Check Yourself

What MIPS instruction does this represent? Choose from one of the four options below.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. `sub $t0, $t1, $t2`
2. `add $t2, $t0, $t1`
3. `sub $t2, $t1, $t0`
4. `sub $t2, $t0, $t1`

2.6 Logical Operations

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

Lewis Carroll, *Alice's Adventures in Wonderland*, 1865

Although the first computers operated on full words, it soon became clear that it was useful to operate on fields of bits within a word or even on individual bits. Examining characters within a word, each of which is stored as 8 bits, is one example of such an operation (see Section 2.9). It follows that operations were added to programming languages and instruction set architectures to simplify, among other things, the packing and unpacking of bits into words. These instructions are called logical operations. Figure 2.8 shows logical operations in C, Java, and MIPS.

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bit-by-bit AND	&	&	`and`, `andi`
Bit-by-bit OR	|	|	`or`, `ori`
Bit-by-bit NOT	~	~	`nor`

FIGURE 2.8 C and Java logical operators and their corresponding MIPS instructions. MIPS implements NOT using a NOR with one operand being zero.

The first class of such operations is called *shifts*. They move all the bits in a word to the left or right, filling the emptied bits with 0s. For example, if register `$s0` contained

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001_{two} = 9_{ten}$$

and the instruction to shift left by 4 was executed, the new value would be:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1001\ 0000_{two} = 144_{ten}$$

The dual of a shift left is a shift right. The actual name of the two MIPS shift instructions are called *shift left logical* (`sll`) and *shift right logical* (`srl`). The following instruction performs the operation above, assuming that the original value was in register `$s0` and the result should go in register `$t2`:

```
sll  $t2,$s0,4  # reg $t2 = reg $s0 << 4 bits
```

We delayed explaining the *shamt* field in the R-format. Used in shift instructions, it stands for *shift amount*. Hence, the machine language version of the instruction above is

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

The encoding of `sll` is 0 in both the op and funct fields, rd contains 10 (register `$t2`), rt contains 16 (register `$s0`), and shamt contains 4. The rs field is unused and thus is set to 0.

Shift left logical provides a bonus benefit. Shifting left by *i* bits gives the same result as multiplying by 2*i*, just as shifting a decimal number by *i* digits is equivalent to multiplying by 10*i*. For example, the above `sll` shifts by 4, which gives the same result as multiplying by 2^4 or 16. The first bit pattern above represents 9, and $9 \times 16 =$ 144, the value of the second bit pattern.

AND A logical bit-by-bit operation with two operands that calculates a 1 only if there is a 1 in *both* operands.

Another useful operation that isolates fields is **AND**. (We capitalize the word to avoid confusion between the operation and the English conjunction.) AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. For example, if register `$t2` contains

$$0000\ 0000\ 0000\ 0000\ 0000\ 1101\ 1100\ 0000_{two}$$

and register `$t1` contains

$$0000\ 0000\ 0000\ 0000\ 0011\ 1100\ 0000\ 0000_{two}$$

then, after executing the MIPS instruction

```
and $t0,$t1,$t2    # reg $t0 = reg $t1 & reg $t2
```

the value of register `$t0` would be

$$0000\ 0000\ 0000\ 0000\ 0000\ 1100\ 0000\ 0000_{two}$$

As you can see, AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern. Such a bit pattern in conjunction with AND is traditionally called a *mask*, since the mask “conceals” some bits.

To place a value into one of these seas of 0s, there is the dual to AND, called **OR**. It is a bit-by-bit operation that places a 1 in the result if *either* operand bit is a 1. To elaborate, if the registers `$t1` and `$t2` are unchanged from the preceding example, the result of the MIPS instruction

```
or $t0,$t1,$t2 # reg $t0 = reg $t1 | reg $t2
```

is this value in register `$t0`:

$0000\ 0000\ 0000\ 0000\ 0011\ 1101\ 1100\ 0000_{two}$

OR A logical bit-by-bit operation with two operands that calculates a 1 if there is a 1 in *either* operand.

The final logical operation is a contrarian. **NOT** takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. Using our prior notation, it calculates $\overline{x}$.

NOT A logical bit-by-bit operation with one operand that inverts the bits; that is, it replaces every 1 with a 0, and every 0 with a 1.

In keeping with the three-operand format, the designers of MIPS decided to include the instruction **NOR** (NOT OR) instead of NOT. If one operand is zero, then it is equivalent to NOT: A NOR 0 = NOT (A OR 0) = NOT (A).

If the register `$t1` is unchanged from the preceding example and register `$t3` has the value 0, the result of the MIPS instruction

```
nor $t0,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)
```

is this value in register `$t0`:

$1111\ 1111\ 1111\ 1111\ 1100\ 0011\ 1111\ 1111_{two}$

NOR A logical bit-by-bit operation with two operands that calculates the NOT of the OR of the two operands. That is, it calculates a 1 only if there is a 0 in *both* operands.

