Chapter 15

MIPS Assembly Language

15–1 This register provides the constant zero in an efficient way. Since many programs use zero for loop counts, initialization of variables, determining positive or negative value, and so on, providing constant zero has received special attention from the designers.

15–2 The two special-purpose registers—called HI and LO—are used to hold the results of integer multiply and divide instructions:

- In the integer multiply operation, HI and LO registers hold the 64-bit result, with the higher-order 32 bits in the HI and the lower-order 32 bits in the LO register.
- In integer divide operations, the 32-bit quotient is stored in the LO and the remainder in the HI register.

15–3 Registers `$t0` to `$t9` are temporary registers that need not be preserved across a procedure call. These registers are assumed to be saved by the caller. On the other hand, registers `$s0` to `$s7` are callee-saved registers that should be preserved across procedure calls.

15–4 Registers `$v0` and `$v1` are used to return results from a procedure.

15–5 Registers `$a0` to `$a3` are used to pass the first four arguments to procedures.

15–6 The bare machine provides only a single memory addressing mode: `disp(Rx)`, where displacement `disp` is a signed, 16-bit immediate value. The address is computed as `disp` + contents of base register `Rx`. Thus, the MIPS provides only the based/indexed addressing mode. In the MIPS, we can use any register as the base register.

The virtual machine supported by the assembler provides additional addressing modes for load and store instructions to help in assembly language programming. The following table shows the addressing modes supported by the virtual machine.

Format	Address computed as
(Rx)	Contents of register Rx
imm	Immediate value imm
imm(Rx)	imm + contents of Rx
symbol	Address of symbol
symbol± imm	Address of symbol ±imm
symbol± imm(Rx)	Address of symbol ±(imm + contents of Rx)

15–7 The MIPS instruction set consists of *instructions* and *pseudoinstructions*. The MIPS processor supports only the instructions. Pseudoinstructions are provided by the assembler for convenience in programming. The assembler translates pseudoinstructions into a sequence of one or more processor instructions.

15–8 The SPIM provides two directives to allocate storage for strings: .ASCII and .ASCIIZ. The .ASCII directive can be used to allocate space for a string that is not terminated by the NULL character. The statement

```
.ASCII  string
```

allocates a number of bytes equal to the number of characters in string. For example,

```
.ASCII  "Toy Story"
```

allocates nine bytes of contiguous storage and initializes it to “Toy Story”.

Strings are normally NULL-terminated as in C. For example, to display a string using print_string service, the string must be NULL-terminated. Using .ASCIIZ instead of ASCII stores the specified string in the NULL-terminated format. The .ASCII directive is useful for breaking a long string into multiple string statements as shown in the following example:

```
.ASCII    "Toy Story is a good computer-animated movie. \n"
.ASCII    "This reviewer recommends it to all kids \n"
.ASCIIZ   "and their parents."
```

15–9 The MIPS stack implementation has some similarities to the Pentium implementation. For example, the stack grows downward (i.e., as we push items onto the stack, the address decreases). Thus, when reserving space on the stack for pushing values, we have to decrease the sp value. For example, to push registers a0 and ra, we have to reserve eight bytes of stack space and use sw to push the values as shown below:

```
sub    $sp,$sp,8      # reserve 8 bytes of stack
sw     $a0,0($sp)     # save registers
sw     $ra,4($sp)
```

This sequence is typically used at the beginning of a procedure to save registers. To restore these registers before returning from the procedure, we can use the following sequence:

```
lw     $a0,0($sp)     # restore the two registers
lw     $ra,4($sp)
addu   $sp,$sp,8      # clear 8 bytes of stack
```

15–10 The branch instruction `b exit_loop` is translated using `bgez` instruction as follows:

```
[0x004000a4] bgez $0 12
```

(The values in [ ] is the address of the instruction.) The offset 12 in the instruction is the relative offset to `exit_loop` computed as `exit_loop-0x004000a4`.

The jump instruction `j loop` is translated as follows:

```
j 0x004000a4
```

In contrast to the branch instruction, the jump instruction uses the absolute address (the value `0x004000a4` is the `loop` address).

15–11 (a)

```
srl $1, $9, 29
sll $9, $9, 3
or $9, $9, $1
```

(b)

```
ori $8, $0, 5
srlv $1, $9, $1
sllv $9, $9, $8
or $9, $9, $1
```

(c)

```
ori $1, $0, 9
mult $2, $1
mflo $8
```

(d)

```
ori $1, $0, 5
div $8, $1
mflo $8
```

(e)

```
ori $1, $0, 5
div $8, $1
mfhi $8
```

(f)

```
bne $6, $5, 12
ori $4, $0, 1
beq $0, $0, 8
slt $4, $5, $6
```

(g)

```
bne $6, $5, 12
ori $4, $0, 1
beq $0, $0, 8
slt $4, $6, $5
```

(h)

```
bne $6, $5, 12
ori $4, $0, 1
beq $0, $0, 8
sltu $4, $6, $5
```

(i)

```
slt $4, $5, $6
```

(j)

```
sltu $4, $5, $6
```

(k)

```
addu $4, $0, $8
```

15–12 MIPS uses registers `$a0` to `$a3` to pass the first four arguments to procedures. Thus, we can use `a0` to pass the parameter count. In contrast, Pentium uses the stack. In addition, compared to Pentium, MIPS allows more flexible access to parameters. In the Pentium, because the return address is pushed onto the stack, we had to use the `BP` register to access the parameters. In addition, we could not remove the numbers from the stack. The stack had to be cleared in the main program by manipulating the `SP` register.

15–13 The main reason is that MIPS has more registers than the Pentium. For example, registers `$v0` and `$v1` are used to return results from a procedure. Registers `$a0` to `$a3` are used to pass the first four arguments to procedures.

15–14 The main reason is that MIPS uses registers to pass parameters and return values. Furthermore, the return address is also stored in a register. For example, registers `$v0` and `$v1` are used to return results from a procedure. Registers `$a0` to `$a3` are used to pass the first four arguments to procedures. Thus, to invoke a procedure with four parameters, we don't need use the stack.

15–15 One of the main differences is the "scale factor" provided by the Pentium. By using the scale factor, we can access arrays using subscripts as opposed byte displacements (provided the array elements are 2, 4, or 8 bytes in size). Another difference is that the Pentium provides based-indexed addressing mode, which is not supported by MIPS.