesis
*-------------------------------------------------------*/
.extern  mem_start,4  /*start loc. for mem test (idtglobals.c) */
.extern  mem_end,4  /*end loc. for mem test (idtglobals.c) */
.extern  user_int_fast,4  /* pointer to user fast int rt (idtglobals.c)*/
.extern  user_int_normal,4 /* pointer to user normal int rt.(idtglobals.c) */
.extern parity_error,4  /* global parity error count (idtglobals.c) */
.extern debug_mode,4  /* remode debugger flag (idtglobals.c) */
.extern idb,4       /* idtc remode debugger flag (idbdebug.c) */
.extern idb_remote,4  /* remode file access flag (idbdebug.c) */
.extern status_base,4  /* contains value to be loaded into status */
                                /* register on startup (idtglobals.c) */
.extern _fbss,4     /* this is defined by the linker */
.extern end,4       /* this is defined by the linker */
.extern cputype,4  /* types #defined in  ../COMMON/c_asm/idtglobals.c  */

        .text
/*-------------------------------------------------------
** prom entry point table
*-------------------------------------------------------*/
FRAME(start,sp,0,ra)


        .globl  __start
__start:
idtstart:
        .set    noreorder
        li      v0,SR_CU1|SR_DE# first clear ERL & enable FPA
        mtc0    v0,C0_SR                # state unknown on reset
        mtc0    zero,C0_CAUSE           # clear software ints
        li      v0,CFG_C_NONCOHERENT# init default cache mode
        mtc0    v0,C0_CONFIG
        nop
```

Notes

```
        nop

/* the memory system may need up to 120us to start up... */
        li      v0,128          /* ~256us */
1:      bne     v0,zero,1b
        subu    v0,1            # BDSLOT
#if defined(P4000_RAM)
/* copy .text section to RAM location */
      .set noreorder
      la     v0, _ftext
      la     v1, _etext
      li     t0, 0xbfc00000
      or     t0,v0
      addiu  v0,-4
    1:
      lw     t2,0(t0)
      addiu  v0,4
      sw     t2,0(v0)
      blt    v0,v1,1b
      addiu  t0,4
      la     v0,1f
      la     v1,_ftext
      or     v0,v0,v1
/* Shoot from RAM location */
      jr     v0
      nop
    1:
      .set   reorder
#endif


        li      v0,K1BASE # verify that ram can be accessed
        li      t0,0xaaaa5555
        sw      t0,0(v0)
        sw      zero,M_BUSWIDTH(v0)#different pattern on bus
        lw      t1,0(v0)
        nop
        beq     t1,t0,2f  # is memory accessable
        nop
/* memory not accessable */
1:      b       1b
        nop

2:      li      t0,-1
        sw      t0,M_BUSWIDTH*2(v0)
        sw      zero,M_BUSWIDTH(v0)
        lw      t1,M_BUSWIDTH*2(v0)
        nop
        bne     t0,t1,1b
        nop
        .set    reorder
        jal     _init_cache     # lowlevel board initialisation
        jal     initmem                 # initializes sp
        sw      zero,parity_error# clear parity error count
        jal     init_dev_tab    # moves device table to RAM
        jal     initialize
        jal     init_cmd_tab    # moves command table to RAM
        jal     clear_brkpts
        jal     main
        j       promexit
ENDFRAME(start)


/*
** prominit -- reinitialize monitor command entry pt
*/
```

Notes

```
FRAME(prominit,sp,0,ra)
        .set    noreorder
        li      v0,SR_CU1|SR_DE# first clear ERL and enable FPA
                        # not proper, but fixes init command on
                                # 64-bit RISController
        mtc0    zero,C0_CAUSE # clear software interrupts
        mtc0    v0,C0_SR
        .set    reorder

        jal     _init_cache # lowlevel board initialisation

        jal     initmem                 # initmem initializes sp
        jal     init_dev_tab
        jal     initialize
        jal     init_cmd_tab
        jal     clear_brkpts
        jal     main
        j       promexit        # should not get here
ENDFRAME(prominit)

/*
** promexit()
**
** client programs may return here to reenter the prom monitor
** It assumes that the proms bss area has not been trashed.
** The normal area for the prom bss area is 0xa0000100 to
** 0xa0010000. The prom stack is reinitialized so no restraints
** are imposed on its condition.
*/
FRAME(promexit,sp,0,ra)
        .set    noreorder
        lw      v0,status_base
        mtc0    zero,C0_CAUSE # clear software interrupts
        mtc0    v0,C0_SR        # back to a known sr
        .set    reorder
        DISPLAY('E','X','I','T')
        sw      zero,user_int_fast
        sw      zero,user_int_normal
        li      v0,MODE_MONITOR
        la      v1,client_regs
        sreg    v0,R_MODE*R_SZ(v1)
        la      v1,end          # end of bss
        addu    v1,v1,P_STACKSIZE-(4*4)+7 /*end prom stack */
        and     v1,~7           /* rounded up to 8 byte boundary */
        sw      v1,fault_stack  /* top of fault stack */
        subu    sp,v1,P_STACKSIZE/4/* monitor stack top */

        jal     config_cache    # determine cache sizes
        jal     flush_cache     # flush cache
        jal     move_exc_code
        la      v0,1f   # switch to cached space if so linked
        jr      v0
1:
        lw      v0,idb
        bne     v0,zero,1f
        jal     init_io
        DISPLAY('P','R','O','M')
        jal     main
        li      a0,0
        j       promexit        # should not get here
1:
        jal     reenter_idb
ENDFRAME(promexit)

/*
** initmem -- config and init cache, clear prom bss
```

Notes

```
** clears memory between PROM_STACK-0x2000 and PROM_STACK-4 inclusive
*/
#define INITMEMFRM ((4*4)+8)
FRAME(initmem,sp, INITMEMFRM, ra)
        la      v0,_fbss # clear bss
        la      v1,end          # end of bss

        .set    noreorder
1:      sw      zero,0(v0)/* clear bss */
        bltu    v0,v1,1b
        addu    v0,4

        /* Get memory limit from given sp */
        and     v0,sp,K0SIZE-1
        sw      v0,mem_size
        addu    v0,-4
        or      v0,K1BASE
        sw      v0,mem_end

/*
**      Initialize stack
*/
        addu    v1,v0,P_STACKSIZE+7/* end of prom stack */
        and     v1,~7           /* rounded up to 8 byte boundary */
        sw      v1,mem_start
        subu    v1,v1,(4*4)
        sw      v1,fault_stack  /* top of fault stack */
        subu    sp,v1,P_STACKSIZE/4/* monitor stack top */
        subu    sp,INITMEMFRM
1:      sw      zero,0(v0)
        bltu    v0,v1,1b
        addu    v0,4
/*
**      check to see if an fpu is really plugged in
*/
        sw      ra,INITMEMFRM-4(sp)

        li      v0,R4600        #default = RC4600(64-bit RISController)

        mfc0    t0,C0_PRID
        nop
        nop
        andi    t0,0xff0
        li      t1,0x00000440   # RC4400 ?
        bne     t0,t1,no44
        nop

        li      v0,R4400        # it is 4400

        j       st_typ
        nop

no44:
        mfc0    t0,C0_PRID
        nop
        nop
        andi    t0,0xff00
        li      t1,0x00002200#4650 ?
        bne     t0,t1,st_typ#must be 64-bit RISController
        nop
        li      v0,R4650#its RC4650
st_typ:
        sw      v0,cputype

#ifndef P3
        mfc0    t2,C0_SR
```

Notes

```
        nop
        li      t3,SR_CU1
        or      t3,t2,t3
        mtc0    t3,C0_SR
#else
 /*     In P3, we need to set FR bit in SR so we can do
        lwc1/swc1 a0,fp1  without exception */
        mfc0    t2,C0_SR
        nop
        li      t3,SR_CU1|SR_FR
        or      t2,t2,t3
        mtc0    t2,C0_SR
#endif

        li      t3,0xaaaa5555# put a's and 5's in t3
        mtc1    t3,fp0              # put 0xaaaa5555 in f0
        nop
        mtc1    zero,fp2# put zero in f1
        nop
        mfc1    t0,fp0
        nop
        mfc1    t1,fp2
        nop

        beq     t0,t3,1f  # br if fpu
        nop
        beq     t1,zero,1f# br if fpu
        nop
        li      t3,~SR_CU1# set fpu unusable
        and     t2,t3
1:
        sw      t2,status_base
        mtc0    t2,C0_SR
        nop
        nop
        .set    reorder

        la      v1,client_regs
        li      v0,MODE_MONITOR
        sreg    v0,R_MODE*R_SZ(v1)
        DISPLAY('C','F','C','H') /* UK */
#ifndef P3
        jal init_tlb
#endif
        jal     config_cache    /* determine cache sizes */
        jal     flush_cache     /* flush cache */
        lw      ra,INITMEMFRM-4(sp)
        addu    sp,INITMEMFRM
        j       ra
ENDFRAME(initmem)

/*
** initialize -- initialize prom state
*/
#define  INITFRM ((4*4)+8)
FRAME(initialize,sp, INITFRM,ra)
        subu    sp,INITFRM
        sw      ra,INITFRM-4(sp)
        jal     move_exc_code
        jal     init_io                 /* initialize io */
        li      v0,MODE_MONITOR
        la      v1,client_regs
        sreg    v0,R_MODE*R_SZ(v1)
        sw      zero,user_int_fast
        sw      zero,user_int_normal
        sw      zero,debug_mode
```

Notes

```
        sw      zero,idb
        sw      zero,idb_remote
        jal     init_memory     /* initialize memory and tlb */
        lw      v0,mem_size
        addu    v0,-4                   /* last word of memory */
        or      v0,K1BASE
        sw      v0,mem_end      /* command in the diagnostic mode */
        la      v0,1f
        j       v0
1:
        lw      ra,INITFRM-4(sp)
        addu    sp,INITFRM
        j       ra
ENDFRAME(initialize)
```

Probing and Recognizing the CPU

The *PRId* register Imp and Rev fields is useful for the first check in the RC30xx family, to differentiate the RC3041 from other family members. The Imp field will be "3" for the RC3051, RC3052, RC3071 and RC3081 (indicating that their control register sets are identical to the RC3000A), but "7" for the RC3041, which has no MMU and assigns some control register numbers differently. Diagnostic software should also make the "Rev" field visible.

The *PRId* register Imp field values are "0x20" for RC4600, "0x21" for RC4700, "0x22" for RC4650, and "0x26" for the RC32364.

Software can investigate the presence of FPA hardware. The "official" technique is to set CU1 in *SR* to enable co-processor 1 operations, and use a *cfc1* instruction from co-processor 1 register 0, which is defined to hold the revision ID. A non-zero value in bits 15-8 indicates the presence of FPA hardware; the value "3" is standard for the FPA type which partners the RC3000 CPU.

In the RC4xxx family the RC4600's floating point implementation register (FEIR) is 0x20; the RC4700's is 0x21; the RC4650's is 0x22 in the Imp field; the RC5000's is 0x23. The Imp fields in the FEIR for the RC4xxx/RC5000 are as follows:

- *RC4600:0x20*
- *RC4650: 0x22*
- *RC4700: 0x20*
- *RC5000: 0x23*

Remember to reset CU1 in *SR* afterwards. You may wish to follow up by confirming that it is possible to store and retrieve data from the FPA registers.

The size of the on-chip caches can be determined, as described in chapter 5. The programmer can NOT assume the cache sizes based on the value of the PrId register; instead, the cache sizes must be explicitly measured.

The test for the presence of a TLB in an RC30xx family CPU is that the TS bit will be clear in *SR* following a hardware reset.

It is often useful to work out the system clock rate. This can be accomplished by running a loop of known length, cached, which will take a fixed large number of CPU cycles, and comparing with "before" and "after" values of a counter which increments at known speed.

Printing out the CPU type, clock rate and cache sizes as part of a sign-on message may be useful.

Bootstrap Sequences

Start-up code suffers from the clash of two opposing but desirable goals:

- *Make minimal assumptions about the integrity of the hardware, and attempt to check each sub-system before using it (think of climbing a ladder and trying to check each rung before putting weight on it);*

Notes

- *Minimize the amount of tricky assembler code. Bootstrap sequences are almost never performance-sensitive, so an early change to a high-level language is desirable. But high-level language code tends to require more subsystems to be operational.*

After basic initialization (like setting up *SR* so that the CPU can at least perform loads and stores) the major question is how soon read/write memory is available to the program, which is essential for calling functions written in C.

Software has an option here. IDT's CPUs all have data cache on chip, and it is reasonable to regard on-chip resources as the lowest rungs on the ladder. The data cache can provide enough storage for C functions during bootstrap; memory might be read or written, but provided software uses less than a cache-size chunk of memory space it will never need to read memory data back from main memory.

Starting Up an Application

To be able to start a C application the system needs:

- *Stack space: assign a large enough piece of writable memory and initialize the sp register to its upper limit (aligned to an 8-byte boundary). Working out how large the stack should be can be difficult, so a large guess helps.*
- *Many systems determine the amount of system RAM available and assign the stack to the top of physical RAM. This is the technique used by IDT/sim. With such a strategy, the stack can have as much RAM as is available in the system, after the program.*
- *Initialized data: normally the C data area is initialized by the program loader to set up any variables which have been allocated values. Some compilation systems permit read-only data (implicit strings and data items declared const) to be managed in a separate "segment" of object code and put into ROM memory.*

  Initialized writable data can be used only if the compilation system and run-time system co-operate to arrange to copy writable data initialization from ROM into RAM. IDT/sim provides code which does this for the IDT/c and MIPS compilers.
- *Zeroed data (bss): in C all static and extern data items which are not explicitly initialized will be set up with a zero value. The compilation system may provide a routine for use at run time which zeroes the data segments.*
- *global pointer initialization: some compilation systems use the gp register for more efficient access to global variables. If the system software is compiled with this option, the OS must set the register to the right value.*
- *Extra effort needed: routines which may cause non-fatal exceptions require more run-time support. In particular, software should be aware that the architecture permits the FPA to abort an instruction with the "illegal pocked" trap when confronted with legal (but unusual) operand values (see the chapter on FPA architecture, later in this manual). Many ordinary arithmetic operations will produce an exception if they overflow.*

Floating Point Co-processor

Notes

In the RC30xx family, the RC3081 contains the 3010A FPA device, which provides a combination of large caches, high-performance integer and floating-point computation.

The RC4600/RC4700 offers complete 64-bit support, doubling the number of registers as compared to the RC30xx. The RC4700 has a multiply unit more powerful than that in the RC4600. While any floating point multiply in the RC4600 takes 8 cycles, the RC4700 takes 4/5 cycles for single/double precision multiply. In the RC4xxx, integer multiply / divide are also performed in the floating point unit. Note that the terms FPA and FPU are used interchangeably in this chapter. The RC4700 performs integer multiply 2 cycles quicker than the RC4600.

The RC4650 offers only single precision floating point support in its FPU. Double precision floating point math, if needed, must be performed in software.

What is Floating Point?

This section describes the various components of the data (always using the same bit-arrangement as does the MIPS implementation) and what they mean. Many readers will feel familiar with these concepts already; however, this section can still prove useful in providing insight to the MIPS treatment of these concepts.

Scientists wanting to write numbers which may be very large or very small are used to using exponential notation; so the distance from Earth to the Sun is:

$$93 \times 10^6 \text{miles}$$

The number is defined by “93”, the *mantissa*[1], and “6”, the *exponent*. Of course the same distance can be written:

$$9.3 \times 10^7 \text{miles}$$

Numerical analysts like to use the second form; a decimal exponential with a mantissa between 1.0 and 9.999... is called *normalized*. The normalized form is useful for computer representation, since it doesn’t require separate information about the position of the decimal point.

Floating point numbers are an exponential form, but base 2, not base 10. Not only are the mantissa and exponent held as binary fields, but the number is formed differently. The distance quoted above is:

$$1.385808 \times 2^{26} \text{miles}$$

The mantissa can be expressed as a binary “decimal”, which is just like a real decimal:

$$1.385808 = 1 + 3 \times 1/10 + 8 \times 1/100 + 5 \times 1/1000 + \ldots$$

is the same value as binary:

$$1.0110001 = 1 + 0 \times 1/2 + 1 \times 1/4 + 1 \times 1/8 + \ldots$$

The IEEE 754 Standard and its Background

Floating point deals with the approximate representations of numbers (similar to decimals); early computer implementations differed in the details of their behavior with very small or large numbers. This meant that numerical routines, identically coded, might behave differently. In some sense these differences shouldn’t have mattered; systems would only produce different answers in circumstances where no implementation could really produce a “correct” answer.

1. The mantissa may also be called “the fractional part” or “fraction”

Notes

Numerical routines are hard to prove correct. Small differences in values could accumulate and could mean, for example, that a routine relying on repeated approximation might converge to the correct result on one CPU, and fail to do so on another.

The ANSI/IEEE Std 754–1985 IEEE Standard for Binary Floating-Point Arithmetic defined standard floating-point representations, operations, and results for program portability. This standard defines exactly what result will be produced by a small class of basic operations, even under extreme situations, ensuring that programmers can obtain identical results from identical inputs regardless of the machine used.

The operations regulated by IEEE 754 include every operation that any MIPS RC3000 FPA can do in hardware as well as some that must be emulated by software.

The IEEE 754 specifies:

- *Rounding and precision of results: even results of the simplest operations may not be representable as finite fractions – in decimals is infinitely recurring and can't be written precisely. IEEE 754*

  $$1/3 = 0.3333\ldots$$

  allows the user to choose between four options: round up, round down, round towards zero and round to nearest. The rounded result will be that which would have been achieved by computing with infinite precision and then rounding. This would leave an ambiguity in "round to nearest" when the infinite-precision result is exactly half-way between two representable forms; the rules provide that in this case, rounding towards zero is proper.
- *When is a result exceptional?: IEEE 754 has its own meaning for the word "exception". A computation can produce a result which is:*
  - *nonsense, such as the square root of -1 (NaN);*
  - *"division by zero" is given special treatment;*
  - *too big to represent ("overflow");*
  - *so small that its representation becomes problematic and precision is lost ("underflow");*
  - *not perfectly represented, like 1/3 ("inexact"). This is usually ignored.*

  All these are bundled together and described as "exceptional".
- *Action taken on IEEE exception: for each exception class listed above the user can opt:*
  - *To ignore the problem, in which case the standard lays down what value will be produced. Overflows and division by zero generate "infinity" (with a positive and negative type); invalid operations generate "NaN" (for Not a Number) in two flavors called "Quiet" and "Signalling".*
    The standard defines the results when operations are carried out on exceptional values (most often a NaN). A Quiet Nan as operand will not cause another exception (though the result will be a NaN too). A Signalling NaN causes an exception whenever it is used.
  - *To have the computation interrupted, and the user program signalled in some OS- and language-dependent manner.*

  Most programs leave all the IEEE exceptions off, but do rely on the system producing the right exceptional values.

IEEE Exponent Field and Bias

The exponent is not stored as a signed binary number, but *biased* so that the exponent field remains positive for the most negative legitimate exponent value; for the 64-bit IEEE format the exponent field is 11 bits long, so the bias is:

$$2^{10} - 1 = 1023$$

For a number

$$\text{mantissa} \times 2^{\text{exp}}$$

the exponent field will contain:

$$\text{exponent} + 1023$$

Only exponents from 1 through 2046 represent ordinary numbers; the biggest and smallest exponent field values (all-zeroes and all ones) are reserved for special purposes, described later.

Notes

IEEE Mantissa and Normalization

The IEEE format defines a single sign bit separate from the mantissa, (0 for positive, 1 for negative). So the stored mantissa only has to represent positive numbers. All properly-represented numbers in IEEE format are normalized, so

$$1 \le \text{mantissa} < 2$$

This means that the most significant bit of the mantissa (the single binary digit before the point) is always a "1" – so it doesn't actually need to be stored. The IEEE standard calls this the *hidden* bit.

So now the number 93,000,000, whose normalized representation has a binary mantissa of 1.01100010110001000101 and a binary exponent of 26, is represented in IEEE 64-bit format by setting the fields:

$$\text{mantissafield} = 01100010110001000101000\ldots$$

$$\text{exponentfield} = 1049 = 10000011001$$

Looking at it the other way; a 64-bit IEEE number with an exponent field of *E* and a mantissa field of *m* represents the number *num* where:

$$\text{num} = 1.\text{m} \times 2^{\text{E}-1023}$$

("1.m" represents the binary fraction with 1 before the point and the mantissa field contents after it).

Reserved Exponent Values

The smallest and biggest exponent field values are used to represent otherwise-illegal quantities:

- *E == 0: used to represent zero (with a zero mantissa) and "demoralizing" forms, where the number is too small. The denormalized number with E zero and mantissa m represents num where:*

  $$\text{num} = 0.\text{m} \times 2^{-1022}$$

  No RC3000 series MIPS FPA is able to cope with either generating or computing with denormalized numbers, and operations creating or involving them will be handled by the software exception handler. The RC4600 can be configured to replace denormalized results by zero and keep going.
- *E == 111...1: (i.e. the binary representation of 2047 in the 11-bit field used for an IEEE double) is used to represent:*
  - *with the mantissa zero, the values +inf, -inf (distinguished by the usual sign bit);*
  - *with the mantissa non-zero, it is a NaN. For MIPS, the most significant bit of the mantissa determines whether the NaN is quiet (ms bit zero) or signalling (ms bit one).*

MIPS FP Data Formats

The MIPS architecture uses two FP formats recommended by IEEE 754:

- *Single precision: fitted into 32 bits of storage. Compilers for MIPS use single precision for float variables.*
- *Double precision: uses 64 bits of storage. C compilers use double precision for C double types.*

The memory and register layout is shown in Table 8.1 with some examples of how the data works out. Note that the *float* representation can't hold a number as big as 93,000,000 exactly.

Notes

	31	30	23	22	0
single	sign	exponent		mantissa	
93000000	0	0001 1010		101 1000 1011 0001 0001	
0	0	0000 0000		000 0000 0000 0000 0000	
+infinity	0	1111 1111		000 0000 0000 0000 0000	
-infinity	1	1111 1111		000 0000 0000 0000 0000	
Quiet NaN	x	1111 1111		0xx xxxx xxxx xxxx xxxx	
Signalling NaN	x	1111 1111		1xx xxxx xxxx xxxx xxxx	

	high-order word					low-order word	
	31	30	20	19	0	31	0
double	sign	exponent		mantissa			
93000000	0	000 0001 1010		1011 0001 0110 0010 0010 1000 0000			
0	0	000 0000 0000		0000 0000 0000 0000 0000 0000			
+infinity	0	111 1111 1111		0000 0000 0000 0000 0000 0000			
-infinity	1	111 1111 1111		0000 0000 0000 0000 0000 0000			
Quiet NaN	x	111 1111 1111		0xxx xxxx xxxx xxxx xxxx xxxx			
Signalling Nan	x	111 1111 1111		xxx xxxx xxxx xxxx xxxx			

Table 8.1 Floating Point Data Formats

The way that the two words making up a double are ordered in memory depends on the CPU configuration; for “big-endian” configuration the high-order word is at the lowest, 8-byte aligned location; for little endian the low-order word is at the lower location.

MIPS Implementation of IEEE 754

IEEE 754 is quite demanding and sets two major problems:

- *Reporting exceptions makes pipelining harder: If the user opts to be told when an IEEE exception happens, then to be useful this should happen synchronously[1]; after the trap, the user will want to see all previous instructions complete, all FP registers still in the pre-instruction state, and will want to be sure that no subsequent instruction has had any effect.*

  In the MIPS architecture hardware traps (as noted in an earlier chapter) are always like this. This does limit the opportunities for pipelining FP operations, because the CPU cannot commit the following instruction until the hardware can be sure that the FP operation will not produce a trap. To avoid adding to the execution time, an FP operation must decide to trap or not in the first clock phase after the operands are fetched. This is possible for most kinds of exceptional result; but if the FPA is configured to trap on the IEEE inexact exception all FP pipelining is inhibited, and everything slows down.

- *Denormalized numbers: The representation of very small (“denormalized”) numbers and the exceptional values is too awkward for the FPA hardware to attempt, and they are instead passed on to the exception handler.*

Note that *the MIPS architecture does not prescribe exactly what calculations will be performed without software intervention*. A complete software floating point emulator may be required for some systems. In practice, the FPA traps only on a very small proportion of the calculations that a program is likely to produce.

1. Elsewhere in this manual and the MIPS documentation this will be referred to as a “precise exception”. But since both “precise” and “exception” are used to mean different things by the IEEE standard, this chapter will describe them as a “synchronous trap”.

Notes

Existing RC30xx family FPAs take the unimplemented trap whenever an operation should produce any IEEE exception or exceptional result other than “inexact” and “overflow”. For overflow, the hardware will generate an infinity or a largest-possible value (depending on the current rounding mode). The FPA hardware will cause an exception or produce denormalized numbers or NaNs.

Floating Point Registers (RC30xx)

The RC30xx defines 16 FP registers, given even numbers *$f0 - $f30.* There are also 16 odd-numbered registers, each of which hold the high-order bits of a 64-bit *double* value stored in the preceding even-numbered register. The odd-numbered registers can be accessed by move and load/store instructions; the FPA will Trap any floating-point operation that references odd numbered registers.

Floating Point Registers (RC4xxx/RC5000)

The RC4600/RC4700/RC5000 Floating Point Unit (FPU) has a set of 32 physical FGRs (Floating-point General-purpose Registers), each 64-bits wide. The RC4650 also offers 32 FGRs but each of them is only 32-bit wide.

These FGRs can be accessed in following ways:

- *as 32 general purpose registers (FGR0 - FGR31)*
  - *If FR-bit in CPU status register is set to 0, each FGR is treated to be 32-bit*
  - *If FR-bit in CPU status register is set to 1, each FGR is treated to be 64-bit*
- *as floating point registers (FPR0 - FPR31)*
  - *if FR-bit in CPU status register is set to 0, only lower 32-bits of each physical FGR are accessible in the RC4600/RC4700. The RC4650 has only 32-bits to start with. A pair of FGRs makes one logical FPR in the RC4600/RC4700; 16 FPRs, each 64-bit wide, are available. They can be accessed as even numbered FPRs only - FPR0, FPR2,...,FPR30. Each FPR can hold either a single or a double precision (except in RC4650) value. In the RC4650, accessed registers are only 32-bit wide and odd numbered FGRs are simply not available in this mode. This mode is provided in the RC4650 for compatibility purposes.*
  - *If FR-bit in CPU status register is set to 1, all 64-bits of each physical FGR are accessible in the RC4600/RC4700. Therefore 32 FPRs, each 64-bit wide (32-bit in RC4650),are available. They can be accessed as FPR0, FPR1,FPR2,...,FPR30,FPR31. Each FPR will hold a single or a double precision (except in the RC4650) value.*
  - *The RC5000 implements coprocessor 3 as another coprocessor COP1X. This is used to implement more floating point instructions, which are listed later in this chapter.*

Floating Point Exceptions/Interrupts

Floating point “exceptions” (enabled IEEE traps, or the “unimplemented operation” trap) are reported with an exception or an interrupt.

In the RC3081, one of the CPU interrupts will be dedicated to the FPA; the interrupt bit used is programmed in the RC3081 Configuration register, defined in chapter 3. In the RC4xxx, a floating point exception is treated as a specific exception, not an interrupt, and the *FPE* code bit in the CPU *cause* register is set. The common exception vector is used to handle the floating point exceptions.

In addition to the 32 *FGRs, both* the RC4xxx and RC3xxx FPUs have 32 *Control* registers, out of which 30 are reserved and only 2 serve any useful purpose. One of the two *control* register *FCR31* is called the *Control/Status* register. *FCR31*:

- *controls which types of exceptions are enabled*
- *holds the cause of the last exception*
- *holds cumulative flags to indicate types of exception case which occurred but did not cause exceptions because those types of exceptions were not enabled*
- *Controls rounding mode*

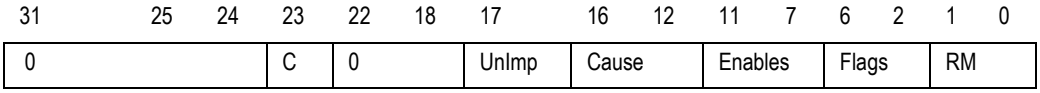
Notes

Provided the corresponding interrupt-enable bit in the CPU status register *SR* is set in the RC30xx (in the RC4xxx floating point exceptions are *not* maskable) a floating point exception will happen "immediately"; no FP or integer operation following the FP instruction which caused the exception will have had any effect. At this point *epc* will point to the correct place to restart the instruction. As described earlier, *epc* will either point to the offending instruction, or to a branch instruction immediately preceding it. If it is the branch instruction, the BD bit will be set in the CPU cause register.

If software performs FP operations with the FPA's interrupt disabled (RC30xx) the system cannot guarantee IEEE 754 compliance; even with all the IEEE traps disabled, the hardware will still attempt to trap on some conditions and will not produce IEEE 754-approved results.

The Floating Point Control/Status Register

The floating point control/status register (shown below) is coprocessor 1 control register 31 (mnemonic FCR31) and is accessed by *mtc1*, *mfc1* instructions.



Fields marked "0" will read as zero, and they must be written as zero.

In the **RC4xxx**, bit 24 is the *FS* bit. When set, denormalized results are flushed to 0 instead of causing an unimplemented operation exception.

- *C: condition bit. This is set only by FP compare operations and tested by conditional branches.*
- *RM: rounding mode, as required by IEEE 754. The values are:*

RM Value	Description
0	"RN" (round to nearest). Round a result to the nearest representable value; if the result is exactly half way between two representable values, round to zero.
1	"RZ" (round towards zero). Round a result to the closest representable value whose absolute value is less than or equal to the infinitely accurate result.
2	"RP" (round up, or towards +infinity). Round a result to the next representable value up.
3	"RM" (round down, or towards -infinity). Round a result to the next representable value down.

Most systems define "RN" as the default behavior.

- *UnImp: This bit is called the **"E"** bit in some IDT hardware manuals. following an FPA trap, this bit will be set to mark an "unimplemented instruction" exception[1].*

  This bit will be set and an interrupt raised whenever:
  - *there really is no instruction like this which the FPA will perform (but it is a "coprocessor 1" encoding); OR*
  - *the FPA is not confident that it can produce IEEE 754-correct result and/or exception signalling on this operation, with these operands.*

  For whatever reason, when "UnImp" is set the offending instruction should be re-executed by a software emulator.

  If FP operations are run without the interrupt enabled, then any FPA operation which wants to take an exception will leave the destination register unaffected and the FP Cause bits undefined.

- *Cause/Enables/Flags: Each of these is a 5-bit field, one bit for each IEEE exception type:*

  Bit4 invalid operation.

  Bit3 division by zero.

  Bit2 overflow.

  Bit1 underflow.

  Bit0 inexact.

[1] The MIPS documentation looks slightly different because it treats this as part of the ''Cause'' field.

Notes

The three different fields operate as follows:

Cause bits are set (by hardware or emulation software) if and only if the last FP instruction executed resulted in that kind of exception.

Flag bits are "sticky" versions of the Cause bits, and are left set by any instruction encountering that exception. The Flag bits can only be zeroed again by writing FPC31.

Enable bitswhen set, allow the corresponding Cause field bit to signal an interrupt. Note that there is no enable for Unimplemented Operation. Setting Unimplemented Operation always generates a FP exception. Another issue to be aware of is that before returning from a floating point exception, or doing a CTC1 to test condition, software must first clear the enabled Cause bits to prevent a repeat of the interrupt.

User mode programs will, therefore, never be able to probe for enabled Cause bits set. This information must be passed on to the user mode handler if needed in some manner other than through the Status register.

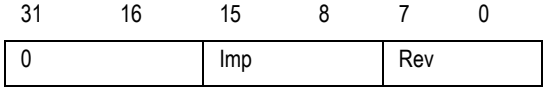
The architecture specifies that if the FPA doesn't set the "UnImp" bit but does set a Cause bit, then both the "Cause" bit setting and the result produced (if the corresponding "Enable" bit is off) are in accordance with the IEEE 754 standard. The FPA will always rely on software emulation (i.e. uses the "unimplemented" trap) for some situations:

- *Any operation which is given a denormalized operand or "underflow" (produces a denormalized result) will trap to the emulator. The emulator itself must test whether the "Enable underflow" bit is set, and either cause an IEEE-compliant exception or produce the correct result.*
- *Operations which should produce the ''invalid'' trap are correctly identified; so if the trap is enabled the emulator must do nothing. But if the "invalid" bit is disabled the software emulator is invoked to generate the appropriate result (usually a Quiet NaN).*
- *Exactly the same is done with a Signalling NaN operand.*
- *FP hardware can handle overflow on arithmetic (producing either the extreme finite value or a signed infinity, depending on the rounding mode). But the software emulator is needed to implement a convert to integer operation which overflows.*

The "Cause" bits are not reliable after an unimplemented exception. A full emulator (capable of delivering IEEE-compatible arithmetic on a CPU with no FPA fitted) to back up the FPA hardware may prove necessary in certain applications. FP Control instructions require care with the pipeline. See the appendix on pipeline hazards to see when the results are available to software.

Floating-point Implementation/Revision Register

This read-only register's fields are shown in below.



This register is co-processor 1 control register 0 (mnemonic FCR0), and is accessed by *ctc1* and *cfc1* instructions.

Unlike the CPU's field, the "Imp" field is useful. In the RC30xx family it will contain one of two values:

0 No FPA is available. Reading this register is the recommended way of sensing the presence of an FPA. Note that software must enable ''coprocessor 1'' instructions before trying to read this register.

3 The FPA is compatible with that used for the RC3000 CPU and its successors.

In the RC4xxx family the "Imp" field is 0x20 for RC4600, 0x21 for RC4700 and 0x22 for the RC4650.

The "Rev" field contains no relevant software data. The "Rev" field is a value of the form *y.x*, where *y* is the major revision number (bits 7:4) and *x* is a minor revision number (bits 3:0). Do not rely on this field.

Notes

Guide to FP Instructions

Load/store

These operations load or store 32 bits (also 64 bits in RC4600/RC4700/RC5000) of memory in or out of an FP register. The RC5000 allows loading an FPR from memory using two GPRs. General notes:

- *The data is unconverted and uninspected, so no exception can occur even if the data does not represent a valid FP value.*
- *These operations can specify the odd-numbered FP registers.*
- *The load operation has a delay of one clock, which—like loading to an integer register—is not interlocked. The compiler and/or assembler will usually take care of this; but it is undefined for an FP load to be immediately followed by an instruction using the loaded value.*
- *When writing in assembly, use the synthetic instructions. It is permissible to use any addressing mode which the assembler can understand (as described below).*

Machine Instructions (disp is signed 16-bit, index is a register):	
ldc1 fd, disp(rs)	fd <- *(rs + disp) (64-bit)
sdc1 fs, disp(rs)	*(rs + disp) <- fs; (64-bit)
lwc1 fd, disp(rs)	fd <- *(rs + disp)
swc1 fs, disp(rs)	*(rs + disp) <- fs;
ldcx1 fd, index(base)	fd <- *(base + index) (64-bit)
sdcx1 fs, index(base)	*(base + index) <- fs
lwcx1 fd, index(base)	fd <- *(base + index) (32-bit)
swcx1 fs, index(base)	*(base + index) <- fs (32-bit)
Synthesized by assembler:	
l.d fd, addr	fd = (double)*addr;
l.s fd, addr	fd = (float)*addr;
s.d fs, addr	(double)*addr = fs;
s.s fs, addr	(float)*addr = fs;

Move Between Registers

No data conversion is done here (bit patterns are copied as-is) and no exception results from any value. These instructions can specify the odd-numbered FP registers. The RC5000 also allows conditional moves.

Between Integer and FP Registers	
dmtc1 rs, fd	/* 64-bits uninterpreted */ fd = rs;
dmfc1 rd, fs	rs = fd;
mtc1 rs, fd	/* 32-bits uninterpreted */ fd = rs;
mfc1 rd, fs	rs = fd;
movt rd, rs, cc	rd = rs if cc is true;
movf rd, rs, cc	rd = rs if cc is false;
Between FP registers	
mov.d fd,fs	/* move 64-bits between reg pairs (or real 64-bit registers in RC4600/RC4700 if FR-bit=1 in SR)*/ fd = fs;
mov.s fd,fs	/* 32-bits between registers */ fd = fs;
movf.s fd,fs,cc	if cc is true, fd = fs;
movf.d fd,fs,cc	if cc is true, fd = fs;
movn.s fd,fs,cc	if cc is false, fd = fs;
movn.d fd,fs,cc	if cc is false, fd = fs;

Notes

3-operand Arithmetic Operations

- *All arithmetic operations can cause any IEEE exception type, and may result in an "unimplemented" trap if the hardware is not happy with the operands.*
- *All these instructions come in single-precision (32-bit, C float) and double-precision (64-bit, C double) format; the instructions are distinguished by a ".s" or ".d" on the opcode.*

  Software can't mix formats; both source values and the result will all be either single or double. To mix singles and doubles use explicit conversion operations.

add.d fd,fs1,fs2	fd = fs1 + fs2
add.s fd,fs1,fs2	
div.d fd,fs1,fs2	fd = fs1/fs2
div.s fd,fs1,fs2	
mul.d fd,fs1,fs2	fd = fs1 x fs2
mul.s fd,fs1,fs2	
sub.d fd,fs1,fs2	fd = fs1 - fs2
sub.s fd,fs1,fs2	

4-operand Arithmetic Operations

- *The RC5000 adds the following instructions. These may generate any IEEE exception type and may result in an "unimplemented" trap if the hardware can not accept the operands.*
- *All these instructions come in single-precision (32-bit, C float) and double-precision (64-bit, C double) format; the instructions are distinguished by a ".s" or ".d" on the opcode.*

Software can't mix formats; both source values and the result will all be either single or double. To mix singles and doubles use explicit conversion operations.

madd.s fd,fr,fs,ft madd.d fd,fr,fs,ft	fd = fr + (fs∗ft)
msub.s fd,fr,fs,ft msub.d fd,fr,fs,ft	fd = (fs∗ft) - fr
nmadd.s fd,fr,fs,ft nmadd.d fd,fr,fs,ft	fd = - ((fs∗ft) + fr)
nmsub.s fd,fr,fs,ft nmsub.d fd,fr,fs,ft	fd = - ((fs∗ft) - fr)

Unary (sign-changing) Operations

Although nominally arithmetic functions, these operations only change the sign bit and so can't produce most IEEE exceptions. They can produce an "invalid" trap if fed with a Signalling NaN value.

abs.d fd,fs abs.s fd,fs	fd = abs(fs)
neg.d fd,fs neg.s fd,fs	fd = -fs

Conversion Operations

Note that "convert from single to double" is written "cvt.d.s". All these use the current rounding mode, even when converting to and from integers. When converting data from CPU integer registers, the move from FP to CPU registers must be coded separately from the conversion operation.

Conversion operations can result in any IEEE exception

Notes

```
cvt.d.s fd,fs       fd = (double) fs;  /* float -> double */
cvt.d.w fd,fs       fd = (double) fs;  /* int -> double */
cvt.d.l fd,fs       fd = (double) fs;  /* 64-bit long-> double */
cvt.s.d fd,fs       fd = (float) fs;  /* double -> float */
cvt.s.w fd,fs       fd = (float) fs;  /* int -> float */
cvt.s.l fd,fs       fd = (float) fs;  /* 64-bit long-> float */
cvt.l.s fd,fs       fd = (int) fs;  /* float -> 64-bit long*/
cvt.l.d fd,fs       fd = (int) fs;  /* double -> 64-bit long*/
cvt.w.s fd,fs       fd = (int) fs;  /* float -> int */
cvt.w.d fd,fs       fd = (int) fs;  /* double -> int */
```

Note that when converting from FP formats to 32/64-bit integers, the result produced depends on the current rounding mode.

Conditional Branch and Test Instructions

The FP test and branch instructions are separate. A test instruction compares two FP values and set the FPA condition bit accordingly (C in the FP status register); the branch instructions branch on whether the bit is set or unset.

The branch instructions are:

