Implementing Conditional Statements

Introduction

We're going to translate some easy conditional statements.

```
if ( i == j )
  i++ ;
j-- ;
```

Translating conditional statements is interesting. In C, for example, when the condition is true, you execute the body. This is the *fall-through* case. That is, you execute the next statement. When the condition is false, you don't execute the body, you jump over it. This is the *jump* case.

Therefore, you jump when the condition is false.

In ISA programming, you jump when the condition is true. Thus, we often need to negate the condition.

Here's the translation of the above if-statement, assuming **$r1** stores **i** and **$r2** stores **j**.

```
    bne  $r1, $r2, L1    # branch if ! ( i == j )
    addi $r1, $r1, 1     # i++
L1: addi $r2, $r2, -1    # j--
```

The label L1 has the same address as the instruction immediately following the colon. Thus, the above code is the same as:

```
    bne  $r1, $r2, L1    # branch if ! ( i == j )
    addi $r1, $r1, 1     # i++
L1:
    addi $r2, $r2, -1    # j--
```

Even though it appears that label **L1** has an empty instruction, it doesn't. It is still associated with the second **addi** instruction.

Translating if-else

Let's translate

```
if ( i == j )
  i++ ;
else
  j-- ;
j += i ;
```

Let's think about what happens. As before, if the condition is false, we want to jump. This time, we want to jump to the *else*. Thus, we write the code like:

```
      bne  $r1, $r2, ELSE   # branch if ! ( i == j )
      addi $r1, $r1, 1      # i++
ELSE: addi $r2, $r2, -1     # j--
```

This code is wrong however. Why? The if-body contains **i++**. Once we're done with that, we need to jump over the else-body. This jump is unconditional, so we use **j** instruction to jump over the else-body.

```
      bne  $r1, $r2, ELSE   # branch if ! ( i == j )
      addi $r1, $r1, 1      # i++
      j L1                  # jump over else  (ADD THIS!!!)
ELSE: addi $r2, $r2, -1     # j--
L1:   add $r2, $r2, $r1     # j += i
```

Translating if-else with &&

Translating **&&** is interesting because there's short-circuiting involved.

To see how this works, let's translate:

```
if ( i == j && i == k )
  i++ ;  // if-body
else
  j-- ;  // else-body
j = i + k ;
```

To make this easier to read, let **<cond1>** stand for **i == j** and **<cond2>** stand for **i == k**.

```
if ( <cond1> && <cond2> )
  i++ ;   // if-body
else
  j-- ;   // else-body
j = i + k ;
```

Short-circuiting occurs when **<cond1>** evaluates to false. The control-flow then *jumps* over **<cond2>** (that is, **<cond2>** is not evaluated), and continues executing in the else-body.

If **<cond1>** evaluates to true, we want to fall-through and check **<cond2>**. If **<cond2>** evaluates false, we again *jump*, this time over the if-body, and to the else-body.

If **<cond2>** is true, we fall-through to the if-body.

Notice that we jump when the condition evaluates to false for both cases, so we'll be interested in jumping on negations of conditions.

Here's the translated code, assuming **$r3** stores **k**.

```
      bne  $r1, $r2, ELSE   # cond1: branch if ! ( i == j )
      bne  $r1, $r3, ELSE   # cond2: branch if ! ( i == k )
      addi $r1, $r1, 1      # if-body: i++
      j L1                  # jump over else
ELSE: addi $r2, $r2, -1     # else-body: j--
L1:   add $r2, $r1, $r3     # j = i + k
```

Usually, it's good to comment each line of assembly code. We've commented a little more than usual (by adding **cond1**, **cond2**, **if-body**, **else-body**) to make it easier to understand.

Assembly language code is often much harder to read, so commenting every line is not unusual. The idea is to comment it with C code, whenever possible.

Translating if-else with ||

Translating **&&** is interesting because there's short-circuiting involved, and the short-circuiting for || is even more interesting than for &&.

Let's translate

```
if ( i == j || i == k )
  i++ ;   // if-body
else
  j-- ;   // else-body
j = i + k ;
```

Again, let's use **<cond1>** to stand for **i == j** and **<cond2>** to stand for **i == k**.

```
if ( <cond1> || <cond2> )
  i++ ;  // if-body
else
  j-- ;  // else-body
j = i + k ;
```

Short-circuiting occurs when **<cond1>** evaluates to true. That is, we want to *jump* over checking the second condition and into the if-body. Notice that we go to the if-body when the condition evaluates to true. When the operator was &&, we jumped to the *else-body* when **<cond1>** evaluated to false.

If **<cond1>** is false, we want to fall-through and check **<cond2>**. If **<cond2>** is false, we now *jump* to the else-body.

If **<cond2>** is true, we fall-through to the if-body.

Notice that we jump when **<cond1>** evaluates to true (to the **if-body**) and when **<cond2>** evaluates to false (to the else-body).

