Chapter 4

The MIPS R2000 Instruction Set

by Daniel J. Ellard

4.1 A Brief History of RISC

In the beginning of the history of computer programming, there were no high-level languages. All programming was initially done in the native machine language and later the native assembly language of whatever machine was being used.

Unfortunately, assembly language is almost completely nonportable from one architecture to another, so every time a new and better architecture was developed, every program anyone wanted to run on it had to be rewritten almost from scratch. Because of this, computer architects tried hard to design systems that were backward-compatible with their previous systems, so that the new and improved models could run the same programs as the previous models. For example, the current generation of PC-clones are compatible with their 1982 ancestors, and current IBM 390-series machines will run the same software as the legendary IBM mainframes of the 1960's.

To make matters worse, programming in assembly language is time-consuming and difficult. Early software engineering studies indicated that programmers wrote about as many lines of code per year *no matter what language they used*. Therefore, a programmer who used a high-level language, in which a single line of code was equivalent to five lines of assembly language code, could be about five times more productive than a programmer working in assembly language. It's not surprising, therefore, that a great deal of energy has been devoted to developing high-level languages where a single statement might represent dozens of lines of assembly language, and will run

without modification on many different computers.

By the mid-1980s, the following trends had become apparent:

- Few people were doing assembly language programming any longer if they could possibly avoid it.
- Compilers for high-level languages only used a fraction of the instructions available in the assembly languages of the more complex architectures.
- Computer architects were discovering new ways to make computers faster, using techniques that would be difficult to implement in existing architectures.

At various times, experimental computer architectures that took advantage of these trends were developed. The lessons learned from these architectures eventually evolved into the *RISC* (Reduced Instruction Set Computer) philosophy.

The exact definition of RISC is difficult to state[1], but the basic characteristic of a RISC architecture, from the point of view of an assembly language programmer, is that the instruction set is relatively small and simple compared to the instruction sets of more traditional architectures (now often referred to as *CISC*, or Complex Instruction Set Computers).

The MIPS architecture is one example of a RISC architecture, but there are many others.

4.2 MIPS Instruction Set Overview

In this and the following sections we will give details of the MIPS architecture and SPIM environment sufficient for many purposes. Readers who want even more detail should consult *SPIM S20: A MIPS R2000 Simulator* by James Larus, *Appendix A, Computer Organization and Design* by David Patterson and John Hennessy (this appendix is an expansion of the SPIM S20 document by James Larus), or *MIPS R2000 RISC Architecture* by Gerry Kane.

The MIPS architecture is a register architecture. All arithmetic and logical operations involve only registers (or constants that are stored as part of the instructions). The MIPS architecture also includes several simple instructions for loading data from memory into registers and storing data from registers in memory; for this reason, the

[1]It seems to be an axiom of Computer Science that for every known definition of RISC, there exists someone who strongly disagrees with it.

MIPS architecture is called a *load/store* architecture. In a load/store (or *load and store*) architecture, the only instructions that can access memory are the *load* and *store* instructions– all other instructions access only registers.

4.3 The MIPS Register Set

The MIPS R2000 CPU has 32 registers. 31 of these are general-purpose registers that can be used in any of the instructions. The last one, denoted register `zero`, is defined to contain the number zero at all times.

Even though any of the registers can theoretically be used for any purpose, MIPS programmers have agreed upon a set of guidelines that specify how each of the registers should be used. Programmers (and compilers) know that as long as they follow these guidelines, their code will work properly with other MIPS code.

Symbolic Name	**Number**	**Usage**
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers $1 \cdots 4$.
t0 - t9	8 - 15, 24 - 25	Temporary Registers $0 \cdots 9$.
s0 - s7	16 - 23	Saved Registers $0 \cdots 7$.
k0 - k1	26 - 27	Kernel Registers $0 \cdots 1$.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

4.4 The MIPS Instruction Set

This section briefly describes the MIPS assembly language instruction set.

In the description of the instructions, the following notation is used:

- If an instruction description begins with an $\circ$, then the instruction is not a member of the native MIPS instruction set, but is available as a *pseudoinstruction*. The assembler translates pseudoinstructions into one or more native instructions (see section 4.7 and exercise 4.8.1 for more information).