```
bc1f disp       Branch if C bit "false" (zero)
bc1t disp       Branch if C bit "true" (one)
```

Like the CPU's other conditional branch instructions *disp* is PC-relative, with a signed 16-bit field as a word displacement. *disp* is usually coded as the name of a label, which is unlikely to end up more than 128Kbytes away.

But before executing the branch, the condition bit must be set appropriately. The comparison operators are:

```
c.<cond>.d fs1,fs2      Compare fs1 and fs2 and set C
c.<cond>.s fs1,fs2
```

Where <cond> is any of 16 conditions called: eq, f, le, lt, nge, ngl, ngle, ngt, ole, olt, seq, sf, ueq, ule, ult, un. Why so many? These test for any "OR" combination of three mutually incompatible conditions:

fs1 <fs2
fs1 == fs2
unordered (fs1, fs2)

The IEEE standard defines ''unordered'', and this relation is true for values such as infinities and NaN which do not compare meaningfully.

To test for conditions like ''greater than'' and ''not equal'', invert the test and then use a *bc1f* rather than a *bc1t* branch.

In addition to these combinations, each test comes in two flavors: one which takes an invalid trap if the operands are unordered, and one which never takes such a trap.

Notes

C bit is set if....	Mnemonic	
	trap	no trap
always false	f	sf
unordered(fs1,fs2)	un	ngle
fs1 == fs2	eq	seq
fs1 == fs2 || unordered(fs1,fs2)	ueq	ngl
fs1 <fs2	olt	lt
fs1 <fs2 || unordered(fs1,fs2)	ult	nge
fs1 <fs2 || fs1 == fs2	ole	le
fs1 <fs2 || fs1 == fs2 || unordered(fs1,fs2)	ule	ngt

The compare instruction produces its result too late for the branch instruction to be the immediately following instruction; a delay slot is required.

For example:

```
if (f0 <= f2) goto foo;/* and don't branch if unordered */

        c.le.d    $f0, $f2
        nop                       # the assembler will do this
        bc1t      foo


if (f0 > f2) goto foo;/* and trap if unordered */

        c.ole.d $f0, $f2
        nop                       # the assembler will do this...
        bc1f      foo
```

Fortunately, many assemblers recognize and manage this delay slot properly.

Other Floating Point Instructions

This section lists other floating point instructions not covered in the preceding sections. These are from the RC4600 and RC5000 instruction set. These, too, can generate any IEEE floating point exception.

sqrt.s fd,fs sqrt.d fd,fs	fd = sqrt(fs)
recip.s fd,fs recip.d fd,fs	fd = 1.0/fs
rsqrt.s fd,fs rsqrt.d fd,fs	fd = 1.0/sqrt(fs)

Instruction Timing Requirements

FP arithmetic instructions are interlocked (the instruction flow "stalls" automatically until results are available; the programmer does not need to be explicitly aware of execution times), and there is no need to interpose "nops" or to reorganize code for correctness. However, optimal performance will be achieved by code which lays out FP instructions to make the best use of overlapped execution of integer instructions, and the FP pipeline. Also note that in the RC4xxx, integer multiply in FPU can occur in parallel with other non-multiply floating point operations.

However, the compiler, assembler or (in the end) the programmer must take care about the timing of:

Notes

- *Operations on the FP control and status register: moves between FP and integer registers complete late, and the resulting value cannot be used in the following instruction.*
- *FP register loads: like integer loads, take effect late. The value can't be used in the following instruction.*
- *Test condition and branch: the test of the FP condition bit using the **bc1t**, **bc1f** instructions must be carefully coded, because the condition bit is tested a clock earlier than might be expected. So the conditional branch cannot immediately follow a test instruction.*

Instruction Timing for Speed

The FPA takes more than one clock for most arithmetic instructions, and so the pipelining becomes visible. The pipeline can show up in three ways:

- *Hazards: where the software must ensure the separation of instructions to work correctly;*
- *Interlocks: where the hardware will protect the software by delaying use of an operand until it is ready, but knowledgable re-arrangement of the code will improve performance;*
- *Overlapping: where the hardware is prepared to start one operation before another has completed, provided there are no data dependencies. This is discussed later.*

Hazards and interlocks arise when instructions fail to stick to the general MIPS rule of taking exactly one clock period between needing operands and making results ready. Some instructions either need operands earlier (branches, particularly, do this), or produce results late (e.g. loads). All instructions which can cause trouble are tabulated in an appendix of this manual.

Initialization and Enable on Demand

Reset processing will normally initialize the CPU's *SR* register to disable all optional co-processors, which includes the FPA (alias coprocessor 1). The *SR* bit CU1 has to be set for the FPA to work. In addition, in the RC4600/RC4700, the *FR* bit ought to be set for full access to all 64-bit FPRs.

To determine availability of a hardware FPA, software should read the FPA implementation register; if it reads zero, no FP is fitted and software should run the system with CU1 off[1]. Once CU1 is enabled, software should setup the control/status register *FCR31* with the system choice of rounding modes and trap enables.

Once the FPA is operating, the FP registers should be saved and restored during interrupts and context switches. Since this is (relatively) time-consuming, software can optimize this:

- *Leave the FPA disabled by default when running a new task. Since the task cannot now access the FPA, the OS doesn't have to save and restore registers.*
- *On a FP instruction trap, mark the task as an FP user and enable the FP before returning to it.*
- *Disable FP operations while in the kernel, or in any software called directly or indirectly from an interrupt routine. This avoids saving FP registers on an interrupt; instead FP registers need be saved only when context-switching to or from an FP using task.*

Floating Point Emulation

The low-cost members of the RC30xx family do not have a hardware FPA. The RC4650 does not have double precision capability in the FPU. Floating point functions for these processors are provided by software, and are slower than the hardware. Software FP is useful for systems where floating point is employed in some rarely-used routines.

There are two approaches:

- *Soft-float: Some compilers can be requested to implement floating point operations with software. In such a system, the instruction stream does not contain actual floating point operations; instead,*

[1] Some systems may still enable CP1, to use the BrCond(1) input pin as an input port. The software must then insure that no FPA operations are actually required, since the CPU will presume that they are actually executed.

Notes

when the software requests floating point from the compiler, the compiler inserts a call to a dedicated floating point library. This eliminates the overhead of emulating a floating point register file, and also the overhead of decoding the requested operation. IDT compilers are implemented in this manner and a FP emulation library is provided in its object form with the package.

- *Run-time emulation: The compiler can produce the regular FP instruction set. The CPU will then take a trap on each FP instruction, which is caught by the FP emulator. The emulator decodes the instruction and performs the requested operation in software.*

  Part of the emulator's job will be emulating the FP register set in memory.

  This technique is much slower than the soft-float technique; however, the binaries generated will automatically gain significant performance (in the RC30xx world) when executed by an RC3081, simplifying system upgrades.

As described above, a run-time emulator may also be required to back up FP hardware for very small operands or obscure operations; and, for maximal flexibility that emulator is usually complete. However, it will be written to ensure exact IEEE compatibility and is only expected to be called occasionally, so it will probably be coded for correctness rather than speed.

Compiled-in floating point (soft-float) is much more efficient on integer only chips; the emulator has a high overhead on each instruction from the trap handler, instruction decoder, and emulated register file.

Notes

Assembler Language Programming

Notes

This chapter details the techniques and conventions associated with writing and reading MIPS assembler code. This is different from just looking at the list of machine instructions because:

1. MIPS assemblers typically provide a large number of extra "synthetic" instructions which provide a richer instruction set than in fact exists at the machine level.
2. Programmers need to know the exact syntax of directives to start and end functions, define data, control instruction ordering and optimization, etc.

For a quick review of the low-level machine instruction set, data types, addressing modes, and conventional register usage, refer to Chapter 2, "MIPS Architecture."

Syntax Overview

Appendix D of this manual contains the formal syntax for the original MIPS assembler; most assemblers from other vendors follow this closely, although they may differ in their support of certain directives. These directives and conventions are similar to those found in other assemblers, especially a UNIX[1] assembler.

Key Points to Note

- *The assembler allows more than one statement on each line, separated by semi-colons.*
- *"White space" (tabs and spaces) is permitted between any symbols.*
- *All text from a '#' to the end of the line is a comment and is ignored, but do not put a '#' in column 1.*
- *Identifiers for labels, variables, etc. can be any combination of alpha-numeric characters plus '$', '_' and '.' (label must end with ':'), except for the first character which must not be numeric.*
- *The assembler allows the use of numbers (decimal between 1-99) as a label. These are treated as "temporary", and are "re-usable". In a branch instruction "1f" (forward) refers to the next "1:" label in the code, and "1b" (back) refers to the previous "1:" label.*
- *This eliminates the need for inventing unique—but meaningless— names for little branches and loops.*
- *The MIPS assembler, among others, provides the conventional register names (a0, t5, etc.) as C pre-processor macros; thus, the programmer must pass the source through the C preprocessor and include the file <iregdef.h>[2].*
- *If the C preprocessor is used, then typically it is permitted to also use C-style /* comments */ and macros.*
- *Hexadecimal constants are numbers preceded by "0x" or "0X"; octal constants must be preceded by "0"; be careful not to put a redundant zero on the front of a decimal constant.*
- *Pointer values can be used; in a word context, a label or relocatable symbol stands for its address as a 32-bit integer. The identifier '.' (dot) represents the current location counter. Many assemblers allow some limited arithmetic on pointers.*
- *Character constants and strings can contain the following special characters, introduced by the backslash '\' escape character:*

1. UNIX is a trademark of AT&T.
2. In IDT/c version 5.0 and later, the header files exist in the directory in which IDT disk/tape is copied. The preprocessor is automatically invoked if the extension of the filename is anything other than ".s". To force the preprocessor to be used with ".s" files, use the switch "-xassembler-with-cpp" in the command line.

Notes

Character	Generated Code	ASCII Values
\a	alert (bell)	0x07
\b	backspace	0x08
\e	escape	0x1B
\f	formfeed	0x0C
\n	newline	0x0A
\r	carriage return	0x0D
\t	horizontal tab	0x09
\v	vertical tab	0x0B
\\	backslash	0x5C
\'	single quote	0x27
\"	double quote	0x22
\0	null (integer 0)	0x00

A character can be represented as a one-, two-, or three-digit octal number (\ followed by octal digits), or as a one-, two-, or three-digit hexadecimal number (\x followed by hexadecimal digits).

- *The precedence of binary and unary operations in constant expressions follows the C definition.*

Register-to-Register Instructions

MIPS three-register operations are arithmetic or logical functions with two inputs and one output, for example:

rd = rs + rt

- *rd: is the destination register, which receives the result of functions op;*
- *rs: is a source register (operand);*
- *rt: is a second source register.*

In MIPS assembly language these type of instructions are written:

opcode rd, rs, rt

For example:

```
addu    $2, $4, $5                      # $2 = $4 + $5
```

Of course any or all of the register operands may be identical. The assembler will do this automatically if *rs* is omitted.

```
addu    $4, $5          →       addu    $4, $4, $5      # $4 = $4 + $5
```

Unary operations (e.g. **neg, not**) are always synthesized from one or more of the three-register instructions. The assembler expects maximum of two operands for these instructions (*dst* and *src*):

```
neg     $2, $4          →       sub     $2, $0, $4      # $2 = -$4
not     $3              →       nor     $3, $0, $3      # $3 = ~$3
```

Probably the most common register-to-register operation is **move**. This instruction is implemented by an **addu** with the always zero-valued register *$0*:

```
move    $3, $5          →       addu    $3, $5, $0      # $3 = $5
```

Notes

Immediate (Constant) Operands

An immediate operand is the term for a constant value found in a field of the instruction. Many of the MIPS arithmetic and logical operations have an alternative form which use a 16-bit immediate in place of *rt*. The immediate value is first sign-extended or zero-extended to 32-bits, for arithmetic or logical operations respectively.

Although an immediate operand implies different low-level machine instruction from its three-register version (e.g. **addi** instead of **add**), there is no need for the programmer to write this explicitly. The assembler will recognize when the final operand is an immediate and use the correct machine instruction. For example:

```
add     $2, $4, 64      →     addi    $2, $4, 64
```

If an immediate value is too large to fit into the 16-bit field in the machine instruction, then the assembler loads the constant into the *assembler temporary* register *$at ($1)* and performs the operation using that.

```
add     $4, 0x12345     →     li      $at, 0x12345
                              add     $4, $4, $at
```

Note the **li** (*load immediate*) instruction is a heavily-used synthetic macro instruction, which loads a 32-bit integer value into a register without the programmer having to worry about how it gets there:

- *When the 32-bit value lies between ±32K it can use a single **addiu** with **$0**; when bits 31-16 are all zero it can use **ori**; when the bits 15-0 are all zero it will be **lui**; and when none of these is possible it will be a an **lui/ori** pair:*

```
li      $3, -5      →     addiu   $3, $0, -5

li      $4, 0x8000  →     ori     $4, $0, 0x8000

li      $5, 0x120000→     lui     $5, 0x12

li      $6, 0x12345 →     lui     $6, 0x1
                          ori     $6, $6, 0x2345
```

Multiply/Divide Instructions

The multiply and divide machine instructions:

- *Do not accept immediate operands;*
- *Do not perform overflow or divide-by-zero tests;*
- *Operate asynchronously – so other instructions can be executed while they do their work;*
- *Store their results in two separate result registers (hi and lo), which can only be read with the two special instructions **mfhi** and **mflo**;*

  The RC4650, however, offers a true 3-operand multiply instruction "mul rd, rs, rt" (rd = rs x rt). The hi and lo registers are undefined after execution of this instruction. The RC4650 also provides 2 additional instructions "mad rs, rt" and "made rs, rt" which offer multiply-accumulate with hi and lo registers as accumulator.
- *The result registers are interlocked – they can be read at any time after the operation is started, and the processor will stall until the result is ready.*

However, the conventional assembler multiply/divide instructions will hide this: they are macro instructions which simulate a three-operand instruction and perform overflow checking. A signed divide may generate about 13 instructions in the RC3xxx, but they execute in parallel with the hardware divider so that no time is wasted (the divide itself takes 35 cycles).

Notes

Instruction	Description
mul	simple unsigned multiply, no checking; in RC4650: 3-operand multiply
mulo	signed multiply, checks for overflow above 32-bits
mulou	unsigned multiply, checks for overflow above 32-bits
mad	signed multiply added to hi,lo accumulator
madu	unsigned multiply added to hi,lo accumulator
div	signed divide, checks for zero divisor or divisor of -1 with most negative dividend.
divu	unsigned divide, checks for zero divisor
rem	signed remainder, checks for zero divisor or divisor of -1 with most negative dividend.
remu	unsigned remainder, checks for zero divisor

Note that doubleword versions of most of the instructions are available for RC4xxx. Their names are the same as above with an additional “d” prefix e.g. **ddiv, dmult**, etc.

Some MIPS assemblers will convert constant multiplication, and division/remainder by constant powers of two, into the appropriate shifts, masks, etc. Don’t rely on this though, as most toolchains expect the compiler or assembly-language programmer to spot this sort of optimization.

To explicitly control the multiplication or division, specify a *dst* of *$0*. The assembler will issue the raw machine instruction to start the operation; it is then up to the programmer to fetch the result from *hi* and/or *lo* and, if required, perform overflow checking.

Load/Store Instructions

The following table lists all the assembler’s load/store instructions. The signed load instructions sign-extend the memory data to 32/64-bits; the unsigned instructions zero-extend.

Load		Store	Description
Signed	**Unsigned**		
ld		**sd**	doubleword (**MIPS-III or later ISA**)
lw	lwu	**sw**	word **(MIPS-III or later ISA**)
lh	**lhu**	**sh**	halfword
lb	**lbu**	**sb**	byte
ulw		**usw**	unaligned word
ulh	**ulhu**	**ush**	unaligned halfword
ldl		sdl	doubleword left **(MIPS-III or later ISA**)
ldr		sdr	doubleword right (**MIPS-III or later ISA**)
lwl		**swl**	word left
lwr		**swr**	word right
flush		**invalidate**	word right (same as lwr and swr)
ll		sc	word load-linked / store conditional **(MIPS-III or later ISA)**
lld		scd	doubleword load-linked / store conditional **(MIPS-III or later ISA)**
l.d		**s.d**	double precision floating-point
l.s		**s.s**	single precision floating-point
sync		sync	finish all load/store fetched (multiproc **MIPS-III or later ISA**)

Notes

Note the architectural constraints of load/store instructions:

- *Alignment: addresses must be aligned correctly (i.e. a multiple of 8 for doublewords (in RC4xxx), 4 for words, and 2 for halfwords), except for the special left, right and unaligned variants (described below), or else they will cause an exception.*
- *Load delay:*

Unaligned Load and Store Instructions

Information and procedures for unaligned load and store using the assembler are discussed at length in Chapter 2 of this manual.

Addressing Modes

As discussed above, the hardware supports only one addressing mode: *base_reg+offset*, where *offset* is in the range –32768 to 32767. However the assembler simulates *direct* and *direct+index-reg* addressing modes by using two or three machine instructions, and the assembler-temporary register.

```
lw    $2, ($3)        →   lw    $2, 0($3)

lw    $2, 8+4($3)     →   lw    $2, 12($3)

lw    $2, addr        →   lui   $2, %hi_addr
                          lw    $2, %lo_addr($2)

sw    $2, addr($3)    →   lui   $2, %hi_addr
                          addu  $at, $at, $3
                          sw    $2, %lo_addr($2)
```

The store instruction is written with the source register first and the address second, to look like a load; for other operations the destination is first.

The symbol *addr* in the above examples can be any of these things:

- *a relocatable symbol – the name of a label or variable (whether in this module or elsewhere);*
- *a relocatable symbol ± a constant expression;*
- *a 32-bit constant expression (e.g. the absolute address of a device register).*

The constructs "%hi_" and "%lo_" do not actually exist in the assembler, but represent the high and low 16-bits of the address. This is not quite the straightforward division into low and high words that it looks, because the 16-bit offset field of a **lw** is treated as signed.

If the "addr" value is such that bit 15 is a "1," then the %lo_addr value will act as negative, and the assembler needs to increment %hi_addr to compensate:

addr	%hi_addr	%lo_addr
0x12345678	0x1234	0x5678
0x10008000	0x1001	0x8000

The **la** (*load address*) macro instruction provides a similar service for addresses as the **li** instruction provides for integer constants:

```
la    $2, 4($3)       →   addiu $2, $3, 4

la    $2, addr        →   lui   $2, %hi_addr
                          addiu $2, $2, %lo_addr($2)

la    $2, addr($3)    →   lui   $2, %hi_addr
                          addiu $2, $2, %lo_addr($2)
                          addu  $2, $2, $3
```

Notes

GP-relative Addressing

Loads and stores to global variables or constants usually require at least two instructions, e.g.:

```
lw    $2, addr      →    lui   $2, %hi_addr
                         lw    $2, %lo_addr($2)

sw    $2, addr($3)  →    lui   $2, %hi_addr
                         addu  $2, $3
                         sw    $2, %lo_addr($2)
```

A common low-level optimization supported by many toolchains is to use *gp-relative addressing*. This technique requires the cooperation of the compiler, assembler, linker and run-time start-up code to pool all of the "small" variables and constants into a single region of maximum size 64Kb, and then set register *$28* (known as the *global pointer* or *gp* register) to point to the middle of this region[1]. With this knowledge the assembler can reduce the number of instructions used to access any of these small variables, e.g.:

```
lw    $2, addr      →    lw    $2, (addr – _gp)($gp)

sw    $2, addr($3)  →    addu  $2, $gp, $3
                         sw    $2, (addr – _gp)($gp)
```

By default most toolchains consider objects less than or equal to 8 bytes in size to be "small". This limit can usually be controlled by the '**-G n**' compiler/assembler option; specifying '**-G 0**' will switch this optimization off altogether.

While it is a useful optimization, there are some pitfalls to beware of:

- *The programmer must take special care when writing assembler code to declare global data items correctly:*
  - *Writable, initialized data of 8 bytes or less must be put explicitly into the .sdata section.*
  - *Global common data must be declared with the correct size, e.g:*

    ```
    .comm  smallobj, 4
    .comm  bigobj, 100
    ```

  - *Small external variables should also be explicitly declared, e.g:*

    ```
    .extern  smallext, 4
    ```

  - *Since most assemblers are effectively one-pass, make sure that the program declares data before using it in the code, to get the most out of the optimization.*
- *In C, global variables must be declared correctly in all modules which use them. For external arrays either omit the size (e.g. extern int extarray[]), or give the correct size (e.g. int cmnarray[NARRAY]). Don't just give a dummy size of 1.*
- *A very large number of small data items or constants may cause the 64Kb limit to be exceeded, causing strange relocation errors when linking. The simplest solution here is to completely disable gp-relative addressing (i.e. use **–G 0**).*
- *Some real-time operating systems, and many PROM monitors, can be entered by direct subroutine calls, rather then via a single "system call" interface. This makes it impossible (or at least very difficult) to switch back and forth between the two different values of gp that will be used by the application, and by the o/s or monitor. In this case either the applications or the o/s (but not necessarily both) must be built with **–G 0**.*
- *When the **–G 0** option has been used for compilation of any set of modules, then it is usually essential that all libraries should also be compiled that way, to avoid relocation errors.*

[1] The actual handling may be toolchain dependent; this is the most common technique.

Notes

Jumps, Subroutine Calls and Branches

The MIPS architecture follows standard nomenclature:

- *PC-relative instructions are called "branch", and absolute-addressed instructions "jump"; the operation mnemonics begin with a **b** or **j, respectively**.*
- *A subroutine call is "jump and link" or "branch and link", and the mnemonics end ..**al**.*
- *All the branch instructions, including branch-and-link, are conditional, testing one or comparing two registers. They are therefore described in the next section. However, unconditional versions can be readily synthesized, e.g.: **beq $0, $0, label**.*
- *The MIPS II ISA adds the concept of "branch likely." The mnemonics add an additional **l** at the end of an equivalent conditional branch mnemonic. Specifically, these instructions are **beql, bnel, blezl, bgtzl, bltzl, bgezl, bltzall, bgezall, bc0tl,** and **bc1fl**. In a "branch likely" instruction a branch is taken in the same way as in a regular branch instruction, i.e. when the condition for branching is met. However, if the branch is not taken, the instruction in the delay slot is nullified.*

  In a regular branch instruction, the delay slot instruction is always executed. In a "branch likely" instruction, the delay slot instruction is executed only if the branch occurs. This offers assembler programmers an opportunity to speed up code execution based on their knowledge of the statistical probability of conditional branches taking place in their actual system.

  *Put something that needs to get executed only if the branch is taken in the delay slot, if branches are more likely to occur than to not occur. This saves one cycle from the subroutine execution time. The penalty for not branching is obvious, so use these instructions carefully where the probability of branch occurring is very high as compared to branch not occurring. Position independent subroutine calls can use the **bal**, **bgezal** and **bltzal** instructions.*

Jump instructions are:

- ***j**: this instruction (jump) transfers control unconditionally to an absolute address. Actually, **j** doesn't quite manage a 32-bit address; the top 4 address bits of the target are not defined by the instruction and the top 4 bits of the current "PC" value is used instead.*

  Most of the time this doesn't matter: 28-bits still gives a maximum code size of 256 Mb. It can be argued that it is useful in system software, because it avoids changing the top 3 address bits which select the address segment (described earlier in this manual).

  *To reach a really long way away, use the **jr** (jump to register) instruction, which is also used for computed jumps.*
- ***jal**, **jalr**: these instructions implement a direct and indirect subroutine call. As well as jumping to the specified address, they store the current pc + 8 in register $31 (ra). Remember that jump instructions, like branches, always execute the following instruction (at pc + 4), so the return address is the instruction after the branch delay slot. Subroutine return is normally done with **jr $31**.*

Conditional Branches

The MIPS architecture does not include a condition code register. Conditional branch machine instructions test one or two registers; and, together with a small group of compare-and-set instructions, are used to synthesize a complete set of arithmetic conditional branches.

Conditional branches are always PC-relative.

Branch instructions are listed below. Again there are architectural considerations:

- *Limited branch offset for PC-relative branches: the maximum branch displacement is ±32768 instructions (±128K bytes), because a 16-bit field is used for the offset.*
- *Branch delay slot: the instruction immediately after a standard branch (or jump) is always executed in the MIPS architecture, whether or not the branch is taken. Many assemblers will normally hide this from the programmer and will try to fill the branch delay slot with a useful instruction, or a **nop** if this is not possible. For the MIPS-II ISA instructions, described in the previous section, if a branch does not occur, the delay slot instruction is nullified.*

Notes

- *No carry flag: due to the lack of condition codes; if software needs to check for carry, compare the operands and results to work out when it occurs (typically, this requires only one **slt** instruction).*
- *No overflow flag: though the add and subtract instructions are available in an optional form, which causes a trap if the result overflows into the sign bit, C compilers typically won't generate those instructions, but Fortran might.*

Coprocessor Conditional Branches

There are four pairs of branches, testing true/false on four "coprocessor condition" values CPCOND0-3. In devices with an internal FPA, CPCOND1 is an internal flag which tests the floating point condition set by the FP compare instructions. Note that the coprocessor must be enabled for the branch instruction to be executed.

Compare and Set

The compare-and-set instructions conform to the C standard; they set their destination to 1 if the condition is true, and zero otherwise. Their mnemonics start with an ''**s**'': so **seq rd, rs, rt** sets **rd** to a 1 or zero depending on whether **rs** is equal to **rt**. These instructions operate just like any 3-operand MIPS instruction.

Floating point comparisons are done quite differently, and are described in the Floating-Point Accelerator chapter.

Coprocessor Transfers

CPU control functions are provided by a set of registers, which the instruction set accesses as "co-processor 0" data registers. These registers deal with catching exceptions and interrupts, and accessing the memory management unit and caches. A RC30xx family CPU has at least 12 registers, the RC4x00 has 23, and the RC4650 has 19. There is much more about this in earlier chapters.

The floating point accelerator is "co-processor 1", and is described in an earlier chapter. In the RC30xx, it has 16 64-bit registers to hold single- or double-precision FP values, which come apart into 32 32-bit registers when doing loads, stores and transfers to/from the integer registers. There are also two floating point control registers accessed with **ctc1**, **cfc1** instructions.

''Co-processor'' instructions are encoded in a standard way, and the assembler doesn't have to know much about what they do.

There are a range of instructions for moving data to and from the coprocessor data and control registers. The assembler expects numbers specified with ''$'' in front (except for floating point registers, which are called $f0 to $f31); but most toolchains provide a header file for the C pre-processor which provides meaningful names for the CPU control and FP control registers.

The assembler syntax makes no special provisions for "co-processor" registers; so if the program contains "obvious" mistakes (like reversing the CPU and special register names) the assembler will not return an error.

Instruction	Description
mfc0 dst, dr	move from CPU control register (to integer register)
dmfc0 dst, dr	move from 64-bit CPU control register (to integer register) (**MIPS-III**)
mtc0 src, dr	move to CPU control register (from integer register)
dmtc0 src, dr	move to 64-bit CPU control register (from integer register) (**MIPS-III**)
cfc1 dst, cr	move from fpa control register (to integer register)
ctc1 src, cr	move to fpa control register (from integer register)
mfc1 dst, dr	move from FP register to integer register
dmfc1 dst, dr	move doubleword from FP register to integer register (**MIPS-III**)

Notes

Instruction	Description
mtc1 src, dr	move to FP register from integer register
dmtc1 src, dr	move doubleword to FP register from integer register (**MIPS-III**)
sdc1 dr, offs(base)	store 64-bit FP register (to memory) (**MIPS-III**)
swc1 dr, offs(base)	store FP register (to memory)
ldc1 dr, offs(base)	load 64-bit FP register (from memory) (**MIPS-III**)
lwc1 dr, offs(base)	load FP register (from memory)
ldxc1 dr, index(base)	load 64-bit FP register (from memory) (**MIPS-IV**)
sdxc1 dr, index(base)	store 64-bit FP register (to memory) (**MIPS-IV**)
lwxc1 dr, index(base)	load 32-bit FP register (from memory) (**MIPS-IV**)
swxc1 dr, index(base)	store 32-bit FP register (to memory) (**MIPS-IV**)

Like conventional load instructions, there must always be one instruction after the move before the result can be used (the load-delay slot), whichever direction data is being moved.

The MIPS-IV ISA also implements coprocessor-to-coprocessor transfers.

Coprocessor Hazards

A pipeline hazard occurs when the architecture definition allows the internal pipelining to "show through" and affect the software: examples being the load and branch delay slots. Most MIPS assemblers will usually shield the programmer from hazards by moving instructions around or inserting **NOP**s, to ensure that the code executes as written.

However some CPU control register writes have side-effects which require pipeline-aware programming; since most assemblers don't understand anything about what these instructions are doing, they may not help.

One outstanding example is the use of interrupt control fields in the *Status* and *Cause* registers. In these cases the programmer must account for any side-effects, and the fact that they are delayed for up to three instructions. For example, after an **mtc0** to the *Status* register which changes an interrupt mask bit, it will be two further instructions before the interrupt is actually enabled or disabled. The same is also true when enabling or disabling floating-point coprocessor instructions (i.e. changing the CU1 bit).

To cope with these situations usually requires the programmer to take explicit action to prevent the assembler from scheduling inappropriate instructions after a dangerous **mtc0**. This is done by using the **.set noreorder** directive, discussed below.

A comprehensive summary of pipeline hazards can be found in the chapter titled "Instruction Timing And Optimization".

Notes

Assembler Directives

Sections

The names of, and support for different code and data sections, are likely to differ from one toolchain to another. Most will at least support the original MIPS conventions, which are illustrated (for ROMable programs) below

	ROM	
		etext
	.rdata	
	read-only data	
	.text	
1fc0000	*program code*	_ftext
	RAM	
????????		
	stack	
	goes down from top of memory	
	heap	
	goes up towards stack	
		end
	.bss	
	uninitialized writable data	
	.sbss	
	uninitialized writable small data	_fbss
		edata
	.lit8	
	64-bit floating point constants	
	.lit4	
	32-bit floating point constants	
	.sdata	
	writable small data	
	.data	
00000200	*writable data*	_fdata
	exception vectors	
00000000		

Within an assembler program the sections are selected as shown in Figure 9.1.

Notes

missing figure

Figure 9.1 Program Segments in Memory

.text, .rdata, .data

Put the appropriate section name before the data or instructions, for example:

```
        .rdata
msg:    .asciiz  "Hello world!\n"

        .data
table:  .word    1
        .word    2
        .word    3

        .text
func:   sub      sp, 64
        ...
```

.lit4, .lit8

These sections cannot be selected explicitly by the programmer. They are read-only data sections used implicitly by the assembler to hold floating-point constants which are given as arguments to the **li.s** or **li.d** macro instructions. Some assemblers and linkers will save space by combining identical constants.

.bss

This section is used to collect *uninitialized* data,the equivalent of Fortran's *common* data. An uninitialized object is declared, together with its size. The linker then allocates space for it in the **.bss** section, using the maximum size from all those modules which declare it. If any module declares it in a real, *initialized* data section, then all the sizes are ignored and that definition is used.

```
.comm  dbgflag, 4      # global common variable, 4 bytes
.lcomm sum, 4          # local common variable, 8 bytes
.lcomm array, 100      # local common variable, 100 bytes
```

Notes

"Uninitialized" is actually a misnomer: although these sections occupy no space in the object file, the run-time start-up code or operating-system must clear the **.bss** area to zero before entering the program; C programs will rely on this behavior. Many tool chains will accommodate this need through the start up file provided with the tool, to be linked with the user program[1].

.sdata, .sbss

These sections are equivalent to the **.data** and **.bss** sections above, but are used in some toolchains to hold *small*[2] data objects. This was described earlier in this chapter, when the use of the *gp* was discussed.

Stack and Heap

The *stack* and *heap* are not real sections that are recognized by the assembler or linker. Typically they are initialized and maintained by the run-time system by setting the *sp* register to the top of physical memory (aligned to an 8-byte boundary), and setting the initial *heap* pointer (used by the *malloc* functions) to the address of the **end** symbol.

Special Symbols

Figure 9.1 also shows a number of special symbols which are automatically defined by the linker to allow programs to discover the start and end of their various sections. Some of these are part of the normal UNIX environment expected by many programs; others are specific to the MIPS environment.

Symbol	Standard?	Value
_ftext		start of text (code) segment
etext	3	end of text (code) segment
_fdata		start of initialized data segment
edata	3	end of initialized data segment
_fbss		start of uninitialized data segment
end	3	end of uninitialized data segment

Data Definition and Alignment

Having selected the correct section, the data objects themselves are specified using the directives described in this section.

.byte, .half, .word, .short

These directives output integers which are 1, 2, or 4 bytes long, respectively. *.short* is the same as *.word*. A list of values may be given, separated by commas. Each value may be repeated a number of times by following it with a colon and a repeat count. For example.

```
.byte   3              # 1 byte:3
.half   1, 2, 3        # 3 halfwords:1 2 3
.word   5 : 3, 6, 7    # 5 words:5 5 5 6 7
```

Note that the section's location counter is automatically aligned to the appropriate boundary before the data is emitted. To actually emit unaligned data, explicit action must be taken using the **.align** directive described below.

1. IDT/c provides this code in the file "/idtc/idt_csu.S".
2. The default for "small" is 8 bytes. This number can be changed with the "-G" compiler/assembler switch.

Notes

.hword *expressions*, .int *expressions* , .long *expressions*

For each expression, emit a number that, at run time, is the value of that expression. Byte order and number of bits depends on the target endianess and integer size setting of the assembler. Both *.int* and *.long* produce identical results. *.hword* produces 16 bit results.

.single, .float, .double

These output single or double precision floating-point values, respectively. *.single* is the same as *.float*. Multiple values and repeat counts may be used in the same way as the integer directives.

```
.float   1.4142175          # 1 single-precision value
.double  1e+10, 3.1415      # 2 double-precision values
```

.ascii, .asciiz “str”

These directives output ASCII strings, either without or with a terminating null character respectively. The following example outputs two identical strings:

```
.ascii   "Hello\0"
.asciiz  "Hello"
```

.string “str”

Copy the string “str” to the object file. More than one string separated by commas are accepted. Strings are assumed to be zero terminated.

```
.string  "Hello\0"
```

.align

This directive allows the programmer to specify an alignment greater than that which would normally be required for the next data directive. The alignment is specified as a power of two, for example:

```
        .align   4      # align to 16-byte boundary (2^4)
var:    .word    0
```

If a label (var in this case) comes immediately before the **.align**, then the label will still be aligned correctly. For example, the following is exactly equivalent to the above:

```
var:    .align   4      # align to 16-byte boundary (2^4)
        .word    0
```

For “packed” data structures this directive allows the programmer to override the automatic alignment feature of **.half**, **.word**, etc., by specifying a zero alignment. This will stay in effect until the next section change. For example:

```
.half    3              # correctly aligned halfword
.align   0              # switch off auto-alignment
.word    100            # word aligned on halfword boundary
```

.comm, .lcomm

These directives declare a *common*, or *uninitialized* data object by specifying the object’s name and size.

An object declared with **.comm** is shared between all modules which declare it: it is allocated space by the linker, which uses the largest declared size. If any module declares it in one of the initialized **.data, .sdata** or **.rdata** sections, then all the sizes are ignored and the initialized definition is used instead[1].

Notes

An object declared with **.lcomm** is local to the current module, and is allocated space in the "uninitialized" **.bss** (or **.sbss**) section by the assembler.

```
.comm   dbgflag, 4      # global common variable, 4 bytes
.lcomm  array, 100      # local uninitialized object, 100 bytes
```

.space *size, fill*

The **.space** directive increments the current section's location counter by *size* number of bytes each of value *fill* (default *fill* = 0), for example:

```
struc:  .word   3
        .space  120     # 120 byte gap
        .word   -1
```

For normal data and text sections it just emits that many zero bytes, but in assemblers which allow the programmer to declare new sections with labels but no real content (like **.bss**), it will just increment the location counter without emitting any data.

Symbol Binding Attributes

Symbols (i.e. labels in one of the code or data segments) can be made visible and used by the linker which joins separate modules into a single program. The linker *binds* a symbol to an address and substitutes the address for assembler-language references to the symbol.

Symbols can have three levels of visibility:

- *Local: invisible outside the module they are declared in, and unused by the linker. The programmer does not need to worry about whether the same local symbol name is used in another module.*
- *Global: made public for use by the linker. Programs can refer to a global symbol in another module without defining any local space for it, using the* ***.extern*** *directive.*
- *Weak global: obscure feature provided by some toolchains. This allows the programmer to arrange that a symbol nominally referring to a locally-defined space will actually refer to a global symbol, if the linker finds one. If the linked program has no global symbol with that name, the local version is used instead.*

  The preferred programming practice is to use the ***.comm*** *directive whenever possible.*

.globl *symbol*, .global *symbol*

Unlike C, where module-level data and functions are automatically *global* unless declared with the static keyword, all assembler labels have *local* binding unless explicitly modified by the **.globl** or **.global** directive. They both mean the same thing.

To define a *symbol* (label) as having *global* binding that is visible to other modules, use the directive as follows:

```
        .data
        .globl  status          # global variable
status: .word   0

        .text
        .globl  set_status# global function
set_status:
        subu    sp,24
        ...
```

Note that **.globl** is not required for objects declared with the **.comm** directive; these automatically have global binding.

[1] The actual handling may be toolchain dependent; this is the most common technique.

Notes

.extern

All references to labels which are not defined within the current module are automatically assumed to be references to globally-bound symbols in another module (i.e. *external* symbols). In some cases the assembler can generate better code if it knows how big the referenced object is (e.g. the global pointer, described earlier). An external object's size is specified using the **.extern** directive, as follows:

```
.extern  index, 4
.extern  array, 100
lw       $3, index          # load a 4 byte (1 word) external
lw       $2, array($3)      # load part of a 100 byte external
sw       $2, value          # store in an unknown size external
```

.weakext

Some assemblers and toolchains support the concept of *weak* global binding. This allows the program to specify a provisional binding for a symbol, which may be overridden if a normal, or *strong* global definition is encountered. For example:

```
         .data
         .weakext errno
errno:   .word    0

         .text
         lw       $2,errno          # may use local or external                #
definition
```

This module, and others which access *errno*, will use this local definition of *errno*, unless some other module also defines it with a **.globl**.

It is also possible to declare a local variable with one name, but make it weakly global with a different name:

```
         .data
myerrno: .word0
         .weakext errno, myerrno

         .text
         lw       $2,myerrno       # always use local definition
         lw       $2,errno         # may use local definition, or             #
other
```

Function Directives

Some MIPS assemblers expect the programmer to mark the start and end of each function, and describe the stack frame which it uses. In some toolchains this information is used by the debugger to perform stack backtraces and the like.

.ent, .end

These directives mark the start and end of a function. A trivial *leaf* function might look like this:

```
         .text
         .ent     localfunc
localfunc:
         addu     v0,a1,a2         # return (arg1 + arg2)
         j        ra
         .end     localfunc
```

Notes

The label name may be omitted from the **.end** directive, which then defaults to the name used in the last **.ent**. Specifying the name explicitly allows the assembler to check that the programmer did not miss earlier **.ent** or **.end** directives.

.aent

Some functions may provide multiple, alternative entry-points. The **.aent** directive identifies labels as such. For example:

```
        .text
        .globl  memcpy
        .ent    memcpy
memcpy:move     t0,a0           # swap first two arguments
        move    a0,a1
        move    a1,t0

        .globl  bcopy
        .aent   bcopy
bcopy:  lb      t0,0(a0)        # very slow byte copy
        sb      t0,0(a1)
        addu    a0,1
        addu    a1,1
        subu    a2,1
        bne     a2,zero,bcopy
        j       ra
        .end    memcpy
```

.frame, .mask, .fmask

Most functions need to allocate a stack frame in which to:

- *save the return address register ($31);*
- *save any of the registers s0 - s9 and $f20 - $f31 which they modify (known as the callee-saves registers);*
- *store local variables and temporaries;*
- *pass arguments to other functions.*

In some CISC architectures the stack frame allocation, and possibly register saving, is done by special purpose *enter* and *leave* instructions, but in the MIPS architecture it is coded by the compiler or assembly-language programmer. However debuggers need to know the layout of each stack frame to do stack back-traces and the like, and in the original MIPS toolchain these directives provided this information; in other toolchains they may be quietly ignored, and the stack layout determined at run-time by disassembling the function prologue. Putting them in the code is therefore not always essential, but does no harm and may make the code more portable. Many toolchains supply a header file <*asm.h*>, which provides C-style macros to generate the appropriate directives, as required (the procedure call protocol, and stack usage, is described in a later chapter).

The **.frame** directive takes 3 operands:

- *framereg: the register used to access the local stack frame – usually $sp.*
- *returnreg: the register which holds the return address. Usually this is $0, which indicates that the return address is stored in the stack frame, or $31 if this is a leaf function (i.e. it doesn't call any other functions) and the return address is not saved.*
- *framesize: the total size of stack frame allocated by this function; it should always be the case that $sp + framesize = previous $sp.*

```
.frame  framereg, framesize, returnreg
```

Notes

The **.mask** directive indicates where the function saves general registers in the stack frame; **.fmask** does the same for floating-point registers. Their first argument is *regmask*, a bitmap of which registers are being saved (i.e. bit 1 set = *$1*, bit 2 set = *$2*, etc.); the second argument is *regoffset*, the distance from *framereg* + *framesize* to the start of the register save area.

