MIPS Addressing Mode Summary

addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.

Multiple forms of addressing are generically called **addressing modes**. The MIPS addressing modes are the following:

1. **Register addressing**, where the operand is a register
2. **Base** or **displacement addressing**, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
3. **Immediate addressing**, where the operand is a constant within the instruction itself
4. **PC-relative addressing**, where the address is the sum of the PC and a constant in the instruction
5. **Pseudodirect addressing**, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Hardware Software Interface

Although we show the MIPS architecture as having 32-bit addresses, nearly all microprocessors (including MIPS) have 64-bit address extensions (see Appendix D). These extensions were in response to the needs of software for larger programs. The process of instruction set extension allows architectures to expand in a way that lets software move compatibly upward to the next generation of architecture.

Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`addi`) and register (`add`) addressing. Figure 2.24 shows how operands are identified for each addressing mode. In More Depth shows other addressing modes found in the IBM PowerPC.

Decoding Machine Language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at a core dump. Figure 2.25 shows the MIPS encoding of the fields for the MIPS machine language. This figure helps when translating by hand between assembly language and machine language.

Warning: Programs that use these syscalls to read from the terminal should not use memory-mapped I/O (see Section A.8).

`sbrk` returns a pointer to a block of memory containing *n* additional bytes. `exit` stops the program SPIM is running. `exit2` terminates the SPIM program, and the argument to `exit2` becomes the value returned when the SPIM simulator itself terminates.

`print_char` and `read_char` write and read a single character. `open`, `read`, `write`, and `close` are the standard UNIX library calls.

A.10 MIPS R2000 Assembly Language

A MIPS processor consists of an integer processing unit (the CPU) and a collection of coprocessors that perform ancillary tasks or operate on other types of data such as floating-point numbers (see Figure A.10.1). SPIM simulates two coprocessors. Coprocessor 0 handles exceptions and interrupts. Coprocessor 1 is the floating-point unit. SPIM simulates most aspects of this unit.

Addressing Modes

MIPS is a load-store architecture, which means that only load and store instructions access memory. Computation instructions operate only on values in registers. The bare machine provides only one memory-addressing mode: `c(rx)`, which uses the sum of the immediate `c` and register `rx` as the address. The virtual machine provides the following addressing modes for load and store instructions:

Format	Address computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
label	address of label
label ± imm	address of label + or – immediate
label ± imm (register)	address of label + or – (immediate + contents of register)

Most load and store instructions operate only on aligned data. A quantity is *aligned* if its memory address is a multiple of its size in bytes. Therefore, a half-

```
lui $at, 4096
addu $at, $at, $a1
lw $a0, 8($at)
```

The first instruction loads the upper bits of the label's address into register `$at`, which is the register that the assembler reserves for its own use. The second instruction adds the contents of register `$a1` to the label's partial address. Finally, the load instruction uses the hardware address mode to add the sum of the lower bits of the label's address and the offset from the original instruction to the value in register `$at`.

Assembler Syntax

Comments in assembler files begin with a sharp sign (#). Everything from the sharp sign to the end of the line is ignored.

Identifiers are a sequence of alphanumeric characters, underbars (_), and dots (.) that do not begin with a number. Instruction opcodes are reserved words that *cannot* be used as identifiers. Labels are declared by putting them at the beginning of a line followed by a colon, for example:

```
        .data
item:   .word 1
        .text
        .globl main      # Must be global
main:   lw          $t0, item
```

Numbers are base 10 by default. If they are preceded by *0x*, they are interpreted as hexadecimal. Hence, 256 and 0x100 denote the same value.

Strings are enclosed in doublequotes ("). Special characters in strings follow the C convention:

- newline `\n`
- tab `\t`
- quote `\"`

SPIM supports a subset of the MIPS assembler directives:

`.align n`	Align the next datum on a 2^n byte boundary. For example, `.align 2` aligns the next value on a word boundary. `.align 0` turns off automatic alignment of `.half`, `.word`, `.float`, and `.double` directives until the next `.data` or `.kdata` directive.
`.ascii str`	Store the string *str* in memory, but do not null-terminate it.