Here's the translated code:

```
      beq  $r1, $r2, IF     # cond1: branch if ( i == j )
      bne  $r1, $r3, ELSE   # cond2: branch if ! ( i == k )
IF:   addi $r1, $r1, 1      # if-body: i++
      j L1                  # jump over else
ELSE: addi $r2, $r2, -1     # else-body: j--
L1:   add $r2, $r1, $r3     # j = i + k
```

switch statements

switch statements are interesting. One should note that false, we now *jump* to the else-bodyswitch works on a very limited number of types (int and char, primarily). It doesn't work on strings (even if students wish they did).

switch evaluates case-by-case. When one condition fails, the next is checked. When a condition is true, the code associated with that condition is run. However, if you don't put **break** the code for the next condition will also run. Unfortunately, most people expect a **break** to occur when the condition is done. They don't expect a fall-through case.

Here's an example of a switch.

```
switch( i ) {
case 1: i++ ;  // falls through
case 2: i += 2 ;
        break;
case 3: i += 3 ;
}
```

Here's the translated code.

```
         addi $r4, $r0, 1      # set temp to 1
         bne $r1, $r4, C2_COND # case 1 false: branch to case 2 cond
         j C1_BODY             # case 1 true: branch to case 1
C2_COND: addi $r4, $r0, 2      # set temp to 2
         bne $r1, $r4, C3_COND # case 2 false: branch to case 2 cond
         j C2_BODY             # case 2 true: branch to case 2 body
C3_COND: addi $r4, $r0, 3      # set temp to 3
         bne $r1, $r4, EXIT    # case 3 false: branch to exit
         j C3_BODY             # case 3 true: branch to case 3 body
C1_BODY: addi $r1, $r1, 1      # case 1 body: i++
C2_BODY: addi $r1, $r1, 2      # case 2 body: i += 2
         j EXIT                # break
C3_BODY: addi $r1, $r1, 3      # case 3 body: i += 3
EXIT:
```

Notice that EXIT, the last label, doesn't have an instruction after it. That's OK. The assembler can still figure out the address of EXIT (by pretending there is an instruction there).

Translating switch, at least as I've done it, involves a large prelude. The first half of the code determines where to jump to. The second half is pretty much the code itself.

The real problem is the fall-through cases. When **case 1** is true, you want to run **i++**, then **i += 2**. You don't want to test for **case 2**, because that's now how the semantics of **switch** works.

bge, bgt, blt, ble

bge, **bgt**, **blt**, **ble** are all pseudo-instructions. That is, there is no corresponding machine code to these instructions. Nevertheless, you can use them in assembly language programs because most assemblers support these pseudo-instructions. They translate them to real instructions.

Let's see an example of how the assembler might translate **bge**. The key is to use **slt** which means "set on less than". Here is the syntax of **slt**.

```
slt $r1, $r2, $r3   # R[1] = R[2] < R[3] ? 1 : 0
```

The semantics are shown in the comments: if **R[2] < R[3]** false, we now *jump* to the else-bodythen **R[1] = 1**, otherwise it's assigned to 0.

Here is the syntax and semantics of **bge**:

```
bge $r1, $r2, LABEL   # jump to LABEL if R[1] >= R[2]
```

If **R[1] >= R[2]** we know that this is equivalent to **!(R[1] < R[2])**. Thus, if we check **R[1] < R[2]** using **slt**, we expect it to be false.

Here's the translation of **bge**.

```
slt $r3, $r1, $r2     # check if R[1] < R[2]
beq $r3, $r0, LABEL   # branch if previous condition is false
```

As an exercise, you should translate the other three pseudo-instructions (and you really should, otherwise, you will waste time on an exam trying to figure it out---working it out ahead of time will let you gain speed while answering such questions on an exam).

Using DeMorgan's Law

In order to branch on negated condition, you need to know the negation of various conditions. Here's a chart:

Condition	Negated Condition
x > y	x <= y
x >= y	x < y
x < y	x >= y
x <= y	x > y
<cond1> && <cond1>	! <cond1> || ! <cond2>
<cond1> || <cond1>	! <cond1> && ! <cond2>

The last two are DeMorgan's Law. Every computer science major should know the above chart by heart.

Summary

As you translate conditional statements, you will see that they require branch statements. The branches are often branching on the negation of the condition, which is why you use the above table.

In the end, to make sure you are translating correctly, do the following:

- Don't try to understand the code too much. Don't optimize, or notice "there's an infinite loop". Just translate it like a (non-optimizing) compiler would.
- Trace the code, to see if it behaves correctly. It's easy to write the code, but tracing gives you some confidence you've written it correctly.
- Try to remember the patterns of translating conditions. You would like to look at C code, and know exactly the strategy you want to use, instead of reinventing the wheel. It's surprising, when exams are graded to notice students not following the examples as done in class, but trying to figure out some contorted versions of their own. This often leads to overly lengthy, and possibly incorrect code.