```
.mask   regmask, regoffset
.fmask  fregmask, fregoffs
```

How these directives relate to the stack frame layout, and examples of their use, can be found in the next chapter. Remember that the directives do not create the stack frame, they just describe its layout; that code still has to be written explicitly by the compiler or assembly-language programmer.

Assembler Control (.set)

The original MIPS assembler is an ambitious program which performs intelligent macro expansion of synthetic instructions, delay-slot filling, peephole optimization, and sophisticated instruction reordering, or scheduling, to minimize pipeline stalls. Many assemblers will be less complex: modern optimizing compilers usually prefer to do these sort of optimizations themselves. However in the interests of source code compatibility, and to make the programmer's life easier, most MIPS assemblers perform macro expansion, insert extra **nops** as required to hide branch and load delay-slots, and prevent pipeline hazards in normal code (pipeline hazards are described in detail later).

With a reordering assembler it is sometimes necessary to restrict the reordering, to guarantee correct timing, or to account for side-effects of instructions which the assembler cannot know about (e.g. enabling and disabling interrupts). The **.set** directives provide this control.

.set noreorder/reorder

By default most assemblers are in *reorder* mode, which allow them to reorder instructions to avoid pipeline hazards and (perhaps) to achieve better performance; in this mode it will not allow the programmer to insert **nops**. Conversely, code that is an a *noreorder* region will not be optimized or changed in any way. This means that the programmer can completely control the instruction order, but the downside is that the code must now be scheduled manually, and delay slots filled with useful instructions or **nops**. For example:

```
.set noreorder
lw      t0, 0(a0)
nop                     # LDSLOT
subu    t0, 1
bne     t0, zero, loop
nop                     # BDSLOT
.set    reorder
```

.set volatile/novolatile

Any load or store instruction within a *volatile* region will not be moved with respect to other loads and stores. This can be important for accesses to memory mapped device registers, where the order of reads and writes is important. For example, if the following code fragment did not use **.set volatile**, then the assembler might decide to move the second **lw** before the **sw**, to fill the first load delay-slot. Hazard avoidance and other optimizations are not affected by this option.

```
.set volatile
lw      t0,0(a0)
sw      t0,0(a1)
lw      t1,4(a0)
.set novolatile
```

Notes

.set noat/at

The assembler reserves register *$1* (known as the *assembler temporary*, or *$at* register) to hold intermediate values when performing macro expansions; if code attempts to use the register, a warning or error message will be sent. It is not always obvious when the assembler will use *$at*, and there are certain circumstances when the programmer may need to ensure that it does not (for example in exception handlers before *$1* has been saved). Switching on **noat** will make the assembler generate an error message if it needs to use *$1* in a macro instruction, and allows the programmer to use it explicitly without receiving warnings. For example:

```
xcptgen:
        .set noat
        subu    k0,sp,XCP_SIZE
        sw      $at,XCP_AT(k0)
        .set at
```

.set nomacro/macro

Most of the time the programmer will not care whether an assembler statement generates more than one real machine instruction, but of course there are exceptions. For instance when manually filling a branch delay-slot in a *noreorder* region, it would almost certainly be wrong to use a complex macro instruction; if the branch was taken, only the first instruction of the macro would be executed. Switching on **nomacro** will cause a warning if any statement expands to more than one machine instruction. For example, compare the following two code fragments:

```
.set noreorder
blt     a1,a2,loop
.set nomacro
li      a0,0x1234       # BDSLOT
.set macro
.set reorder
```