Figure 2.8 above shows the relationship between the C and Java operators and the MIPS instructions. Constants are useful in AND and OR logical operations as well as in arithmetic operations, so MIPS also provides the instructions *and immediate* (`andi`) and *or immediate* (`ori`). Constants are rare for NOR, since its main use is to invert the bits of a single operand; thus, the MIPS instruction set architecture has no immediate version of NOR.

Elaboration: The full MIPS instruction set also includes exclusive or (XOR), which sets the bit to 1 when two corresponding bits differ, and to 0 when they are the same. C allows *bit fields* or *fields* to be defined within words, both allowing objects to be packed within a word and to match an externally enforced interface such as an I/O device. All fields must fit within a single word. Fields are unsigned integers that can be as short as 1 bit. C compilers insert and extract fields using logical instructions in MIPS: `and`, `or`, `sll`, and `srl`.

Elaboration: Logical AND immediate and logical OR immediate put 0s into the upper 16 bits to form a 32-bit constant, unlike add immediate, which does sign extension.

Check Yourself

Which operations can isolate a field in a word?

1. AND
2. A shift left followed by a shift right

The utility of an automatic computer lies in the possibility of using a given sequence of instructions repeatedly, the number of times it is iterated being dependent upon the results of the computation. . . . This choice can be made to depend upon the sign of a number (zero being reckoned as plus for machine purposes). Consequently, we introduce an [instruction] (the conditional transfer [instruction]) which will, depending on the sign of a given number, cause the proper one of two routines to be executed.

Burks, Goldstine, and von Neumann, 1947

2.7 Instructions for Making Decisions

What distinguishes a computer from a simple calculator is its ability to make decisions. Based on the input data and the values created during computation, different instructions execute. Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels. MIPS assembly language includes two decision-making instructions, similar to an *if* statement with a *go to*. The first instruction is

```
beq register1, register2, L1
```

This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`. The mnemonic beq stands for *branch if equal*. The second instruction is

```
bne register1, register2, L1
```

It means go to the statement labeled `L1` if the value in `register1` does *not* equal the value in `register2`. The mnemonic `bne` stands for *branch if not equal*. These two instructions are traditionally called **conditional branches**.

EXAMPLE

Compiling *If-then-else* into Conditional Branches

In the following code segment, `f`, `g`, `h`, `i`, and `j` are variables. If the five variables `f` through `j` correspond to the five registers `$s0` through `$s4`, what is the compiled MIPS code for this C *if* statement?

```
if (i == j) f = g + h; else f = g - h;
```

ANSWER

Figure 2.9 shows a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want the branch if registers are equal instruction (`beq`). In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label `Else` is defined below) and so we use the branch if registers are *not* equal instruction (`bne`):

```
bne $s3,$s4,Else   # go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as `j` (the label `Exit` is defined below).

```
j Exit     # go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label `Else` to this instruction. We also show the label `Exit` that is after this instruction, showing the end of the *if-then-else* compiled code:

```
Else:sub $s0,$s1,$s2  # f = g - h (skipped if i = j)
Exit:
```

conditional branch An instruction that requires the comparison of two values and that allows for a subsequent transfer of control to a new address in the program based on the outcome of the comparison.

Notice that the assembler relieves the compiler and the assembly language programmer from the tedium of calculating addresses for branches, just as it does for calculating data addresses for loads and stores (see Section 2.12).

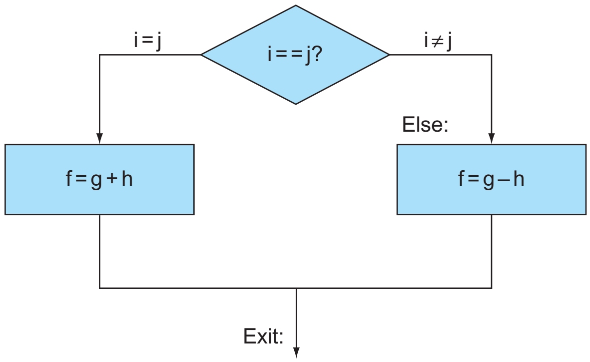

FIGURE 2.9 Illustration of the options in the if statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

Hardware/Software Interface

Compilers frequently create branches and labels where they do not appear in the programming language. Avoiding the burden of writing explicit labels and branches is one benefit of writing in high-level programming languages and is a reason coding is faster at that level.

Loops

Decisions are important both for choosing between two alternatives—found in *if* statements—and for iterating a computation—found in loops. The same assembly instructions are the building blocks for both cases.

EXAMPLE

Compiling a *while* Loop in C

Here is a traditional loop in C:

```
while (save[i] == k)
      i += 1;
```

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C segment?