`.asciiz str`	Store the string *str* in memory and null-terminate it.
`.byte b1,..., bn`	Store the *n* values in successive bytes of memory.
`.data <addr>`	Subsequent items are stored in the data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.double d1, ..., dn`	Store the *n* floating-point double precision numbers in successive memory locations.
`.extern sym size`	Declare that the datum stored at *sym* is *size* bytes large and is a global label. This directive enables the assembler to store the datum in a portion of the data segment that is efficiently accessed via register `$gp`.
`.float f1,..., fn`	Store the *n* floating-point single precision numbers in successive memory locations.
`.globl sym`	Declare that label *sym* is global and can be referenced from other files.
`.half h1, ..., hn`	Store the *n* 16-bit quantities in successive memory halfwords.
`.kdata <addr>`	Subsequent data items are stored in the kernel data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.ktext <addr>`	Subsequent items are put in the kernel text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.set noat` and `.set at`	The first directive prevents SPIM from complaining about subsequent instructions that use register `$at`. The second directive reenables the warning. Since pseudoinstructions expand into code that uses register `$at`, programmers must be very careful about leaving values in this register.
`.space n`	Allocate *n* bytes of space in the current segment (which must be the data segment in SPIM).

`.text <addr>`	Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.word w1, ..., wn`	Store the *n* 32-bit quantities in successive memory words.

SPIM does not distinguish various parts of the data segment (`.data`, `.rdata`, and `.sdata`).

Encoding MIPS Instructions

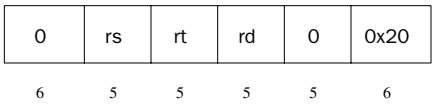
Figure A.10.2 explains how a MIPS instruction is encoded in a binary number. Each column contains instruction encodings for a field (a contiguous group of bits) from an instruction. The numbers at the left margin are values for a field. For example, the `j` opcode has a value of 2 in the opcode field. The text at the top of a column names a field and specifies which bits it occupies in an instruction. For example, the `op` field is contained in bits 26–31 of an instruction. This field encodes most instructions. However, some groups of instructions use additional fields to distinguish related instructions. For example, the different floating-point instructions are specified by bits 0–5. The arrows from the first column show which opcodes use these additional fields.

Instruction Format

The rest of this appendix describes both the instructions implemented by actual MIPS hardware and the pseudoinstructions provided by the MIPS assembler. The two types of instructions are easily distinguished. Actual instructions depict the fields in their binary representation. For example, in

Addition (with overflow)

`add rd, rs, rt`



the `add` instruction consists of six fields. Each field's size in bits is the small number below the field. This instruction begins with 6 bits of 0s. Register specifiers begin with an *r,* so the next field is a 5-bit register specifier called `rs`. This is the same register that is the second argument in the symbolic assembly at the left of this line. Another common field is $\texttt{imm}_{16}$, which is a 16-bit immediate number.

Pseudoinstructions follow roughly the same conventions, but omit instruction encoding information. For example:

Multiply (without overflow)

```
mul rdest, rsrc1, src2        pseudoinstruction
```

In pseudoinstructions, `rdest` and `rsrc1` are registers and `src2` is either a register or an immediate value. In general, the assembler and SPIM translate a more general form of an instruction (e.g., `add $v1, $a0, 0x55`) to a specialized form (e.g., `addi $v1, $a0, 0x55`).

Arithmetic and Logical Instructions

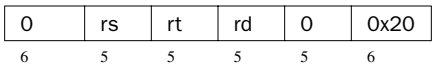
Absolute value

```
abs rdest, rsrc              pseudoinstruction
```

Put the absolute value of register `rsrc` in register `rdest`.

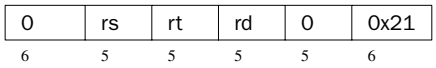
Addition (with overflow)

```
add rd, rs, rt
```



Addition (without overflow)

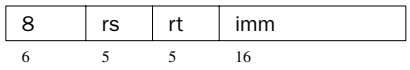
```
addu rd, rs, rt
```



Put the sum of registers `rs` and `rt` into register `rd`.

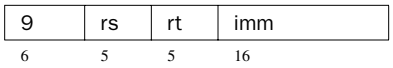
Addition immediate (with overflow)

```
addi rt, rs, imm
```



Addition immediate (without overflow)

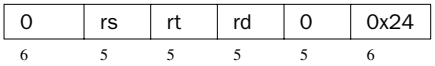
```
addiu rt, rs, imm
```



Put the sum of register `rs` and the sign-extended immediate into register `rt`.