```
.set noreorder
blt     a1,a2,loop
.set nomacro
li      a0,0x12345      # BDSLOT
.set macro
.set reorder
```

The first will assemble successfully, but the second will generate an assembler error message, because its **li** is expanded into two machine instructions (**lui** and **ori**). Some assemblers will catch this mistake automatically.

.set nobopt/bopt

Setting the **nobopt** control prevents the assembler from carrying out certain types of branch optimization. It is usually used only by compilers.

.set mips*n*

Advanced assemblers support this mechanism of setting the acceptable MIPS ISA level on the fly. Legal values of n are from 0 to 4. When n is 0, the MIPS ISA level is set to the default for the tool-chain or that set by the command line. 1 through 4 allow instructions of that particular level and below to be accepted by the assembler from that point on in the code. In addition to accepting and rejecting instructions, this irective also results in different ways of expanding some macros depending on the IPS ISA level chosen. This feature can be used to allow certain RC4xxx instructions while coding for 32-bit mode.

Notes

Listing Controls

Assembler listings are generated with a command line option while running the assembler (or preprocessor for it). In IDT/c, for example, the option is "-a".

.eject

Force a page-break at this point when generating assembly listing.

.list

Enable generating of assembly listing.

.nolist

Disable generating of assembly listing.

.subttl "subheading"

Use "subheading" text as the title (third line, immediately after the title line) when generating assembly listings. Affects subsequent pages as well.

.psize *lines, columns*

Define number of lines and (optionally) columns per page of listing. Default *lines* = 60, default *width* = 200 columns. If *lines* = 0 , no form-feeds are ever generated except those specified by *.eject.* Good practice for saving paper.

.title "heading"

Use "heading" text as the title (second line, immediately after the source filename and page number) when generating assembly listings.

The Complete Guide to Assembler Instructions

For *each* mnemonic defined by the MIPS assemblers for the MIPS IV instruction set, the assembler instructions listed below show how it is implemented and what it does. Some naming conventions in the assembler may appear confusing:

- *Unsigned versions: a "**u**" suffix on the assembler mnemonic is usually to be read as "unsigned". Usually this follows the conventional meaning; but the most common **u**-suffix instructions are **addu** and **subu**: and here the **u** means that overflow into the sign bit will not cause a trap. Regular **add** is never generated by C compilers.*
- *Many compilers, not expecting there to be a run-time system to handle overflow traps, will always use the "u" variant.*
- *However, because the integer multiply instructions **mult** and **multu** generate 64-bit results the signed and unsigned versions are different – and neither of the machine instructions produce a trap under any circumstances.*
- *Immediate operands: as mentioned above, the programmer can use immediate operands with most instructions (e.g. **add rd, rs, 1**); quite a few arithmetic/logic instructions really do have "immediate" versions (called **addi** etc.). Most assemblers do not require the programmer to explicitly know which machine instructions support immediate variants.*
- *Building addresses, %lo_ and %hi_: synthesis of addressing modes is described earlier. The table typically will list only one address-mode variant for each instruction in the table.*
- *What it does: the function of each instruction is described using "C" expression syntax; it is easy to get a rough idea, but a thorough knowledge of C allows the exact behavior to be understood.*

Notes

The assembler descriptions use the following conventions:

Word	Used For
rs,rt	CPU registers used as operands
rd	CPU register which receives the result
fs,ft	floating point register operands
fd	floating point register which receives the result
imm	16-bit ‘‘immediate’’ constant
label	the name of an entry point in the instruction stream
addr	one of a number of different address expressions
%hi_addr %lo_addr	where addr is a symbol defined in the data segment, ‘‘%hi_addr’’ and ‘‘%lo_addr’’ are as described above; that is, they are the high and low parts of the value which can be used in an **lui/addui** sequence.
%gpoff_addr	the offset in the ‘‘small data’’ segment of an address
$at	register $1, the ‘‘assembler temporary’’ register
$zero	register $0, which always contains a zero value
$ra	the ‘‘return address’’ register **$31**
RETURN	the point to where control returns to after a subroutine call; this is the next instruction but one after the branch/jump to subroutine, and is normally loaded into $ra by the ‘‘.. and link’’ instructions.
trap (CAUSE, code)	Take a CPU trap; ‘‘CAUSE’’ determines the setting of the Cause register, and ‘‘code’’ is a value not interpreted by the hardware, but which system software can obtain by looking at the trap instruction. CAUSE values can be BREAK; FPINT (for floating point exception); SYSCALL.
unordered(fs,ft)	some exceptional floating point values cannot be sensibly compared; it is not sensible to ask whether one NaN is bigger than another (NaN, ‘‘not a number’’, is produced when the result of an operation is not defined). The IEEE754 standard requires that for such a pair that ‘‘fs <ft’’, ‘‘fs == ft’’ and ‘‘fs > ft’’ shall all be false. ‘‘unordered(fs,ft)’’ returns true for an unordered pair, false otherwise.
fpcond	the floating point ‘‘condition bit’’ found in the FP control/status register, and tested by the **bc1f** and **bc0t** instructions.

Assembler	Expands To	What It Does
move rd,rs	**addu rd,rs,$zero**	rd = rs;
movn rd,rs, rt		if (rt ≠ 0) then rd ← rs; (**MIPS-IV**)
movz rd,rs, rt		if (rt = 0) then rd ← rs; (**MIPS-IV**)
Branch instructions (PC-relative, all conditional):		
b label	**beq $zero,$zero,label**	goto label;
bal label	**bgezal $zero,label**	ra = RETURN; goto label;
beq rs,rt,label		if (rs == rt) goto label;
beql rs,rt,label		if (rs == rt) goto label else nullify delay slot instruction; (**MIPS-II**)
beqz rs,label	**beq rs,$zero,label**	if (rs == 0) goto label;
bge rs,rt,label	**slt $at,rs,rt** **beq $at,$zero,label**	if ((signed) rs >= (signed) rt) goto label;

Notes

Assembler	Expands To	What It Does
bgel rs,rt,label	**slt $at,rs,rt** **beqzl $at,$zero,label**	if ((signed) rs >= (signed) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
bgeu rs,rt,label	**sltu $at,rs,rt** **beq $at,$zero,label**	if ((unsigned) rs >= (unsigned) rt) goto label;
bgeul rs,rt,label	**sltu $at,rs,rt** **beqzl $at,$zero,label**	if ((unsigned) rs >= (unsigned) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
bgez rs,label		if ((signed) rs >= 0) goto label;
bgezal rs,label		if ((signed) rs >= 0) { ra = RETURN; goto label;}
bgezall rs,label		if ((signed) rs >= 0) { ra = RETURN; goto label;} (**MIPS-II**)
bgezl rs,label		if ((signed) rs >= 0) goto label; else nullify delay slot instruction; (**MIPS-II**)
bgt rs,rt,label	**slt $at,rt,rs** **bne $at,$zero,label**	if ((signed) rs > (signed) rt) goto label;
bgtl rs,rt,label	**slt $at,rt,rs** **bnezl $at,$zero,label**	if ((signed) rs > (signed) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
bgtu rs,rt,label	**slt $at,rt,rs** **beq $at,$zero,label**	if ((unsigned) rs > (unsigned) rt) goto label;
bgtul rs,rt,label	**sltu $at,rt,rs** **bnezl $at,label**	if ((unsigned) rs > (unsigned) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
bgtz rs,label		if ((signed) rs > 0) goto label;
bgtzl rs,label		if ((signed) rs > 0) goto label; else nullify delay slot instruction; (**MIPS-II**)
ble rs,rt,label	**sltu $at,rt,rs** **beq $at,$zero,label**	if ((signed) rs <= (signed) rt) goto label;
blel rs,rt,label	**slt $at,rt,rs** **beqzl $at,label**	if ((signed) rs <= (signed) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
bleu rs,rt,label	**sltu $at,rt,rs** **beq $at,$zero,label**	if ((unsigned) rs <= (unsigned) rt) goto label;
bleul rs,rt,label	**sltu $at,rt,rs** **beqzl $at,label**	if ((unsigned) rs <= (unsigned) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
blez rs,label		if ((signed) rs <= 0) goto label;
blezl rs,label		if ((signed) rs <= 0) goto label; else nullify delay slot instruction; (**MIPS-II**)
blt rs,rt,label	**slt $at,rs,rt** **bne $at,$zero,label**	if ((signed) rs <(signed) rt) goto label;
bltl rs,rt,label	**slt $at,rs,rt** **bnezl $at,label**	if ((signed) rs <(signed) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)

Notes

Assembler	Expands To	What It Does
bltu rs,rt,label	**sltu $at,rs,rt** **bne $at,$zero,label**	if ((unsigned) rs <(unsigned) rt) goto label;
bltul rs,rt,label	**sltu $at,rs,rt** **bnezl $at,label**	if ((unsigned) rs <(unsigned) rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
bltz rs,label		if ((signed) rs <0) goto label;
bltzl rs,label		if ((signed) rs <0) goto label; else nullify delay slot instruction; (**MIPS-II**)

bltzal rs,label		if ((signed) rs <0) {
		ra = RETURN;
		goto label;
		}

bltzall rs,label		if ((signed) rs <0) { ra = RETURN;
		goto label;
		}
		else nullify delay slot instruction; (**MIPS-II**)

bne rs,rt,label		if (rs != rt) goto label;
bnel rs,rt,label		if (rs != rt) goto label; else nullify delay slot instruction; (**MIPS-II**)
bnez rs,label	**bne rs,$zero,label**	if (rs != 0) goto label;
Unary arithmetic/logic operations:		
abs rd,rs	**sra $at,rs,31** **xor rd,rs,$at** **sub rd,rd,$at**	rd = rs <0 ? -rs: rs;
abs rd	**sra $at,rd,31** **xor rd,rd,$at** **sub rd,rd,$at**	rd = rd <0 ? -rd: rd;
neg rd,rs	**sub rd,$zero,rs**	rd = -rs; /* trap on overflow */
neg rd	**sub rd,$zero,rd**	rd = -rd; /* trap on overflow */
dneg rd,rs	**sub rd,$zero,rs**	rd = -rs; /* doubleword, trap on overflow */ (**MIPS-III**)
negu rd,rs	**subu rd,$zero,rs**	rd = -rs; /* no trap */
negu rd	**subu rd,$zero,rd**	rd = -rd; /* no trap */
dnegu rd	**subu rd,$zero,rd**	rd = -rd; /* doubleword,no trap */ (**MIPS-III**)
not rd,rs	**nor rd,rs,$zero**	rd = ~rs;
not rd	**nor rd,rd,$zero**	rd = ~rd;
Binary arithmetic/logical operations:		
add rd,rs,rt		rd = rs + rt; /* trap on overflow */
add rd,rs	**add rd,rd,rs**	rd += rs; /* trap on overflow */
dadd rd,rs,rt		rd=rs+rt; /*doubleword, trap on overflow */ (**MIPS-III**)
addu rd,rs,rt		rd = rs + rt; /* no trap on overflow */
daddu rd,rs,rt		rd=rs+rt;/*doubleword, no trap on overflow */ (**MIPS-III**)

Notes

Assembler	Expands To	What It Does
addu rd,rs		rd += rs; /* no trap on overflow */
and rd,rs,rt		rd = rs & rt;
and rd,rs	**and rd,rd,rs**	rd &= rs;
div rd,rs,rt	**div rs,rt** **bne rt,$zero,1f** **nop** **break 7** **1:** **li $at,-1** **bne rt,$at,2f** **nop** **lui $at,0x8000** **bne rs,$at,2f** **nop** **break 6** **2:** **mflo rd**	rd = rs/rt; /* trap divide by zero */ /* trap overflow conditions */
ddiv rd,rs,rt	**ddiv rs,rt** **bnez rt,1f** **nop** **break 7** **1:** **daddiu $at,$zero, -1** **bne rt,$at,2f** **daddiu $at,$zero,1** **dsll32 $at,$at,0x1f** **bne rs,$at,2f** **nop** **break 6** **2:** **mflo rd**	rd = rs/rt; /* doubleword */ (**MIPS-III**) /* trap divide by zero */ /* trap overflow conditions */
divu rd,rs,rt	**divu rs,rt** **bne rt,$zero,1f** **nop** **break 7** **1:** **mflo rd**	 rd = rs/rt; /* trap on divide by zero */ /* no check for overflow */
ddivu rd,rs,rt	**ddivu $zero,rs,rt** **bnez rt,1f** **nop** **break 7** **1:** **mflo rd**	 rd = rs/rt; /* doubleword */ /* trap on divide by zero */ /* no check for overflow */ (**MIPS-III**)
or rd,rs,rt		rd = rs | rt;
mul rd,rs,rt	**multu rs,rt** **mflo rd**	rd = rs*rt; /* no checks */ */* mul has a different definition in **RC4650/RC32364** as described later in the category: Multiply/divide unit machine instructions */*
dmul rd,rs,rt	**dmultu rs,rt** **mflo rd**	rd = rs*rt; /* doubleword, no checks */ (**MIPS-III**)

Notes

Assembler	Expands To	What It Does
mulo rd,rs,rt	**mult rs,rt** **mfhi rd** **sra rd,rd,31** **mflo $at** **beq rd,$at,1f** **nop** **break 6** **1:** **mflo rd**	rd = rs * rt; /* signed */ /* trap on overflow */
dmulo rd,rs,rt	**dmult rs,rt** **mflo rd** **dsra rd,rd,0x1f** **mfhi $at** **beq rd,$at,1f** **nop** **break 6** **1:** **mflo rd**	rd = rs * rt; /* signed doubleword*/ (**MIPS-III**) /* trap on overflow */
mulou rd,rs,rt	**multu rs,rt** **mfhi $at** **mflo rd** **beq $at,$zero,1f** **nop** **break 6** 1:	rd = (unsigned) rs * rt; /* trap on overflow */
dmulou rd,rs,rt	**dmultu rs,rt** **mfhi $at** **mflo rd** **beqz $at,1f** **nop** **break 6** 1:	rd = (unsigned) rs * rt; /* doubleword, trap on overflow */ (**MIPS-III**)
nor rd,rs,rt		rd = ~(rs | rt);
rem rd,rs,rt	**div rs,rt** **bne rt,$zero,1f** **nop** **break 7** **1:** **li $at,-1** **bne rt,$at,2f** **nop** **lui $at,0x8000** **bne rs,$at,2f** **nop** **break 6** **2:** **mfhi rd**	rd = rs%rt; /* trap if rt == 0 */ /* trap if it will overflow */

Notes

Assembler	Expands To	What It Does
drem rd,rs,rt	**ddiv $zero,rs,rt** **bnez rt,1f** **nop** **break 7** **1:** **daddiu $at,$zero,-1** **bne rt,$at,2f** **daddiu $at,$zero,1** **dsll32 $at,$at,0x0x1f** **bne rs,$at,2f** **nop** **break 6** **2:** **mfhi rd**	rd = rs%rt; /*doubleword */ (**MIPS-III**) /* trap if rt == 0 */ /* trap if it will overflow */
remu rd,rs,rt	**divu rs,rt** **bne rt,$zero,1f** **nop** **break 7** **1:** **mfhi rd**	/* unsigned operation, ignore overflow */ rd = rs%rt; /* trap if rt == 0 */
dremu rd,rs,rt	**ddivu $zero,rs,rt** **bnez rt,1f** **nop** **break 7** **1:** **mfhi rd**	/* unsigned operation, ignore overflow */ rd = rs%rt; /* doubleword */ (**MIPS-III**) /* trap if rt == 0 */
rol rd,rs,rt	**negu $at,rt** **srlv $at,rs,$at** **sllv rd,rs,rt** **or rd,rd,$at**	/* rd = rs rotated left by rt */
ror rd,rs,rt	**negu $at,rt** **sllv $at,rs,$at** **srlv rd,rs,rt** **or rd,rd,$at**	/* rd = rs rotated right by rt */
seq rd,rs,rt	**xor rd,rs,rt** **sltiu rd,rd,1**	rd = (rs == rt) ? 1: 0;
sge rd,rs,rt	**slt rd,rs,rt** **xori rd,rd,1**	rd = ((signed)rs >= (signed)rt) ? 1: 0;
sgeu rd,rs,rt	**sltu rd,rs,rt** **xori rd,rd,1**	rd = ((unsigned)rs >= (unsigned)rt) ? 1: 0;
sgt rd,rs,rt	**slt rd,rt,rs**	rd = ((signed)rs > (signed)rt) ? 1: 0;
sgtu rd,rs,rt	**sltu rd,rt,rs**	rd = ((unsigned)rs > (unsigned)rt) ? 1: 0;
sle rd,rs,rt	**slt rd,rt,rs** **xori rd,rd,1**	rd = ((signed)rs <= (signed)rt) ? 1: 0;
sleu rd,rs,rt	**sltu rd,rt,rs** **xori rd,rd,1**	rd = ((unsigned)rs <= (unsigned)rt) ? 1: 0;
slt rd,rs,rt		rd = ((signed)rs <(signed)rt) ? 1: 0;
sltu rd,rs,rt	**sltu rd,rs,rt** **xor rd,rs,rt**	rd = ((unsigned)rs <(unsigned)rt) ? 1: 0;
sne rd,rs,rt	**sltu rd,$zero,rd**	rd = (rs == rt) ? 1: 0;

Notes

Assembler	Expands To	What It Does
sll rd,rt,rs	**sllv rd,rt,rs**	rd = rt<<rs;
dsll rd,rt,rs	**dsllv rd,rt,rs**	rd = rt <<rs; /* doubleword */ (**MIPS-III**)
dsll32 rd,rt,imm		rd = rt <<(imm+32); /* doubleword */ (**MIPS-III**)
sra rd,rt,rs	**srav rd,rt ,rs**	rd = ((signed) rt) >>rs;
dsra rd,rt,rs	**dsrav rd,rt ,rs**	rd = ((signed) rt) >>rs;/* doubleword */ (**MIPS-III**)
dsra32 rd,rt,imm		rd = ((signed) rt) >>(imm+32);/* doubleword */ (**MIPS-III**)
srl rd,rt,rs	**srlv rd,rt,rs**	rd = ((unsigned) rt) >>rs;
dsrl rd,rt,rs	**dsrlv rd,rt,rs**	rd = ((unsigned) rt) >>rs; /* doubleword */ (**MIPS-III**)
dsrl32 rd,rt,imm		rd = ((unsigned) rt) >>(imm+32); /* doubleword */ (**MIPS-III**)
sub rd,rs,rt		rd = rs - rt; /* trap on overflow */
dsub rd,rs,rt		rd = rs - rt; /* doubleword trap overflow */ (**MIPS-III**)
subu rd,rs,rt		rd = rs - rt; /* no trap on overflow */
dsubu rd,rs,rt		rd = rs-rt; /* doubleword no trap overflow */ (**MIPS-III**)
xor rd,rs,rt	**xor rd,rs,rt**	rd = rs ^ rt;
Binary instructions with one constant operand (“immediate”): **(addi** opcode is legal but unnecessary)		
add rd,rs,imm	**addi rd,rs,imm**	/* “add” traps on overflow */ /* when -32768 <= imm <32768 */ rd = rs + (signed) imm;

	lui rd,hi_imm
	ori rd,rd,lo_imm
	add rd,rs,rd
		/* for big values add and ALL signed ops
		* expand like this */
		rd = imm & 0xFFFF0000;
		rd |= imm & 0xFFFF;
		rd = rs + rd;

dadd rd,rs,imm	**daddi rd,rs,imm**	rd = rs + (signed) imm; (64-bit) (**MIPS-III**)
addu rd,rs,imm	**addiu rd,rs,imm**	/* “addu” won’t trap on overflow */ /* will expand if imm bigger than 16 bit */ rd = rs + (signed) imm;
daddu rd,rs,imm	**daddiu rd,rs,imm**	/* “daddu” won’t trap on overflow */ rd = rs + (signed) imm; (64-bit) (**MIPS-III**)
sub rd,rs,imm	**addi rd,rs,-imm**	/* trap on overflow */ /* will expand if imm bigger than 16 bit */ rd = rs - (signed) imm;
dsub rd,rs,imm	**daddi rd,rs,-imm**	rd = rs - (signed) imm; /* doubleword trap overflow */ (**MIPS-III**)
subu rd,rs,imm	**addiu rd,rs,-imm**	/* no trap on overflow */ /* will expand if imm bigger than 16 bit */ rd = rs - (signed) imm;
dsubu rd,rs,imm	**daddiu rd,rs,-imm**	/* no trap on overflow */ /* will expand if imm bigger than 16 bit */ rd = rs - (signed) imm;

Notes

Assembler	Expands To	What It Does
and rd,rs,imm	**andi rd,rs,imm**	rd = rs & imm; /* 0 <= imm <65535 */

	lui rd,hi_imm
	ori rd,rd,lo_imm
	and rd,rs,rd
		/* for big values add and ALL unsigned ops
		* expand like this */
		rd = imm & 0xFFFF0000;
		rd |= imm & 0xFFFF;
		rd = rs & rd;

or rd,rs,imm	**ori rd,rs,imm**	rd = rs | imm; /* 0 <= imm <65535 */
slt rd,rs,imm	**slti rd,rs,imm**	/* -32768 <= imm <32768 */ rd = ((signed) rs <(signed) imm) ? 1: 0; /* expanded as for add if imm big */
sltu rd,rs,imm	**sltiu rd,rs,imm**	rd = ((unsigned) rs <(unsigned) imm) ? 1: 0; /* expanded as for "and"if imm big */
xor rd,rs,imm	**xori rd,rs,imm**	rd = rs ^ imm;
li rd,imm	**ori rd,$zero,imm**	rd = (unsigned) imm; /* imm <= 65335 */

	lui rd,hi_imm
	ori rd,$zero,lo_imm
		/* for big imm value expand to... */
		rd = imm & 0xFFFF0000;
		rd |= imm & 0xFFFF;

lui rd,imm		rd = imm << 32;
Multiply/divide unit machine instructions:		
mad rs, rt		HI,LO = HI,LO + rs*rt /*signed, never trap */ **RC4650/RC32364** only
madu rs, rt		HI,LO = HI,LO + rs*rt /*unsigned, never trap */ **RC4650/ RC32364** only
msub rs,rt		HI,LO = HI,LO - rs*rt /*signed, never trap */ **RC32364** only
msubu rs,rt		HI,LO = HI,LO - rs*rt /*unsigned, never trap */ **RC32364** only
clz rt,rs		rt = leading zeros **RC32364** only
clo rt,rs		rt = leading ones **RC32364** only
mul rd, rs, rt		32-bit rd = rs * rt /* never trap */ **RC4650/RC32364** only
mult rs,rt		/* Start signed multiply of rs and rd. * Result can be retrieved, in a while, * using mfhi/mflo */
dmult rs,rt		/* Start signed multiply of rs and rd; doubleword * Result can be retrieved, in a while, * using mfhi/mflo (**MIPS-III**) */
multu rs,rt		/* start unsigned multiply of rs and rd */
dmultu rs,rt		/* start unsigned doubleword multiply of rs and rd */ (**MIPS-III**)
div rs,rt		/* start signed divide rs/rt */
ddiv rs,rt		rd = rs/rt; /*doubleword, trap on errors */ (**MIPS-III**)
divu rs,rt		/* start unsigned divide rs/rd */
ddivu rs,rt		/* doubleword, start unsigned divide rs/rd */ (**MIPS-III**)

Notes

Assembler	Expands To	What It Does
mfhi rd		/* retrieve remainder from divide or high-* order word of result of multiply */
mflo rd		/* retrieve result of divide or low-order * word of result of multiply */
mthi rs		/* load multiply unit "hi" register */
mtlo rs		/* load multiply unit "lo" register */
Unconditional (absolute) branch and call:		
jal label		ra = RETURN; goto label;
jalr rd,rs		rd = RETURN; goto *rs;
jalr rs	**jalr rs,$ra**	ra = RETURN; goto *rs;
jal rd,addr	**lui $at,%hi_addr** **addiu $at,$at,%lo_addr** **jalr rd,$at**	rs = RETURN; goto label; goto *at;
j label		goto label;
jr rs		goto *rs;
No-op:		
nop	**sll $zero,$zero,$zero**	/* no-op, instruction code == 0 */
Load address:		
la rd,label	**lui rd,%hi_label** **addiu rd,rd,%lo_label**	rd = %hi_addr <<32 rd += (signed) %lo_label;
Address mode implementation for load/store:		
lw rd,label	**lui rd,%hi_label** **lw rd,%lo_label(rd)**	/* link-time determined location */ /* note can use rd or $at for lw */
	lw rd,%gpoff_addr($gp)	/* link-time location, in gp segment */
lw rd,offset(rs)	**lw rd,offset(rsO)**	/* single instruction if offset fits * in 16 bits */
	lui rd,%hi_offset **addu rd,rd,rs** **lw rd,%lo_offset(rd)**	/* sequence for big offset */
Load and store instructions:		
ld rd,addr		/* load doubleword */ rd = *((long long*) addr); (**MIPS-III**)
lw rd,addr		/* load word */ rd = *((int *) addr);
lh rd,addr		/* load half-word,sign-extend */ rd = *((short *) addr);
lhu rd,addr		/* load half-word,zero-extend */ rd = *((unsigned short *) addr);
lb rd,addr		/* load byte, sign-extend */ rd = *((signed char *) addr);

Notes

Assembler	Expands To	What It Does
lbu rd,addr		/* load byte, sign-extend */ rd = *((unsigned char *) addr);
ld $t2,addr	**lui $at,%hi_addr** **addiu $at,$at,%lo_addr** **lw $t2,0($at)** **lw $t3,4($at)**	/* load 64-bit integer into pair of regs */ (**MIPS-III**)
sd rs,addr		/* store doubleword */ *((long long*) addr) = rs; (**MIPS-III**)
sw rs,addr		/* store word */ *((int *) addr) = rs;
sh rs,addr		/* store half-word */ *((short *) addr) = rs;
sb rs,addr		/* store byte */ *((char *) addr) = rs;
sd $t2,addr	**lui $at,%hi_addr** **addiu $at,$at,%lo_addr** **sw $t2,0($at)** **sw $t3,4($at)**	/* store 64-bit integer */ (**MIPS-III**)
ulw rd,addr	**lui $at,%hi_addr** **addiu $at,$at,%lo_addr** **lwl rd,0($at)** **lwr rd,3($at)**	/* load word unaligned */ (**MIPS-III**) /* if addr is aligned, does same load * twice */
usw rs,addr	**lui $at,%hi_addr** **addiu $at,$at,%lo_addr** **swl rs,0($at)** **swr rs,3($at)**	/* store word unaligned */ (**MIPS-III**) /* if addr is aligned, does same store * twice */
lwl rd,addr		load/store word left/right, see “Unaligned Load and Store Instructions” on page 9-5
lwr rd,addr		
swl rs,addr		
swr rs,addr		
ldl rd,addr		load/store doubleword left/right, see “Unaligned Load and Store Instructions” on page 9-5 (**MIPS-III**)
ldr rd,addr		
sdl rs,addr		
sdr rs,addr		
l.s fd,addr	**lui $at,%hi_addr** **lwc1 fd,%lo_addr($at)**	/* load FP single */ fd = *((float *) addr);
l.d $fd,addr	**lui $at,%hi_addr** **addiu $at,$at,%lo_addr** **lwc1 $fd+1,0($at):** (**MIPS-I**) **lwc1 $fd,4($at):** (**MIPS-I**) **ldc1 $fd,0(at):** (**MIPS-II**)	/* load FP double into reg pair */ fd = *((double *) addr);
s.s $fs,addr	**swc1 fs,addr**	/* store FP single */ *((float *) addr) = fs;

Notes

Assembler	Expands To	What It Does
s.d $fd,addr	**lui $at,%hi_addr** **addiu $at,$at,%lo_addr** **swc1 $fd+1,0($at) :**(**MIPS-I**) **swc1 $fd,0(at) :**(**MIPS-I**) sdcl $fd,0(at) **:**(**MIPS-II**)	/* store FP double from reg pair */ *((double *) addr) = fs;
lwxcl $fd,index(base) ldxcl $fd,index(base)		load a word or doubleword from memory to FPR (GPR+GPR addressing) fd=memory[base+index] (**MIPS-IV**)
swxcl $fs,index(base) sdxcl $fs,index(base)		store a word or doubleword to memory from FPR (GPR+GPR addressing) fs=memory[base+index] (**MIPS-IV**)
Co-processor ''condition'' tests:		
bc0t label **bc2t label** **bc3t label**		/* goto label if corresponding BrCond * input is active */
bc0f label **bc2f label** **bc3f label**		/* goto label if corresponding BrCond * input is inactive */
bcztl label		/* goto label if coprocessor "z" conditional signal true ; else nullify delay slot instruction*/ (**MIPS-II**)
bczfl label		/* goto label if coprocessor "z" conditional signal false; else nullify delay slot instruction*/ (**MIPS-II**)
Trap instructions:		
break code		trap(BREAK, code);
sdbbp		software debug breakpoint (**RC32364** only)
syscall		trap(SYSCALL, 0)
teq rs,rt		if (rs == rt) trap exception occurs; (**MIPS-II**)

teq rs,rt,code	**bne rs,rt,1f**
	nop
	break code
	1:
		/* RC4xxx compatibility instruction for (**MIPS-I)**
		if (rs == rt)
		trap(BREAK, code);

teqi rs,imm		if (rs == (sign extended)imm) trap exception occurs; (**MIPS-II**)
tge rs,rt,code	**slt $at,rs,rt** **bne $at,$zero,1f** **nop** **break code** **1:**	if ((signed)rs >= (signed)rt) trap(BREAK, code);
tge rs,rt		if ((signed)rs >= (signed)rt) trap exception occurs; (**MIPS-II**)
tgei rs,rt		if ((signed)rs >= (sign extended)imm) trap exception occurs; (**MIPS-II**)

Notes

Assembler	Expands To	What It Does
tgeu rs,rt,code	**sltu $at,rs,rt** **bne $at,$zero,1f** **nop** **break code** **1:**	if ((unsigned)rs >= (unsigned)rt) trap(BREAK, code);
tgeu rs,rt		if ((unsigned)rs >= (unsigned)rt) trap exception occurs; (**MIPS-II**)
tgeiu rs,rt		if ((unsigned)rs >= (unsigned)immediate value) trap exception occurs; (**MIPS-II**)
tlt rs,rt,code	**slt $at,rs,rt** **beq $at,$zero,1f** **nop** **break code** **1:**	if ((signed)rs <(signed)rt) trap(BREAK, code);
tlt rs,rt		if ((signed)rs <(signed)rt) trap exception occurs; (**MIPS-II**)
tlti rs,rt		if ((signed)rs <(sign extended)immediate value) trap exception occurs; (**MIPS-II**)
tltu rs,rt,code	**sltu $at,rs,rt** **beq $at,$zero,1f** **nop** **break code** **1:**	if ((unsigned)rs <(unsigned)rt) trap(BREAK, code);
tltu rs,rt		if ((unsigned)rs <(unsigned)rt) trap exception occurs; (**MIPS-II**)
tltiu rs,rt		if ((unsigned)rs <(unsigned)immediate value) trap exception occurs; (**MIPS-II**)
tne rs,rt,code	**beq rs,rt,1f** **nop** **break code** **1:**	if (rs != rt) trap(BREAK, code);
tne rs,rt		if (rs != rt) trap exception occurs; (**MIPS-II**)
Floating point instructions: (All come in both ".d" (64-bit) and ".s" (32-bit) forms; only ".d" are listed.)		
Test and set condition flag instructions:		

c.f.d		if (unordered(fs,ft))
		trap(FPINT);
		fpcond = 0;

c.sf.d		fpcond = 0;

c.un.d		if (unordered(fs,ft))
		trap(FPINT);
		fpcond = unordered(fs,ft);

c.ngle.d		fpcond = unordered(fs,ft);

c.eq.d		if (unordered(fs,ft))
		trap(FPINT);
		fpcond = (fs == ft);

c.seq.d		fpcond = (fs == ft);

Notes

Assembler	Expands To	What It Does
c.ueq.d		if (unordered(fs,ft)) fpcond = (fs == ft) || unordered(fs,ft);
c.ngl.d		fpcond = (fs == ft) || unordered(fs,ft);

c.olt.d		if (unordered(fs,ft))
		trap(FPINT);
		fpcond = (fs <ft);

c.lt.d		fpcond = (fs <ft);

c.ult.d		if (unordered(fs,ft))
		trap(FPINT);
		fpcond = (fs <ft) || unordered(fs,ft);

c.nge.d		fpcond = (fs <ft) || unordered(fs,ft);

c.ole.d		if (unordered(fs,ft))
		trap(FPINT);
		fpcond = (fs <= ft);

c.le.d		fpcond = (fs <= ft);

c.ule.d		if (unordered(fs,ft))
		trap(FPINT);
		fpcond = (fs <= ft) || unordered(fs,ft);

c.ngt.d		fpcond = (fs <= ft) || unordered(fs,ft);
FP move		
mov.d fd,fs		fd = fs;
Conditional FP move ((MIPS-IV) only): (Only double precision is shown here, but single precision exists also.)		
movt.d fd,fs,cc		if fp condition cc=1, fd=fs (**MIPS-IV**)
movf.d fd,fs,cc		if fp condition cc=0, fd=fs (**MIPS-IV**)
mov2.d fd,fs,rt		if GPR rt=0, fd=fs (**MIPS-IV**)
movn.d fd,fs,rt		if GPR rt≠0, fd=fs (**MIPS-IV**)
Unary arithmetic: These operations are implemented by operating only on the sign bit, so invalid values are not a concern, and they never trap.		
abs.d fd,fs		fd = (fs > 0) ? fs: -fs;
abs.d fd	**abs.d fd,fd**	fd = (fd > 0) ? fd: -fd;
neg.d fd,fs		fd = -fs;
neg.d fd	**neg.d fd,fd**	fd = -fd;
Convert between formats: (cvt.X.Y should be read "convert TO X FROM Y")		
cvt.d.s fd,fs		fd = (double) ((float) fs);
cvt.d.s fd	**cvt.d.s fd,fd**	fd = (double) ((float) fd);
cvt.d.l fd,fs		fd = (double) (long long) fs); **(MIPS-III)**
cvt.d.l fd	**cvt.d.l fd,fd**	fd = (double) ((long long) fd); **(MIPS-III)**
cvt.d.w fd,fs		fd = (double) ((int) fs);
cvt.d.w fd	**cvt.d.w fd,fd**	fd = (double) ((int) fd);

Notes

Assembler	Expands To	What It Does
cvt.s.d fd,fs		fd = (float) ((double) fs);
cvt.s.d fd	**cvt.s.d fd,fd**	fd = (float) ((double) fd);
cvt.s.l fd,fs		fd = (float)((long long) fs); **(MIPS-IV)**
cvt.s.l fd	**cvt.s.l fd,fd**	fd = (float)((long long) fd); **(MIPS-IV)**
cvt.s.w fd,fs		fd = (float)((int) fs);
cvt.s.w fd	**cvt.s.w fd,fd**	fd = (float)((int) fd);
cvt.l.d fd,fs		/* note 64-bit fixed point value is chosen * according to rounding mode */ fd = (long long)((double) fs); **(MIPS-IV)**
cvt.l.d fd	**cvt.l.d fd,fd**	fd = (long long)((double) fd); **(MIPS-IV)**
cvt.l.s fd,fs		fd = (long long)((float) fs); **(MIPS-IV)**
cvt.l.s fd	**cvt.l.s fd,fd**	fd = (long long)((float) fd); **(MIPS-IV)**
cvt.w.d fd,fs		/* note integer value is chosen * according to rounding mode */ fd = (int)((double) fs);
cvt.w.d fd	**cvt.w.d fd,fd**	fd = (int)((double) fd);
cvt.w.s fd,fs		fd = (int)((float) fs);
cvt.w.s fd	**cvt.w.s fd,fd**	fd = (int)((float) fd);
Convert from floating-point to integer using an explicit rounding mode: *Note: **rt** is used as a temporary.*		
ceil.l.d fd,fs		fd = (long long) ceil((double) fd); **(MIPS-III)**
ceil.w.d fd,fs,rt	**cfc1 rt,$31** **nop** **ori $at,rt,3** **xori $at,$at,1** **ctc1 $at,$31** **nop** **cvt.w.d fd,fs** **ctc1 rt,$31**	fd = ceil((double) fd);
floor.l.d fd,fs		fd = (long long)floor((double) fd); **(MIPS-III)**
floor.w.d fd,fs,rt	**cfc1 rt,$31** **nop** **ori $at,rt,3** **xori $at,$at,0** **ctc1 $at,$31** **nop** **cvt.w.d fd,fs** **ctc1 rt,$31**	fd = floor((double) fd);
round.l.d fd,fs,rt		fd = (long long) round((double) fd); **(MIPS-III)**
round.w.d fd,fs,rt	**cfc1 rt,$31** **nop** **ori $at,rt,3** **xori $at,$at,2** **ctc1 $at,$31** **nop** **cvt.w.d fd,fs** **ctc1 rt,$31**	fd = round((double) fd);

Notes

Assembler	Expands To	What It Does
trunc.l.d fd,fs		fd = (long long) ((double) fd); **(MIPS-III)**
trunc.w.d fd,fs,rt	**cfc1 rt,$31** **nop** **ori $at,rt,3** **xori $at,$at,2** **ctc1 $at,$31** **nop** **cvt.w.d fd,fs** **ctc1 rt,$31**	fd = (int) ((double) fd);
ceil.l.s fd,fs		fd = (long long)ceil((float) fd); **(MIPS-III)**
ceil.w.s fd,fs,rt	***see above***	fd = ceil((float) fd);
floor.l.s fd,fs		fd = (long long)floor((float) fd); **(MIPS-III)**
floor.w.s fd,fs,rt	***see above***	fd = floor((float) fd);
round.l.s fd,fs,rt		fd = (long long)round((float) fd); **(MIPS-III)**
round.w.s fd,fs,rt	***see above***	fd = round((float) fd);
trunc.l.s fd,fs		fd = (long long) ((float) fd); **(MIPS-III)**
trunc.w.s fd,fs,rt	***see above***	fd = (int) ((float) fd);
Arithmetic operations: (All can trap under some circumstances.)		
add.d fd,fs,ft		fd = fs + ft;
add.d fd,fs	**add.d fd,fd,fs**	fd += fs;
div.d fd,fs,ft		fd = fs/ft;
div.d fd,fs	**div.d fd,fd,,fs**	fd /= fs;
mul.d fd,fs,ft		fd = fs*ft;
mul.d fd,fs	**mul.d fd,fd,fs**	fd *= fs;
sqrt.d fd,fs		fd = square-root of (double)fs; **(MIPS-III)**
sqrt.s fd,fs		fd = square-root of (float)fs; **(MIPS-III)**
sub.d fd,fs,ft		fd = fs - ft;
sub.d fd,fs	**sub.d fd,fd,fs**	fd -= fs;
recip.s fd,fs		fd = 1.0/fs (float); **(MIPS-IV)**
recip.d fd, fs		fd = 1.0/fs (double); **(MIPS-IV)**
rsqrt.s fd, fs		fd = 1.0/sqrt(fs) (float); **(MIPS-IV)**
rsqrt.d fd, fs		fd = 1.0/sqrt(fs) (double); **(MIPS-IV)**
madd.s fd,fr,fs,ft		fd = (fs ∗ ft) + fr (float); **(MIPS-IV)**
madd.d fd,fr,fs,ft		fd = (fs ∗ ft) + fr (double); **(MIPS-IV)**
msub.s fd,fr,fs,ft		fd = (fs ∗ ft) - fr (float); **(MIPS-IV)**
msub.d fd,fr,fs,ft		fd = (fs ∗ ft) - fr (double); **(MIPS-IV)**
nmadd.s fd,fr,fs,ft		fd = ((fs ∗ ft) + fr) (float); **(MIPS-IV)**
nmadd.d fd,fr,fs,ft		fd = -((fs ∗ ft) + fr) (double); **(MIPS-IV)**
nmsub.s fd, fr, fs, ft		fd = -((fs ∗ ft) - fr) (float); **(MIPS-IV)**

Notes

Assembler	Expands To	What It Does
nmsub.d fd, fr, fs, ft		fd = -((fs * ft) - fr) (double); **(MIPS-IV)**
Conditional branch following test:		
bc1f label		if (!fpcond) goto label;
bc1fl label		if (!fpcond) goto label; else nullify delay slot instruction **(MIPS-II)**
bc1t label		if (fpcond) goto label;
bc1tl label		if (fpcond) goto label; else nullify delay slot instruction; **(MIPS-II)**
Move data between FP and integer register:		
mfc1 rd,fs		/* no format conversion done, just copies * bits. Can use odd-numbered fp registers */ rd = fs;
mtc1 rs,fd		/* no format conversion done, just copies * bits. Can use odd-numbered fp registers */ fd = rs;
dmfc1 $t2,$f2		(**MIPS-III**)
dmtc1 $t2,$f2		(**MIPS-III**)
mtc1.d $t2,$f2	**mtc1 $t2,$f3** **mtc1 $t3,$f2**	/* move a double value (just bits, no * conversion)from integer register pair *to FP reg pair */
CPU control instructions (privileged mode only):		
mfc0 rd, nn		rd = (contents of CPU control reg nn);
mtc0 rs, nn		(CPU control reg nn) = rs;
tlbr **tlbwi** **tlbwr** **tlbpr**		These instructions are used to setup the TLB (memory management hardware) and are described in Chapters 2 & 3.
eret		Used at the end of an exception routine in **RC4xxx / RC32364 / RC5000** Next instruction is not executed (unlike in *rfe*) Do not place *eret* itself in delay slot
rfe		Used at the end of an exception routine in **RC30xx** Restores kernel-mode and global interrupt enable bits from the 3-level “stack” in the status register SR. See chapter 3.
deret		Debug Exception Return; only in **RC32364**
cache op, addr		operate directly on primary cache; **RC4xxx / RC32364**
wait		reduce power consumption ; **RC32364/RC4600/RC4650/ RC4700/RC5000**

Notes

Alphabetic List of Assembler Instructions

In this list real hardware instructions are marked with a dagger. The instructions in this list are supported by all MIPS CPUs. Instructions uniquely supported in later ISAs are described in the next section.

List of RC30xx Instructions

abs rd,rs: integer absolute value
abs.d fd,fs†: FP double precision absolute value
abs.s fd,fs†: FP single precision absolute value
add rd,rs,rt_imm†: add, trap on overflow
add.d fd,fs,ft†: FP double precision add
add.s fd,fs1,fs2†: FP single precision add
addi rd,rs,imm†: add immediate, trap on overflow
addiu rd,rs,imm†: add immediate, never trap
addu rd,rs,rt_imm†: add, never trap
and rd,rs,rt_imm†: logical AND
andi rd,rs,imm†: logical AND immediate
b label: PC-relative subroutine call
bal label: PC-relative subroutine call
bc0f offset†: branch if CPCOND input signal inactive
bc0t offset†: branch if CPCOND input signal active
bc1f label†: branch if FP condition bit clear
bc1t label†: branch if FP condition bit set
beq rs,imm,label: branch if rs == immediate value
beq rs,rt,label†: branch if rs == rt
beqz rs,label: branch if rs is zero
bge rs,rt,label: branch if rs ≥ rt (signed compare)
bge rs,imm,label: branch if rs ≥ immediate value(signed compare)
bgeu rs,rt,label: branch if rs ≥ rt (unsigned compare)
bgeu rs,imm,label: branch if rs ≥ immediate value (unsigned compare)
bgez rs,label†: branch if rs ≥ 0 (signed)
bgezal rs,label†: branch to subroutine if rs == 0
bgt rs,rt,label: branch if rs > rt (signed)
bgt rs,imm,label: branch if rs > immediate value (signed)
bgtu rs,rt,label: branch if rs > rt (unsigned)
bgtu rs,imm,label: branch if rs > immediate value (unsigned)
bgtz rs,label†: branch if rs > 0 (signed)
ble rs,rt,label: branch if rs ≤ rt (signed)
ble rs,imm,label: branch if rs ≤ immediate value(signed)
bleu rs,rt,label: branch if rs ≤ rt (unsigned)
bleu rs,imm,label: branch if rs ≤ immediate value(unsigned)
blez rs,label†: branch if rs ≤ 0 (signed)
blt rs,rt,label: branch rs <rt (signed)
blt rs,imm,label: branch rs < immediate value(signed)
bltu rs,rt,label: branch rs <rt (unsigned)
bltu rs,imm,label: branch rs < immediate value(unsigned)
bltz rs,label†: branch if rs <0 (signed)
bltzal rs,label†: branch to subroutine if rs <0 (signed)
bne rs,rt,label†: branch if rs not equal to rt
bne rs,imm,label: branch if rs not equal to immediate value
bnez rs,label: branch if rs not zero
break†: trap with "breakpoint" Cause field
c.*XXX*.d fs1,fs2†: FP compare, set FP condition (double). **c.***XXX*.s fs1,fs2†: FP compare, set FP condition (single)

Notes

ceil.w.d fd, fs†: arithmetically convert double fs to single fixed-point format and put it in fd; round to +infinity
ceil.w.s fd, fs†: arithmetically convert single fs to single fixed-point format and put it in fd; round to +infinity
cfc1 rd, crs†: move FP control register "crs" contents to rd
ctc1 rs, crd†: move rs contents to FP control register "crs"
cvt.*X.Y* fd,fs†: FP convert from format Y to X. Y and X can be "d" for double-precision, "s" for single-precision, and "w" for 32-bit signed integer value held in an FP register. **div rd,rs,rt**†: rd = rs/rt, trap division by zero or overflow
div.d fd,fs,ft†: FP double precision divide
div.s fd,fs1,fs2†: FP single precision divide
divu rd,rs,rt†: rd = rs/rt; trap divide by zero but not overflow
dmfc1 rd,fs: move contents of FP register fs to rd and next reg
flush rd,offset(rs): same as lwr
floor.w.d fd, fs†: arithmetically convert double fs to single fixed-point format and put it in fd; round to -infinity
floor.w.s fd, fs†: arithmetically convert single fs to single fixed-point format and put it in fd; round to -infinity
invalidate rs2,offset(rs1): same as swr
j label†: jump to label
j rs: indirect jump to address stored in rs
jal label†: call subroutine at label (return address in ra/$31)
jal rd,label: call subroutine but put return address in rd
jalr rs†: call subroutine who's address is in rs (return in ra/$31)
jalr rd,rs†: call subroutine at rs but put return address in rd
jr rs†: indirect jump to address stored in rs
l.d fd, offset(rs): load 64 bits to FP register
l.s fd, offset(rs): load 32 bits to FP register
la rd,label: load rd with address of label
lb rd,offset(rs)†: load byte from memory and sign-extend
lbu rd,offset(rs)†: load byte from memory and zero-extend
lcache rd,offset(rs): same as lwl
lh rd,offset(rs)†: load half-word (16bits) from memory and sign-extend
lhu rd,offset(rs)†: load half-word (16bits) from memory and sign-extend
li rd,imm: load constant value "imm" into rd
li.d rt, imm: load 64-bit FP constant to general register
li.d fd, imm: load 64-bit FP constant to FP register
li.s rt, imm: load 32-bit FP constant to general register
li.s fd, imm: load 32-bit FP constant to FP register
lui rd,imm†: load "imm" into the high bits of rd, zeroing low bits
lw rd,offset(rs)†: load word (32bits) into register
lwu rd,offset(rs)†: load unsigned word (32bits) into register **lwc1 fd,offset(rs)**†: load 32-bits from memory to FP register
lwl rd,offset(rs)†: load word left, used for unaligned loads.
lwr rd,offset(rs)†: load word right, used for unaligned loads.
mfc0 rd, crs†: move contents of CPU control register crs to rd
mfc1 rd,fs†: move contents of FP register fs to rd
mfhi rd†: put multiply result high word or divide's remainder in rd
mflo rd†: put multiply result low word or divide result in rd
mov.d fd,fs†: move FP double from fs to fd
mov.s fd,fs†: move FP single from fs to fd
move rd,rs: move data from register rs to rd
mtc0 rs, crd†: put contents of rs into CPU control register crd
mtc1 rs,fd†: put bits from rs into FP register
mtc1.d $t2,$f2: put 64 bits from register pair starting at rs to FP register
mthi rs†: put contents of rs into multiply unit "hi" register
mtlo rs†: put contents of rs into multiply unit "lo" register
mul rd,rs,rt: rd = rs*rt, signed multiply, no overflow trap

Notes

mul rd,rs,imm: rd = rs*immediate value, signed multiply, no overflow trap
mul.d fd,fs,ft†: FP double precision multiply
mul.s fd,fs1,fs2†: FP single precision multiply
mulo rd,rs,rt: rd = rs*rt, signed, will trap if overflows
mulo rd,rs,imm: rd = rs*immediate value, signed, will trap if overflows
mulou rd,rs,rt: rd = rs*rt unsigned, will trap if overflows
mulou rd,rs,imm: rd = rs*immediate value unsigned, will trap if overflows
mult rs,rt†: start multiplying rs*rt as signed values
multu rs1, rs2†: start multiplying rs*rt as unsigned values
neg rd,rs: rd = -rs, trap on overflow
neg.d fd,fs†: fd = -fs, double FP, never traps
neg.s fd,fs†: fd = -fs, single FP, never traps
negu rd,rs: rd = -rs, no overflow check
nor rd,rs,rt†: rd = logical NOR of rs and rt
nor rt,rs,imm: rd = logical NOR of rs and immediate value
not rd,rs: rd = ~rs, logical NOT
or rd,rs,rt_imm†: rd = rs | rt, logical OR
ori rd,rs,imm†: logical OR, immediate form (don't need to code this)
rem rd,rs,rt: rd = remainder of rs/rt, signed, trap divide by zero and overflow
rem rd,rs,imm: rd = remainder of rs/immediate value, signed, trap divide by zero and overflow
remu rd,rs,rt: rd = remainder of rs/rt, unsigned, trap divide by zero
remu rd,rs,imm: rd = remainder of rs/immediate value, unsigned, trap divide by zero
rfe†: restores CPU status register at end of exception processing; not available in RC4xxx and RC32364
rol rd,rs,rt: rd = rs rotated left by rt
rol rd,rs,imm: rd = rs rotated left by immediate value
ror rd,rs,rt: rd = rs rotated right by rt
ror rd,rs,imm: rd = rs rotated right by immediate value
round.w.d fd, fs†: arithmetically convert double fs to single fixed-point format and put it in fd; round to nearest
round.w.s fd, fs†: arithmetically convert single fs to single fixed-point format and put it in fd; round to nearest
s.d fs,offset(rs): store 64 bits from FP register
s.s fs,offset(rs): store 32 bits from FP register
sb rs2,offset(rs1)†: store byte to memory
scache rs2,offset(rs1): same as swl
seq rd,rs,rt: set rd to 1 if rs == rt, 0 otherwise
seq rd,rs,imm: set rd to 1 if rs == immediate val, 0 otherwise
sge rd,rs,rt: set rd to 1 if rs ≥ rt (signed), 0 otherwise
sge rd,rs,imm: set rd to 1 if rs ≥ imm val(signed), 0 otherwise
sgeu rd,rs,rt: set rd to 1 if rs ≥ rt (unsigned), 0 otherwise
sgeu rd,rs,imm: set rd to 1 if rs ≥ immediate value (unsigned), 0 otherwise
sgt rd,rs,rt: set rd to 1 if rs > rt (signed), 0 otherwise
sgt rd,rs,imm: set rd to 1 if rs > immediate value(signed), 0 otherwise
sgtu rd,rs,rt: set rd to 1 if rs > rt (unsigned), 0 otherwise
sgtu rd,rs,imm: set rd to 1 if rs > immediate value(unsigned), 0 otherwise
sh rs2,offset(rs1)†: store half-word (16bits) to memory
sle rd,rs,rt: set rd to 1 if rs ≤ rt (signed), 0 otherwise
sle rd,rs,imm: set rd to 1 if rs ≤ immediate value(signed), 0 otherwise
sleu rd,rs,rt: set rd to 1 if rs ≤ rt (unsigned), 0 otherwise
sleu rd,rs,imm: set rd to 1 if rs ≤ immediate value (unsigned), 0 otherwise
sll rd,rs,rt†: rd = rs shifted left (bigger) by rt (max 31)
sllv rd,rs1,rs2†: rd = rs shifted left (bigger) by rt (max 31)
slt rd,rs,rt_imm†: set rd to 1 if rs <rt_imm (unsigned), 0 otherwise
slti rd,rs,imm†: set rd to 1 if rs <imm (signed), 0 otherwise
sltiu rd,rs,imm†: set rd to 1 if rs <imm (unsigned), 0 otherwise

Notes

sltu rd,rs,rt_imm†: set rd to 1 if rs <rt_imm (unsigned), 0 otherwise
sne rd,rs,rt: set rd to 1 if rs not equal to rt, 0 otherwise
sne rd,rs,imm: set rd to 1 if rs not equal to immediate value, 0 otherwise
sra rd,rs,rt†: rd = rs shifted right by rt, sign bit propagates down
srav rd,rs,rt†: rd = rs shifted right by rt, sign bit propagates down
srl rd,rs,rt†: rd = rs shifted right by rt, zeroes from top
srlv rd,rs,rt†: rd = rs shifted right by rt, zeroes from top
sub rd,rs,rt_imm†: rd = rs – rt_imm, trap if overflows
sub.d fd,fs,ft†: FP double precision subtract
sub.s fd,fs1,fs2†: FP single precision subtract
subu rd,rs,rt†: rd = rs – rt, no trap on overflow
subu rd,rs,imm: rd = rs – immediate value, no trap on overflow
sw rs2,offset(rs1)†: store word (32 bits) to memory
swc1 fs, offset(rs)†: store FP register value to memory
swl rs2,offset(rs1)†: store word left, used for unaligned stores
swr rs2,offset(rs1)†: store word right, used for unaligned stores
syscall†: trap with a "syscall" cause value
tlbp†: TLB (memory management unit) maintenance instruction
tlbr†: TLB (memory management unit) maintenance instruction
tlbwi†: TLB (memory management unit) maintenance instruction
tlbwr†: TLB (memory management unit) maintenance instruction
trunc.w.d fd, fs†: arithmetically convert double fs to single fixed-point format and put it in fd; round to zero
trunc.w.s †: arithmetically convert single fs to single fixed-point format and put it in fd; round to zero
xor rd,rs,rt_imm†: rd = bitwise exclusive-OR of rs and rt_imm
xori rd,rs,imm†: explicit immediate form of "xor"

Alphabetic List of Rc4xxx Assembler Instructions

In addition to the instructions in the previous section, the RC4xxx CPUs support the following instructions. Instructions marked with a "*****" are also supported by RC32364.

List of RC4xxx Instructions

***bcztl label**†: branch if COP "z" condition signal true, else nullify delay slot instruction; "z" = 1 for FPU
***bczfl label**†: branch if COP "z" condition signal false likely, else nullify delay slot instruction; "z" = 1 for FPU
***beql rs, rt, label**†: branch if rs == rt, else nullify delay slot instruction
***beql rs, imm, label**†: branch if rs == immediate value, else nullify delay slot instruction
bgel rs, rt, label: branch if rs ≥ rt(signed); else nullify delay slot instruction
bgel rs, imm, label: branch if rs ≥ immediate value (signed); else nullify delay slot instruction
bgeul rs,rt,label: branch if rs ≥ rt (unsigned compare); else nullify delay slot instruction
bgeul rs,imm,label: branch if rs ≥ immediate value (unsigned compare); else nullify delay slot instruction
***bgezall rs, label**†: unconditionally put return address in *r31* and branch if rs ≥ 0 (signed); else nullify delay slot instruction
***bgezl rs, label**†: branch if rs ≥ 0 (signed); else nullify delay slot instruction
bgtl rs,rt,label: branch if rs > rt (signed); else nullify delay slot instruction
bgtl rs,imm,label: branch if rs > immediate value (signed); else nullify delay slot instruction
bgtul rs,rt,label: branch if rs > rt (unsigned); else nullify delay slot instruction
bgtul rs,imm,label: branch if rs > immediate value (unsigned); else nullify delay slot instruction
***bgtzl rs, label**†: branch if rs > 0 (signed); else nullify delay slot instruction
blel rs,rt,label: branch if rs ≤ rt (signed); else nullify delay slot instruction
blel rs,imm,label: branch if rs ≤ immediate value(signed) ; else nullify delay slot instruction
bleul rs,rt,label: branch if rs ≤ rt (unsigned) ; else nullify delay slot instruction
bleul rs,imm,label: branch if rs ≤ immediate value(unsigned); else nullify delay slot instruction
***blezl rs, label** †: branch if rs ≤ 0 (signed); else nullify delay slot instruction

Notes

bltl rs,rt,label: branch rs <rt (signed); else nullify delay slot instruction
bltl rs,imm,label: branch rs < immediate value(signed); else nullify delay slot instruction
bltul rs,rt,label: branch rs <rt (unsigned) ; else nullify delay slot instruction
bltul rs,imm,label: branch rs < immediate value(unsigned); else nullify delay slot instruction
***bltzall rs, label** †: unconditionally put return address in *r31* and branch if rs < 0 (signed); else nullify delay slot instruction
***bltzl rs, label** †: branch if rs < 0 (signed); else nullify delay slot instruction
***bnel rs, rt, label** †: branch if rs not equal to rt; else nullify delay slot instruction
***bnel rs, imm, label** †: branch if rs not equal to immediate value; else nullify delay slot instruction
***cache op, offset(rs)†:** perform an operation on primary cache
ceil.l.d fd, fs †: arithmetically convert double fs to 64-bit fixed-point format and put it in fd; round to +infinity
ceil.l.s fd, fs†: arithmetically convert single fs to 64-bit fixed-point format and put it in fd; round to +infinity
cvt.d.l fd, fs†: FP convert from 64-bit fixed format to double
cvt.d.l fd: FP convert from 64-bit fixed format to double
cvt.s.l fd, fs†: FP convert from 64-bit fixed format to single
cvt.s.l fd: FP convert from 64-bit fixed format to single
cvt.l.d fd, fs†: FP convert from double to 64-bit fixed format
cvt.l.d fd: FP convert from double to 64-bit fixed format
cvt.l.s fd, fs†: FP convert from single to 64-bit fixed format
cvt.l.s fd: FP convert from single to 64-bit fixed format
dadd rd, rs, rt†: doubleword add, trap on overflow
dadd rt, rs, imm: doubleword add immediate, trap on overflow
daddi rt, rs, imm†: doubleword add immediate, trap on overflow
daddiu rt, rs, imm†: doubleword unsigned add immediate, never trap
daddu rd, rs, rt†: doubleword unsigned add, never trap
daddu rt, rs, imm: doubleword unsigned add immediate, never trap
ddiv rs, rt†: divide rs by rt, both signed, never trap, result in LO special register, remiander in HI special register
ddiv rd, rs, rt: divide rs by rt, both signed, never trap, result in rd, remiander in HI special register
ddiv rd, rs, imm: divide rs by immediate value, both signed, never trap, result in rd, remiander in HI special register
ddivu rs, rt†: divide rs by rt, both unsigned, never trap, result/remainder in special registers LO/HI
ddivu rd, rs, rt: divide rs by rt, both unsigned, never trap, result in rd, remiander in HI special register
ddivu rd, rs, imm: divide rs by immediate val, both unsigned, never trap, result in rd, remiander in HI special register
dmfcz rt, rd†: move doubleword from register rd of coprocessor “z” to general register rt
dmtcz rt, rd†: move doubleword from general register rt to register rd of coprocessor “z”
dmul rd,rs,rt: start multiplying rs*rt as double signed values
dmul rd, rs, imm: start multiplying rs*immediate value as double signed values
dmulo rd,rs,rt: rd = rs*rt, double signed values, will trap on overflows
dmulou rd,rs,rt: rd = rs*rt, double unsigned values, will trap on overflows
dmult rs, rt†: start multiplying rs*rt as double signed values
dmultu rs, rt†: start multiplying rs*rt as double unsigned values
dneg rd,rs: rd = -rs, double values, trap on overflow
dnegu rd,rs: rd = -rs, double values, no trap check
drem rd,rs,rt: rd = remainder of rs/rt, signed double, trap divide by zero and overflow
drem rd,rs,imm: rd = remainder of rs/immediate value, signed double, trap divide by zero and overflow
dremu rd,rs,rt: rd = remainder of rs/rt, unsigned double, trap divide by zero
dremu rd,rs,imm: rd = remainder of rs/immediate value, unsigned double, trap divide by zero
dsll rd, rt, sa†: shift left contents of rt by sa bits and put result in rd, insert zeros in low order bits
dsll rd, rt, rs: similar to **dsll** above, shift amount in lowest 6 bits of rs
dsllv rd, rt, rs†: similar to **dsll** above, shift amount in lowest 6 bits of rs
dsll32 rd, rt, sa†: similar to **dsll** above, shift amount is 32+sa bits
dsra rd, rt, sa†: shift right contents of rt by sa bits, put result in rd, sign extend high order bits
dsra rd, rt, rs: similar to **dsra** above, shift amount in lowest 6 bits of rs

Notes

dsrav rd, rt, rs†: similar to **dsra** above, shift amount in lowest 6 bits of rs
dsra32 rd, rt, sa†: similar to **dsra** above, shift amount is 32+sa bits
dsrl rd, rt, sa†: shift right contents of rt by sa bits and put result in rd, insert zeros in low order bits
dsrl rd, rt, rs: similar to **dsrl** above, shift amount in lowest 6 bits of rs
dsrlv rd, rt, rs†: similar to **dsrl** above, shift amount in lowest 6 bits of rs
dsrl32 rd, rt, sa†: similar to **dsrl** above, shift amount is 32+sa bits
dsub rd, rs, rt†: doubleword subtract rd = rs - rt, trap on overflow
dsub rd, rs, imm: doubleword subtract rd = rs - immediate value, trap on overflow
dsubu rd, rs, rt†: doubleword unsigned subtract rd = rs - rt, never trap
dsubu rd, rs, imm: doubleword unsigned subtract rd = rs -immediate value, never trap
***eret** †: return from interrupt, exception or error trap
floor.l.d fd, fs†: arithmetically convert double fs to 64-bit fixed-point format and put it in fd; round to -infinity
floor.l.s fd, fs†: arithmetically convert single fs to 64-bit fixed-point format and put it in fd; round to -infinity
ld rt, offset(rs)†: load doubleword(64bits) into register rt
ldcz rt, offset(rs)†: load doubleword(64bits) into register rt of coprocessor "z"
ldl rt, offset(base)†: load doubleword left, used for unaligned loads
ldr rt, offset(base)†: load doubleword right, used for unaligned loads
***ll rt, offset(base)**†: load linked word into rt, implicitly performs a SYNC operation
lld rt, offset(base)†: load linked doubleword into rt, implicitly performs a SYNC operation
***mad rs, rt**†: RC4650 only; signed; (hi-lo) = (hi-lo) + rs*rt
***madu rs, rt**†: RC4650 only; unsigned; (hi-lo) = (hi-lo) + rs*rt
***mul rd, rs, rt**†: RC4650 only; rd=rs*rt (all 32 bits only)
round.l.d fd, fs†: arithmetically convert double fs to 64-bit fixed-point format and put it in fd; round to nearest
round.l.s fd, fs†: arithmetically convert single fs to 64-bit fixed-point format and put it in fd; round to nearest
***sc rt, offset(base)**†: store conditional word from general register rt, implicitly performs a SYNC operation
scd rt, offset(base)†: store conditional doubleword from general register rt, implicitly performs a SYNC operation
sd rt, offset(base)†: store doubleword (64bits) into register rt
sdcz rt, offset(base)†: store doubleword (64bits) into register rt of coprocessor "z"
sdl rt, offset(base)†: store doubleword left, used for unaligned loads
sdr rt, offset(base)†: store doubleword right, used for unaligned loads
sqrt.d fd, fs†: square root of double fs put in fd
sqrt.s fd, fs†: square root of single fs put in fd
***sync** †: in multiprocessor environment, finish all loads/stores fetched upto this point before allowing further loads/stores
***teq rs, rt** †: trap if rs equals rt
***teq rs, imm**: trap if rs equals immediate value
***teqi rs, imm**†: trap if rs equals sign-extended immediate value
***tge rs, rt** †: trap if signed rs ≥ rt
***tge rs, imm**: trap if signed rs ≥ immediate value
***tgei rs, imm**†: trap if signed rs ≥ immediate value
***tgeiu rs, imm**†: trap if unsigned rs ≥ immediate value
***tgeu rs, rt**†: trap if unsigned rs ≥ rt
***tgeu rs, imm**: trap if unsigned rs ≥ immediate value
***tlt rs, rt**†: trap if signed rs < rt
***tlt rs, imm**: trap if signed rs < immediate value
***tlti rs, imm**†: trap if signed rs < immediate value
***tltiu rs, imm**†: trap if sign extended rs < immediate value
tltu rs, rt†: trap if unsigned rs < rt
tltu rs, imm: trap if unsigned rs <immediate value
***tne rs, rt**†: trap if rs is not equal to rt
***tne rs, imm**: trap if rs is not equal to immediate value
***tnei rs, imm**†: trap if rs is not equal to sign extended immediate value

Notes

trunc.l.d †: arithmetically convert double fs to 64-bit fixed-point format and put it in fd; round to zero
trunc.l.s †: arithmetically convert single fs to 64-bit fixed-point format and put it in fd; round to zero
***wait** †: reduce power consumption by halting the pipeline

Alphabetic List of RC5000 Assembler Instructions

In addition to the instructions in the previous section, the RC5000 CPUs support the following instructions. Hardware instructions are identified with a dagger. Instructions marked with a "*" are also supported by RC32364.

List of RC5000 Instructions

ldxcl fd,index(base)†:load doubleword from memory to FPR using GPR+GPR addressing
lwxcl fd,index(base)†:load word from memory to FPR using GPR+GPR addressing
madd.d fd,fr,fs,ft†:double precision multiply-add and write result to fd
madd.s fd,fr,fs,ft†:single precision multiply-add and write result to fd
msub.d fd,fr,fs,ft†:double precision multiply-subtract
msub.s fd,fr,fs,ft†:single precision multiply-subtract
movf.d fd,fs,cc†:conditional fp move: if cc=0, fd=fs (double precision)
movf.s fd,fs,cc†:conditional fp move: if cc=0, fd=fs (single precision)
movn rd,rs,rt†:conditional move: if rt¼0, move 'rs' to 'rd'
movn.d fd,fs,rt†:conditional fp move: if GPR rt¼0, fd=fs (double precision)
movn.s fd,fs,rt†:conditional fp move: if GPR rt¼0, fd=fs (single precision)
movt.d fd,fs,cc†:conditional fp move: if cc=0, fd=fs (double precision)
movt.s fd,fs,cc†:conditional fp move: if cc=0, fd=fs (single precision)
***movz rd,rs,rt†**:conditional move: if rt=0, move 'rs' to 'rd'
movz.d fd,fs,rt†:conditional move: if GPR rt=0, fd=fs (double precision)
movz.s fd,fs,rt†:conditional move: if GPR rt=0, fd=fs (single precision)
nmadd.d fd,fr,fs,ft†:negated multiply-add (double precision)
nmadd.s fd,fr,fs,ft†:negated multiply-add (single precision)
nmsub.d fd,fr,fs,ft†:negated multiply-subtract (double precision)
nmsub.s fd,fr,fs,ft†:negated multiply-subtract (single precision)
***pref hint,offset(rs)†**:prefetch
recip.d fd,fs†:double precision approximate reciprocal of fs
recip.s fd,fs†:single precision approximate reciprocal of fs
rsqrt.d fd,fs†:double precision approximate reciprocal sqrt of fs
rsqrt.s fd,fs†:single precision approximate reciprocal sqrt of fs
sdxcl fs, index(base)†: store doubleword from FP register into memory using GPR+GPR addressing
swxcl fs, index(base)†: store word from FP register into memory using GPR+GPR addressing

Alphabetic List of RC32364 Assembler Instructions

In addition to the instructions marked with a "*" in the previous sections, the RC32364 RISController supports the following instructions. Hardware instructions are identified with a dagger.

List of RC32364 Instructions

clo rt,rs†:count leading ones
clz rt,rs†:count leading zeros
deret†:debug exception return
msub rs,rt†:multiply subtract
msubu rs,rt†:multiply subtract unsigned
sdbbp†:software debug breakpoint

C Programming

Notes

An efficient C run-time environment relies on conventions (enforced by compilers and assembly language programmers) about register usage within C-compatible functions.

The Stack, Subroutine Linkage, Parameter Passing

Many MIPS programs are written in mixed languages—for embedded systems programmers, this is most likely to be a mix of C and assembler.

MIPS Corporation established a set of conventions about how to pass arguments to functions (pass parameters to subroutines), and how to return values from functions.

These complex conventions start off quite simply: all arguments are allocated space in a data structure on the stack, but the first few arguments are placed in CPU registers and the stack contents left undefined. In practice, this optimization means that for most function calls the arguments are all passed in registers; but the stack data structure is the best starting point for understanding the process.

Stack Argument Structure

The MIPS hardware does not directly support a stack, but the calling convention defines one. The stack grows downwards and the current stack bottom is kept in register *sp* (alias $29). Any OS which is providing protection and security will make no assumptions about the user's stack, and the value of *sp* doesn't really matter except at the point where a function is called. But it is conventional to keep *sp* at or below the lowest stack location your function has used.

At the point where a function is called, *sp* must be 8-byte aligned (not required by RC3xxx CPU hardware, but defined to simplify compatibility and part of the rules).

To call a subroutine according to the MIPS standard, the caller creates a data structure on the stack to hold the arguments and sets *sp* to point to it. The first argument (left-most in the C source) is lowest in memory. A *word* in the MIPS architecture is 32 bits long. Each argument is expanded to at least 1 word. *Double* (double-precision floating point) values are aligned on an 8-byte boundary (as are data structures which contain a *double* field).

The argument structure really does look like a C *struct*, but there are two differences:

- *There are always at least 4 words worth of bytes in the structure, even if the arguments would fit in less;*
- *each partial word (char or short) argument appears in the structure as what is effectively an int (word length) in memory. This does not apply to partial-word fields inside a struct argument.*

Which Arguments Go in Which Registers?

Arguments assigned in the first 4 words of the argument structure are passed in registers, and the caller can and does leave the first 4 words worth of bytes in the structure undefined. The called function can save the values back in memory if it needs to reconstruct memory-held arguments.

The four words of register argument values go in *a0* through *a3* respectively ($4 through $7), except where the caller can be sure that the data would be better loaded into floating point (FP) registers:

- *Unless the first argument takes a FP value, the FP registers are not used. This ensures that traditional functions like printf still work, although the number and type of arguments are variable. Moreover, it is relatively harmless: the majority of simple FP routines take only FP arguments.*
- *Only two FP values may be passed in registers: and will be in FP registers $f12 and $f14 (implicitly using $f13 and $f15 for double-precision values).*

Notes

Two doubles occupy 4 words, which is all the data expected to be in registers. Historically, functions with lots of single-precision arguments are not frequent enough to make another rule.

If a function returns a structure type, then the return-value convention involves the invention of a pointer as the implicit first argument before the first (visible) argument; this is described in detail below.

Note that the following examples assume 32-bit word size. For 64-bit word size, all addresses would be doubled (the stack pointer increments would be by 8 instead of by 4 as shown below).

Examples from the C library

thesame = strncmp(“bear”, “bearer”, 4);

Leads to an argument structure whose fields are allocated as:

Location	Contents	In register
sp+12	<undefined>	-
sp+8	4	a2
sp+4	address of "bearer"	a1
sp+0	address of "bear"	a0

There are less than 4 words of arguments, so they all fit in registers.

That seems like a complex way of deciding to put three arguments into the usual registers. However, its value is clearer in the case of something a bit more tricky from the math library:

double ldexp (double, int);

y = ldexp(x, 23); /* y = x * (223) */**

The arguments come out as

Location	Contents	In register
sp+12	<undefined>	-
sp+8	23	a2
sp+4	(double) x	$f12/$f13
sp+0		

Passing Structures

C allows the programmer to use structure types as arguments (it is much more common practice to pass pointers to structures instead, but the language supports both). In MIPS the structure forms part of the ‘‘argument structure’’. In the following example:

```
struct thing {
        char letter;
        short count;
        int value;
} = {"z", 46, 100000};