ANSWER

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 88 in the prior section). We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll  $t1,$s3,2    # Temp reg $t1 = i * 4
```

To get the address of `save[i]`, we need to add `$t1` and the base of save in `$s6`:

```
      add $t1,$t1,$s6     # $t1 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
      lw $t0,0($t1)       # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i] ≠ k`:

```
      bne $t0,$s5, Exit   # go to Exit if save[i] ≠ k
```

The next instruction adds 1 to `i`:

```
addi $s3,$s3,1      # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
      j     Loop           # go to Loop
    Exit:
```

(See the exercises for an optimization of this sequence.)

Hardware/Software Interface

Such sequences of instructions that end in a branch are so fundamental to compiling that they are given their own buzzword: a **basic block** is a sequence of instructions without branches, except possibly at the end, and without branch targets or branch labels, except possibly at the beginning. One of the first early phases of compilation is breaking the program into basic blocks.

basic block A sequence of instructions without branches (except possibly at the end) and without branch targets or branch labels (except possibly at the beginning).

The test for equality or inequality is probably the most popular test, but sometimes it is useful to see if a variable is less than another variable. For example, a *for* loop may want to test to see if the index variable is less than 0. Such comparisons are accomplished in MIPS assembly language with an instruction that compares two registers and sets a third register to 1 if the first is less than the second; otherwise, it is set to 0. The MIPS instruction is called *set on less than,* or `slt`. For example,

```
slt    $t0, $s3, $s4    # $t0 = 1 if $s3 < $s4
```

means that register `$t0` is set to 1 if the value in register `$s3` is less than the value in register `$s4`; otherwise, register `$t0` is set to 0.

Constant operands are popular in comparisons, so there is an immediate version of the set on less than instruction. To test if register `$s2` is less than the constant 10, we can just write

```
slti    $t0,$s2,10     # $t0 = 1 if $s2 < 10
```

Hardware/Software Interface

MIPS compilers use the `slt`, `slti`, `beq`, `bne`, and the fixed value of 0 (always available by reading register `$zero`) to create all relative conditions: equal, not equal, less than, less than or equal, greater than, greater than or equal.

Heeding von Neumann's warning about the simplicity of the "equipment," the MIPS architecture doesn't include branch on less than because it is too complicated; either it would stretch the clock cycle time or it would take extra clock cycles per instruction. Two faster instructions are more useful.

Hardware/Software Interface

Comparison instructions must deal with the dichotomy between signed and unsigned numbers. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll soon take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

EXAMPLE

Signed versus Unsigned Comparison

Suppose register `$s0` has the binary number

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

and that register `$s1` has the binary number

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

ANSWER

The value in register `$s0` represents -1_{ten} if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1_{ten} in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y.

Bounds Check Shortcut

EXAMPLE

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$s1 ≥ $t2` or if `$s1` is negative.

ANSWER

The checking code just uses `u` to do both checks:

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

Case/Switch Statement

Most programming languages have a *case* or *switch* statement that allows the programmer to select one of many alternatives depending on a single value. The simplest way to implement *switch* is via a sequence of conditional tests, turning the *switch* statement into a chain of *if-then-else* statements.

Sometimes the alternatives may be more efficiently encoded as a table of addresses of alternative instruction sequences, called a **jump address table** or **jump table**, and the program needs only to index into the table and then jump to the appropriate sequence. The jump table is then just an array of words containing addresses that correspond to labels in the code. The program loads the appropriate entry from the jump table into a register. It then needs to jump using the address in the register. To support such situations, computers like MIPS include a *jump register* instruction (`jr`), meaning an unconditional jump to the address specified in a register. Then it jumps to the proper address using this instruction. We'll see an even more popular use of `jr` in the next section.

jump address table Also called **jump table**. A table of addresses of alternative instruction sequences.

Hardware/Software Interface

Although there are many statements for decisions and loops in programming languages like C and Java, the bedrock statement that implements them at the instruction set level is the conditional branch.

Elaboration: If you have heard about *delayed branches*, covered in Chapter 4, don't worry: the MIPS assembler makes them invisible to the assembly language programmer.

Check Yourself

I. C has many statements for decisions and loops, while MIPS has few. Which of the following do or do not explain this imbalance? Why?

1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

II. Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while MIPS doesn't?

1. Logical operations AND and OR implement & and |, while conditional branches implement && and ||.
2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.

ABSTRACTION

2.8 Supporting Procedures in Computer Hardware

procedure A stored subroutine that performs a specific task based on the parameters with which it is provided.

A **procedure** or function is one tool programmers use to structure programs, both to make them easier to understand and to allow code to be reused. Procedures allow the programmer to concentrate on just one portion of the task at a time; parameters act as an interface between the procedure and the rest of the program and data, since they can pass values and return results. We describe the equivalent to procedures in Java in Section 2.15, but Java needs everything from a computer that C needs. Procedures are one way to implement **abstraction** in software.