AND

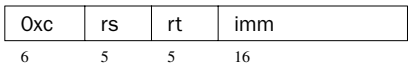
`and rd, rs, rt`



Put the logical AND of registers `rs` and `rt` into register `rd`.

AND immediate

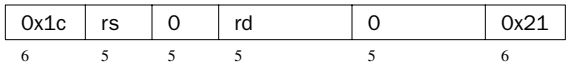
`andi rt, rs, imm`



Put the logical AND of register `rs` and the zero-extended immediate into register `rt`.

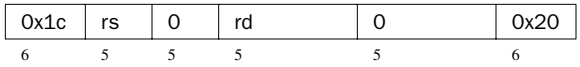
Count leading ones

`clo rd, rs`



Count leading zeros

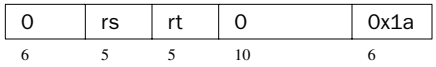
`clz rd, rs`



Count the number of leading ones (zeros) in the word in register `rs` and put the result into register `rd`. If a word is all ones (zeros), the result is 32.

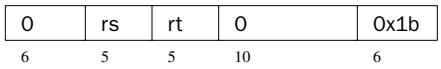
Divide (with overflow)

`div rs, rt`



Divide (without overflow)

`divu rs, rt`



Divide register `rs` by register `rt`. Leave the quotient in register `lo` and the remainder in register `hi`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.

Divide (with overflow)

```
div rdest, rsrc1, src2          pseudoinstruction
```

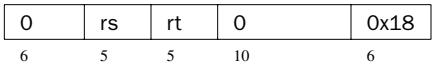
Divide (without overflow)

```
divu rdest, rsrc1, src2         pseudoinstruction
```

Put the quotient of register `rsrc1` and `src2` into register `rdest`.

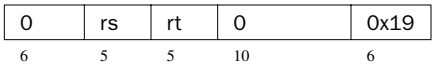
Multiply

`mult rs, rt`



Unsigned multiply

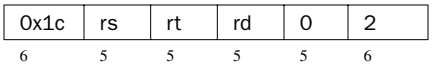
`multu rs, rt`



Multiply registers `rs` and `rt`. Leave the low-order word of the product in register `lo` and the high-order word in register `hi`.

Multiply (without overflow)

`mul rd, rs, rt`



Put the low-order 32 bits of the product of `rs` and `rt` into register `rd`.

Multiply (with overflow)

```
mulo rdest, rsrc1, src2         pseudoinstruction
```

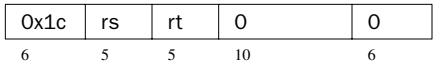
Unsigned multiply (with overflow)

```
mulou rdest, rsrc1, src2        pseudoinstruction
```

Put the low-order 32 bits of the product of register `rsrc1` and `src2` into register `rdest`.

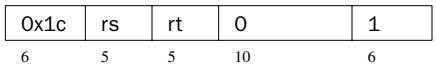
Multiply add

```
madd rs, rt
```



Unsigned multiply add

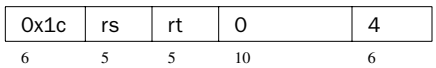
```
maddu rs, rt
```



Multiply registers `rs` and `rt` and add the resulting 64-bit product to the 64-bit value in the concatenated registers `lo` and `hi`.

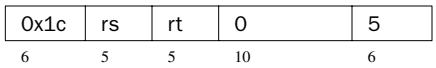
Multiply subtract

```
msub rs, rt
```



Unsigned multiply subtract

```
msub rs, rt
```



Multiply registers `rs` and `rt` and subtract the resulting 64-bit product from the 64-bit value in the concatenated registers `lo` and `hi`.

Negate value (with overflow)

```
neg rdest, rsrc
```

pseudoinstruction

Negate value (without overflow)

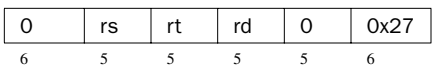
```
negu rdest, rsrc
```

pseudoinstruction

Put the negative of register `rsrc` into register `rdest`.