(void) processthing (thing);
```

Location	Contents			In register
sp+4	100000			a1
sp+0	‘‘z’’	<pad>	46	a0

In a big-endian CPU, the result of this is that the *char* value in the structure should end up in the most-significant 8 bits of the argument register, but packed together with the *short*.

Notes

How printf() and varargs work

Consider this example, which assumes 32-bit registers:

printf ("length = %f, width = %f, num = %dn", 1.414, 1.0, 12);

Location	Contents	In register
sp+24	12	<value here>
sp+20	(double) 1.0	<value here>
sp+16		
sp+12	(double) 1.414	a3
sp+8		a2
sp+4	<padding>	-
sp+0	pointer to format string	a0

Note:

- *The padding at sp +4 is required to get correct alignment of the double values (the C rule is that floating point arguments are always passed as double unless the programmer explicitly asks otherwise with a typecast or function prototype).*
- *Because the first argument is not a floating point value, the compiler doesn't use an FP register for the second argument either. The data will instead be loaded into the two registers a2 and a3.*

The *printf*() subroutine is defined with the ''stdarg'' or ''varargs'' macro package, which provides a portable cover for the register and stack manipulation involved. The *printf* routine picks off the arguments by taking the address of the first or second argument, and then can advance up the argument structure to find further arguments.

However, the macro package also has to persuade the C compiler to copy *a0* through *a3* into their "shadow" locations in the argument structure. Some compilers will detect the use of the address of an argument and take the hint; ANSI C compilers should react to "..." in the function definition; others may need a "pragma".

This should clarify the value of placing the *double* value into the integer registers; that way ''stdarg'' and the compiler can just store the registers *a0- a3* into the first 16 bytes of the argument structure, regardless of the type or number of the arguments.

Returning Value from a Function

An integer or pointer return value will be in register *v0* ($2). Register *v1* ($3) is reserved by the MIPS ABI but many compilers don't use it. However, expect it to be used for returning 64-bit integer values in certain compilers (probably as a *long long* data type).

Any floating point result comes back in register *$f0* (implicitly using *$f1* if the value is double precision).

If a function is declared in C as returning a structure value, that value is not returned in registers. Instead an additional implicit argument, a pointer to a caller-supplied structure template, is prepended to the explicit arguments; and the called function copies its return value to the template. Following the normal rules for arguments, the "implicit" first argument will be in register *a0* when the function is called. On return *v0* points to the returned structure, too.

Stack-frame Allocation

Functions can be divided into three classes; three different approaches satisfy most programming needs.

Notes

Leaf functions

Functions which contain no calls to other functions are called *leaf* functions. Because of this, they don't have to worry about setting up argument structures and can safely maintain data in the non-preserved registers *t0 – t7, a0 – a3* and *v0 – v1, t8 – t9, and AT* and may use the stack for storage if required. They can leave the return address in register *ra* and return directly to it.

Most functions written in assembler for tuning reasons, or as convenience functions for accessing features not visible in C, will be leaf functions. The declaration of such a function is very simple. For example:

```
#include <idtc/asm.h>
#include <idtc/regdef.h>

LEAF(myleaf)
        ...
        <system specific code goes here>
        ...
        j       ra
END(myleaf)
```

Most toolchains can pass assembler source code through the C macro pre-processor before assembling it. The files <*idtc/asm.h*> and <*idtc/regdef.h*> include useful macros (like LEAF and END, above) for declaring global functions and data; they also allow the use of software register names, e.g. *a0* instead of *$4.* If using the MIPS Corp. toolchain, for example, the above fragment would be expanded to:

```
        .globl  myleaf
        .ent    myleaf,0
        ...
        <system specific code goes here>
        ...
        j       $31
        .end    myleaf
```

Other toolchains may have different definitions for these macros, as appropriate to their needs.

Non-leaf functions

Non-leaf functions are those which contain calls to other functions. Normally the function starts with code (the "function prologue") to reset *sp* to the low-water mark of argument structures for any functions which may be called, and to save the incoming values of any of the registers *s0 – s8* which the function uses. Stack locations must also be reserved for *ra,* automatic (i.e. stack-based local) variables, and any further registers whose value this function needs preserved over its own calls (if the values of the argument registers *a0 – a3* need to be preserved, they can be saved into their standard positions on the "argument structure").

Note that, since *sp* is set only once (in the function prologue) all stack-held locations can be referenced by fixed offsets from *sp.*

This is illustrated in the non-leaf function listed below, in conjunction with the picture of the stackframe in Figure 10.1.

Notes

Figure 10.1 Stackframe for a Non-leaf Function

```
#include <idtc/asm.h>
#include <idtc/regdef.h>

#
# myfunc (arg1, arg2, arg3, arg4, arg5)
#

# framesize = locals + regsave (ra,s0) + pad + fregsave (f20/21) + args + pad
myfunc_frmsz= 4 + 8 + 4 + 8 + (5 * 4) + 4

NESTED(myfunc, myfunc_frmsz, zero)
        subu    sp,myfunc_frmsz
        .mask   0x80010000, -4
        sw ra,myfunc_frmsz-8(sp)
        sw s0,myfunc_frmsz-12(sp)
        .fmask  0x00300000, -16
        s.d     $f20,myfunc_frmsz-24(sp)
        ...
        <your code goes here, e.g>
        # local = otherfunc (arg5, arg2, arg3, arg4, arg1)
        sw      a0,16(sp)                       # arg5 (out) = arg1 (in)
        lw      a0,myfunc_frmsz+16(sp)# arg1 (out) = arg5 (in)
        jal     otherfunc
        sw      v0,myfunc_frmsz-4(sp)# local = result
        ...
        l.d $f20,myfunc_frmsz-24(sp)
        lw s0,myfunc_frmsz-12(sp)
        lw ra,myfunc_frmsz-8(sp)
        addu    sp,myfunc_frmsz
        jr ra
END(myfunc)
```

Analyzing the above example, one step at a time:

```
#
# myfunc (arg1, arg2, arg3, arg4, arg5)
#
```

Notes

The function myfunc expects five arguments: on entry the first four of these will be in registers *a0 – a3,* and the fifth will be at *sp+16.*

```
# framesize = locals + regsave (ra,s0) + pad + fregsave (f20/21) + args + pad
myfunc_frmsz= 4 + 8 + 4 + 8 + 20 + 4
```

The total frame size is calculated as follows:

- *locals (4 bytes): keep one local variable on the stack, rather than in a register; the example may need to pass the address of the variable to another function.*
- *regsave (8 bytes): save the return address register ra, because this function calls another function; this function also plans to use the callee-saved register s0.*
- *pad (4 bytes): the rules say that double precision floating-point must by 8-byte aligned, so add one word of padding to align the stack.*
- *fregsave (8 bytes): the function plans to use $f20, which is one of the callee-saved floating-point registers.*
- *argsize (20 bytes): this function is going to call another function which needs five argument words; this size must never be less than 16 bytes if a nested function is called, even if it takes no arguments.*
- *pad (4 bytes): the rules say that the stack pointer must always be 8-byte aligned, so add another word of padding to align it.*

```
NESTED(myfunc, myfunc_frmsz, zero)
        subu    sp,myfunc_frmsz
```

In the MIPS Corp. toolchain this would be expanded to:

```
.globl  myfunc
.ent    myfunc,0
.frame  $29,myfunc_frmsz,$0
subu    $29,myfunc_frmsz
```

This declares the start of the function, and makes it globally accessible. The **.frame** function tells the debugger the size of stack frame to be created, and finally the **subu** instruction creates the stack frame itself.

```
.mask   0x80010000, -4
sw ra,myfunc_frmsz-8(sp)
sw s0,myfunc_frmsz-12(sp)
```

The function must save the return address and any *callee-saved* integer registers used, in the stack frame. The **.mask** directive tells the debugger which registers will be saved*($31* and *$20),* and the offset from the top of the stack frame to the top of the save area: this corresponds to *regoffs.* The **sw** instructions then save the registers: the higher the register number, the higher up the stack it is placed (i.e. the registers are saved in order).

```
.fmask  0x00300000, -16
s.d     $f20,myfunc_frmsz-24(sp)
```

The code then does the same thing for the *callee-saved* floating-point registers *$f20* and (implicitly) *$f21.* The **.fmask** offset corresponds to *fregoffs*, i.e. local variable area + integer register save area + padding word.

```
# local = otherfunc (arg5, arg2, arg3, arg4, arg1)
sw      a0,16(sp)                   # arg5 (out) = arg1 (in)
```

Notes

```
        lw      a0,myfunc_frmsz+16(sp) # arg1 (out) = arg5 (in)
        jal     otherfunc
```

This program calls the function otherfunc. Its arguments 2 to 4 are the same as this programs' arguments 2 to 4, so these can pass straight through without being moved. However, the code must swap argument 5 and argument 1, so it copies:

- *its input arg1 (in register a0) to the arg5 position in the outgoing argument build area (new sp + 16).*
- *its input arg5 (at old sp + 16) to outgoing argument 1 (register a0).*

```
        sw      v0,myfunc_frmsz-4(sp)# local = result
```

The return value from otherfunc is stored in the local (automatic) variable, allocated the top 4 bytes of the stack frame.

```
        l.d $f20,myfunc_frmsz-24(sp)
        lw s0,myfunc_frmsz-12(sp)
        lw ra,myfunc_frmsz-8(sp)
        addu    sp,myfunc_frmsz
        jr ra
END(myfunc)
```

Finally the function epilogue reverses the prologue operations: restores the floating-point, integer and return address registers; pops the stack frame; and returns.

Functions Needing Run-time Computed Stack Locations

In some languages dynamic variables can be created whose size varies at run-time. Some C compilers support this, by using the library function *alloca*. This means that *sp* has been lowered by an amount unknown at compile time, so the compiler can't use it to reach stack locations. In this case, the function prologue uses another register, *s8,* also known as *fp,* and points it to the post-prologue value of *sp.*

Since *fp* is one of the saved registers, the prologue must also save its old value. In the function body, all stack location references to automatic variables, and saved-register positions are made via *fp.* But when calling other functions, and putting data into the argument structure, that will be done with relation to *sp.*

When creating space with *alloca* the address returned is actually a bit higher than *sp,* since the compiler has still reserved space for the largest argument structure required by any function call.

This example is a slightly modified version of the function used in the last section, with the addition of a "call" to *alloca*.

```
#include <idtc/asm.h>
#include <idtc/regdef.h>

#
# myfunc (arg1, arg2, arg3, arg4, arg5)
#

# framesize = locals + regsave (ra,s8,s0) + fregsave (f20/21) + args + pad
myfunc_frmsz= 4 + 12 + 8 + (5 * 4) + 4

        .globl  myfunc
        .ent    myfunc,0
        .frame  fp,myfunc_frmsz,$0

        subu    sp,myfunc_frmsz
        .mask   0xc0010000, -4
        sw ra,myfunc_frmsz-8(sp)
        sw fp,myfunc_frmsz-12(sp)
        sw s0,myfunc_frmsz-16(sp)
        .fmask  0x00300000, -16
        s.d     $f20,myfunc_frmsz-24(sp)
```

Notes

```
        move    fp,sp                           # save bottom of fixed frame
        ...
        # t6 = alloca (t5)
        addu    t5,7                            # make sure that size
        and     t5,~7                           # is multiple of 8
        subu    sp,t5                           # allocate stack
        addu    t6,sp,20                        # leave room for args
        ...
        <your code goes here, e.g>
        # local = otherfunc (arg5, arg2, arg3, arg4, arg1)
        sw      a0,16(sp)               # arg5 (out) = arg1 (in)
        lw      a0,myfunc_frmsz+16(fp)# arg1 (out) = arg5 (in)
        jal     otherfunc
        sw      v0,myfunc_frmsz-4(fp)# local = result
        ...
        move    sp,fp                           # restore stack pointer
        l.d $f20,myfunc_frmsz-24(sp)
        lw s0,myfunc_frmsz-16(sp)
        lw fp,myfunc_frmsz-12(sp)
        lw ra,myfunc_frmsz-8(sp)
        addu    sp,myfunc_frmsz
        jr ra
END(myfunc)
```

There are a few notable differences from the previous example:

```
        .globl  myfunc
        .ent    myfunc,0
        .frame  fp,myfunc_frmsz,$0
```

The function can't use the NESTED macro any more, since it is using a separate frame pointer which must be explicitly declared using the **.frame** directive.

```
        .mask   0xc0010000, -4
        sw ra,myfunc_frmsz-8(sp)
        sw fp,myfunc_frmsz-12(sp)
        sw s0,myfunc_frmsz-16(sp)
```

Since the program will modify *fp (= s8 = $30),* it must save it in the stackframe too.

```
        # t6 = alloca (t5)
        addu    t5,7                    # make sure that size
        and     t5,~7                   # is multiple of 8
        subu    sp,t5                   # allocate stack
        addu    t6,sp,20                # leave room for args
```

This sequence allocates a variable number of bytes on the stack, and sets a register *(t6)* to point to it. The program must make sure that the size is rounded up to a multiple of 8, so that the stack stays correctly aligned. In addition, it must add 20 to the stack pointer, to leave room for the five argument words that will be used in future calls.

```
        sw      a0,16(sp)               # arg5 (out) = arg1 (in)
        lw      a0,myfunc_frmsz+16(fp)# arg1 (out) = arg5 (in)
        jal     otherfunc
        sw      v0,myfunc_frmsz-4(fp)# local = result
```

When building another function's arguments, use the *sp* register; but when accessing input arguments or local variables the program must use the *fp* register.

```
        move    sp,fp                           # restore stack pointer
        l.d $f20,myfunc_frmsz-24(sp)
```

Notes

```
lw s0,myfunc_frmsz-16(sp)
lw fp,myfunc_frmsz-12(sp)
```

Finally, at the start of the function epilogue, restore the stack pointer to its post-prologue position, and then restore the registers (not forgetting to restore the old value of *fp,* of course).

Shared and Non-shared Libraries

A C object library is a collection of pre-compiled modules, which are automatically linked into a program's binary when it refers to a function or variable whose name is defined in the module. Many standard C functions like printf are defined in libraries.

Libraries provide a simple and powerful way of extending the language; but in a multi-tasking OS every program will carry its own copy of the library function. Modern library functions may be huge; for example the graphics interface libraries to the widely-used X window system add about 300Kbytes to the size of a MIPS object, dwarfing the application code of many simpler programs.

In response to this problem most modern OS' provide some way in which library code may be shared between different applications. There are different approaches:

Sharing Code in Single-address Space Systems

In a single address-space OS like VxWorks[1], programs can be linked to library functions by deferring the link operation (which actually fixes up the program code) until the program is loaded into system memory. In this kind of system the library function becomes part of a single large program. But:

- *The libraries must be written to be "re-entrant"; they may be used by different tasks, and one task may be suspended in the middle of a library function and that function re-used by another.*

  For simple operations, re-entrancy is easily achieved by avoiding any use of static modifiable data (so that all computation is done on the stack and in machine registers). However, where library functions must maintain internal data life gets much more complicated; accesses to shared variables must use the programming technique of critical regions protected by semaphores.

  This does mean that library programmers must respect these rules, and can't just recompile existing code into libraries without modification.
- *The run-time system must maintain a symbol table for loading. System utilities such as the debugger also need access to the symbol table and relocation information.*

In such a system a little extra work at load time allows a single copy of a library function to be freely used by the OS kernel, drivers and any number of application tasks. Simple functions suffer very little run-time overhead (the convenient gp-relative addressing optimization, described in the last chapter, cannot be used); the critical region overhead for shared data is unavoidable.

Sharing Code Across Address Spaces

In a "protected" OS where separate applications run in separate virtual address spaces, the problems are quite different. This section will outline the way in which Unix-like systems conforming to the MIPS/ABI standard provide libraries which can be shared between different applications, with no restriction on how the libraries and applications can be programmed.

Every MIPS/ABI application runs in its own virtual address space. The application code is fixed to particular locations in this address space when it is linked. Library code is not built in; the application carries a table of the names of library functions and variables which are used, but not yet included. In addition, the application's symbol table defines public items which may be called from the library; under MIPS/ABI, library routines may freely refer to public data, or call public functions, in application code[2].

1. VxWorks is a trademark of Wind River Systems, Inc.
2. Though this may not be good programming practice.

Notes

In the MIPS/ABI model the binary application code must not be modified; it may itself be shared by multiple invocations of the application by multiple users.

It is not possible to predefine the actual virtual addresses at which a library's code and data will be located, but the offset from the start of its code to the start of its data is fixed, and this permits a number of tricks to be used.

- *Position-independent code: the compiler and assembler (by a command line option, used for library functions) can generate fully "position independent code" (PIC). All MIPS branch instructions are PC-relative; somewhat more complex sequences must be used to load a PC-relative address into a register, but if necessary it can be done:*

```
la rd, label      –>      bgezal $zero, 1f
                          nop
                   1:     addu rd, $31, label – 1b
```

- *Indirection and the Global offset table: PIC is suitable for references to code within a single module of a library (because the module's code is loaded as a single entity into consecutive virtual addresses). Data, or external functions, will be at locations which cannot be determined until the application and library are loaded, and so their addresses cannot be embedded in the program text.*

  Such addresses are held in a table built in the each library's per-process data space, the "global offset table" (GOT). Since the data space is not shared and is writable, the table can be built as the application and its libraries are loaded.

  A library function refers to a variable or external function through the GOT at a table index fixed when the library was compiled and linked. A load of the external integer type "errno" will come out as:

```
lw rd, errno      →       la gp, ThisLibsGOTBase
                          lw rd, errno_offset(gp)
                          nop
                          lw rd, 0(rd)
```

  Similarly, invocation of the shared-library function exit() would look like this:

```
/* setup argument */
jal exit          →       la gp, ThisLibsGOTBase
                          lw t9, exit_offset(gp)
                          nop
                          jalr t9
```

  The register gp (or $28) is a good choice for the table base. Because of its role in providing fast access to short variables it is not modified by standard functions. As an optimization it is calculated only once per function, in the function prologue. The calculation uses the fact that the function's actual virtual address will be in t9 (see previous example), and that the library's GOT is at a fixed offset from its code. So a position-independent function prologue might start like this:

```
func:
        la      gp, _gp_disp
        addu    gp, gp, t9
        addu    sp, sp, framesize
        sw      gp, 32(sp)
```

  In the above example, _gp_disp is a magic symbol which is recognized by the linker when building a shared library: it's value will be the offset between the instruction and the GOT. The calculated value is saved on the stack, and must be restored from there after a call to an external function, since that function may itself have modified gp.

Notes

There is much more that could be said about the way in which the MIPS/ABI implementation is optimized. For example, no attempt is made to link in libraries when an application is first loaded into memory; dummy GOT entries are used instead. When and if the application uses a library module, the reference is caught and fixed up in much the same way as a virtual-memory system incrementally pages-in a program image.

An Introduction to Optimization

The compiler writer's first responsibility is to ensure that the generated code does precisely what the language semantics say it should; and that is hard enough. In modern compilers, the optimizer has a secondary purpose, which is to allow the compiler's basic code generator to be simple (and therefore easier to implement correctly).

Common Optimizations

Most compilers will do all of the following. Occasionally the assembler may get in on some of them too.

- *Common sub-expression elimination (CSE): this detects when the code is doing the same work twice. At first sight this looks like it is just making up for dumb programming; but in fact CSE is critically important, and tends to be run many times to tidy up after other stages:*
  - *It is CSE which gives the compiler the ability to optimize across the function. The basic code generator works through the program expression-by-expression; even for well-written source-code, the expansion of simple C statements into multiple MIPS instructions will lead to a lot of duplicated effort. The very first CSE pass factors out the duplication and clears the way for register allocation.*
  - *Most memory-reference optimization is actually done by CSE – the code which fetches a variable from memory is itself a sub-expression.*
    The enemy of CSE is unpredictable flow of control: the conditional branch. The compiler can find it difficult to know what computation has run before which, with some straightforward exceptions, CSE can really only operate inside basic blocks (a piece of code delimited by, but not containing, either an entry point or a conditional branch). CSE markedly improves both code density and run-time performance.
    Similar to CSE are the optimizations of constant folding, constant propagation and arithmetic simplification. These pre-compute arithmetic performed on constants, and modify other expressions using standard algebraic rules so as to permit further constant folding and better CSE.
- *Jump optimization: removes redundant tests and jumps. Code produced by earlier compiler stages often contains jumps to jumps, jumps around unreachable code, redundant conditional jumps, and so on. These optimizations will remove this redundancy.*
- *Strength reduction: means the replacement of computationally expensive operations by cheaper ones. For example; multiplication by a constant value can be replaced by a series of shifts and adds. This actually tends to increase the code size while reducing run-time.*
- *Loop optimization: studies loops in the code, starting with the inner ones (which, the compiler guesses, will be where most time is spent). There are a number of useful things which can be done:*
  - *Sub-expressions which depend on variables which don't change inside the loop can be pre-computed before the loop starts.*
  - *Expressions which depend in some simple way on a loop variable can be simplified. For example, in:*

```
int i, x[NX];

for (i = 0; i <NX; i++)
        x[i]++;
```

the array index (which would normally involve a multiplication and addition) can be replaced by an incrementing pointer.
This kind of optimization will usually recognize only a particular set of stylized opportunities.

Notes

- *Loops can be "unrolled", allowing the work of 2 to a few iterations of the loop to be performed in line. On some processors where branches are inherently slow, this is inherently effective; but this isn't true for the RC30xx family.*
  But the unrolled loop offers much better opportunity for other optimizations (CSE and register allocation being the main beneficiaries).
  Loop unrolling may significantly increase the size of the compiled program, and usually must be requested as a specific compiler option.

- *Function inlining: the compiler may guess that some small functions can be expanded in-line, like a macro, rather than calling them. This is another optimization which increases the size of the program to give better performance, and usually requires an explicit compiler option. Some compilers may recognize the inline keyword used in C++ to allow the programmer to specify which functions should be "inlined".*

- *Register allocation: by far the most important optimization stage is to make the best possible use of the 32 general purpose registers, to make code faster and smaller. The compiler identifies global variables (static and external data stored in memory); automatic variables (defined within a function, and notionally stored on the stack); and intermediate products of expression evaluation.*

  Any variable must eventually be assigned to a machine register, and input data copied to that register, before the CPU can do anything useful with it. The register allocator's job is to minimize the amount of work done in shuffling data in and out of registers; it does this by maintaining some variables in registers for all or part of a function's run-time.

  Note:
  - *This process usually entirely ignores the old-fashioned "C" register attribute. It might be used as a hint; but most compilers figure out for themselves which variables are best kept in registers, and when.*
  - *The MIPS convention provides the compiler with 9 registers s0-s8 which can be freely used as automatic variables. Any function using one of these must save its value on entry, and restore it on exit. These registers tend to be suitable for long-term storage of user variables.*
    It also has a set of 10 "temporary" registers t0-t9 which are typically used for intermediate values in expression evaluation. The "argument" registers a0-a3 and "result" registers v0-v1 can be freely used too. However, these values don't survive a function call; if data is to be kept past a function call it is more efficient to use one of the "callee saved" registers s0-s8, because then the work of saving and restoring the value will be done only if a called function really wants to use that register.
  - *C's semantics mean that any write through a pointer could potentially alter almost any memory location; so a compiler's ability to maintain a user-defined variable in a register is strictly limited. It is safe to do so for any function variable (automatic variable) which is nowhere subject to the "address-of" operator "&". It may be able to do this for a variable inside a loop where there is neither a store-through-pointer operation nor a function call.*

- *Pipeline-specific code re-scheduling: the compiler or assembler can sometimes move the logical instruction flow around so as to make good use of the branch and load "delay slots". In practice, the delay slots are fine grain and tied to specific machine instructions; and this can only be done late in the compilation process.*

  The most obvious techniques are:
  - *If the instruction succeeding a load doesn't depend on the loaded value, just leave out the **nop** which would have been placed in the delay slot.*
  - *Move the logically-preceding instruction into the delay slot. The optimizer may be able to find an instruction a few positions preceding the branch or load, provided there are no intervening entry points.*
  - *The register-register architecture makes it fairly simple to pick out instructions which depend on each other and cannot be re-sequenced.*
  - *For a load, the optimizer may be able to find an instruction in the code after the load which is independent of the load value and is able to be moved into the delay slot.*
  - *Moving the instruction just before a branch into the branch delay slot.*
  - *Duplicating the instruction at a branch target into the branch delay slot, and fixing up the branch to go one more instruction forward.*
  - *This is particularly effective with loop-closing instructions. If the branch is conditional, though, the compiler can only do it if the inserted instruction can be seen to be harmless when the branch is not taken.*

Notes

How to Prevent Unwanted Effects from Optimization

Some code may rely on system effects invisible to the compiler. Examples include software intended to poll the status register of a serial port and send a character when it's ready:

```
unsigned char *usart_sr = (unsigned char *) 0xBFF00000;
unsigned char *usart_data = (unsigned char *) 0xBFF20000;
#defined TX_RDY 0x40

void putc (ch)
char ch;
{
 while ((*usart_sr & TX_RDY) == 0)
          ;

 *usart_data = ch;
}
```

A compiler, left to optimize this as for any other program, may send 2 characters and then enter an infinite loop. The compiler sees the memory reference implied by *usart_sr as a loop-invariant fetch; there are no stores in the "while" loop so this seems a safe optimization. The compiler has actually coded for:

```
void putc (ch)
char ch;
{
 tmp = (*usart_sr & TX_RDY);

 while (tmp)
          ;

 *usart_data = ch;
}
```

With most compilers, this particular problem is prevented by defining registers carefully:

```
volatile unsigned char
          *usart_sr = (unsigned char *) 0xBFF00000;
volatile unsigned char
          *usart_data = (unsigned char *) 0xBFF20000;
```

A similar situation can exist if software must examine a variable that is modified by an interrupt or other exception handler. Again, declaring the variable as ''volatile'' should fix the problem.

Although the C rules describe the operation of ''volatile'' as implementation dependent, most compilers which ignore the ''volatile'' keyword are expected to play safe.

There are other, more subtle, ways in which optimizations can break a program. For example, it may change the order in which some loads and stores occur. It may be easier to write and maintain hardware driver code in C than in assembler, but it's the programmer's responsibility to know exactly what the compiler did, and to make sure it's what was wanted.

Optimizer-unfriendly Code and How To Avoid It

Certain kinds of C programs will cause problems for a MIPS CPU and its optimizing compiler, and will cause unnecessary loss of performance. Some things to avoid are:

- *Sub-word arithmetic: use of short or char variables in arithmetic operations is less efficient than using full word arithmetic. The MIPS CPU lacks sub-word arithmetic functions and will have to do extra work to make sure that expressions overflow and wrap around when they should. The int data type represents the optimum arithmetic type for the RC30xx family; most of the time short and char values can be correctly manipulated by int automatic variables.*
- *Taking the address of a local variable: the compiler will now have to consider the possibility that any*

Notes

function call or write through a pointer might have changed the variable's value; so it won't live long in a machine register.

Perhaps the best way of seeing this is that defining a variable local to a function (and whose address is not taken) is essentially free. It will be assigned to a register, which would have been needed in any case for the intermediate result.

- *Nested Function calls: in the MIPS architecture the direct overhead of a function call is very small (2-3 clocks). But the function call makes it difficult for the compiler to make good use of registers, so may be much more costly in terms of lost optimization opportunity.*

Portability Considerations

Notes

This chapter discusses three facets of portability:

- *Migrating existing software from another CPU architecture to the MIPS family.*
- *Writing code that can readily be used on multiple MIPS family members.*
- *Writing code that will be portable to future family members.*

This manual focuses on the architecture-specific portability issues. And since most modern embedded programming uses the "C" programming language, this chapter will begin with a review of the portability concerns that are associated with this programming language.

This chapter also reviews some of the historical obstacles to program portability: byte ordering conventions, word sizes, alignment constraints, etc.; it will discuss the manner in which the MIPS architecture deals with these issues. This review is intended to discuss the issues which complicate porting existing code, developed for execution on other architectures, to the MIPS family.

Finally, this chapter will discuss generating an environment to support multiple family members, both existing and possible future members, to enable the investment in porting to be applied to a wide variety of system cost-performance points.

Writing Portable C

"C" is one of a class of languages which originally aimed to abstract the abilities of a class of simple minicomputers, to add some terse and powerful syntax for flow of control, and to provide simple but adequate mechanisms for data structuring.

C lets the underlying architecture show through; it is possible to write portable C by programming discipline, but it is not enforced by the language.

C's low-level origins contribute to its power and efficiency, but make it prone to non-portability. Some good examples follow:

- *Basic data types: change in their size (i.e. the number of bits of precision) between different implementations.*
- *Pointers: (inevitably implemented as real machine pointers) expose the memory layout of data, which is implementation-dependent.*

Some things have got easier with time; early C implementations had to target machines with 7-, 8- and 9-bit *char* types, and with 36-bit machine words. It is now reasonable to assume that targets will have an 8-bit *char* which is the smallest addressable unit of memory, and other basic types will be 16-, 32- or 64-bits in size.

C Language Standards

C has evolved continuously since its early days. It has definitely gone up-level; most changes have tended to increase the amount of abstraction and checking. To date, there are three main "variants", or standards, for the C language.

- *K&R: named after the C Programming Manual by Brian Kernighan and Dennis Richie, reflects the standard used for the first few years of Unix life. It has little type-checking, many defaults, and the compilers rarely complain. However, it provides a useful common base: most compilers will (sometimes unwillingly and with warnings) correctly translate programs written to K&R.*

  In practice the language was, during this period, defined by a single implementation: AT&T Bell Lab's Portable C compiler.
- *ANSI: the ANSI standard gathers the improvements that have been made over the years and then regulates them. ANSI defines syntax allowing the programmer to make more well-defined declara-*

Notes

tions of functions and checks programmer usage against them. ANSI compilers tend to produce more warning messages than K&R compilers, reflecting the greater amounts of type-checking performed.

A number of compilers use "compliance test suites" to "guarantee" ANSI compliance. A common test suite is the "Plum-Hall" test suite, which includes modules to test a compiler (and its libraries) compliance to the ANSI rules. The IDT/C compiler uses this compliance suite to validate its ANSI compliance.

- *GNU: the Free Software Foundation's GNU compiler is set to restore the dominance of a single implementation of the compiler, and thus permit the emergence of a new dialect. Note that the GNU compiler does support ANSI compliance.*

  GNU also adds a number of very valuable features; including function inlining, a robust "asm" statement, alloca().

GNU provides the benefit of being available across multiple hosts and target architectures. Thus, porting applications developed using the GNU toolchain (up on which IDT/c is based) from some other architecture to MIPS will avoid the porting problems associated with compiler PRAGMAs, compiler directives, and the like.

Similarly, porting ANSI compliant code from a different architecture should be relatively straight-forward. However, differences in supported PRAGMAs, and other environmental differences, may cause a higher level of porting activity.

C Library Functions and POSIX

C supports separate compilation of modules: C libraries are bunches of pre-compiled object code defining common functions. The "standard" C library of functions is effectively part of the language.

The ANSI standard addresses a subset of common library functions and defines their function. But this deliberately steers clear of OS-dependent functions; and these include the simplest input/output routines.

The POSIX (IEEE1003.4) standard is probably the best candidate, defining a standard C language interface to a workable IO system. POSIX has its problems:

- *it does not yet cover all OS features*
- *its definers occasionally felt obliged to standardize an "improvement" of current practice, so POSIX compliance is still hard to find even in a large OS.*

But it is a huge improvement on earlier single-camp standards and will undoubtedly become important. Programs adhering to POSIX should be able to be rebuilt on a large range of OS, including Desktop OS'es (such as UNIX) and RTOS environments. Using POSIX compliant library functions will further enhance portability across toolchains and architectures.

Data Representations and Alignment

The MIPS architecture can only load multi-byte data which is naturally aligned – a 4-byte quantity only from a 4-byte boundary, etc. The compiler ensures that data lands up in the right place, which requires:

- *Padding between fields of data structures.*
- *Defensive alignment; base addresses of structures, or stack frames, are aligned to the largest unit to which the architecture is sensitive (4 or 8 bytes in the MIPS architecture).*

The toolchain used for previous development, targeted to a different CPU architecture, may do this differently.

Consider the following example:

```
struct foo {
        char small;
        short medium;
        char again;
        int big;
}       char smallz
```

Notes

This will be laid out in memory as shown in Figure 11.1:

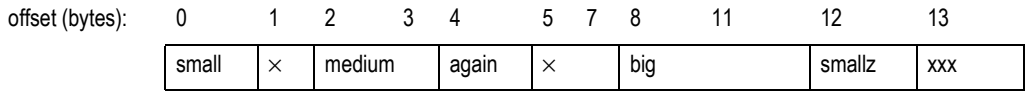


Figure 11.1 Example of Data Alignment in Memory

Notes on Structure Layout and Padding

These notes should be taken as typical of what a good compiler will do. They are required by, for example, a MIPS/ABI compliant compiler; but beware that a compiler *could* still be fully compliant with C standards and use different data representations, as long as these were internally consistent.

- *Alignment of structure base address: A structure's alignment is that of its most demanding subfield. struct foo contains an int requiring 4-byte alignment, so the structure itself will be 4-byte aligned. IDT/c for RC4xxx offers a mode in which all integers and/or addresses are treated as true 64-bit mode entities, where all of the above discussion should be read to refer to 8-byte instead of 4-byte.*

  Dynamic memory allocation, either on the stack or by software routines such as malloc() could give rise to alignment problems; so they are specified to return pointers aligned to the largest size which the architecture cares about. In the case of the RC30xx family, this need only be 4 bytes, but it is usually 8 bytes.
- *Memory order: fields within structures are stored into memory in the order declared.*
- *Padding: is generated whenever the next field would otherwise have the "wrong" alignment with respect to the structure's base address.*
- *Endianness: has no effect on the picture shown by Figure 11.2. Endianness determines how an integer value is related to the values of its constituent bytes (when they are considered separately); it does not affect the relative byte locations used for storing those values.*

  Endianness does affect C bitfields, which are discussed below.

The memory representation of data is compiler dependent, and the programmer should not expect it to be in any way portable – even between two different compilers for the same architecture. In general, it is reasonable to expect to be able to exchange an array of *char*s (each taking a value between 0 and 255), but not more.

ANSI compilers may support an option using the "pack" PRAGMA:

```
#pragma pack(1)
struct foo {
        char small;
        short medium;
        char again;
        int big;
}
```

This has the effect of causing the compiler to omit all padding and produce the layout shown in Figure 11.2:

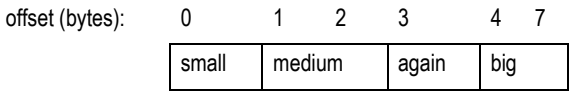


Figure 11.2 Example of "pack" PRAGMA Layout

A structure packed like this has no inherent alignment, so in addition to the lack of any padding, the structure base address may also be unaligned. The compiler will always generate load and store sequences to its fields which are alignment independent (and therefore to some extent inefficient) – even though, in this particular case, the big field happens to have the correct 4-byte alignment from the structure base.

Notes

Structure packing is most frequently used when storing large files of a particular structure in memory; for example, when storing the "description" of a font in the ROMs of a printer. By eliminating padding, more font structures can be saved in a smaller amount of memory; the cost of doing this occurs at run-time, when more conservative code sequences must be used to read fields from the structure.

The "1" in pack(1) refers to the maximum alignment which must be respected, so "pack(2)" means align only to 2-byte boundaries:

```
#pragma pack(2)
struct foo {
        char small;
        short medium;
        char again;
        int big;
}
```

The preceding code fragment has the effect of causing the compiler to pad items of 2 bytes or larger to 2-byte boundaries, producing the layout shown in Figure 11.3:

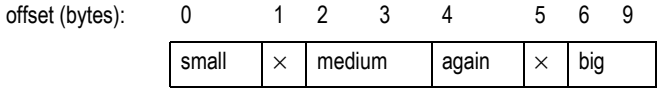


Figure 11.3 Example of "pack" PRAGMA Effect

The #pragma pack feature is not the only potential source of data representation incompatibility; endianness, discussed below, can also pose a significant portability issue. Nonetheless, used with care this feature can reduce the amount of difference between sources for two different architectures. Another issue with porting data is discussed in Chapter 2, "Data types in memory and registers."

Isolating System Dependencies

Most programs depend on an environment implemented by underlying independent software (perhaps from a 3rd party); this may be bound in at run time (an operating system or system monitor shared library), or at link time (library functions, "include" files). Quite often, sources may not be available; sometimes they will just be more trouble to port than to reproduce.

If only the boundary between the "application" program and its environment consisted of well-defined standard calls and include files, the job would be trivial. It isn't, usually.

Locating System Dependencies

In general, the "core application" consists of the code which is NOT:

- *Supplied as part of an OS the new system won't be using*
- *A library function which is not available (with exactly the same semantics) in the target compilation system*
- *Not licensed for use on the new target system*

There are two "concentric" boundaries which can be drawn, and in a sense they divide the original code into three parts

- *The inner part is the application to be ported. The new system may carry this code through unchanged except where portability problems mean the code needs to be changed. After porting, this code should still be usable on the original system.*
- *System dependent code, libraries, OS etc. which are clearly not going to be taken to the new system. Porting should not be an issue for these.*
- *Glue functions and data which join the two up. These will have to be modified, or sometimes re-implemented, to adapt the application to the new environment.*

Notes

The glue probably represents 10% of the code, but requires 90% of the work. In a program that has been ported often, the glue will be neatly separated; in a program which evolved in a single system, the glue may be rather deeply mixed with the application.

Fixing Up Dependencies

To remove these dependencies, the programmer must first try to find the best boundaries and then divide the code into application, glue, and environment, since there will be a new "environment" on the new system, the latter code is more or less irrelevant (and is likely written in assembler, to a great extent). This is art as well as science because there is no single correct way to do it. The objective will be to minimize the scope for introducing new errors, while minimizing the amount of work done.

The "application" part should be recompilable on the new system, generating a list of unresolved definitions which need to be patched up. Some of these, when investigated, will turn out to be used in code which really belonged in the "glue"; move the boundary and iterate until the list of unresolved names makes sense. The glue now needs to be re-implemented for the new environment. For each function, the programmer has two choices:

- *Recompile the function, using some "underglue" definitions or functions to mimic the behavior of the old environment using the new one. In a sense, the programmer is pragmatically deciding that what was seen as glue is now application.*
- *Reimplement the function (using the old one for inspiration and as a source for cut-and-paste), aiming to mimic the function as a "black box".*

For each function or module, choose one of these strategies. It is always a bad idea to mix strategies in the same module.

Isolating Non-Portable Code

In general, it is difficult to write a "stand-alone" program portably. In the desktop environment, programmers write programs to an OS standard; thus, porting a program to a new system is limited to porting that OS.

As examples, it is easy to write a portable routine to calculate prime numbers; it is much harder to write a portable routine to accept typed input, providing line editing and simple argument parsing (are characters 7- or 8-bit? Is the language English? What accented characters are acceptable? How does the display device implement backspace?)

The best programs hide the nonportable parts of code in modules, whose interfaces consist of stable data declarations and functions whose operation can be expressed clearly and succinctly.

Using Assembler

There are three reasons for using assembler:

- *Efficient implementation of critical functions: scheduling bandwidth to a tightly controlled memory region may be critical in some systems.*
- *Access to instructions not supported by the compiler: e.g. access to control registers. These can sometimes be replaced by using "tiny" subroutines; and sometimes by C asm statements. Tiny subroutines are particularly apt when, although the implementation will be completely machine dependent the desired effect is machine-independent – prefer a "disable interrupts" function to a "set status register bits" function.*
- *Some critical environmental deficiency: (most commonly) inability to provide the free use of CPU registers and the stack which the compiler relies on. Classic examples are interrupt handlers. To maximize ease of portability, the programmer can at least make it a priority, in these routines, to build an environment from which software can call C functions.*

Notes

Endianness

The word "endianness" was introduced by a famous short paper[1] in the Journal of the ACM, in the early 1980's. The author observed that computer architectures had divided up into two camps, based on an "arbitrary" choice of the way in which byte and word addressing are related. In "Gulliver's Travels" the little-endians and big-endians fought a war over the correct end at which to start eating a boiled egg; a war notable for the inability of the protagonists to comprehend the arbitrary nature of their difference. The joke was appreciated, and the name has stuck.

The problem comes up in both software and hardware fields – but slightly differently:

- *Endianness – hardware visibility: this arises when a byte-addressed system is wired up with buses which are multiple-bytes wide. When the system transfers a multi-byte datum across the bus, each byte of that datum has its own, individual address.*

  So:

  If the lowest-addressed byte in the datum travels on the 8 bus lines ("byte lane") with the lowest bit-numbers, the bus is little-endian.

  If the lowest-addressed byte in the datum travels on the byte lane with the highest bit-numbers, the bus is big-endian.

  With the exception of Hewlett Packard and IBM, note that there is little dispute in the industry as to how bit numbers relate to arithmetic significance; high bit numbers are always most significant. In particular, this means that bits-within-byte have an unambiguous meaning.

  All byte addressable CPUs announce themselves as either big- or little-endian every time they transfer data. Intel and DEC CPUs are little-endian; Motorola, Sun SPARC and most IBM CPUs are big-endian. MIPS CPUs can be either, as configured from power-up. In MIPS CPUs endianness is only apparent for partial word writes. Instructions are still accepted with the same endianness, regardless of whether the CPU is configured to be big- or little-endian.

  For a hardware engineer, endianness only matters when a system includes buses, CPUs or peripherals whose endianness doesn't match.

  The choice facing the hardware engineer is not a happy one; if two components or buses don't match, the system designer must choose one of two undesirable situations:

  – *If the data buses are connected to preserve byte address, then bit numbering for multi-byte data moving through the system will be inconsistent; so multi-byte data is likely to require re-interpretation by software.*
  – *If the data buses are connected with matching bit numbers, then the two sides will see the sequence of bytes in memory differently. This problem can be managed by keeping all data strictly word-aligned, and "byteswapping" before and after transfer.*
    Where a system includes a MIPS CPU which can be configured with either endianness with no external hardware provided, option (b) is what happens whenever the CPU configuration is changed to mismatch the rest of the system.

- *Endianness – software visibility: software engineers writing in a high level language apparently have no need to number bits, so might believe themselves immune from this problem. But on closer inspection, it turns out that normal binary numbers (i.e. 2-s complement integers) bigger than 8 bits implicitly define an ordering – some bits are arithmetically more significant.*

  In software:

  An architecture where the lowest addressed byte of a multi-byte integer holds the least-significant bits is called little-endian.

  An architecture where the lowest addressed byte of a multi-byte integer holds the most significant bits is called big-endian.

  Software problems occur on any system afflicted by hardware incompatibility; but the software problem also emerges when a program deals with "foreign" data originating from a system using the opposite convention. The data may arrive on a communications link, on a tape or floppy disk.

- *Why is it so confusing? It is difficult even to describe the problem without taking a side. The origin of*

1. "On holy wars and a plea for peace", Danny Cohen, IEEE Computer, October 1981 pp. 48-54

Notes

the two types lies in two different ways of drawing the pictures and describing the data; both natural in different contexts.

Big-endians typically draw their pictures organized around words (32 bits in a MIPS system), like Figure 11.2. What's more, big-endians see words as a sort of number, so they put the highest bit number (most significant) on the left, like our familiar Arabic numbers. And a big endian sees memory extending up and down the page from the picture in Figure 11.4.