- If the op contains a `(u)`, then this instruction can either use signed or unsigned arithmetic, depending on whether or not a `u` is appended to the name of the instruction. For example, if the op is given as `add(u)`, then this instruction can either be `add` (add signed) or `addu` (add unsigned).
- *des* must always be a register.
- *src1* must always be a register.
- *reg2* must always be a register.
- *src2* may be either a register or a 32-bit integer.
- *addr* must be an address. See section 4.4.4 for a description of valid addresses.

4.4.1 Arithmetic Instructions

	Op	Operands	Description
○	`abs`	*des, src1*	*des* gets the absolute value of *src1*.
	`add(u)`	*des, src1, src2*	*des* gets *src1* + *src2*.
	`and`	*des, src1, src2*	*des* gets the bitwise and of *src1* and *src2*.
	`div(u)`	*src1, reg2*	Divide *src1* by *reg2*, leaving the quotient in register `lo` and the remainder in register `hi`.
○	`div(u)`	*des, src1, src2*	*des* gets *src1* / *src2*.
○	`mul`	*des, src1, src2*	*des* gets *src1* × *src2*.
○	`mulo`	*des, src1, src2*	*des* gets *src1* × *src2*, with overflow.
	`mult(u)`	*src1, reg2*	Multiply *src1* and *reg2*, leaving the low-order word in register `lo` and the high-order word in register `hi`.
○	`neg(u)`	*des, src1*	*des* gets the negative of *src1*.
	`nor`	*des, src1, src2*	*des* gets the bitwise logical **nor** of *src1* and *src2*.
○	`not`	*des, src1*	*des* gets the bitwise logical negation of *src1*.
	`or`	*des, src1, src2*	*des* gets the bitwise logical **or** of *src1* and *src2*.
○	`rem(u)`	*des, src1, src2*	*des* gets the remainder of dividing *src1* by *src2*.
○	`rol`	*des, src1, src2*	*des* gets the result of rotating left the contents of *src1* by *src2* bits.
○	`ror`	*des, src1, src2*	*des* gets the result of rotating right the contents of *src1* by *src2* bits.
	`sll`	*des, src1, src2*	*des* gets *src1* shifted left by *src2* bits.
	`sra`	*des, src1, src2*	Right shift arithmetic.
	`srl`	*des, src1, src2*	Right shift logical.
	`sub(u)`	*des, src1, src2*	*des* gets *src1* - *src2*.
	`xor`	*des, src1, src2*	*des* gets the bitwise exclusive **or** of *src1* and *src2*.

4.4.2 Comparison Instructions

	Op	Operands	Description
○	`seq`	*des, src1, src2*	*des* ← 1 if $src1 = src2$, 0 otherwise.
○	`sne`	*des, src1, src2*	*des* ← 1 if $src1 \neq src2$, 0 otherwise.
○	`sge(u)`	*des, src1, src2*	*des* ← 1 if $src1 \geq src2$, 0 otherwise.
○	`sgt(u)`	*des, src1, src2*	*des* ← 1 if $src1 > src2$, 0 otherwise.
○	`sle(u)`	*des, src1, src2*	*des* ← 1 if $src1 \leq src2$, 0 otherwise.
	`stl(u)`	*des, src1, src2*	*des* ← 1 if $src1 < src2$, 0 otherwise.

