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;
```

Figure 2.9 is 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 instruction that requi the comparison of tw values and that allows a subsequent transfer control to a new addr 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).

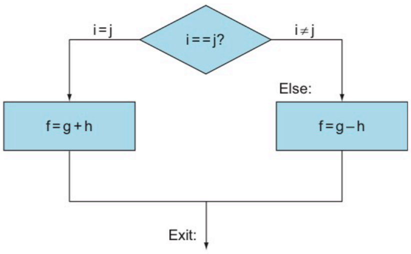

FIGURE 2.9 Illustration of the options in the if statement above. The left box corresponds to the *then* part of the *if* statement, and the right box corresponds to the *else* part.

例题 • 将 if-then-else 语句编译成条件分支指令

在下面这段代码中，f、g、h、i、j 都是变量，设该 5 个变量依次对应于从 $s0 到 $s4 的寄存器，求这条 C 语言 if 语句编译后形成的 MIPS 代码。

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

答案

图 2-9 是 MIPS 代码执行过程的流程图。第一个表达式比较 i 和 j 是否相等，需要一条 beq 指令。通常，通过测试分支的相反条件来跳过 if 语句后面的 then 部分，代码的效率会更高（标签 Else 将在后面定义）所以我们使用 bne 指令：

```
bne $s3,$s4,Else    # go to Else if i ≠ j
```
90

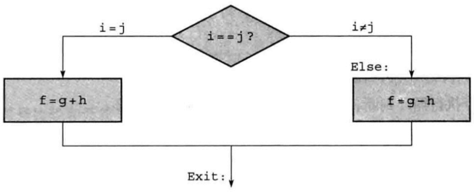

图 2-9 上述 if 语句的程序流程图。左边方框对应 if 语句的 then 部分，右边方框对应 if 语句的 else 部分

下一个赋值语句执行一个单操作，如果所有的操作数都分配给寄存器，那么它只是一条指令：

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

在 if 语句的结尾部分，需要引入另一种分支指令，通常叫作无条件分支指令（unconditional branch）。当遇到这种指令时，程序必须分支。为了区分条件分支和无条件分支，MIPS 将无条件分支指令命名为 jump，简写成 j（标签 Exit 将在后面定义）。

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

if 语句中 else 部分的赋值语句也可编译成一条指令。我们只需将标签 Else 加在这条指令前、标签 Exit 加在该条指令后面，表示 if-then-else 编译的代码结束：

```
Else:sub $s0,$s1,$s2  # f = g - h (skipped if i = j)
Exit:
```
□ 91

注意，就像汇编器完成存数/取数指令的数据地址计算一样，它也完成分支指令的地址计算，这使得编译器和汇编语言程序员摆脱了乏味的地址计算任务（参见 2.12 节）。

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2 ^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative

	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1;
			else $s1 = 0
				Compare less than; for beq, bne

	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1;
			else $s1 = 0
				Compare less than unsigned

	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1;
			else $s1 = 0
				Compare less than constant

	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1;
			else $s1 = 0
				Compare less than constant unsigned

Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

FIGURE 2.1 MIPS assembly language revealed in this chapter. This information is also found in Column 1 of the MIPS Reference Data Card at the front of this book.

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?

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 byte addressing. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by 2^2 or 4 (see page 94 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.)

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

假设 i 和 k 存放在寄存器 $s3 和 $s5 中，数组 save 的基址存放在寄存器 $s6 中。求这段 C 程序对应的 MIPS 汇编代码。

答案

第一步需要将 save [i] 读入一个临时寄存器中。在读入之前，需要计算它的地址。在将 i 加到 save 数组基址以形成访存地址前，由于系统按照字节寻址的缘故，先要将 i 乘以4。幸运的是，我们可以使用逻辑左移指令实现这一乘法，因为左移2 位等价于乘4（见2.6 节）。需要在该指令前增加一个标签 Loop，以便在循环末端能够跳回该指令。

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

为了得到 save [i] 的地址，需要将 $t1 和 $s6 中 save 的基址相加：

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

现在可用该地址将 save [i] 读入一个临时寄存器中：

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

下一条指令执行循环判断，如果 save [i] ≠k 则退出循环：

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

再下一条指令将 i 加1：

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

在循环的末尾，程序跳转到循环的开始。随后增加了一个 Exit 标签，这样就完成了全部编译：

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

（见练习题中对该指令序列的优化。） □

硬件/软件接口 以分支指令结束的这类指令序列对编译非常重要，因此它们有对应的专用术语：基本块。基本块（basic block）是没有分支（可能出现在末尾者除外）并且没有分支目标/分支标签（可能出现在开始者除外）的指令序列。编译最初阶段的任务之一就是将程序分解为若干基本块。