```
union either {
        int as_int;
        short as_short[2];
        char as_char[4];
};
```

bit no:	31 24	23 16	15 8	7 0
	as_int			
	as_short[0]		as_short[1]	
	as_char[0]	as_char[1]	as_char[2]	as_char[3]
byte offset:	0	1	2	3
	3	2	1	0

Figure 11.4 Big Endian Data Structure

Little-endians are little-endians because they think in bytes, so the same data structure looks like Figure 11.4. Little-endians don't think of computer data as primarily numeric, so they tend to put all the low numbers (bits, bytes, whatever) on the left. A little endian sees memory extending off to the left and right of the picture.

What it Means to the Programmer

Software can very easily find out if it is executing as a big-endian, or little-endian, CPU – by a piece of deliberately non-portable code:

```
union either {
        int as_int;
        short as_short[2];
        char as_char[4];
};

either.as_int = 0x12345678;

if (sizeof(int) == 4 && either.as_char[0] == 0x78) {
        printf ("Little endian\n");
}
else if (sizeof(int) == 4 && either.as_char[0] == 0x12 {
        printf ("Big endian\n");
}
else {
        printf ("Probably not MIPS architecture\n");
}
```

In application software, so long as software doesn't carelessly access the same piece of data as two different integer types, endianness should create no problems. But as soon as the program needs to know how data is stored in memory, it is very important.

Bitfield Layout and Endianness

C permits programs to define bitfields in structures; as an example, the chapter on floating point used a bitfield structure to map the fields of an IEEE floating point value stored in memory. An FP single value is multi-byte, so this definition is expected to be endianness-dependent. It looked like this:

Notes

```
struct ieee754sp_konst {
 unsignedsign:1;
 unsignedbexp:8;
 unsigned mant:23;
};
```

C bitfields are always packed. But bitfields may not span word boundaries (usually corresponding to the size of a long: 32 bits for the MIPS family). The structure and mapping for a big-endian CPU is shown in Figure 11.5 (using a typical big-endian's picture); for a little-endian version it is shown in Figure 11.6.

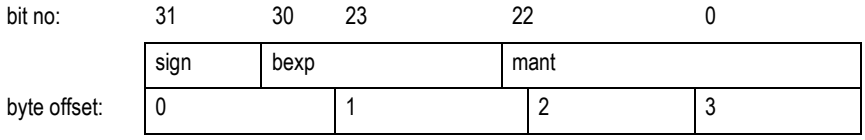


Figure 11.5 Data Structure and Mapping for a Big-endian CPU

The little-endian version of the structure defines the fields in the other direction; the C compiler insists that, even for bitfields, items declared first in the structure occupy lower addresses:

To make that work, as shown in Figure 11.6 that in little-endian mode the compiler packs bits into structures starting from low-numbered bits.

```
struct ieee754sp_konst {
 unsigned mant:23;
 unsignedbexp:8;
 unsignedsign:1;
};
```

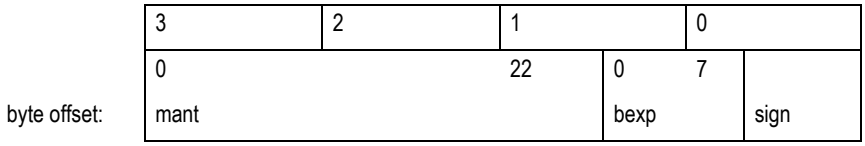


Figure 11.6 Data Structure and Mapping for a Little-endian CPU

Changing the Endianness of a MIPS CPU

Programming a board which can be configured with either byte ordering is tricky, but possible.

The MIPS CPU doesn't have to do too much to change endianness. The only parts of the instruction set which recognize objects smaller than 32 bits are partial-word loads and stores. The instruction:

```
lbu $t0, 1($zero)
```

takes the byte at byte program address 1, loads it into the least-significant bits (0 through 7) of register *$t0,* and fills the rest of the register with zero bits.

This *description* is endianness-independent; and the signals produced by the CPU are identical in the two cases – the address will be the appropriate translation of the program address "1", and the transfer-width code will indicate 1 byte. But: *in big-endian mode the data loaded into the register will be taken from bits 23-16 of the CPU data bus; in little-endian mode the byte is loaded from bits 8-15 of the CPU data bus.*

It is exactly this shift of byte-lane associated with a particular byte address, no more or less, which implements the endianness switch.

The default effect of this switch on a system built for the other endianness is that the CPU's view of byte addressing becomes scrambled with respect to the rest of the system; *but the CPU's view of bit numbering within aligned 32-bit words continues to match the rest of the system.* This is the case described in (b) above; and it has some advantages.

Notes

Complementing the chip's ability to reconfigure itself, most MIPS compilers can produce code of either byte-ordering convention.

Designing and specifying for configurable endianness

Some hard thinking and good advice before the design is committed, may help a great deal. To summarize:

- *Read-only instruction memory: should be connected to the CPU with bit-number-preserving connections, regardless of configuration. Even if the ROM is less than 32 bits wide, the way in which ROM data is built into words should also be independent of the CPU configuration.*
- *IO system or external world connection: if the system makes any connection to a standard bus, or connects to a memory which gets filled by an agent other than the CPU, or uses a multibyte-wide DMA controller, then it may be appropriate to include a configurable byte-lane swapper between the CPU and IO.*
- *Local writable memory: normally it is best to let this attach in a simple bit-number-preserving way to the CPU bus. If there is a byte-lane swapper in the system, it should also swap lanes between the IO system and the local memory.*

Read-only instruction memory

All MIPS instructions are aligned 32-bit words. If a read-only program memory is attached to the CPU by bit-number-preserving connections which are unaltered between modes, then big-endian and little-endian CPUs run the same instruction set, bit for bit.

The endianness mode shows up only when the CPU attempts a partial-word operation; so a program written without partial-word operations will run the same in either mode. It is reasonably straightforward to build a PROM which could bootstrap the system in either mode.

Algorithmics have used this to build enough "bi-directional" code to at least display an error message when the rest of the PROM program discovers that it mismatches the CPU configuration:

```
.align    4
.ascii "remEcneg\000\000\000y"
```

that's what the string "Emergency" (with its standard C terminating null and two bytes of essential padding) looks like when viewed with the wrong endianness. It would be even worse if it didn't start on a 4-byte aligned location. Figure 11.7 (drawn from the bit-orientated point of view of a confirmed big-endian) shows what is going on.

	31 24	23 16	15 8	7 0
	'r'	'e'	'm'	'E'
byte address from BE CPU:	0	1	2	3
byte address from LE CPU:	3	2	1	0
	'c'	'n'	'e'	'g'
byte address from BE CPU:	4	5	6	7
byte address from LE CPU:	7	6	5	4
	×	×	'\000'	'y'
byte address from BE CPU:	8	9	10	11
byte address from LE CPU:	11	10	9	8

Figure 11.7 Example of Bit-orientation with Wrong Endianness

Notes

Writable (volatile) memory

The above applies to any program memory; but the system may want to treat volatile program memory differently. Why?

Volatile program memory must be loaded at run-time. Most loading processes ultimately involve fetching instructions from a file, and most files are defined as byte sequences. Thus the 32-bit instruction words must be constructed (one way or the other) from byte sequences. The standard way of storing code in files *does* change between the two options: big-endian code is stored with the most significant byte of each instruction first, and little endian code with the least significant byte first.

Byte-lane swapping

It may happen that somewhere in the system there is a bus or device whose byte-order doesn't change when the CPU's does. The best solution (from a software engineer's perspective), is to persuade the hardware designer to put a programmable byte-lane swapper between the CPU and the IO system. The way this works is shown diagrammatically in Figure 11.8.

This is referred to as a byte-lane swapper, not a byte-swapper, to emphasize that it does not discriminate on a per-transfer basis, and in particular it is not switched on and off for transfers of different sizes. Such discrimination would be futile; the hardware transfer size does not consistently reflect the way in which software is interpreting data (for example, cache-line refills may contain byte values). *There is no external hardware mechanism which can hide endianness problems.*

What a byte-lane swapper *does* achieve is to ensure that, when the CPU configuration is changed, the relationship between the CPU and the now non-matching external bus or device is one where byte sequence is preserved.

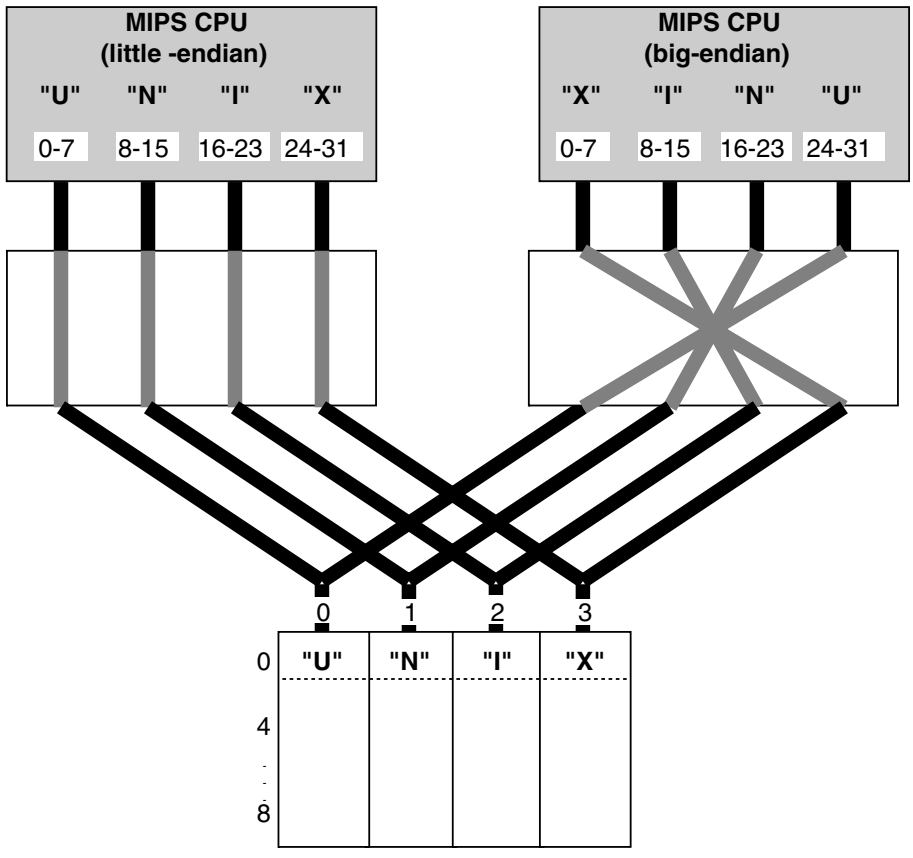


Figure 11.8 Byte-lane swapper

Notes

When the system includes a byte-lane swapper between the CPU and some memory, it is probably not viable to swap it when using cached memory. It really can be used only:

1. at system configuration time;
2. when talking to uncached, IO system locations. The system could discriminate (to swap or not to swap) based on the address regions which select various sub-buses or sub-devices.

The system doesn't normally need to put the byte-lane swapper between the CPU and its local memory; avoiding the use of one in this path is desirable, because the CPU/local memory connection is fast and wide, so the swapper will be expensive. Since the swapper configuration is determined at reset time, and the memory is then completely undefined, the system can treat the CPU/local memory as a unit; the swapper is installed between the CPU/memory unit and the rest of the system. In this case the relationship between bit number and byte order in the local memory changes with the CPU, but this fact is concealed from the rest of the world.

Configurable IO controllers

Some newer IO controllers can themselves be configured into big-endian and little-endian modes. Use of such devices must be done carefully, particularly when using it not as a static (design-time) option but rather a jumper (reset-time) option.

It is quite common for such a feature to affect only data transfers, leaving the programmer to handle other endianness issues, such as access to bit-coded device registers.

Portability and Endianness-independent Code

Any code which exposes data to two different views will be endianness-dependent (and likely to be architecture- and compiler-dependent too). Many MIPS compilers define the symbols MIPSEB or MIPSEL so that programmers can include endianness dependent code, such as:

```
#if defined (MIPSEB)
/* big-endian version */
#else
/* little-endian version */
#endif
```

With ingenuity and patience the programmer can probably represent the difference with common code but conditional data declarations; that should be more maintainable. However, endianness-independent code should be used wherever possible.

Endianness-independent code

All data references which pick up data from an "external" source or device are potentially endianness-dependent. But according to how the system is wired, software may be able to work both ways:

- *If the device is byte-sequence compatible: then it should be programmed strictly with byte operations.*

  If ever, for reasons of efficiency or necessity, the system must transfer more than one byte at a time, the programmer must figure out how those bytes should pack in to a machine word. This code will be explicitly endianness-dependent, and can be made conditional.
- *If the device is bit-number compatible: then program it strictly with word (32-bit) operations. This may well mean that device data comes and goes into slightly inconvenient parts of a CPU register; 8-bit registers in system originally conceived as big-endian are commonly wired via bits 31–24. So software may need to shift them up and down appropriately.*

Compatibility Within the MIPS Family

It is relatively straightforward to make programs compatible across the entire IDT family. The device user's manuals detail potential areas of incompatibility, most of which can easily be accommodated by software.[1] The software-visible differences in these CPUs are as follows:

Notes

- *Cache size: all CPUs have separate I- and D-caches each of between 512 bytes and 16Kbytes. All D-caches are write-through, so the only cache maintenance operation required is that of invalidating an entry. The cache management software uses the same basic code sequences for all family members (which follows the original RC3000), using status-register control bits to "isolate" and "swap" the caches.*

  To maximize portability, system software should measure the cache size at system initialization, as described earlier. Do not rely on the CPU type and revision fields in the ID register.

  To simplify porting to other MIPS devices, such as the IDT RC4600, software should probably structure cache invalidation software as using a single entry point. Thus, when porting to these upscale devices, the amount of software to change is minimized.
- *Cache line size: In the RC30xx, all caches are direct mapped. The D-cache always has a line size of one word, and all I-caches have a 4-word line size. The 4-word line size does offer the potential for a faster I-cache invalidation routine; but invalidating each word of a region still works correctly with a 4-word line. With the cache instruction in the RC4xxx/RC5000/RC32364, cache operations on entire blocks become very straight forward.*
- *Cache-hit write policy: All of the MIPS CPUs will use a read-modify-write sequence when performing a partial-word write to a location already present in the D-cache. This can lead to some curious problems if another memory master is simultaneously accessing the same word; all software should assume that the read-modify-write sequence might occur.*
- *Write buffer differences and wbflush(): To make the write-through cache efficient, all MIPS CPUs have a four deep write buffer, which holds the address/data of a write while the CPU runs on. The operation of the write buffer should be invisible when writing and reading regular, side-effect free memory; but it can have effects when accessing IO buffers.*

  The programmer only needs an implementation of wbflush(); a routine defined to hold the CPU in a loop until all pending writes have been completed. In the RC30xx family, wbflush() can be implemented by performing an uncached read (for example, to the reset exception vector location, since the programmer is assured that the system will provide uncached memory at that location). An example of wbflush() is presented in Chapter 5.
- *FP hardware: Currently, only the RC3081 and the RC4xxx integrate the hardware FPA on-chip. For occasional FP instructions, trap-based software emulators may be appropriate; the use of the emulator can be completely software-transparent, but slow.*
- *MMU hardware: If present, it is always the same software-refilled TLB and control set, as described above. Base versions provide consistent mappings for kuseg and kseg2; however, maximum portability is achieved when programs only use the kseg0, kseg1 regions which are supported by all processors (including the RC4600/RC4700/RC5000/RC32364).*
- *Integrated IO devices: Some future CPUs may integrate timers, DRAM controllers, DMA and other memory-mapped peripherals. If the programmer isolates such code into "driver" modules for existing systems, porting to these devices will be simplified.*
- *Perform device-type identification at boot-time: The reset chapter discussed how to identify the particular CPU being used at reset time. Performing device identification allows the software to then branch to the appropriate device specific initialization code (e.g. to initialize the RC3041 control registers, or CP1 usability for the RC3081). Providing this basic structure as part of reset only enables software to be quickly adapted to support other family members.*
- *Isolate CP0 code from applications code: The MIPS architecture allows CP0 to vary by implementation. By writing the code modularly, so that system and exception management functions are modularized out of the application code, porting to new generations of processors is simplified (e.g. the RC4600, which uses a slightly different exception state management mechanism and slightly different vectors, but is otherwise very familiar to an RC30xx programmer).*
- *MIPS ISA level: In order to keep the assembly code as portable as possible, the programmer may tend to use the lowest possible MIPS ISA. While this is understandable, an attempt should be made to use the right ISA for the CPU, tune applications to the CPU, and keep assembly code #ifdef'd.*

[1] Perhaps the most notable exception has to do with the TLB. Software environments that use kuseg and/or kseg2 will probably not be able to subsittute "E" and base-version, or RC4600/RC4700/RC5000/RC32364 and RC4650, for each other.

Notes

Porting to MIPS: Frequently Encountered Issues

The following issues have come up fairly frequently:

- *Moving from 16-bit int: a significant number of programs are being moved up from x86 or other CPUs whose standard mode is 16-bit, so that the C int is a 16-bit value. Such programs may rely, very subtly, on the limited size and overflow characteristics of 16-bit values. While the programmer can get correct operation by translating such types into short, this may be very inefficient. Take particular care with signed comparisons.*
- *Negative pointers: when running in unmapped mode on a MIPS CPU all pointers are in the kseg0 or kseg1 areas; and both use pointers whose 32-bit value has the top bit set. It is therefore extremely important that any implicit aliasing of integer and pointer types (quite common in C) specify an unsigned integer type (preferably an unsigned long).*

  Unmapped programs on certain other architectures deal with physical addresses, which are invariably a lot smaller than 2GB.
- *Signed vs. unsigned characters: K&R C made the default char type (used for strings, and so on) signed char; this is consistent with the convention for larger integer values. However, as soon as programmers have to deal with character encodings using more than 7-bit values, this is dangerous when converting or comparing. So the ANSI standard determines that char declarations should, by default, be unsigned char.*

  If the old program may depend on the default sign-extension of char types, there is often a compiler option to restore the traditional convention.
- *Data alignment and memory layout: if a program makes assumptions about memory layout (such as using C struct declarations to map input files, or the results of data communications) the programmer should review and check the structure declarations. It will often not be possible to interpret such data without a conversion routine (for example, to convert little-endian format integers to big-endian).*

  It is probably better to remove such dependencies; but it may be possible to work around them. By setting up the RC30xx system to match the software's assumptions about endianness, and judicious use of the #pragma pack(xx) feature, the problem may be avoided.
- *Stack issues – varargs/alloca: as pointed out above. The C stack is synthesized using standard register/register instructions to form a single stack containing both return addresses and local variables; but the stack frame may not be generated in functions which don't need it.*

  If the C code thinks it knows something about the stack, it may not work. However, two standards-conformant macro/library operations are available:
  - *varargs: use this include file based macro package to implement routines with a variable number of parameters. C code should make no other assumptions about the calling stack.*
  - *alloca: use this "library function" (it is implemented as a built-in by many compilers) to allocate memory at run-time, which is "on the stack" in the sense that it will be automatically freed when the function allocating the memory returns. Don't assume that such memory is actually at an address with some connection with the stack.*
- *Argument passing – autoconversions: arguments passed to a function, and not explicitly defined by a function prototype, are often promoted; typically to an int type, for sub-word integers. This can cause surprises, particularly when promoting data unexpectedly interpreted as signed.*
- *Endianness: the system architect may be able to configure the MIPS system to match the endianness of the existing system, to save the many trials described above.*
- *Ambiguous behavior of library functions: library functions may behave unexpectedly at the margins*
  - *a classic example is using the memcpy() routine (defined in many C environments) to copy bytes, and accidentally feeding it a source and destination area which overlap.*
- *Include file usage: this is closer to a system dependency; but the programmer can spend hours trying to untangle an incompatible forest of ".h" files. Moral: if a program is supposed to be largely OS-independent, try not to use the OS' standard include files.*

Notes

Considerations for Portability to Future Devices

In general, it is difficult to perfectly plan for future, unknown devices. However, the techniques described above should minimize the effort required to take advantage of changing technology:

- *develop code portable across existing family members. Future family members may continue to vary cache sizes, TLB structures, inclusion of FPA, etc. However, many of them can be expected to be compatible with the basic CP0 mechanisms described in the earlier chapters. Code which is independent of cache size, resides in kseg0 and kseg1, and which allows the inclusion of new/additional device drivers is likely to be readily portable to newer family members.*
- *Use modular programming. Specifically, map device specific functions such as cache invalidation, device initialization exception decoding and exception service dispatch, to independent modules (rather than intertwine these functions throughout the program). This will facilitate the porting to family members such as the RC4600, which offer different CP0 architectures.*
- *Isolate the key algorithms to be device independent. For example, image rasterization of routing table look-up should be implemented in code which is device independent (but may rely on underlying, independent exception or cache structures).*

Writing Power-On Diagnostics

Notes

Large companies with established product lines will already have guidelines for systems diagnostics; programmers may find this chapter useful for particular information about the MIPS architecture and how its features can be employed.

However, a large number of engineers will be dealing not just with a new CPU architecture, but also with a new level of system complexity. For those, this chapter is a pragmatic, hands-on guide to producing usable diagnostics. There is much academic literature about the efficiency and thoroughness of tests (particularly memory tests) which won't be addressed in this manual.

Golden Rules For Diagnostics Programming

- *Test only the minimum required at each stage: tests which run very early must be written in an environment which makes the programmers' life difficult. Whole chunks of the hardware cannot be trusted, the CPU may not be able to run at full speed, and it may be impossible to use high level languages.*

  The structure of the early tests is therefore pretty much unaffected by the hardware specification; they are focussed on getting enough confidence in the CPU, program memory and writable memory (and, more importantly, the interconnects between them) to make it safe to use high-level language routines.
- *Keep it simple: diagnostic routines are particularly hard to prove, since the only way to check them is to simulate hardware faults. When the hardware really does go wrong, the diagnostics are quite likely to crash silently; a computer going wrong frequently goes so badly wrong that not even the most paranoid test will get running.*

  Routines so simple that they are almost certain to be correct by inspection will probably be robust when needed; and the programmer will be more confident in pointing the finger at the hardware.
- *Find some way to communicate: the worst thing any diagnostic can do in the face of an error is to say nothing. But since most faults are near-catastrophic, this worst case happens often. The diagnostics programmer will therefore do everything possible to get diagnostic routines to do something visible with the absolute minimum of hardware.*

  Many hardware products are fitted with some kind of write-only output device with diagnostics in mind – perhaps an LED, a 7-segment display or (if the designer could afford its space and cost) a miniature alphanumeric LED or liquid crystal display showing 4 or more characters. This device should be wired up so that, provided the CPU and ROM memory are functional, the minimal amount of further hardware has to work for the display to show something.

  Don't forget that even where software can't flash an LED, it can make software's activity visible to a simple piece of test equipment—a voltmeter, oscilloscope or logic analyzer. For example, the IDT Micromonitor will perform a software loop at an "error address"; a logic analyzer can then trigger on this address to see the sequence of events immediately prior to the error.
- *Never poll anything forever: of course, it is common practice in simple device drivers to code a loop which is exited when some status bit changes. But when dealing with unproven subsystems, it is best to keep some track of real time so that the code can recognize that the status bit is not going to change, and report it.*
- *Good diagnostics are fast: some fault conditions are dynamic or pattern-sensitive, and careful, slow diagnostics won't ever find them.*
- *Fighting past programmable hardware: one major problem for the diagnostics programmer is the use of software-configurable hardware. For example, Algorithmics' SL–3000 single-board computer uses a VLSI component (VAC068) for the external bus address path. This component integrates a programmable address decoder and wait-state generator. This is convenient and saves a lot of ran-*

Notes

dom logic; BUT this means that even the simplest operations (e.g. access to a UART register) won't work until the VAC068 has been configured.

The hardware engineer should have been talking with the systems programmers about this as the system was designed, since it is quite possible to build a system which cannot be bootstrapped.

- *Work with the user: good diagnostic tests may well be able to give a clear indication of where a problem lies. But never forget that diagnostics are meant to be run and watched by a knowledgeable person. Give the user a chance and inform them of what is happening. If a test prints out "Trying master access from Ethernet chip" and then nothing more, it is much more helpful than silently sticking in an infinite loop trying to figure out something more specific to say.*

What Should Tests Do?

- *Diagnostics versus go/no-go: a major conceptual difference; is the test intended to direct service or repair effort to a particular subsystem, or is it merely intended to come up with a "yes/no" answer?*

  In practice most test software seems to be expected to do both. This is not a major problem in terms of what is tested and how, but there is one big difference – time. A power-on "yes/no" test needs to be completed before it exhausts the patience of the person operating the power switch (empirically, 20 or 30 seconds seems about the limit).

  A diagnostic test can run for much longer. To address both needs with one test, find some way of configuring the test so that it can be asked to be more thorough at the cost of taking longer.

- *Black boxes and internals: in theory each subsystem can be treated as a "black box", purely in terms of its logical functions, and tested at that level without regard for its implementation. However, perfect tests usually require too long to run, and thus shortcuts are needed; knowing what shortcuts will be sensible is usually based on the internal design.*

  Build a simple logical block diagram of complex subsystems, working with the hardware designer, and refer to it when figuring out a test sequence.

  Bear in mind that malfunctioning hardware can behave in ways which have no relation to its correct function. Note that this can cause "false positives"; for example when a write/read-back test returns correct data which has been retained by stray capacitance on a set of undriven signal wires (this is a fairly common occurrence in tests designed to determine the amount of system RAM available).

  Hardware engineers will have some feel for what may happen inside a component when it is abused; for example, it is useful to know that certain kinds of timing violation will cause the loss of data in a whole "row" of cells inside a dynamic RAM chip.

- *Connections are more unreliable than components: probably 10-50 times more unreliable. Short-circuits between signals are fairly common (very common on boards which have not been auto-tested) and can produce subtle and peculiar behavior.*

- *Microcontrollers and other smart hardware: any independently-acting programmable subsystem causes testing problems; this is probably the best reason for keeping subsystems dumb whenever possible. The same principles apply to test software executing on an intelligent subsystem, as to the whole test software. But communicating results to the user is often even more difficult.*

- *Testing internals of components: few systems really need to do this, or can do a good job of it. The diagnostics programmer can't find out how VLSI components are really built, so any tests beyond the simplest and most obvious are unlikely to be useful. What is possible is to set out to exercise components up from the most primitive operations they perform as "black boxes", with a view to proving the whole interface between the device and the rest of the system.*

- *Specifying tests: an art form, like any specification. DO agree in advance on how to signal information (LED flash codes, signal levels, logic analyzers); DON'T bother to agree in advance what algorithm to use for memory tests.*

Notes

How to Test the Diagnostic Tests?

Verifying the tests can be extremely difficult; the diagnostics engineer would ideally like to take the tests down all possible paths (e.g. the memory is good vs. the memory is bad). Doing so requires a method to make the test find faults in what may be an actual, good system.

There are two primary techniques for doing this:

- *Software test harnesses: these use some kind of simulator, which can be programmed to be defective.*
- *Hardware test harnesses: with this technique, hardware faults can be very tricky.*

Overview of Algorithmics' Power-on Selftest

This section describes the functions and construction of a set of ROM-resident test routines designed for Algorithmics' SL–3000 VMEbus single-board computer, which is based on an IDT RC3081-40 CPU.

The primary purpose of the tests is as power-up confidence tests, which must run in a short period of time; but they can be configured (using information held in a small nonvolatile writable store) to run slowly and carefully. They are useful as diagnostics, particularly for units which are too faulty to load more sophisticated routines.

Starting Points

Unless a reasonable amount of logic is working correctly the SL–3000 will be unable to run test code. The minimum requirements are:

- *PROM: is correctly readable.*
- *Onboard data and address interconnects: are fault-free, at least between the PROM and CPU (at least when all possible subunits are held in reset.)*
- *CPU: capable of executing code correctly.*

  The tests do not have to assume correct operation of the on-chip caches (they are tested), the FPA (the tests merely look to see whether there is one there, and the test software does not need it to work), and the TLB (memory-management hardware, described earlier.)
- *Error Reporting: the SL–3000 has a front-panel 7-segment "hex" LED display provided mainly for this purpose. Where the console serial port is available, connected and functional it is used to provide fuller information.*

  In some circumstances the diagnostics will also leave warning messages and codes in the nonvolatile memory, for higher-level software to find.

  Under serious failure conditions the tests make a last-ditch attempt to pass back information by a series of writes to PROM space; information is encoded in the store target addresses. The writes have no effect on the hardware[1], but can be monitored with test equipment in laboratory conditions.
- *Underlying hardware: the "minimal" functions described above implicitly require the use of other logic on the board. In particular, the VMEbus interface components (VIC068 and VAC068) integrate a variety of local bus control functions, and code will be impossible to run if these are faulty.*

Control and Environment Variables

The nonvolatile RAM provides configuration and other information shared between several different levels of software. Rather than attempting to legislate for a rigid fixed-field map, the bulk of the NVRAM storage is organized as an ''environment'' modelled after the UNIX facility. This provides a set of key/value pairs, all of which are ASCII strings.

1. Such a methodology may not be compatible with the use of a ROM emulator; instead, it may be appropriate to define an "error reporting space" in the address map, which performs the appropriate handshake back to the CPU, but which does not decode into any actual memory devices.

Notes

The environment is used both to set up options for the power-on tests (e.g. whether to spend time on thorough DRAM tests), and to return information discovered by those tests (e.g. to report the size of the caches).

The integrity of the environment store is protected by a checksum. If the power-on test detects a corrupted NVRAM, it will ignore the NVRAM contents and use a set of default values for the environment variables.

Users have to have some way of inspecting and altering the environment. Normally this will be provided as *setenv*, *getenv* commands implemented by an interactive ROM monitor. The power-on self-test code includes subroutines accessing the environment, but is designed to work with a variety of monitors.

A few NVRAM locations are predefined and strictly reserved for some other piece of software. They are ignored by the power-on tests.

Reporting

Progress through the tests is shown as a sequence of numbers displayed on the front-panel LED. Failure is shown by a (possibly multi-digit) code flashed on the display.

Total collapse of the hardware under test is inferred by failure to keep incrementing the count, so the tests make sure that the display is changed every few seconds (exception: when the user has deliberately set an option variable to request the exhaustive version of a test, the user is expected to be patient).

Usually test progress and results are also reported to the console (always to serial port 0, always at 9600 baud); but most console output can be suppressed by setting an appropriate environment variable, in case some systems have some other equipment permanently attached to the console port. However, fatal error messages will be reported to the console regardless of the environment state.

Unexpected Exceptions During Test Sequence

If something is really wrong with the machine, the CPU will usually get some kind of exception (illegal instruction, illegal or unmapped address). These conditions are usually to be regarded as fatal. They are usually a sign of something very seriously wrong, so the priority is to make the code robust enough that something will get reported.

Exception reporting to the hex display should be done with the most pessimistic assumptions about the state of the machine; i.e. without using memory or the console. Once a minimal report has been made this way, it is permissible to assume memory is working in order to produce a better report to the console.

The boot test sequence will always use the “bootstrap exception vector”, described earlier in this manual, so that exceptions are trapped into PROM space with the instruction cache not used. Since the CPU can be reconfigured to vector exceptions through cached low memory, the test code does not have to provide any software mechanisms for intercepting its own exceptions.

Driving Test Output Devices

Test device software is pessimistic about the status of the hardware it talks to, to ensure that tests cannot be hung-up by malfunctioning outputs. For example, the serial port routines do not wait forever for characters to be transmitted.

Restarting the System

System restart (as far as possible equivalent to a hardware reset) will occur if software jumps to the reset location 0xbfc00000. No ‘‘warm restart’’ is provided for by this code; it is assumed that anyone wanting to preserve machine state will not want to run the test sequence.

Notes

Standard Test Sequence

The tests are summarized in below:

Mnemonic	Test Summary
init	setup CPU and system (from a cold start)
vac-reg	register access tests on VAC068
led	display "8" then "0"
endian	check consistency of bigend jumpers and ROM, stop on error
can use byte variables now	
mem-conf	check memory size and that configuration is OK (there is a jumper which needs to match the type of DRAM chips used)
mem-min	uncached write/read address test on PROM data area
in C from here on...	
prom	checksum PROM sections and warn
nvram	checksum environment region, use defaults if wrong
can use environment variables from here on...	
cache	sizes caches and then performs internal write/read test (address in address)
refill	d-cache from PROM, then d-cache from main memory
vac-timer	check that programmable timers run, and that interrupt signals are reaching the VAC
fpa	test for presence, interrupt wiring
nvram-rtc	check clock (built in with nonvolatile RAM module) for reasonable value, warn if it lost power.
vic-reg	register access tests on VIC068
vic-timeout	check local bus timeout
vic-timer	confirm timer working
vic-int	check that VIC interrupts are getting through to the CPU, and that the interrupt acknowl-edge mechanism works.
vic-scon	Is this system a VMEbus controller? set env variable
mem-best	fast address-based confidence check
mem-parity	check out that the parity check logic is accepting good and detecting bad parity
mem-soak	sequence of ‘‘thorough’’ memory tests
uart-reg	register write/read tests on 72001 (UART)
uart-init	initialize 72001 (suspiciously) and send a character
eth-reg	register access tests on SONIC
eth-read	get SONIC to read memory and check (also detects interrupt)
eth-write	get SONIC to write (or copy) memory and check
scsi-reg	register access tests on 53C710. Also check out the byte-swapper which is available for little-endian mode if required.
scsi-read	get 53C710 to read memory and check (and check its interrupt)
scsi-write	get 53C710 to write (or copy) memory and check

Notes

Notes on the Test Sequence

- *From Reset: The CPU restarts at the usual PROM location, running uncached. This PROM is intended to restart in the same way regardless of whether the starting location was reached by a hardware reset or a software jump; so everything which can be is reset.*

  The sequence is complex and goes like this:

  1. There is a branch instruction at the boot location. A failure to read the ROM correctly will lead to the CPU getting an immediate exception, failing to branch, or branching to the wrong address. All are pretty obvious to an engineer watching addresses on a logic analyzer.
  2. Initialize the status register to place the CPU in a reasonable mode. Software preserves the prior-to-reset values of *ra* and *epc.* They have to be put into general purpose registers, since at this stage the memory can not be trusted.
  3. The part of the ROM containing the test code is now check-summed. If this passes, ROM code should be able to be correctly read and executed. This is a reasonable piece of confidence testing, but in fact if the PROM doesn't work perfectly software would probably never have got here.

Now perform IO system initialization.

4. Write to PROM space (required by the VAC068 chip to drop it out of ''reset mode'' – where ALL cycles are decoded as for ROM).
5. Initialize the VIC and VAC chips (which control onboard IO cycles) with a series of register writes. The register addresses, and the data to be written to them, are defined in a table – which, as it consists only of constant data, can be defined in a C module.
6. The SL–3000 is equipped with a board control register whose outputs hold various subsystems in reset; program it to reset everything which can be.
7. Program the serial ports. They can now be used for reporting any problem (although they cannot yet be trusted to work).
8. Wait 1 second while the user takes in the existing state of the LED (just in case it might be important).

- *vac-reg: a typical first test on an intelligent controller; pick a register which can be written with any 16-bit value, and read back, and which has no harmful side effects. This proves out the basic address paths in the IO system, and (half of) the data bus; and the system will shortly need to program the VAC device before many other parts of the system will work.*
- *led: enable hex display and flash it from "0" to "8". From now on software will go on flashing the display to demonstrate progress.*
- *endian: check that the PROM endianness makes sense (up to this point all the code is "bi-directional", which involves avoiding all partial-word loads and stores). If the board's configuration jumper and the PROM type are mismatched, flash/print an error message and stop.*
- *mem-conf: check that the board is not equipped with small DRAMs but configured for big ones (this state leaves holes in the memory).*
- *mem-min: perform minimal memory test. In the event of any problems, report and carry on (no good can be accomplished by stopping).*

  These tests need only cover uncached accesses to memory made while running uncached from PROM, and can be restricted to that portion of the memory used by the PROM software. They need to be restricted too; since the system is still running uncached, a test of the whole of memory would take too long.

  Once this has passed, the system is capable of supporting compiled test code.
- *prom: compute and compare a simple 32-bit add/carry checksum on each "package" in the PROM, intended to detect single-bit dropout and mis-programming. A zero stored checksum (an impossible result with add/carry) suppresses the check for those who can't be bothered to maintain the checksum during PROM development.*
- *nvram: verify checksum on NVRAM environment area. If it is wrong, use default environment settings. The default settings will cause tests to be more verbose and more thorough.*

  If environment does not suppress console output, print a console sign-on message.
- *cache: figure out the size of the I- and D-caches, using the diagnostic isolate/swap cache features*

Notes

(see the chapter on cache management). The cache size is left in an environment variable, because system software will want to know it later.

Now do simple memory tests in the caches, using an address-in-data test to produce different patterns. The test is coded in C and run uncached, using a tiny assembler subroutine to read/write a single word in the cache; the emphasis is on making the code as obvious as possible. This module cannot be tested except by chance (since all RC30xx family CPUs work and the caches are internal) – so it had better be right by design.

- *Refill from ROM: check out cache refill from PROM. This exercises some logic which puts together ROM cycles into (slow) bursts on request, to allow ROM code to be run cached.*
- *Refill from main memory: the main memory logic provides real high-speed bursts of data. Check that at least a pattern (which is designed to cause each data bit to change as much as possible) can be read.*

  If all cache tests pass, further test software can be run cached where necessary. This is really needed – it is impracticable to run a thorough memory test in a reasonable period of time unless the caches are enabled.
- *vac-timer: see whether the VAC timers will run.*
- *fpa: check for presence and consistent interrupt configuration, but do not expect to perform a functional test.*
- *nvram-rtc: check for a plausible value in the real time clock registers and record it.*
- *vic-reg: write/read test on VIC068 registers.*
- *vic-timeout: the VIC068 is used to timeout local bus accesses to nonexistent locations. Make sure this works and causes a bus error (involves catching the exception).*
- *vic-timer: check that the VIC068 interval timer is giving periodic interrupts.*
- *vic-int: check that VIC interrupts are getting through to the CPU, and that the interrupt acknowledge mechanism works.*
- *vic-scon: obtain whatever detail is available on the VMEbus environment without doing anything. This includes reporting on whether the board is configured as system controller, and the state of the backplane SYSRESET* and ACFAIL* lines.*
- *mem-best: "best-efforts" is necessarily relative to the amount of time allowed for testing memory (Algorithmics believes something around 10s is sensible). This small amount of time allows nothing more complex than an address-in-address test. Speed is probably more useful than theoretical thoroughness.*

  The diagnostic will report the memory size into an environment variable.
- *mem-parity: use the diagnostic area to write bad parity to a memory location, and then test that it is detected and reported.*
- *mem-soak: optionally (enabled by an NVRAM environment entry) run a much more complete memory test. Parity checking can be used to detect errors.*
- *uart-reg: check out 72001 UART connections by write/read registers.*
- *uart-init: check out that serial ports are responding (to the extent possible without writing characters to any but the console).*
- *eth-reg: write and read-back test of register bits. Program up the controller and look for plausible status.*

  Note that no test is made for the presence of a transceiver or a network connection. Higher level bootstrap software should take care to report such conditions.
- *eth-read/eth-int: persuade the SONIC to read memory as master, by issuing a "load CAM" command.*

  Completion of the load will cause an interrupt; track this through the VIC and to the CPU pin. Note that it is quite legitimate to do this with interrupts disabled in the CPU; the CPU can see the state of its pins.

Notes

- *eth-write: persuade the ethernet to write something to memory and check it. This may involve an internal loopback command, but anything which writes memory will do.*

  After the test the ethernet controller will be reset.

- *scsi-reg: register write/read of 53C710 controller.*

  The way the SCSI controller is wired -up allows diagnostic software to check that the IO bus byte swapper is configured as expected by the PROM. This is particularly important because the byte-swapper is mainly used for network and SCSI data, and corruption to these won't be noticed until an embarrassingly long way into bootstrapping. Software records the actual CPU and IO endian-ness in environment variables.

- *scsi-read: persuade the 53C710 to read memory (by persuading it to read a very simple SCRIPT) and check. This causes an interrupt, which the diagnostic checks can be delivered all the way to the CPU pins.*

- *scsi-write: get the 53C710 to write to memory and check it.*

  Leave the SCSI controller reset after the test.

Annotated Examples from the Test Code

These examples concentrate on the first, low-level code which has to be in assembler (since writable memory is not yet trusted, and C code can't be used without some memory for a stack).

- *Starting Up: the PROM is linked with its first module starting like this (observe that the "li" which identifies this as an absolute reset is explicitly placed in the branch delay slot of the jump):*

```
        .text
        .set    noreorder
_stext:
bt_rvec:
        j bt_bootpkg; li a0, 7
        ...
        /* a lot later is the exception vector, 0x180 bytes
         * up
         */
        ...
        j it_bevgen; nop
```

This jumps to start off the real code, which in this case is designed for a PROM space broken up into "packages" each of which is a separately-linked program. But the first few instructions are likely to be required on pretty much any start-up PROM.

Zero is placed into k0 because the exception routine uses this as a flag – a nonzero value in k0 will be taken as the address of a user-installed exception routine.