4.4.3 Branch and Jump Instructions

4.4.3.1 Branch

	Op	Operands	Description
	`b`	*lab*	Unconditional branch to *lab*.
	`beq`	*src1, src2, lab*	Branch to *lab* if $src1 \equiv src2$.
	`bne`	*src1, src2, lab*	Branch to *lab* if $src1 \neq src2$.
○	`bge(u)`	*src1, src2, lab*	Branch to *lab* if $src1 \geq src2$.
○	`bgt(u)`	*src1, src2, lab*	Branch to *lab* if $src1 > src2$.
○	`ble(u)`	*src1, src2, lab*	Branch to *lab* if $src1 \leq src2$.
○	`blt(u)`	*src1, src2, lab*	Branch to *lab* if $src1 < src2$.
○	`beqz`	*src1, lab*	Branch to *lab* if $src1 \equiv 0$.
○	`bnez`	*src1, lab*	Branch to *lab* if $src1 \neq 0$.
	`bgez`	*src1, lab*	Branch to *lab* if $src1 \geq 0$.
	`bgtz`	*src1, lab*	Branch to *lab* if $src1 > 0$.
	`blez`	*src1, lab*	Branch to *lab* if $src1 \leq 0$.
	`bltz`	*src1, lab*	Branch to *lab* if $src1 < 0$.
	`bgezal`	*src1, lab*	If $src1 \geq 0$, then put the address of the next instruction into `$ra` and branch to *lab*.
	`bgtzal`	*src1, lab*	If $src1 > 0$, then put the address of the next instruction into `$ra` and branch to *lab*.
	`bltzal`	*src1, lab*	If $src1 < 0$, then put the address of the next instruction into `$ra` and branch to *lab*.

4.4.3.2 Jump

Op	Operands	Description
`j`	*label*	Jump to label *lab*.
`jr`	*src1*	Jump to location *src1*.
`jal`	*label*	Jump to label *lab*, and store the address of the next instruction in `$ra`.
`jalr`	*src1*	Jump to location *src1*, and store the address of the next instruction in `$ra`.

4.4.4 Load, Store, and Data Movement

The second operand of all of the load and store instructions must be an address. The MIPS architecture supports the following addressing modes:

	Format	Meaning
○	*(reg)*	Contents of *reg*.
○	*const*	A constant address.
	const(reg)	*const* + contents of *reg*.
○	*symbol*	The address of *symbol*.
○	*symbol+const*	The address of *symbol* + *const*.
○	*symbol+const(reg)*	The address of *symbol* + *const* + contents of *reg*.

4.4.4.1 Load

The load instructions, with the exceptions of `li` and `lui`, fetch a byte, halfword, or word from memory and put it into a register. The `li` and `lui` instructions load a constant into a register.

All load addresses must be *aligned* on the size of the item being loaded. For example, all loads of halfwords must be from even addresses, and loads of words from addresses cleanly divisible by four. The `ulh` and `ulw` instructions are provided to load halfwords and words from addresses that might not be aligned properly.

	Op	Operands	Description
○	`la`	*des, addr*	Load the address of a label.
	`lb(u)`	*des, addr*	Load the byte at *addr* into *des*.
	`lh(u)`	*des, addr*	Load the halfword at *addr* into *des*.
○	`li`	*des, const*	Load the constant *const* into *des*.
	`lui`	*des, const*	Load the constant *const* into the upper halfword of *des*, and set the lower halfword of *des* to 0.
	`lw`	*des, addr*	Load the word at *addr* into *des*.
	`lwl`	*des, addr*	
	`lwr`	*des, addr*	
○	`ulh(u)`	*des, addr*	Load the halfword starting at the (possibly unaligned) address *addr* into *des*.
○	`ulw`	*des, addr*	Load the word starting at the (possibly unaligned) address *addr* into *des*.

4.4.4.2 Store

The store instructions store a byte, halfword, or word from a register into memory.

Like the load instructions, all store addresses must be *aligned* on the size of the item being stored. For example, all stores of halfwords must be from even addresses, and loads of words from addresses cleanly divisible by four. The `swl`, `swr`, `ush` and `usw` instructions are provided to store halfwords and words to addresses which might not be aligned properly.

	Op	Operands	Description
	`sb`	*src1, addr*	Store the lower byte of register *src1* to *addr*.
	`sh`	*src1, addr*	Store the lower halfword of register *src1* to *addr*.
	`sw`	*src1, addr*	Store the word in register *src1* to *addr*.
	`swl`	*src1, addr*	Store the upper halfword in *src* to the (possibly unaligned) address *addr*.
	`swr`	*src1, addr*	Store the lower halfword in *src* to the (possibly unaligned) address *addr*.
○	`ush`	*src1, addr*	Store the lower halfword in *src* to the (possibly unaligned) address *addr*.
○	`usw`	*src1, addr*	Store the word in *src* to the (possibly unaligned) address *addr*.

