MIPS Assembly/MIPS Details

Contents

Registers

MIPS has 32 general-purpose registers and another 32 floating-point registers. Registers all begin with a dollar-symbol ($). The floating point registers are named $f0, $f1, ..., $f31. The general-purpose registers have both names and numbers, and are listed below. When programming in MIPS assembly, it is usually best to use the register names.

Number	Name	Comments
$0	$zero	Always zero
$1	$at	Reserved for assembler
$2, $3	$v0, $v1	First and second return values, respectively
$4, ..., $7	$a0, ..., $a3	First four arguments to functions
$8, ..., $15	$t0, ..., $t7	Temporary registers
$16, ..., $23	$s0, ..., $s7	Saved registers
$24, $25	$t8, $t9	More temporary registers
$26, $27	$k0, $k1	Reserved for kernel (operating system)
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

In general, there are many registers that can be used in your programs: the ten **temporary registers** and the eight **saved registers**. Temporary registers are general-purpose registers that can be used for arithmetic and other instructions freely, while saved registers must be saved at procedure entry, and restored at procedure exit.

Temporary register names all start with a $t. For instance, there are $t0, $t1 ... $t9. this means there are 10 temporary registers that can be used without worrying about saving and restoring their contents. The saved registers are named $s0 to $s7.

The **zero register**, is named $zero ($0), and is a static register: it always contains the value zero. This register may not be used as the target of a store operation, because its value is hardwired in, and cannot be changed by the program.

There are also several registers to which the programmer does not have direct access. Among these are the Program Counter (PC), which stores the address of the instruction executing, and the "hi" and "lo" registers, which are used in multiplication and division, which have results longer than 32 bits (multiplication may result in a 64-bit product and division results in a quotient and remainder). Programmers change the PC on branches and jumps. There are special instructions to move data to and from the hi and lo registers.

Instruction Formats

There are 3 different types of instructions: R Instructions, I Instructions, and J Instructions.

R Instructions

R Instructions take three arguments: two source registers (**rt** and **rs**), and a destination register (**rd**). R instructions are written using the following format:

instruction rd, rs, rt

where each one stands for as follow:

rd	Destination register specifier
rs	Source register specifier
rt	Source/Destination register specifier

For example,

```
add $t0, $t1, $t2
```

Adds the values of $t1 and $t2 and stores the result in $t0.

When assembled into machine code, an R instruction is represented as follows:

opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

For R-format instructions, the **opcode**, or "operation code" is always zero. **rs**, **rt**, and **rd** correspond to the two source and one destination registers, respectively. **shamt** is used in shift instructions instead of **rt** to make the hardware simpler. In assembly, to shift the value in $t4 two bits to the left and place the result in $t5:

```
sll $t5, $t4, 2
```

Since the opcode is zero for all R-format instructions, **func** specifies to the hardware exactly which R-format instruction to execute. The add example above would be encoded as follows:

```
opcode rs    rt     rd   shamt funct
000000 01001 01010 01000 00000 100000
```

Since it is an R-format instruction, the first six bits (opcode) are 0. The next 5 bits correspond to rs, which in this example is $t1. From the table above, we find that $t1 is $9, which in binary is 01001. Likewise, the next five bits encode $t2 = $10 = 01010. The destination is $t0 = $8 = 01000. We are not performing a shift, so shamt is 00000. Finally, since the func for the add instruction is 100000.

For the shift example above, the opcode field is again 0 since this is an R format instruction. The rs field is unused in shifts, so we leave the next five bits at 0. The rt field is $t4 = $12 = 01100. The rd field is $t5 = $13 = 01101. The shift amount, shamt, is 2 = 00010. Finally, the func field for sll is 000000. Thus, the encoding for **sll $t5, $t4, 2** is:

```
opcode rs    rt     rd    shamt  funct
000000 00000 01100 01101 00010  000000
```

I Instructions

I instructions take two register arguments and a 16-bit "immediate" value. An immediate is a value that is stored as part of the instruction instead of in memory. This makes accessing constants much faster than if we had to put constants in memory and then load them (hence the name). I-format instructions, like R-format instructions, specify the target register (**rt**) first. Next comes one source register (**rs**) and finally the immediate value.

instruction rt, rs, imm

For example, let's say that we want to add the value 5 to the register $t1, and store the result in $t0:

```
addi $t0, $t1, 5
```

I-format instructions are represented in machine code as follows:

opcode	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

The **opcode** specifies which operation is requested. **rs** and **rt** are five bits each, as before, and in the same positions as the R-format instructions. The **imm** field holds the immediate value. Depending on the instruction, the immediate constant may either be sign-extended or zero-extended. If a 32-bit immediate is needed, a special instruction, **lui** ("load upper immediate") exists for loading an immediate into the upper 16 bits of a register. That register can then be logically ORed with another 16-bit immediate to store the final value in that register. That value can then be used in normal R-format instructions. The following sequence of instructions stores the bit pattern 0101 0101 0101 ... into register $t0:

```
lui $t0, 0x5555
ori $t0, $t0, 0x5555
```

Typically, the assembler will automatically split 32-bit constants in this way so the programmer doesn't have to worry about it.

The addi example from above would be encoded as follows. The addi instruction has an opcode of 001000. The source register, $t1, is number 9, or 01001 in binary. The target register, $t0, is number 8, or 01000 in binary. Five is 101 in binary, so **addi $t0, $t1, 5** in machine code is:

```
opcode   rs   rt          imm
001000 01001 01000 0000 0000 0000 0101
```

J Instructions

J instructions are used to transfer program flow to a given, hardcoded offset from the current value of the PC register. J instructions are almost always written with labels: the assembler and linker will convert the label into a numerical value. A J instruction takes only one argument: the address to jump to.

instruction addr

There are two J-format instructions: **j** and **jal**. The latter will be discussed later. The j ("jump") instruction tells the processor to immediately skip to the instruction addressed by **addr**. To example, to jump to a **label1**:

```
j label1
```

A J-format instruction is encoded as

opcode	**addr**
6 bits	26 bits

On MIPS32 machines, addresses are 32-bits wide, so 26 bits may not be enough to specify which instruction to jump to. Fortunately, since all instructions are 32-bits (four bytes) wide, we can assume that all instructions start at a byte that's divisible by 4 (we are in fact guaranteed this by the loader). In binary, a number that is divisible by 4 ends with two zeros (just like a number that's divisible by 100 in decimal always ends in two zeros). Therefore, we can allow the assembler to leave out the last two zeros and have the hardware reinsert them. This effectively makes the address field 28 bits. The final four bits will be borrowed from the address of the current instruction, so we cannot let a program straddle a 256MB boundary, because a jump across the boundary would require a change in the 4 uppermost bits.

In the example above, if label1 specified an instruction as address 120, or 1111000 in binary, we can encode the jump example above in machine code as follows. The opcode for j is 2, or 10 in binary, and we must chop off the last two bits of the jump address, leaving it at 11110. Thus, the machine code for **j 120** is:

```
opcode |---------------addr-----------|
000010 0000 0000 0000 0000 0000 0111 10
```

Retrieved from "https://en.wikibooks.org/w/index.php?title=MIPS_Assembly/MIPS_Details&oldid=3066351"