```
LEAF(bt_bootpkg)

        move    k0,zero

        .set    noreorder

        li      s1,SR_BEV       /* complete SR initialization*/

        mtc0    s1,sr
        nop
        nop
```

After two "nop"s the new status register has taken effect and the CPU can be trusted. Software can now save the epc and ra registers, which are potentially useful in telling users what was happening before reset:

```
        /*
         * save epc & ra so that they can be passed to package
```

Notes

```
        */
        mfc0    s1,epc
        .set    reorder
        move    s2,ra
```

Now read the "package" record, which is a little bit of PROM space at a well known address (1024 bytes above the start of the PROM). Each of 8 possible records contains 4 words of information: a magic number, the start address, end address, a checksum, and a start location.

The register a0 (conventionally used for the first argument of a subroutine) is used to pick one of 8 packages to run, and the 7th points to the start of the power-on tests:

```
        bltu    a0,NPKG,1f      # make sure package is in range
        li      a0,NPKG-1
1:
        /* get pointer to package info */
        sll     a0,PKGSHIFT+2
        la      s0,bt_pkginfo
        addu    s0,a0

        lw      t0,oMAGIC(s0)   # get magic number
        li      a0,BT_BADPKG
        bne     t0,+BTMAGIC,bt_fail# must be same as us
```

Now the diagnostic will calculate a checksum for all the PROM locations for the code and constant data of the power-on test code. Note that, even without a stack, software can call a subroutine; recall that the MIPS hardware implements no stack functions, and the subroutine call instruction ("jal" for jump-and-link) puts the return address into register ra.

```
        lw      a0,oSTART(s0)
        lw      a1,oEND(s0)
        jal     bt_chksum

        lw      t0,oSUM(s0)
        beq     t0,v0,1f        # good checksum?

1:
        /* jump at selected code */
        move    a0,s1
        move    a1,s2
        lw      t0,oENTRY(s0)
        j       t0
END(bt_bootpkg)
```

Now the boot process really gets started. it_main implements the test sequence. Once again it is possible to call one level of subroutine without a problem:

```
/*
 * entry point for integrated tests
 * a0,a1 contains epc,ra
 */
NESTED(it_main,0,ra)

        li      v0,SR_BEV|SR_PE
        .set    noreorder
        mtc0    v0,sr
        .set    reorder

        move    s0,a0
        move    s1,a1

        /*
         * initialize the board and IO systems
         */
```

Notes

```
        jal     sbd_init
        jal     sbd_ioinit


        /* to see LED state */
        li      a0,250
        jal     sbd_msdelay/* a VERY rough 250ms pause */

        jal     sbd_basic/* tests before memory sizing */
        move    s2,v0           /* save memory size */

        li      a0,PA_TO_KVA1(0)
        li      a1,0x10000
        jal     sbd_memmin/* test 1 Mbyte of memory from 0 */
```

Now the software can trust the memory. After saving a few things in their assigned global locations, a stack is defined and the program is written in C:

```
        /* at last put them into memory */
        sw      s0,epc_at_restart
        sw      s1,ra_at_restart
        sw      s2,mem_size

        /*
         * might have usable memory so give up on the
         * assembler and use C
         */
        li      sp,PA_TO_KVA1(0xfffc)
        jal     it_cmain
```

Note that it doesn't really return, just goes off and finds the next package.

```
        jal     sbd_closedown

        /*
         * tests have completed so execute next package
         */
        move    a0,v0
        j       bt_bootpkg
END(it_main)
```

This next section describes how some of the more significant subroutines are implemented.

- *sbd_init: The SL–3000 hardware suffers from intelligent peripheral controllers which require to be programmed in a precise sequence; until this is done many "normal" functions just don't work.*

The code has to do a dummy write to ROM space first (the programmable decoder, from reset, will map every cycle onto ROM space):

```
/*
 * basic initialization
 */
LEAF(sbd_init)
        /* kick VAC068 out of force eprom mode */
        sw      zero,PA_TO_KVA1(LOCAL_PROM)
```

Now the program uses a table of register addresses and values to be written to them. The table itself can be defined in a C module, making it readable and allowing the use of the same header files as for more complex device drivers:

```
        /* initialize VAC registers */
        la      a0,vicvacresettab
vicvacdefloop:
```

Notes

```
                /* v0 gets pointer to VIC/VAC register */
                lw      v0,0(a0)

                beqz    v0,vicvacdefend

                lw      v1,4(a0)
                sw      v1,0(v0)

                add     a0,8
                b       vicvacdefloop
vicvacdefend:
```

Now the board appears to work, so the code kind of starts again. The "BCRR" address is a hardware register whose outputs hold most subsystems in reset:

```
                /*
                 * hold all devices in reset and disable LED
                 */
                li      v0,BCRR_LBLK
                sw      v0,PA_TO_KVA1(BCRR)

                /*
                 * VIC will bus error any accesses made while SYSRST
                 * is active so wait until SYSRST goes away
                 */
                li      v0,PA_TO_KVA1(BCRR)
1:              lw      v1,0(v0)
                and     v1,BCRR_SYSRST
                beqz    v1,1b
```

This breaks the earlier rules (this is a loop which can continue for ever) but with all local bus cycles being terminated with a bus error the system should not hang in an infinite loop.

The VMEbus power-on test convention is that each board should assert the SYSFAIL* signal until it has passed its power-on tests. So for the moment, assert it:

```
                /* make sure that SYSFAIL is asserted with a 'reset'
                 * code
                 */
                li      v0,VIC_SYSFAIL|VIC_STATLRESET
                sw      v0,VIC_VSTATUS

                j       ra
END(sbd_init)
```

- *Doing without a stack: more complex test software would like to be able to call subroutines. But without a memory-based stack, it is impossible to properly track the return address. Therefore, the early tests borrow three of the 32 registers and define a pseudo-stack and a couple of macros to use at the beginning and end of subroutines.*

  These are for use in assembly code, but are implemented with the C preprocessor:

```
#define _t6     $14
#define _t7     $15
#define _gp     $28


#define PUSHRA  move    _gp,_t6; \
                move    _t6,_t7; \
                move    _t7,ra


#define POPRA   move    ra,_t7; \
```

Notes

```
        move    _t7,_t6; \
        move    _t6,_gp;  \
        move    _gp,zero
```

"POPRA" puts zero into the stack bottom; if the program should under run the stack the result will be an attempt to return to address zero, which would be trapped by the memory-management hardware, if fitted.

The MIPS assembler defines the conventional register names using the C preprocessor; so to make sure these registers aren't used, they are "undefined":

```
/* of course this means the programmer can't use these... */
#undef gp
#undef t6
#undef t7
```

- *First test of first device: on the SL–3000 board the VAC068 device (which connects the address lines of the VMEbus) integrates onboard device decode and control functions. Although it is initialized, unless it works nothing else will; so it must be a good place to start:*

```
/*
 * The VAC has already been initialized
 * Here just try writing/reading a VAC register
 */
SLEAF(tst_vacreg)

        /*
         * checkerboard test on VACPIODATO register
         * luckily this does not affect anything on the board
         */
        li      t0,0xaaaa0000
        sw      t0,VACPIODATAO/* store data in register */
        not     t0
        sw      t0,VACID/* complement to VACID (read-only) */
        not     t0
        lw      t1,VACPIODATAO/* reread register */
#ifdef ALLFAIL
        xor     t1,0x80000000
#endif
        and     t1,0xffff0000
        bne     t1,t0,9f   /* was it ok? */
```

Earlier, this chapter discussed the difficulty in testing the test software; the "#ifdef ALLFAIL" can be used to build in automatic failure, so at least the error reporting routines are tested.

```
        /*
         * now try the other bits
         */
        li      t0,0x55550000
        sw      t0,VACPIODATAO/* store data in register */
        not     t0
        sw      t0,VACID/* complement to VACID (read-only) */
        not     t0
        lw      t1,VACPIODATAO/* reread register */
        and     t1,0xffff0000
        bne     t1,t0,9f   /* was it ok? */


        /* read the VAC ID register and check the contents */
        lw      t0,VACID
        and     t1,t0,VAC_IDENTMASK
        bne     t1,VAC_IDENT,9f

        /* return the revision ID */
        and     t0,VAC_REVMASK
```

Notes

```
        srl     t0,16
        j       ra

9:      li      a0,IT_VACREG
        j       _it_signal
SEND(tst_vacreg)
```

The routine *_it_signal*() attempts, by all means available, to communicate the result of a test:

- *Reporting errors without printf:*

```
/* assembler doesn't support character literals */
#define NL0x0a

/*
 * low level error report
 * trashes: a2;a0,v0,a1,v1
 */
LEAF(_it_signal)
        PUSHRA
```

Here is a use of the register-stack macro, allowing the error routines to nest to a depth of four:

```
        jal     _sbd_signal

        jal     sbd_displaycode

        move    a2,a0     /* don't change sbd_printmsg */

        la      a0,errormsg
        jal     sbd_printmsg

        move    a0,a2
        jal     sbd_printcode

        li      a0,NL
        jal     sbd_printc

        POPRA
        j       ra
END(_it_signal)
```

The constituent routines are:

- *_sbd_signal controls one of the system's way of telling the world its troubles – in this case, by placing an error code in an 8-bit register dual-ported to the VMEbus (implemented in the VIC controller), and driving the wire-OR'ed VMEbus SYSFAIL line.*
- *sbd_displaycode uses the LED display to show the same 8-bit error value; it does this by blanking the display momentarily, then showing the byte value as two nibbles (most-significant first).*
- *sbd_printmsg, sbd_printcode between them report the error to the console. Used only for desperate conditions, it entirely ignores the user's expressed wishes about the serial ports – on the grounds that for a fatal error silence is always wrong. The "printcode" routine explains the error code with a message from the table codemessages (tstmessages.c).*

- *Endianness-proof code and testing endianness: the SL–3000 board can be set up (with option jumpers) to run either in big-endian or little-endian mode. Usually, software has to be built for the correct endianness, but Algorithmics wanted to ensure that the power-on test would at least tell the user if the jumpers were set wrongly for the installed ROM.*

  However, MIPS instructions are all 32-bit words, and are all designed as bit codes. Provided the system correctly wires up the bit numbers within each 32-bit word (which is the most "natural" way to wire up a 32-bit MIPS processor), the instruction encoding does not change between big- and little-endian. What does change is the effect of partial-word load and store instructions; but so long as the software doesn't use partial-word operations the code will run in either mode.

Notes

A CPU can easily sense its own endianness by comparing the result of a byte load with the word-value contents of the location:

```
        .rdata
littleflag:
        .word   1
        .text
                .align  2
ycnegreme:.ascii“remEcneg  00:y”
```

It is quite difficult to spell in the wrong endianness...

```
LEAF(tst_endian)
        la      v0,littleflag
        lbu     v0,0(v0)

#if BYTE_ORDER==LITTLE_ENDIAN
        beq     v0,zero,9f
#endif
#if BYTE_ORDER==BIG_ENDIAN
        bne     v0,zero,9f
#endif

        j       ra

9:
        la      a0,ycnegreme/* "Emergency" backwards */
        jal     sbd_printmsg

        li      a0,IT_ENDIAN
        /* message in code table is backwards too */
        jal     sbd_printcode

        li      a0,NL
        jal     sbd_printc

1:      li      a0,IT_ENDIAN
        jal     sbd_displaycode
        b       1b
SEND(tst_endian)
```

Instruction Timing and Optimization

Notes

The great majority of MIPS instructions require their operands by the end of the “RD” (second) pipeline stage, and produce their result at the end of the “ALU” (third) stage. If all instructions could always stick to these rules, any instruction sequence could be correctly run at maximum speed. The great power of the MIPS architecture is that the vast majority of instructions can stick to this rule.

Where this can’t be done for some reason, an instruction taking operands from the immediately preceding instruction may not run correctly. A lot of the time, this will produce unpredictable behavior – a *pipeline hazard*, and it is up to the programmer, compiler and assembler (together) to keep those instruction pairs apart. This can sometimes be done by moving code around, but otherwise the programmer can insert a **nop**.

In other cases, the sequence will work but will result in execution pausing while the desired result is produced – an *interlock*. Compilers, assemblers and programmers would like to move code around to avoid interlocks to maximize performance. Table 13.1 lists all instructions that either require their operands to be delivered earlier than usual or that deliver their results late.

If one instruction delivers a result used by a subsequent instruction, and either instruction is listed in Table 13.1, the sum of the late-result count of the first instruction and the early-operand count of the second gives the number of **nop** or other intervening (independent) instructions required to prevent a hazard or interlock.

A tick in the “hazard” column means that failure to observe these conditions will break a program, and the assembler, unless inhibited, will probably insert **nop** instructions to avoid the problem. No tick means the problem is interlocked.

Instruction	Early Operand	Late Result	Hazard?	Notes
Branch instructions		1	3	where result is new “PC” value, i.e. delayed branch
Load instructions	lw, lh, lhu, lb, lbu, lwc1	1	3	load delay interlock RC4xxx
lwl, lwr	0/-1	1	3	*late* read of value to merge, so no delay needed between **lwl/lwr** pair
mult, multu (RC30xx)		11		result interlocked
mult, multu (RC4600)		10		result interlocked
mult, multu (RC4700)		8		result interlocked
mul, mult/u, mad/u (RC4650/ RC32364) msub/u (RC32364)		3 (16-bit) 4 (32-bit)		result interlocked
dmult, dmultu (RC4600)		12		result interlocked
dmult, dmultu (RC4700)		10		result interlocked
dmult, dmultu (RC4650)		6		result interlocked
div, divu (RC30xx)		35		result interlocked
div, divu (RC4600, RC4700)		42		result interlocked
div, divu (RC4650, RC32364)		36		result interlocked

Table 13.1 Instructions that Require an Operand

Notes

ddiv, ddivu (RC4600, RC4700)		74		result interlocked
ddiv, ddivu (RC4650)		68		result interlocked
Integer/control register moves: **mfc0, mtc0**		1	3	
FP conditional branches: **bc1t, bc1f**	1	1	3	
Integer/FP moves **mfc1, mtc1, ctc1, cfc1**		1	3	
FP addition unit ops **add.s, add.d, sub.s, sub.d**		+1		
mul.s (RC3081)		+3		interlocked
mul.s (RC4600 / RC4650)		+8		interlocked
mul.s (RC4700)		+4		interlocked
mul.d (RC3081)		+4		interlocked
mul.d (RC4600)		+8		interlocked
mul.d (RC4700)		+5		interlocked
div.s (RC3081)		+11		interlocked
div.s (RC4600/RC4700/RC4650)		+32		interlocked
div.d (RC3081x)		+18		interlocked
div.d (RC4600/RC4700)		+61		interlocked
div.d (RC4600/RC4700)		+61		interlocked
sqrt.s (RC4600/RC4700/RC4650)		+31		interlocked
sqrt.d (RC4600/RC4700)		+60		interlocked
cvt.w.s, cvt.w.d, cvt.s.d (RC3081)		+1		interlocked
cvt.w.s, cvt.w.d, cvt.s.d (RRC4600/RC4700)		+4		interlocked
cvt.s.w, cvt.d.w (RC30xx)		+2		interlocked
cvt.s.w, cvt.d.w (RC4600/RC4700)				interlocked

Table 13.1 Instructions that Require an Operand (Continued)

Table 13.2 lists the floating point execution rate for RC5000 operations.

Operation	.S		.D		.W		.L		Other.	
	Latency	Repeat	Latency	Repeat	Latency	Repeat	Latency	Repeat	Latency	Repeat
LWC1/LDC1									2	1
LWXC1/LDXC1									2	1
PREFX									0	2
SWC1/SDC1									2	1
SWXC1/SDXC1									3	2
MTC1/DMTC1									2	1
MFC1/DMFC1									2	1
CTC1									6	3

Table 13.2 RC5000 Floating Point Unit Execution Rate

Notes

Operation	.S		.D		.W		.L		Other.	
	Latency	Repeat	Latency	Repeat	Latency	Repeat	Latency	Repeat	Latency	Repeat
CFC1									2	1
BC1									2	2
ABS/NEG	1	1	1	1						
C.cond	2	1	2	1						
MOV-	1	1	1	1						
MADD/MSUB/ NMADD/NMSUB	4	1	5	2						
ADD/SUB	4	1	4	1						
MUL	4	1	5	2						
ROUND.W/ TRUNC.W	4	1	4	1						
ROUND.L/ TRUNC.L	4	1	4	1						
CEIL.W/ FLOOR.W	4	1	4	1						
CEIL.L/ FLOOR.L	4	1	4	1						
CVT.S			4	1	6	3	6	3		
CVT.D	4	1			4	1	4	1		
CVT.W	4	1	4	1						
CVT.L	4	1	4	1						
DIV	15	15	30	30						
SQRT	15	15	30	30						
RECIP	15	15	30	30						
RCPSQRT	32	32	62	62						

Table Notes:
Round.L, Trunc.L, Ceil.L, Floor.L each trap on greater than 52 bits of significance.
CVT.D.L traps on greater than 53 bits of significance.

Table 13.2 RC5000 Floating Point Unit Execution Rate

Notes and Examples

- *Any branch takes effect late, so the instruction following the branch is always executed. It's often possible to move the last instruction which logically precedes the branch around; clever compilers may be able to figure out that the instruction normally at the branch target can successfully be put in the delay slot, speeding up loops; failing all else, the slot can be filled with a* ***nop****.*
- *A load from memory into any register produces its result late, so a "delay slot" is needed before the result is used:*

```
lwc1    $f0, 42(t0)
nop
add.s   $f4, $f2, $f0
```

- *A branch on FP condition needs the C bit early, so a gap is needed between a "set" instruction and the branch:*

Notes

```
c.eq.s   $f0, $f2
nop
bc1t     thesame
```

- *The sequence below requires two nops (though this sequence may be highly unlikely)*

```
ctc1     t0, $31
nop
nop
bc1t     somewhere
```

Additional Hazards

Early Modification of HI and LO

An interrupt or trap will abort most instructions, and the resulting writeback will be inhibited. But this isn't done in the integer multiply unit; changes to the multiply unit registers cannot be prevented once multiply and divide instructions start.

An exception might occur just in time to prevent an **mfhi** or **mflo** from completing its writeback, but still allow a subsequent multiply or divide instruction to start. By the time the exception completes (or equivalently, by the time the exception routine saves the *lo* or *hi* register values) the multiply/divide could have overwritten the data and exception recovery won't happen properly.

To avoid this ensure that at least two instruction times separate an **mfhi** or **mflo** instruction from a following multiply or divide instruction.

Bitfields in CPU Control Registers

Some of the CPU control registers ("coprocessor 0") contain bitfield values or flags which have side effects on the operation of other instructions. Unless specifically documented below, software must assume that any such side effects will be unpredictable on the three instruction periods following the execution of an **mtc0**.

The following are specifically noted:

- *Enabling/disabling a group of co-processor instructions: use of CP0 instructions in the following two instructions is unpredictable – in particular the CPU may, or may not, trap.*
- *Enabling/disabling interrupts: the enable won't allow an interrupt to affect (i.e. get in before, abort the writeback phase of) the following two instructions; it can happen before the third.*

  Similarly, when disabling interrupts the following two instructions may nonetheless be interrupted.
- *TLB changes and instruction fetches: there is a 2 instruction delay between a change to the TLB and it taking any effect on instruction translation. Additionally, there is a single-entry cache used for instruction translations (called the micro-TLB) which is implicitly flushed by loading EntryHi, which can also delay the effect.*

  The OS should only perform TLB updates in code running in an unmapped space.

Hazards Specific to RC4xxx, RC32364 and RC5000

In Table 13.3 the number of instructions required between instruction A (which places a value in a CP0 register) and instruction B (which uses the same register as a source) is computed using the following formula:

(destination stage of A) - (source stage of B) - 1

Notes

Operation	SOURCE NameStage		DESTINATION NameStage	
MTC0	gpr rt	2(A)	cpr rd	4(W)α
MFC0	cpr rd	2(A)	gpr rt	4(W)α
TLBR	Index, TLB	2(A)	PageMask, EntryHi,EntryLo0, EntryLo1	4(W)
TLBWI TLBWR	Index or Random, PageMask, EntryHi, EntryLo0, EntryLo1	2(A)	TLB	3(D)β
TLBP	PageMask, EntryHi	2(A)	Index	4(W)
ERET	EPC or ErrorEPC, Status.ERL	2(A)	Status.EXL, Status.ERL	4(W)γ
			LLbit	4(W)
CACHE Index Load Tag			TagLo, TagHi, ECC	3(D)
CACHE Index Store Tag	TagLo, TagHi, ECC	3(D)		
Instruction fetch	EntryHi.ASID, Status.KSU, Status.RE, Config.K0C, TLB	0(I)		
	Status.ERL, Status.EXL	0(I)γ		
Instruction fetch exception			EPC, Status, Cause	4(W)
			BadVAddr, Context, EntryHi	1(I)δ
Coprocessor usable test	Status.CU, Status.KSU, Status.EXL, Status.ERL	1(R)		
Interrupt	Cause.IP, Status.IM, Status.IE, Status.EXL, Status.ERL	2(A)		
Load/Store	EntryHi.ASID, Status.KSU, Status.RE, Status.ERL, Status.EXL Config.K0C, TLB	2(A)		
Load/Store exception			EPC, Status, Cause, BadVAddr, Context, EntryHi	4(W)

Notes:
a There must be at least one instruction between a MTC0 and a MFC0.
bTLBW_ instructions will cause a one cycle slip.
gInstructions fetches following an ERET will see a change in EXL or ERL in Stage 2 of the ERET in anticipation of the completion of the ERET. If the ERET does not complete, these instructions are killed before they commit changes in state other than noted by d. The pipestage corresponding to the stage field is given in parentheses.

Table 13.3 Instruction Requirements Between Instructions A & B

Hazards Specific to RC4650

- *A mtc0 CAlg must not change the field corresponding to the currently active address space. The result is undefined.*
- *A mtc0 that changes any base or bounds register must execute in unmapped space. Mapped space cannot be entered for 5 instructions following a change to these registers.*
- *When DWatch is enabled, the two immediately following instructions may not checked for a match with the watch value.*
- *When IWatch is enabled, the five immediately following instructions may not be checked for a match with the watch value.*
- *When the **IL** bit (bit 23) of the status register is changed, refills to set A of I-cache may not be disabled until 5 instructions later.*

Notes

- *When the **DL** bit (bit 24) of the status register is changed, refills to set A of the D-cache may not be disabled until 3 instructions later.*

Hazards Specific to RC32364

- *A mtc0 followed by a mfc0 is undefined. A one instruction delay between mtc0 and mfc0 is needed for proper operation.*
- *When DWatch is enabled, the two instructions immediately following may not be checked for a match with the watch value.*
- *When IWatch is enabled, the five instructions that follow may not be checked for a match with the I match value.*
- *When bit 23 of the Status register is changed, refills to set A may not be disabled until five instructions later.*
- *When bit 24 of the Status register is changed, refills to set A may not be disabled until three instructions later.*

Non-obvious Hazards

There are other device “hazards” which can’t be determined by examining the processor pipeline. In general, these are due to the amount of time required for changes to CP0 registers to “propagate” to the cache, bus interface, or exception controller of the device.

The CPU hardware user’s manual specifies a number of clock cycles, and whether software can operate cached, for modifications to RC3041 and RC3081 specific CP0 registers. The programmer is referred to those manuals for additional information.

One other common “hazard” bears particular note: modifying the IEc and IM bits of the status register in a single CP0 instruction is not recommended. The effects of these bit fields may or may not be seen in different clock cycles; thus, changing both with a single mtc0 or ctc0 instruction may result in side effects such as spurious interrupts (if for example the new value unmasked a previously masked interrupt but was also attempting to clear the global IEc bit).

Software Tools for Board Bring-Up

Notes

This chapter describes the software tools typically used by IDT when debugging a board for the first time. Additional details on system design and debugging are available from IDT in application notes, evaluation boards, and design guides.

Tools Used In Debug

In a typical system, IDT engineers use the following tools for initial board debugging:

- *Logic analyzer: This tool is indispensable for determining why a particular memory sub-system is malfunctioning. Although other diagnostic tools are used to determine which subsystems are misbehaving, ultimately a logic analyzer is used to trace the misbehavior, so a work-around or fix can be applied.*

  Use of the logic analyzer may be complemented through the use of a device specific "pod". These pods are designed to be inserted into the CPU socket and recognize the device bus protocols. These pods typically can disassemble incoming instructions as well, facilitating debug of programs as well as hardware.

- *ROM emulator: IDT frequently applies ROM emulator tools to minimize the hassle of burning new sets of EPROMs as higher levels of code is developed. A word of caution: some ROM emulators may take actions (desired or not) when the ROM space is written to; the hardware designer should review the requirements of the ROM emulator to insure system compatibility.*

- *In-circuit emulator: In some cases, IDT will apply an in-circuit emulator to a debug task.*

  Many developers rely heavily on the use of in-circuit emulation for system debug; others rely exclusively on software-based debug techniques coupled with generic measurement equipment. In-circuit emulation can certainly be a useful tool, although it may prove to be outside the project development budget.

- *IDT Micromonitor: The IDT micromonitor is a small program designed to help discover and debug problems in the system RAM.*

  Since the micromonitor is designed to help debug system RAM, it does not assume that RAM resources are available to it. Thus, the micromonitor is written in assembly and does not require a stack or variable storage; it uses the on-chip register file for temporary data storage.

  For the Micromonitor to operate, the ROM sub-system must work, and the system console must work (typically a UART for serial i/o).

  The Micromonitor contains a number of diagnostics for system RAM, designed to insure that address and data lines are correctly connected; that DRAM refresh works properly; that cached and uncached accesses function properly; etc. Successful use of the micromonitor gives the debugger confidence in the board memory system.

- *IDT/sim (system integration manager): This is a PROM monitor/debugger program, designed to run in a target system. IDT/sim gives the engineer the ability to set breakpoints, peek and poke memory, install new commands, examine machine state, single step, etc.*

  In addition, IDT/sim contains the communications interface to a number of host-resident remote target high-level language debuggers, including GDB and MIPS DBX. With IDT/sim executing on the target board, the programmer can perform high-level language debugging on the target from the development host.

Notes

Initial Debugging

When debugging is first begun, the engineer generally will not even be confident of the proper behavior of the ROM and RAM memory subsystems.

A number of techniques can be used during this initial debug. Some engineers prefer to use an in-circuit emulator with overlay memory to cause the CPU to make repetitive accesses to the memory while the engineer probes it with a logic-analyzer and/or oscilloscope. Other engineers will just "try the boot prom" and use a logic-analyzer to see the first few cycles after reset (typically to the boot prom). Again, a logic-analyzer pod may prove helpful in showing what instruction finally arrives back at the CPU data pins.

Debugging the ROM and UART subsystem are preliminary steps required for the micromonitor, SIM, and remote target debug. There is no particular "mystery" to doing this with the IDT family; just good old-fashioned debugging.

Porting The IDT Micromonitor

Porting the micromonitor typically requires only two steps:

- *determining the UART address: this will be system specific. In micromonitor, there is an assembler directive inside the source file used to define the UART_BASE address. The programmer needs to modify this line to reflect the system address map.*
- *provide the UART driver: if the system uses an 8251/8530/2681/68681/uPD72001(NEC), or compatible UART, then the programmer can use one of the drivers provided with the micromonitor. Otherwise, the programmer needs to provide a rudimentary UART driver for the system UART.*

  There is an advantage to patterning new drivers after UART drivers provided with the micromonitor. In general, a full device driver is probably not required--fixed baud rates, a single receive or transmit character from a CPU register, and programming in assembly are all appropriate to the goals of the micromonitor.

  If selecting one of the existing UART drivers, the programmer should set the appropriate assembly file line to indicate the selected driver.

Running the IDT Micromonitor

The micromonitor documentation describes the proper running of the micromonitor program. In general, the micromonitor should be used until all of the diagnostic tests of system RAM can be completed successfully and repeatedly, running both cached and uncached.

At this time, the engineer is confident that the ROM and RAM systems operate correctly, and can be accessed cacheably (in four word bursts) and non-cacheably. In addition, partial word accesses to the system RAM are now verified.

The engineer is now free to move on to porting SIM, and debugging the remainder of the I/O subsystems.

Initial IDT/SIM Activity

The first goal for running IDT/sim is to merely get to the basic IDT/sim prompt. This should not rely on subsystems other than those already confirmed using micromonitor: the ROM, RAM, and UART. Thus, the only problems that should be expected are programming, not system.

However, there is one common problem that can slow progress:

- *Improper memory sizing algorithm. IDT/sim usually performs a RAM area sizing operation, to determine the amount of system RAM. It then places the stack pointer at the top of system RAM. If the memory sizing algorithm does not work properly, the stack could be placed in non-existent memory, or SIM could be fooled into thinking there is 0kB of memory. In either case, SIM would not boot or execute properly.*

  To avoid this problem many engineers "hard-wire" a memory size into SIM for initial boot and system test. For example, an evaluation board might be populated with 1MB of DRAM, and SIM hard-wired for 1MB of RAM. The memory sizing algorithm could then be debugged later.

Notes

Once IDT/sim is at the system prompt, the engineer can complete the process of system debug. At this point, the ROM, RAM, and console UART subsystems are executing properly.

The engineer may choose to use “Peek” and “Poke” operations into the memory space to test accesses to peripherals, or instead may begin porting drivers and diagnostics. IDT/sim will provide a rich execution environment, including breakpoints, single step, cache and memory housekeeping, in-line assembly, download, etc.

The system engineer can also choose to apply other traditional microprocessor development tools, including ROM emulators, in-circuit emulators, and also use remote target debugging, during the actual system software port.

A Final Note on IDT/KIT

In addition to the functions found in IDT/sim, IDT offers the Kernel Integration Toolkit. IDT/kit contains many bits of “housekeeping” code for the system environment builder, including functions such as cache flushing/management software and exception decode and dispatch. IDT/kit contains the “processor specific” bits and pieces of an operating system, allowing the OS programmer to be freed from many of the details of the CPU architecture and implementation. The entire code is provided in its source format (C and assembler).

Notes

Software Design Examples

Notes

Application Software

This example will use the most common C program, “Hello World,” and will be run in RAM by downloading to an evaluation board using the IDT/sim PROM monitor. This example also illustrates a range of simple application programs and benchmarks that will work regardless of hardware or operating system and require no more than a ANSI C library.

```
#include <stdio.h>

main (int argc, char **argv)
{
 printf ("hello world!\n");
 return (0);
}
```

Memory Map

A simple stand-alone program will usually have all the memory to itself, except for a small amount at the bottom (and possibly the top) which is reserved for use by the PROM monitor.

In such an environment, the programmer will not have to worry about virtual memory: the program can be linked to run in the cacheable kseg0 address region or, to see the program with a logic analyzer, in the uncacheable kseg1 region. These regions map one-to-one with physical memory.

A typical base address for the program code would be 0x80020000 (i.e. at offset 0x20000 in the KSEG0 region). This leaves 128 Kbytes free for the PROM monitor’s own data and stack, which is enough for IDT/sim. Above this will come the program’s initialized data, then BSS (uninitialized data), followed by its “heap” (free memory for use by malloc *et al*). The PROM monitor will usually put the stack pointer near the top of memory, and the stack and heap will grow towards each other.

Starting UP

Having downloaded the program to the evaluation board and told the PROM monitor to start the program, it will set the stack pointer to the top of memory and jump to the program’s *entrypoint*, often defined by a label with a standard name (e.g. start), or simply by jumping to the first address in the program.

The code following the entrypoint has to ensure that the run-time environment required for a C program and library is set up. For a downloaded program this is usually a simple matter of zeroing the BSS segment, and initializing the *$gp* register and stack. It should then call the program’s main function, after ensuring that its *argc*, *argv* and *envp* arguments are initialized. If main returns, then its return value is passed to the exit function, which will close open files and in turn call exit. The exit function should transfer control back to the PROM monitor (the exact manner this is done is system or tool dependent)[1].

The following code fragment shows how a start-up module might be implemented; it is commonly provided as part of the development system.