NOR

```
nor rd, rs, rt
```



Put the logical NOR of registers `rs` and `rt` into register `rd`.

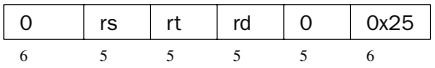
NOT

```
not rdest, rsrc          pseudoinstruction
```

Put the bitwise logical negation of register `rsrc` into register `rdest`.

OR

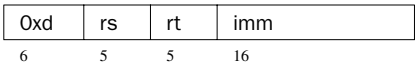
```
or rd, rs, rt
```



Put the logical OR of registers `rs` and `rt` into register `rd`.

OR immediate

```
ori rt, rs, imm
```



Put the logical OR of register `rs` and the zero-extended immediate into register `rt`.

Remainder

```
rem rdest, rsrc1, rsrc2          pseudoinstruction
```

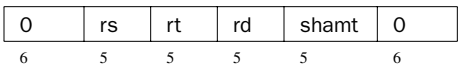
Unsigned remainder

```
remu rdest, rsrc1, rsrc2         pseudoinstruction
```

Put the remainder of register `rsrc1` divided by register `rsrc2` into register `rdest`. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the convention of the machine on which SPIM is run.

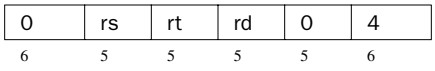
Shift left logical

```
sll rd, rt, shamt
```



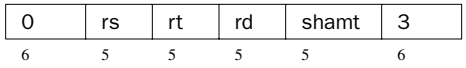
Shift left logical variable

```
sllv rd, rt, rs
```



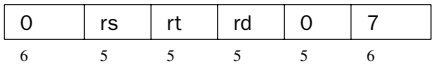
Shift right arithmetic

```
sra rd, rt, shamt
```



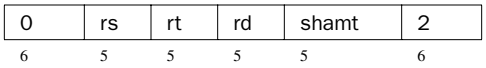
Shift right arithmetic variable

```
srav rd, rt, rs
```



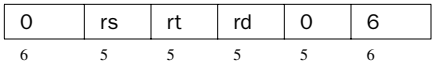
Shift right logical

```
srl rd, rt, shamt
```



Shift right logical variable

```
srlv rd, rt, rs
```



Shift register `rt` left (right) by the distance indicated by immediate `shamt` or the register `rs` and put the result in register `rd`. Note that argument `rs` is ignored for `sll`, `sra`, and `srl`.

Rotate left

```
rol rdest, rsrc1, rsrc2        pseudoinstruction
```

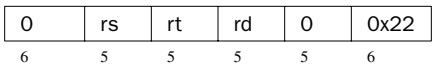
Rotate right

```
ror rdest, rsrc1, rsrc2        pseudoinstruction
```

Rotate register `rsrc1` left (right) by the distance indicated by `rsrc2` and put the result in register `rdest`.

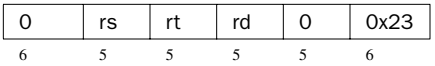
Subtract (with overflow)

```
sub rd, rs, rt
```



Subtract (without overflow)

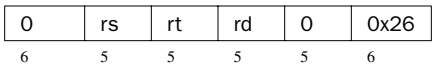
```
subu rd, rs, rt
```



Put the difference of registers `rs` and `rt` into register `rd`.

Exclusive OR

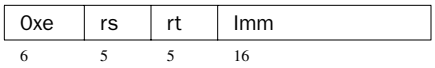
```
xor rd, rs, rt
```



Put the logical XOR of registers `rs` and `rt` into register `rd`.

XOR immediate

```
xori rt, rs, imm
```

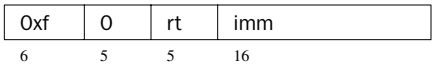


Put the logical XOR of register `rs` and the zero-extended immediate into register `rt`.

Constant-Manipulating Instructions

Load upper immediate

```
lui rt, imm
```



Load the lower halfword of the immediate `imm` into the upper halfword of register `rt`. The lower bits of the register are set to 0.

Load immediate

```
li rdest, imm                    pseudoinstruction
```

Move the immediate `imm` into register `rdest`.

Comparison Instructions

Set less than

```
slt rd, rs, rt
```