4.4.4.3 Data Movement

The data movement instructions move data among registers. Special instructions are provided to move data in and out of special registers such as `hi` and `lo`.

	Op	Operands	Description
○	`move`	*des, src1*	Copy the contents of *src1* to *des*.
	`mfhi`	*des*	Copy the contents of the `hi` register to *des*.
	`mflo`	*des*	Copy the contents of the `lo` register to *des*.
	`mthi`	*src1*	Copy the contents of the *src1* to `hi`.
	`mtlo`	*src1*	Copy the contents of the *src1* to `lo`.

4.4.5 Exception Handling

Op	Operands	Description
`rfe`		Return from exception.
`syscall`		Makes a system call. See 4.6.1 for a list of the SPIM system calls.
`break`	*const*	Used by the debugger.
`nop`		An instruction which has no effect (other than taking a cycle to execute).

4.5 The SPIM Assembler

4.5.1 Segment and Linker Directives

Name	Parameters	Description
`.data`	*addr*	The following items are to be assembled into the data segment. By default, begin at the next available address in the data segment. If the optional argument *addr* is present, then begin at *addr*.
`.text`	*addr*	The following items are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument *addr* is present, then begin at *addr*. In SPIM, the only items that can be assembled into the text segment are instructions and words (via the `.word` directive).
`.kdata`	*addr*	The kernel data segment. Like the data segment, but used by the Operating System.
`.ktext`	*addr*	The kernel text segment. Like the text segment, but used by the Operating System.
`.extern`	*sym size*	Declare as global the label *sym*, and declare that it is *size* bytes in length (this information can be used by the assembler).
`.globl`	*sym*	Declare as global the label *sym*.

4.5.2 Data Directives

Name	Parameters	Description
`.align`	*n*	Align the next item on the next 2^n-byte boundary. `.align 0` turns off automatic alignment.
`.ascii`	*str*	Assemble the given string in memory. Do not null-terminate.
`.asciiz`	*str*	Assemble the given string in memory. Do null-terminate.
`.byte`	*byte1* $\cdots$ *byteN*	Assemble the given bytes (8-bit integers).
`.half`	*half1* $\cdots$ *halfN*	Assemble the given halfwords (16-bit integers).
`.space`	*size*	Allocate *n* bytes of space in the current segment. In SPIM, this is only permitted in the data segment.
`.word`	*word1* $\cdots$ *wordN*	Assemble the given words (32-bit integers).

4.6 The SPIM Environment

4.6.1 SPIM syscalls

Service	Code	Arguments	Result
`print_int`	1	`$a0`	*none*
`print_float`	2	`$f12`	*none*
`print_double`	3	`$f12`	*none*
`print_string`	4	`$a0`	*none*
`read_int`	5	*none*	`$v0`
`read_float`	6	*none*	`$f0`
`read_double`	7	*none*	`$f0`
`read_string`	8	`$a0` (address), `$a1` (length)	*none*
`sbrk`	9	`$a0` (length)	`$v0`
`exit`	10	*none*	*none*

4.7 The Native MIPS Instruction Set

Many of the instructions listed here are not native MIPS instructions. Instead, they are *pseudoinstructions*– macros that the assembler knows how to translate into native

MIPS instructions. Instead of programming the "real" hardware, MIPS programmers generally use the *virtual machine* implemented by the MIPS assembler, which is much easier to program than the native machine.

For example, in most cases, the SPIM assembler will allow *src2* to be a 32-bit integer constant. Of course, since the MIPS instructions are all exactly 32 bits in length, there's no way that a 32-bit constant can fit in a 32-bit instruction word and have any room left over to specify the operation and the operand registers! When confronted with a 32-bit constant, the assembler uses a table of rules to generate a sequence of native instructions that will do what the programmer has asked.

The assembler also performs some more intricate transformations to translate your programs into a sequence of native MIPS instructions, but these will not be discussed in this text.

By default, the SPIM environment implements the same virtual machine that the MIPS assembler uses. It also implements the bare machine, if invoked with the `-bare` option enabled.