```
            .comm   environ,4

            .data
#define     ARGC    1
argv0:      .asciiz "prog"
argvec:     .word   argv0, 0
envvec:     .word   0
```

1. The above functionality is provided by the “idt_csu.S” program provided with IDT/c.

Notes

```
        .text
LEAF(_start)
        /* initialize $gp */
        la      gp,_gp

        /* clear the BSS */
        la      t0,_fbss
        la      t1,end
1:      sw      zero,0(t0)
        addu    t0,4
        bltu    t0,t1,1b

        /* make sure stack is in same KSEG as .data */
        and     t0,sp,0x1fffffff        # get stack physical              #
address
        and     t1,~0x1fffffff          # get KSEG of "end"
        or      sp,t0,t1                # put sp in same KSEG

        /* align to 8 byte boundary and allocate an argsave              area*/
        and     sp,~7
        subu    sp,16

        /* initialize argc, argv, and environ (IDT/sim zeroes            a0-a2)
*/
        li      a0,ARGC                 # dummy argc
        la      a1,argvec               # dummy argv
        la      a2,envvec               # dummy envp
        sw      a2,environ

        /* exit (main (argc, argv, environ)) */
        jal     main
        move    a0,v0
        jal     exit

        /* in case exit returns */
1:      break   1
        b       1b
END(_start)


LEAF(_exit)
        li      ra,0xbfc00000+(17*8)# IDT/sim prom return             #
vector
        j       ra
END(_exit)
```

C Library functions

Many C application programs will expect to have access to a C library which conforms to the ANSI definition, as described in [reference K&R]. Most development systems will supply a library that conforms to at least some parts of this standard. The rest of this section follows Appendix B of [reference K&R] to warn the programmer about those areas where some cross-development system libraries may deviate from the standard – refer to the toolchain documentation for specific information.

Input and Output

The <stdio.h> header file is almost certain to be present, but the library will often provide only a small subset of the expected standard i/o facilities. In particular it will usually provide access to only a single console device via *stdin* and *stdout*, with no file i/o. Some systems may provide remote file access facilities, but this is often via a distinct set of non-standard function calls[1].

[1] The IDT/c toolchain does provide many of these and other referenced functions. The programmer should consult the reference manuals for a particular toolchain to determine which functions are supported.

Notes

- *File operations: are unlikely to be present, and if they are will usually support only the system console device.*
- *Formatted output: the printf functions will usually be present, but may omit some of the newer ANSI formatting options, and may not support floating-point formats.*
- *Formatted input: the scanf functions are often absent.*
- *Character input and output: usually provided, but often only to the system console.*
- *Direct input and output: sometimes provided, but often only to the system console. or serial I/O ports.*
- *File positioning: probably absent.*
- *Error handling: probably absent.*

Character Class Tests

The <ctype.h> header file and its associated functions and/or macros are usually provided. The isxdigit function is sometimes absent or has a different name.

String Functions

The older string functions are usually present, although often not very optimized. Some of the newer string functions such as strspn, strcspn, strpbrk, strstr, strerror and strtok may be absent.

The mem... functions are sometimes absent, and in their place the older bcopy, bcmp and bzero functions may be provided.

Mathematical Functions

The mathematical functions, if provided at all, will often be in a separate math library. If this library is supplied, it will probably implement all of the required functions. Note that it may be impossible, or tricky, to run floating-point code on CPUs which do not have an on-chip FPA. Even if it does have one, the system may still need a trap handler for serious floating-point use. Some compilers such as the IDT/c compiler, can be instructed to implement floating-point operations by making calls to an emulation library (IDT provides this library along with the compiler).

Utility Functions

The strto... functions are sometimes absent, but the older atoi and atol will usually be available. The floating point conversions may be absent. IDT tool-chain provides all of these functions.

The following functions are often absent: rand, srand, atexit, system, getenv, bsearch, qsort, labs, div and ldiv.

The malloc family will probably exist in some form, although realloc is sometimes absent. At the lowest level they will probably call the sbrk function to acquire memory from the system, which the programmer may be required to implement. A simple sbrk will just return consecutive chunks of memory starting from &end (i.e. just after the program's declared data), until it reaches somewhere near the bottom of the stack, as follows:

```
extern char end[];
extern interrno;
static void *curbrk = end;
static void *maxbrk = 0;

#define MAXSTACK (64 * 1024)

void *
sbrk (int n)
{
 void *p;
```

Notes

```
/* calculate limit for curbrk on first call */
if (!maxbrk)
maxbrk = (void *)&n – MAXSTACK; /* &n is approx value of
                                                $sp */

/* check that there is room for this request */
if (curbrk + n > maxbrk) {
        /* no room */
errno = ENOMEM;
return (void *)-1;
}

/* zero the requested region */
memset (curbrk, 0, n);

/* advance curbrk past region and return pointer to it */
p = curbrk;
curbrk += n;
return p;
}
```

Diagnostics

The assert macro is often absent.

Variable Argument Lists

Variable arguments are usually supported, but sometimes only via the old *vararg* mechanism rather than the newer ANSI *stdarg*. IDT tool-chain provides both.

Non-local jumps

The setjmp and longjmp functions are usually supplied.

Signals

It is unlikely that the signal functions will be supported, although sometimes a limited form is provided in order to support SIGINT only.

Date and time

It is likely that none of these functions will be available. Timing benchmarks will often require a stop-watch, or some software mechanism which is very specific to the PROM monitor and/or development system[1].

Running the Program

Having typed in the “hello world” program, the programmer must then compile it, link it, and convert it into a form suitable for downloading to an evaluation board. This process is very dependent on the particular development system, which should provide some sort of automated mechanism: many UNIX-hosted tool-chains provide a set of *makefiles* which control the whole process, via the well-known *make* utility. IDT tool-chain provides this facility even on the DOS platform.

When the compilation has completed successfully, a down-loadable file is created (typically using S-records or other standard format. IDT/sim version 5.1 or later will allow downloading of elf or ecoff files via ethernet, a superfast way of downloading.). Downloading this file will require the use of a terminal emulator (in IDT/sim, use the “load” command on the board, and the “cp” command on the host), or some other special utility, to transmit the file down an RS232 line to the board. More advanced evaluation boards may provide an Ethernet, SCSI or parallel interface in order to download large programs at high speed. Finally, it is then necessary to instruct the PROM monitor to execute the program (*go* command of IDT/sim).

1. IDT provides a seamless platform-independent solution. The function “timer_stop” can provide elapsed time in microseconds since last call to the function “timer_start”, as long as IDT/sim 5.1 or later is installed on the board.

Notes

A complete edit, compile, download and run cycle on a SUN platform using IDT/c might look like this:

On UNIX development system:

C> cd /idt/samples*change to source code directory*
C> vi hello.c*enter/edit the source file*
C> cp MakeBE Makefile*create the makefile from the template*
C> vi Makefile *change "stanford" in template to "hello"*
C> make*compile and link for IDT board ;*

this creates a "hellof.srec" file

On eval board's console:

<IDT> **l -a tty1***srec download via RS232 port #1*

On development system:

C> **cat hellof.srec > /dev/ttyb** *download via ttyb port*

On eval board's console:

<IDT> **go***start the program*

Debugging the Program

Hopefully not too much can go wrong with ''hello world'', but larger application programs may require some debugging before they work.

Most PROM monitors, including IDT/sim, incorporate a command-line driven, machine-level debugger. This will allow the programmer to disassemble the code, examine registers and memory, set breakpoints and single-step through code one machine-instruction at a time.

Source-level debuggers allow the programmer to work in terms of the original source code and data structures instead of MIPS machine instructions. These debuggers run on the host development system – so that they can get at the source files and compiler-generated debugging information. They operate the program on the evaluation board by "remote control", via a serial line or network connection. Many PROM monitors will incorporate a special protocol to support this feature, although some may require that the code for it be downloaded along with the program.

Source-level debuggers may themselves be command-line driven (e.g. MIPS *dbx* and IDT's/GNU's *gdb*), or may offer a multi-window, GUI interface (GNU's 1995 releases). In all cases they are very complex programs, with many different commands and options. The development system's documentation should provide more details of how to use them with a target board.

Embedded System Software

Many aspects of ''embedded software'' are the same as ''application software'', and its early development may be carried out in exactly the same way, on an evaluation board. But ultimately it is likely to be running in EPROM, on custom hardware, and require lower-level access to the processor in order to initialize it, test it, and handle machine traps and interrupts.

Memory Map

Compared to a program which is downloaded into RAM, embedded software will (at least initially) have its code and read-only data in EPROM. The EPROM, and thus the code, should be located at physical address 0x1fc00000, which corresponds to the processor's reset vector of 0xbfc00000. The data area should probably be located near the bottom of RAM (DRAM or SRAM), but just above the area used for the processor's (non-boot) exception vectors: 0x400 should be safe for all existing RISController processors. Device registers should be decoded at high physical addresses, but below 512 Mb. If the hardware engineer suggests putting RAM at anywhere other than zero, or device registers anywhere outside of the bottom 512 Mb, then complain loudly: it will make software much more complicated, and performance may suffer.

Notes

Starting up

After a hardware reset, code will be running in KSEG1 (i.e. uncached), with the caches, TLB (if present), internal registers, and RAM in an undefined state. Its first job is to initialize these resources. A detailed discussion and example of this can be found in earlier chapters of this manual.

For higher performance, code will need to be located in the cacheable KSEG0 region (i.e. at 0x9fc00000), rather than the uncached KSEG1 (0xbfc00000). This has implications for start-up code. Before the caches are initialized, branches and absolute jumps (i.e. **j** and **jal**) are safe, because they do not alter the top four bits of the program counter, but any reference to data, or an attempt take the address of a function for use with **jr** or **jalr** will generate a KSEG0 address before it is valid to do so. The programmer must take care that any such references are explicitly mapped to KSEG1, by logically or-ing in the KSEG1 base address (i.e. 0xa0000000). Once the caches are initialized, switch to running cached by use of an explicit **jr** instruction, as follows:

```
/* switch to running cached, if so linked */
la t0,1f
jr t0
1:
```

Even running cached from EPROM will not give optimal performance, since cache refill cycles from EPROM will be slower than from RAM. A higher performance option is to link the code to run in RAM, and arrange for the start-up code to copy itself and the rest of the software from ROM to RAM. This is also useful when debugging the ROM, since it is not possible to set breakpoints or single-step code which is in ROM. Note, however, that this requires even more careful programming of the start-up code. Even jumps cannot be used until the code has been moved: only pc-relative branches are safe, and the **bal** instruction should be used in place of **jal** (though beware its limited +-128Kb range). Any attempt to access data or take the address of a function must be relocated by explicitly adding in the offset between the code in RAM and its temporary location in EPROM. It is sensible to calculate this offset just once, and keep it in a reserved register, such as *$k1*.

Another complication is initialized data. Initialized data can be declared in assembler or C, e.g.

```
        .data
base:   .word   10
```

or

```
int base = 10;
```

The initialized data is writable, and so must be in RAM. But how does it get there?

Some cross-development toolchains are not very helpful, and require that all data must be either uninitialized, or if initialized then read-only. Other toolchains provide various different mechanisms by which to initialize this data. SDE-MIPS, for example, takes the straight-forward step, when generating a ROM image, of placing a copy of the initialized data segment (i.e. *.data*) at the next 16-byte boundary after the code. It is then easy to copy this from ROM to its final in RAM.

The following code fragment illustrates a flexible mechanism for handling these different possibilities for moving code and data to RAM.

```
_reset_vec:
 b _reset
 ...

_reset:
 move k1,zero           # assume no relocation
 bal 1f                 # ra := current pc
1: la t0,1b             # t8 := linked pc
 beq t0,ra,2f           # when they match, then no reloc is              #
correct

 /* executing at other than the linked address */
 li k1,0xbfc00000       # k1 := actual EPROM base
```

Notes

```
    la t0,_reset_vec        # t8 := linked EPROM base (may be RAM)
    subu k1,t0              # k1 := reloc factor (actual – linked)
2:

    /* initialize CPU, RAM, caches, tlb & stack
            (hardware specific) */
    ...

    /* skip code move if it is linked for ROM */
    and t0,k1,~0x20000000 # ignore simple KSEG1->KSEG0 reloc
    beqz t0,3f

    /* copy code to linked address in RAM */
    la a0,_ftext            # a0 := destination (RAM) address
    addu a1,a0,k1           # a1 := source (ROM) address
    la a2,etext             # a2 := code size (etext – _ftext)
    subu a2,a0
    bal memcpy

3:
    /* copy initialized data to RAM (SDE-MIPS specific) */
    la a0,_fdata            # a0 := destination (RAM) address
    la a1,etext             # a1 := source address (after ROM code)
    addu a1,k1
    addu a1,15              # round address up to 16-byte boundary
    and a1,~15
    la a2,edata             # a2 := data size (edata – _fdata)
    subu a2,a0
    bal memcpy

    /* jump to C start-up at linked address */
    la t0,_start
    j t0
```

Embedded System Library Functions

Embedded system software written in C or C++ will still need access to the MIPS Coprocessor 0 registers and instructions in order to control interrupts, catch exceptions, handle the caches and TLB, and so on. Some cross-compiler vendors will supply a toolkit of low-level library routines to do this, and sometimes it will include full source code. At a minimum such a kit should include assembler functions which read and write each CPU control register, initialize and update the TLB (if present), and initialize and invalidate all or part of the caches. Unfortunately there are no standard interfaces for these functions, and the programmer will have to read the cross-development system's documentation. The examples in this manual could serve as a baseline reference for programmers which choose to generate these functions themselves.[1]

Trap and Interrupt Handling

Beyond routines to manipulate the CPU control registers and caches, the system software may need a mechanism by which to catch machine exceptions (the generic name for traps and interrupts), and cause appropriate C handler functions to be called. Vendor-supplied embedded system toolkits probably contain some code to help with this, although this is often at the very low level, and require more work to interface to C-level functions. SDE–MIPS includes some relatively high-level exception handling code that allows the programmer to route different exceptions to different C functions, and pass them a pointer to a structure which contains the complete CPU context at the time of the exception.

Choices about stacks

1. Alternately, the programmer could obtain IDT/sim and/or IDT/kit from IDT, or a similar product from other 3rd party tools vendors.

Notes

An exception handler has several choices regarding its use of stacks:

1. Remain on the current stack, shared with the main, or current application. This is usually adequate for simple, single-threaded applications.
2. Have an exception stack, which it switches to upon receiving an exception when not already at exception level. This avoids overrunning an application's stack, if it is small, and avoids problems if the exception was caused by a bad value stack pointer value.
3. Have several exception stacks, one per ''process''. This is essential in multi-processing applications.

Simple Interrupt Routines

If any of the CPU's six interrupt pins or two software interrupt bits are active, and not masked by the CPU's *Status* register, the CPU takes an immediate Interrupt exception. Once the generic exception handler has routed the exception to the specific Interrupt exception function, it is the this function's responsibility to sort the interrupts into priority order and dispatch to the correct device's interrupt handler. The simplest technique is to make interrupt priorities correspond directly to interrupt pin number, allowing a simple bit-scan of the *Cause* register.

A very simple, fixed-priority interrupt handler might look something like this:

```
extern void softclock(), softnet();
extern void diskintr();
extern void netintr();
extern void ttyintr();
extern void fpuintr();
extern void clkintr();
extern void dbgintr();

/* interrupt handler table */
void (*intrhand())[8] = {
 softclock,                    /* [0] SInt0: clock */
 softnet,                      /* [1] SInt1: network */
 diskintr,                     /* [2] Intr0: disk controller */
 netintr,                      /* [3] Intr1: network interface */
 ttyintr,                      /* [4] Intr2: uart */
 fpuintr,                      /* [5] Intr3: fpu interrupt */
 clkintr,                      /* [6] Intr4: timer interrupt */
 dbgintr                       /* [7] Intr5: bus errors, debug button, etc. */
};
```

```
/*
 * Interrupt exception handler.
 * 1) The xcp argument points to a structure which maps to
 *    the stack frame in which the CPU context (i.e. all
 *    registers) are saved.
 * 2) On entry all interrupts are masked (disabled).
 * 3) It calls the mips_setsr() function to modify the CPU
 *    Status register.
 */

interrupt (struct xcption *xcp)
{
        unsigned int pend, intrno;

        /* find all pending, unmasked interrupts */
        pend = xcp->cause & xcp->status & SR_IMASK;

        /* dispatch each pending interrupt, starting with
         * highest */
        for (intrno = 7; (pend & SR_IMASK) != 0;
```

Notes

```
                                                pend <<= 1, intrno--) {
                        if (pend & SR_IBIT7) {
                                /* enable only interrupts of higher priority
                                 * than this one. */
                                unsigned int imask = SR_IMASK <<(intrno + 1)
                                mips_setsr (imask | SR_IEC);

                                /* call interrupt handler */
                                *intrhand[intrno]) (xcp);
                        }
                }

                /* disable all interrupts */
                 mips_setsr (0);
        }
```

Floating-point Traps and Interrupts

The previous section shows how to recognize a floating point interrupt. Following the interrupt the *EPC* will either point at the FP instruction or (if the FP instruction is in a branch delay slot) at the immediately preceding branch.

To find out what happened, look first at the CPU *Cause* register. If it shows a "co-processor unusable" condition, then the FPA instruction set is not enabled. In the RC30xx, if it shows an interrupt at the FPA's level, the handler can get further details of exactly what has gone wrong by consulting the floating point status register. In the RC4xxx, floating point exceptions are handled by the normal exception handler which will have to do the things described below once it has figured out that it is dealing with a FP exception. However, there are only three cases of interest:

- *The FPA is disabled (CU1 == 0 in the CPU status register). If the CPU does not have an FPA, the software might want to emulate the instruction. If the FPA is available, the system might have been doing an "enable-on-demand"[1]. If so enable it and return to retry the instruction.*
- *The chip includes an FPA, and it's enabled, and the FCR31 UnImp bit is set. The FPA has interrupted because it can't perform this particular operation, with these particular operands. The normal approach is to emulate the instruction – though in this case software will want to put the result back in the real FP registers.*

  In theory there are a rather restricted range of operations and operands which cause this condition: underflows, operations which should produce the "illegal" NaN value, denormalised operands, NaN operands, and infinite operands.

  The system could put in special case code to handle just these conditions. But it is very hard to get assurances about exactly when the FPA may refuse an operation.
- *The system has an enabled FPA, and the FP status register UnImp bit is clear. It looks as if the FPA operation has produced an IEEE-exception. Software may need to report this to the application, in some OS-dependent manner.*

Emulating Floating Point Instructions

- *Locate the instruction: it will either be at EPC (when the CPU status register, SR bit BD, is clear); or when BD is set, indicating that the exception happened in a branch delay slot, the FP instruction will be at EPC+4.*
- *Decode the instruction: The encoding of FP arithmetic instructions is very regular.*
- *Fetch the operands: the instruction encoding tells which FP registers hold the operand(s).*
- *Call the emulator: to perform the actual operation.*
- *Check for exceptions: if there are any enabled IEEE exceptions. If the system architect knows that IEEE exceptions can't usefully happen (perhaps because there is no mechanism in place for applications to catch them), skip this stage.*
- *Patch the result: back into the appropriate FP destination register.*

[1] Some systems do this to avoid saving FP registers at context switch if the application is not using the FPA.

Notes

- *Hop over the emulated instruction: if BD was clear, just restart at EPC+4.*

  But if BD was set the program is going to have to decode and emulate the branch instruction (at EPC) too, and restart at the branch target location.

Debugging

Once the developer leaves behind the relative safety of a PROM monitor and its debug support to develop the system PROM, finding out why the code is not working correctly may become much more tedious.

At the worst, the programmer will have to use judicious calls to printf, link the program in KSEG1 (i.e. uncached) and monitor CPU addresses with a logic analyzer. Armed with a list of function addresses (e.g. the output of the *nm* utility), and possibly a detailed disassembler listing for the suspect function, it is often possible to deduce the bug. It is seldom necessary to capture data values, although a few bits or a byte can be useful if the analyzer has enough probes.

Many vendors offer RC30xx/RC4xxx disassemblers and special pods for an analyzer to trace both instruction and data accesses.

Another technique is to include support for remote source-level debugging in the new PROM. The use of a ROM emulator device may prove helpful. This would allow the debugger to place “breakpoints” into the ROM code.

UNIX-Like System S/W

It is obviously impossible, in a few pages, to give a comprehensive description of a big operating system. This section will provide some background on what a portable big system does, and how it does it on MIPS – so if the system needs to implement some fragment of this functionality the programmer won’t be starting entirely from scratch. In specific examples shown, the description below relates to the freely redistributable “NetBSD” system, part of the Berkeley family.

The description is arranged as follows:

- *Terminology: key words, often used with very particular meanings in Unix-like systems:*
- *Components of a process:*
- *Protection: how the kernel protects itself and other processes from misbehaving software;*
- *Kernel services:*
- *Virtual memory: how the MIPS architecture is used to build VM.*
- *Interrupts: how the CPU’s features relate to the needs of the OS.*

Terminology

- *Task: a thread of control, identified by a program counter and a stack. In other contexts this may be called a “process” or “thread”.*
- *Address space: the program memory context seen by an application. For MIPS this is a single, simple 32-bit space, divided into two. The lower 2Gbytes is accessible in user mode, but the upper 2Gbytes is not usable except in kernel mode. Note that the address mapping doesn’t change with CPU mode. There are no segments, no separate I- and D-space.*

  This MIPS model fits very well onto the BSD family of Unix-like systems, and was probably conceived with BSD’s requirements in mind.
- *Program: a bunch of code and data initialization, held on disc and loaded when required.*

  A “process” combines all these three: it is a task in an address space running a program.
- *File: a named sequence of bytes coming from “outside”. At its simplest it is just data which can be written out to disc and subsequently read back.*
- *Device: abstract, fairly unified interface to diverse real-world peripherals. Devices are named like files, and offer the same basic byte-stream model. Beneath this interface the kernel buffers data, handles interrupts and hardware details, and also provides an escape mechanism to keep device-*

Notes

specific functions tidy.

"Device drivers" are the lowest layer of kernel software which deals with hardware, and are supposed to isolate dependencies on particular controllers/peripherals.

Network interfaces are handled differently, and networking code is way beyond the scope of this chapter.

- *Page fault: the OS maintains a mapping of program (virtual) addresses to physical addresses. But it doesn't have to keep all the process pages in memory. Access to a page for which no translation is defined causes a trap (a page fault which invokes a piece of software which checks that the address is legitimate, and if so brings the page into memory. When a page is touched for the first time, it will either be loaded from disc (if it is program text or initialized data) or supplied set to zero (if it is uninitialized data or stack).*

Components of a Process

The above description refers to a BSD "process" as a task, address space and program all at the same time. This is a restriction, but it does keep things simpler.

"Processes" are laid out in memory as shown in Figure 15.1, Memory layout of a BSD process.

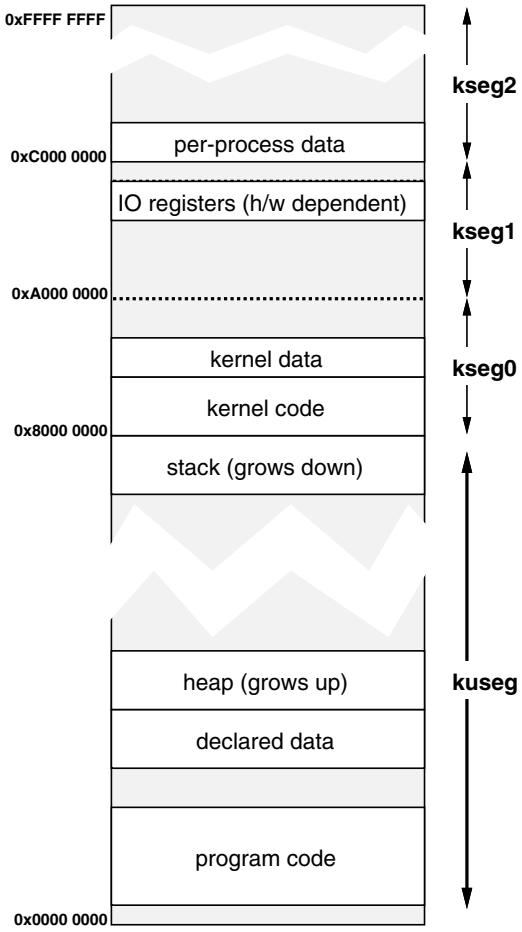


Figure 15.1 Memory layout of a BSD process

- *Program text: every process has a program in memory which it can run (it may be "virtual memory", but to the process it seems to be there).*
- *Stack: every process has a stack, which grows downwards from the top of the user-accessible space. Since the MIPS architecture has no architecture-specified stack pointer, the OS is always*

Notes

willing to allocate pages of memory in the stack region if ever the program gets a page fault.

- *Declared data: the data declared in a C program is noted in the object file, and explicitly accessed by compiled-in code. Initialized data is paged from the program file as needed, uninitialized data is supplied as zero-filled pages.*
- *Heap: this is the traditional name for data space allocated during program run-time. At the top of the data section the kernel maintains a boundary address (the break); on a page fault addresses above this are rejected as invalid. To allocate extra data the process can invoke the sbrk() system call; this is usually done implicitly when calling a free-space manager function such as malloc().*
- *Kernel data structures: when a process in BSD makes a system call the process continues execution, but in kernel mode. Some kernel activity (such as interrupts) doesn't run on a particular process context, but most does.*

  So important parts of the process address space are inside the kernel, and are not accessible while the process is running in user mode. In particular, the process in kernel mode gets access to the whole kernel code and data (mapped into kseg0) and to all IO registers (mapped in kseg1).

  It is a boon that, while a process is running in the kernel, all its user-mode data is accessible at exactly the same addresses as in user mode. Some architectures have to implement a special-case "copy user data to/from kernel space" instruction.
- *proc structure: lurking in the kernel data area are the two key data structures which define the process. Why two? The smaller of these is the proc structure and contains information which may be required even when the process is not itself executing, and;*
- *per-process data area (u area): this is the larger process structure, and is accessible only when the process is active. By a special trick of the MMU, the per-process data area is mapped to a constant virtual address inside the kernel, in the kseg2 region.*
- *kernel stack: attached to the per-process area, mapped into kseg2, is the stack used by the process when executing in the kernel. It is this stack which is "borrowed" by interrupts.*

System Calls and Protection

One of the goals of BSD is protection for robustness; to ensure that a user-level program which goes wrong cannot disrupt the rest of the system. This is basically achieved by the process address space:

- *In user mode, the process can only get at its user-mode virtual addresses, which are only those pages allocated by the kernel.*
- *To get into kernel mode, the process has to drop through a system call trap and can then perform only the function the system call allows. It is the duty of the system call itself to check its arguments for sanity, and to make sure that it behaves properly.*

  Interrupts and inadvertent traps behave much like system calls, albeit ones which don't work on behalf of the user process.

Of course, since the process has the whole kernel mapped it can at any time attempt a reference to kernel code or data; but in user mode this will be immediately trapped, and find its way to a memory reference error handler – which by default will kill the process.

RC30xx security features are pretty much the minimum that will support a BSD-style OS. Many architectures offer much more; but portable OS', since they want to be portable, use only the lowest common denominator of security functions – and since all significant microprocessor OS' are now portable, the extra functionality is wasted.

What the kernel does

In the BSD system the kernel is the essential common ground between processes, and must share out access to any resource for which processes compete (CPU time, memory, disc bandwidth etc.). It must also provide basic mechanisms so that processes which want to co-operate can communicate with each other.

Notes

BSD and other Unix-like systems are traditionally rather kernel-heavy; more modern OS' try to provide only minimum functions in the kernel (which is then often called a *microkernel*), handing over other jobs to distinct "server" processes.

- *File system: the kernel provides access to the file system, which is based on open/read/write/close functions. In practice this splits into two; resolving names and then implementing file I/O.*

  There will usually be multiple file system implementations (but each offering the same service); a file I/O system call will be redirected to the correct code according to whether the file is local, on NFS, on a DOS floppy disc, etc.

- *Scheduling: BSD decides which process to run. Most of the time, processes will run until they need some input – and then they'll make a system call to get the input and block until the input is ready.*

  But sometimes a process needs to compute for longer; in this case it will be time-sliced; it will be allowed to run only for a second or so and then another process will be given a go.

  To prevent a compute-bound process from clogging up the CPU, processes are given priorities, and any process which uses up its time slice has its priority reduced. A priority-based scheduling decision is made often – potentially, after any interrupt.

- *Paging: the kernel shares memory by picking pages of memory which don't appear to have been used for a while, and throwing them out. A data page which has been written by a process since it came in must first be saved to a disc swap file.*

  The MIPS architecture gives no direct help in tracing what happens to pages; in many architectures the MMU hardware notes (separately) whenever a page is either referenced or written. In MIPS this must be simulated; so the kernel picks pages and marks them as (from the point of view of the hardware) "read only" or "invalid". Then it waits; if a process references or writes the page a trap will be generated, and the trap handler will look at the page status and set a software referenced/written bit.

  In this way processes which are not active slowly migrate out of memory.

- *Caching and sharing code: it often happens, particularly in a multi-user system, that there are multiple processes all running the same program. NetBSD treats code pages (i.e. read-only pages marked as loadable from a file) as sharable; when they are kept in memory they are indexed by their disc location. During periods of relatively light load (which is most of the time in most systems) much of memory has nothing very useful in it; so code pages are allowed to stay there, forming a least-recently used cache.*

  This means that a program which is repeatedly re-run to completion goes much faster. Although each time a process must be created and the whole program nominally "paged in", in practice all that is needed is to construct a set of entries referencing the already memory-resident code.

Virtual Memory Implementation for MIPS

The RC30xx and RC4xxx hardware supports an arbitrary (though small) set of translations in their TLBs. When an address is encountered which doesn't match with one of these, the CPU takes an exception (a *tlbmiss*) and software must find a new translation and load it.

"tlbmiss" events can occur very frequently when running large programs, and the trap handler must run quickly. Misses for user-mode addresses in the RC30xx are vectored through a dedicated trap vector, to the *utlbmiss* routine; since MIPS kernels can be built to run largely in the kseg0/kseg1 areas (which don't require the TLB) the vast majority of TLB misses are user ones.

To speed the trap handler, most systems will keep memory-resident tables of page entries, in a format already bit-for-bit compatible with the hardware-determined TLB entries.

It would be nice to do this by keeping a simple array of TLB entries, indexed by virtual address. However, with a 2Gbyte range of user addresses and 4Kbyte pages, the array would require 512K entries, occupying 2Mbytes of memory. Since the program address space has huge "holes" in it, most of this 2Mbytes of memory would be full of nothing – which is a lot of memory to dedicate,

Notes

Two different solutions to this problem are used. MIPS Corp's UMIPS and RISC/os variants use a linear page table but don't keep it all in memory; NetBSD uses a memory-held secondary cache of page table entries supporting a machine-independent data structure:

- *Linear Page table not all in memory: the linear page table is located in the virtual space kseg2. Although the whole page table is very large, most of it is never referenced, never allocated a kseg2 translation, and therefore costs nothing. The active parts of the page table correspond with the stack, data and code parts of the process address space; and for these the kseg2 translation is likely to remain live.*

  The CPU's Context register is explicitly designed to do the work of computing where the desired page table entry lies, saving a few more instructions.

  This does require that the utlbmiss handler can safely suffer a regular trap, to cope with those occasions where the page table read falls on a kseg2 address which is not currently translated by the TLB. This nested exception is not allowed to happen in any other circumstances; but its use here motivates another feature of the MIPS hardware, and a convention:
  - *The status register's internal stack of processor state (2 bits for kernel/user mode and interrupt on/off) is three deep; allowing an exception to occur in an exception handler, before the status register gets saved.*
  - *The "nested" exception overwrites the EPC value (return address) from the original address reference, so the utlbmiss handler saves EPC into the general-purpose register k1; the regular trap handler which deals with kernel TLB misses has to detect the double-exception and return to the right place.*
    This is why there are two registers (k0,k1) reserved for exception handling: most of the time only one is needed.
- *Secondary cache of page table entries: NetBSD uses a different technique. Here the TLB miss handler consults a software cache of recently used page table entries. The software cache is implemented with a simple 2-set hashing function, with a fast path for translations which are in the same set as their predecessor. A modestly large cache gives an excellent hit rate – so those few translations which miss here can be computed by a C-language routine using architecture-independent tables.*

Interrupt Handling for MIPS

Interrupt handling in Unix-like OS' are descended from the priority-based system implemented in hardware by DEC's PDP-11 and VAX architectures. Priorities are numbered from 0 to 7 (though not all are always used) – more recently, the numeric priorities have been getting names.

- *Priority model and spl: kernel code is arranged so that, in general, each piece of code is accessible only at or above a particular priority level. So, for example, once a program is at level 4 the CPU will only accept interrupt requests prioritized at level 5 and above.*

  Most of the kernel code used by system calls runs at level 0.

  Device code which needs to lock itself against asynchronously-occurring interrupt events can call a function such as spl4() (spl stands for "set processor level"): there is a separate call for each level. spl4() returns a value representing the priority level when it was called, so the code sequence:

  ```
  p = spl4();
  /* do something which can't be interrupted */
  splx(p);
  ```

  restores whatever is required to lower the level again.

  Note that interrupt handlers can get called at two points: either as soon as the interrupt signal is activated, or (if the processor is currently at a higher spl) the handler will be called when a call to splx() lowers the level below the interrupt's priority.

How it works

The MIPS interrupt hardware knows nothing of levels, with only an unprioritized mask for the interrupt inputs. But if an spl level can be assigned to each of the interrupt inputs, then each of the *spl..*() routines can be implemented by setting the interrupt mask to a value enabling only those interrupts allocated a higher level.

Assembly Language Programming Tips

Notes

The MIPS ISA is designed for high-frequency, single-cycle instruction operation. Also, as noted earlier, the MIPS architecture does not carry a status register nor does it directly support various addressing formats. As a result, some operations that may have been found in older CISC architectures must by synthesized from multiple instructions in the MIPS architecture.

This chapter describes common programming problems and their implementation in the MIPS architecture. Many of these operations are directly supported by the synthetic instructions, described earlier.

Also note that many of these instructions require the use of $at (the assembler temporary register) described earlier.

32-bit Address or Constant Values

As noted earlier in this manual, the MIPS instruction set does not have enough room in the bit encoding to directly support 32-bit constants or constant address values. Thus, programmers must use combinations of instructions to generate 32-bit values.

Again, these are commonly handled using the synthetic **la** or **li** instructions. Depending on the immediate value, the assembler will generate one or two instructions to implement the immediate load into the register:

Operand	Instruction Sequence
Upper 16 bits all zero	ori rd, $value_{15..0}$
Upper 17 bits all one	addi rd, $0, $value_{15..0}$
Lower 16 bits all zero	lui rd, $value_{31..16}$
All other values	lui rd, $value_{15..0}$ ori rd, $value_{31..16}$

To jump to an absolute 32-bit address, a similar construct must be used. The **la** synthetic instruction is used to load the target address into a register; a **jr** (jump register) is then used to perform the jump.

Note that **j** and **jal** may be used in many instances. However, these instructions take the high-order four bits of the current "PC" as the upper four bits of the target address, and thus limit the program space that can be reached. In practice, this limit may be larger than the address space of most typical embedded applications.

Use of "Set" Instructions

The MIPS ISA provides a very powerful operation to enable the easy synthesis of complex test operations.

The "set" instructions place a value of '1' (true) or '0' (false) into the specified destination register to reflect the outcome of a specified comparison operation. When used with conditional branch operations, complex comparison sequences can be implemented, as well add-with-carry or subtract-with-borrow operation.

Use of "Set" with Complex Branch Operations

The MIPS instruction set directly implements branch comparisons for the following cases:

- *two registers equal*
- *two registers not equal*

Notes

- *register greater-than-or-equal to zero*
- *register less-than-or-equal to zero*
- *register greater-than zero*
- *register less-than zero*

These branch comparisons directly implement a wide range of common test conditions directly in hardware. However, in certain situations the programmer may require a more complicated test between two non-zero registers. This is where the "set" instructions are used.

For example, if the programmer wishes to branch conditionally if one register is less than another, a two instruction sequence is used:

```
slt     $at, $a, $b
bne     $at, $0, target     # branch to target if a < b
```

Using analogous instruction sequences, the programmer can synthesize virtually any comparison between two registers using the various set instructions.

Similarly, comparisons with immediate values can be implemented. For example, to compare whether a register value is less-than-or-equal-to an immediate:

```
slti    $at, $a, imm+1
bne     $at, $0, target     # branch to target of a <= imm
```

Of course, if the immediate value is large, then the programmer must first build it into a register as described earlier in this chapter, and then perform the comparison.

Many of these common operations are already built into the synthetic instruction set supported by a given toolchain assembler package. The programmer is advised to consult the reference manual.

Carry, Borrow, Overflow, and Multi-precision Math

The MIPS ISA does not directly support a carry bit. Instead, the effects of a carry bit can be synthesized when needed using the "set" constructs. This enables the programmer to implement tests for overflow, multi-precision math, and add-with-carry operations.

For example, these constructs enable the programmer to perform tests to determine whether an arithmetic operation resulted in a carry (or borrow).

For add sequences, there are two cases to consider:

Case	Instruction Sequence
No possible carry from previous operation	addu temp, A, B sltu carryout, temp, B # carryout from A + B
Carry-in from previous operation	not temp, A sltu carryout, B, temp xor carryout, 1 # carry-out from A+B+1

Subtract with borrow works analogously:

Case	Instruction Sequence
No borrow-in	sltu borrow, B, A #borrow-out from A-B
Borrow-in from previous	sltu borrow, B, A xor borrow, 1 #borrow out from A-B-1

Testing for overflow also uses the set instructions, coupled with two basic rules:

- *An addition operation has overflowed if:*
  - *the sign of both operands is the same*
  - *the sign of the result differs from the sign of the operands*

Notes

- *A subtraction has overflowed if*
  - *the signs of the two operands are different*
  - *the sign of the result is different from the sign of the minuend*

Testing for these conditions is a simple programming exercise. For example, testing for overflow in signed addition:

```
/* branch to Label if t1+t2 overflows                    */
        addu    t0, t1, t2      /* result in t0*/
        xor     t3, t1, t2      /* check signs of operands*/
        bltz    t3, 1f          /* then no overflow*/

        xor     t3, t0, t1      /* check sign of result */
        bltz    t3, Label       /* overflow...*/

1f:     /* no overflow */
```

RC4xxx Features

The RC4xxx family with its support for MIPS-II and MIPS-III ISA offers tremendous power and flexibility which may be some times overlooked if the programmer is from the RC30xx world and is migrating to the RC4xxx. Newer instructions ought to be used whenever intelligent design and performance are the goals. These instructions include square-root, multiply accumulate(RC4650), three operand multiply (RC4650), trap-on various conditions, direct *cache* access, “double-word” versions of *add/multiply/divide* and virtually all types of *load* and *store* including those pertaining to co-processors and unaligned accesses, *branch-likely*, various methods of arithmetic and logical *shifts* in registers, and the power saving *wait*.

RC5xxx features

Similarly, the RC5000, with its support for the MIPS IV ISA offers even grater power and flexibility. Some of the newer instructions include the floating point multiply-add, multiply-subtract, reciprocal, reciprocal square root, and “conditional move” operations, which minimize the number of branches in programs.

Notes

Assembly Language Syntax

Notes

This chapter describes the assembler syntax valid for most RC30xx assemblers.

The *compiler-dir* directives in the syntax are for use by compilers only, and they are not described in this book.

statement-list:
statement
statement statement-list

statement:
stat \n
stat ;

stat:
label
label instruction
label data
instruction
data
symdef
directive

label:
identifier :
decimal :

identifier:
[A-Za-z.$_][A-Za-z0-9.$_]

instruction:
opcode
opcode operand
opcode operand , operand
opcode operand , operand , operand

opcode:
add
sub
etc.

operand:
register
(register)
addr-immed (register)
addr-immed
float-register
float-const

register:
$decimal

float-register:
$fdecimal

addr-immed:
label-expr
label-expr + expr

Notes

label-expr **-** *expr*
expr

label-expr:
label-ref
label-ref **-** *label-ref*

label-ref:
numeric-ref
identifier
.

numeric-ref:
decimalf
decimalb

data:
data-mode data-list
.ascii *string*
.asciiz *string*
.string *string*
.space *size* **,** *fill*

data-mode:
.byte
.half
.hword
.word
.int
.long
.short
.float
.single
.double
.quad
.octa

data-list:
data-expr
data-list **,** *data-expr*

data-expr:
expr
float-const
expr **:** *repeat*
float-const **:** *repeat*

repeat:
expr

symdef:
constant-id **=** *expr*

constant-id:
identifier

directive:
set-dir
segment-dir
align-dir
symbol-dir
block-dir
compiler-dir

set-dir:
.set [no]volatile

Notes

.set [no]reorder
.set [no]at
.set [no]macro
.set [no]bopt
.set [no]move
.set mips*n*

segment-dir:
.text
.data
.rdata (E(OFF))
.rodata (ELF)
.sdata

listing-dir:
.eject
.list
.nolist
.psize *lines , columns*
.subttl
.title

align-dir:
.align *expr*

symbol-dir:
.globl *identifier*
.extern *identifier , constant*
.comm *identifier , constant*
.lcomm *identifier , constant*

block-dir:
.ent *identifier*
.ent *identifier , constant*
.aent *identifier , constant*
.mask *expr , expr*
.fmask *expr , expr*
.frame *register , expr , register*
.end *identifier*
.end

compiler-dir:
.alias *register , register*
.bgnb *expr*
.endb *expr*
.file *constant string*
.galive
.gjaldef
.gjrlive
.lab *identifier*
.livereg *expr , expr*
.noalias *register , register*
.option flag
.verstamp *constant constant*
.vreg *expr , expr*

expr:
expr binary-op expr
term

term:
unary-operator term
primary

primary:
constant

Notes

(expr)

binary-op: one of

*** / %**
+ –
<< >>
&
^
|

unary-operator: one of

+ – ~ !

constant:

decimal
hexadecimal
octal
character-const
constant-id

decimal:

[1-9][0-9]+

hexadecimal:

0x*[0-9a-fA-F]+*
0X*[0-9a-fA-F]+*

octal:

0*[0-7]+*

character-const:

'x'

string:

"xxxx"

float-const: for example

1.23 .23 0.23 1. 1.0 1.2e10 1.2e-15

Index

X