7.4.2. if Statements in Assembly

Let's take a look at the `getSmallest` function in assembly. For convenience, the function is reproduced below.

```c
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) {
        smallest = y;
    }
    else {
        smallest = x;
    }
    return smallest;
}
```

The corresponding assembly code extracted from GDB looks similar to the following:

```
(gdb) disas getSmallest
Dump of assembler code for function getSmallest:
   0x40059a <+4>:   mov    %edi,-0x14(%rbp)
   0x40059d <+7>:   mov    %esi,-0x18(%rbp)
   0x4005a0 <+10>:  mov    -0x14(%rbp),%eax
   0x4005a3 <+13>:  cmp    -0x18(%rbp),%eax
   0x4005a6 <+16>:  jle    0x4005b0 <getSmallest+26>
   0x4005a8 <+18>:  mov    -0x18(%rbp),%eax
   0x4005ae <+24>:  jmp    0x4005b9 <getSmallest+35>
   0x4005b0 <+26>:  mov    -0x14(%rbp),%eax
   0x4005b9 <+35>:  pop    %rbp
   0x4005ba <+36>:  retq
```

This is a different view of the assembly code than we have seen before. Here, we can see the *address* associated with each instruction, but not the *bytes*. Note that this assembly segment has been lightly edited for the sake of simplicity. The instructions that are normally part of function creation (i.e., `push %rbp`, `mov %rsp, %rbp`) are removed. By convention, GCC places the first and second parameters of a function in registers `%rdi` and `%rsi`, respectively. Since the parameters to `getSmallest` are of type `int`, the compiler places the parameters in the respective component registers `%edi` and `%esi` instead. For the sake of clarity, we refer to these parameters as `x` and `y`, respectively.

Let’s trace through the first few lines of the previous assembly code snippet. Note that we will not draw out the stack explicitly in this example. We leave this as an exercise for the reader, and encourage you to practice your stack tracing skills by drawing it out yourself.

- The first `mov` instruction copies the value located in register `%edi` (the first parameter, `x`) and places it at memory location `%rbp-0x14` on the call stack. The instruction pointer (`%rip`) is set to the address of the next instruction, or 0x40059d.
- The second `mov` instruction copies the value located in register `%esi` (the second parameter, `y`) and places it at memory location `%rbp-0x18` on the call stack. The instruction pointer (`%rip`) updates to point to the address of the next instruction, or 0x4005a0.
- The third `mov` instruction copies `x` to register `%eax`. Register `%rip` updates to point to the address of the next instruction in sequence.
- The `cmp` instruction compares the value at location `%rbp-0x18` (the second parameter, `y`) to `x` and sets appropriate condition code flag registers. Register `%rip` advances to the address of the next instruction, or 0x4005a6.
- The `jle` instruction at address 0x4005a6 indicates that if `x` is less than or equal to `y`, the next instruction that should execute should be at location `<getSmallest+26>` and that `%rip` should be set to address 0x4005b0. Otherwise, `%rip` is set to the next instruction in sequence, or 0x4005a8.

The next instructions to execute depend on whether the program follows the branch (i.e., executes the jump) at address 0x4005a6. Let’s first suppose that the branch was *not* followed. In this case, `%rip` is set to 0x4005a8 (i.e., `<getSmallest+18>`) and the following sequence of instructions executes:

- The `mov -0x18(%rbp), %eax` instruction at `<getSmallest+18>` copies `y` to register `%eax`. Register `%rip` advances to 0x4005ae.
- The `jmp` instruction at `<getSmallest+24>` sets register `%rip` to address 0x4005b9.
- The last instructions to execute are the `pop %rbp` instruction and the `retq` instruction, which cleans up the stack and returns from the function call. In this case, `y` is in the return register.

Now, suppose that the branch was taken at `<getSmallest+16>`. In other words, the `jle` instruction sets register `%rip` to 0x4005b0 (`<getSmallest+26>`). Then, the next instructions to execute are:

- The `mov -0x14(%rbp),%eax` instruction at address 0x4005b0 copies `x` to register `%eax`. Register `%rip` advances to 0x4005b9.
- The last instructions that execute are `pop %rbp` and `retq`, which clean up the stack and returns the value in the return register. In this case, component register `%eax` contains `x`, and `getSmallest` returns `x`.

We can then annotate the preceding assembly as follows:

```
0x40059a <+4>:  mov %edi,-0x14(%rbp)          # copy x to %rbp-0x14
0x40059d <+7>:  mov %esi,-0x18(%rbp)          # copy y to %rbp-0x18
0x4005a0 <+10>: mov -0x14(%rbp),%eax          # copy x to %eax
0x4005a3 <+13>: cmp -0x18(%rbp),%eax          # compare x with y
0x4005a6 <+16>: jle 0x4005b0 <getSmallest+26> # if x<=y goto
<getSmallest+26>
0x4005a8 <+18>: mov -0x18(%rbp),%eax          # copy y to %eax
0x4005ae <+24>: jmp 0x4005b9 <getSmallest+35> # goto <getSmallest+35>
0x4005b0 <+26>: mov -0x14(%rbp),%eax          # copy x to %eax
0x4005b9 <+35>: pop %rbp                      # restore %rbp (clean up
stack)
0x4005ba <+36>: retq                          # exit function (return
%eax)
```

Translating this back to C code yields:

Table 1. Translating `getSmallest()` into goto C form and C code.

goto Form	Translated C code

<pre>int getSmallest(int x, int y) {
    int smallest;
    if (x <= y) {
        goto assign_x;
    }
    smallest = y;
    goto done;
 	
assign_x:
    smallest = x;
 	
done:
    return smallest;
}</pre>
	<pre>int getSmallest(int x, int y) {
	    int smallest;
	    if (x <= y) {
	        smallest = x;
	    }
	    else {
	        smallest = y;
	    }
	    return smallest;
	}</pre>

In Table 1, the variable `smallest` corresponds to register `%eax`. If `x` is less than or equal to `y`, the code executes the statement `smallest = x`, which is associated with the `goto` label `assign_x` in our `goto` form of this function. Otherwise, the statement `smallest = y` is executed. The `goto` label `done` is used to indicate that the value in `smallest` should be returned.

Notice that the preceding C translation of the assembly code is a bit different from the original `getSmallest` function. These differences don't matter; close inspection of both functions reveals that the two programs are logically equivalent. However, the compiler first converts any `if` statement into an equivalent `goto` form, which results in the slightly different, but equivalent, version. Table 2 shows the standard `if` statement format and its equivalent `goto` form:

Table 2. Standard if statement format and its equivalent goto form.

<table>
<tr><th>C if statement</th><th>Compiler's equivalent goto form</th></tr>
<tr><td><pre><code>if (condition) {
    then_statement;
}
else {
    else_statement;
}</code></pre></td><td><pre><code>   if (!condition) {
        goto else;
    }
    then_statement;
    goto done;
else:
    else_statement;
done:</code></pre></td></tr>
</table>

Compilers translating code into assembly designate a jump when a condition is true. Contrast this behavior with the structure of an `if` statement, where a "jump" (to the `else`) occurs when conditions are *not* true. The `goto` form captures this difference in logic.

Considering the original `goto` translation of the `getSmallest` function, we can see that:

- `x <= y` corresponds to `!condition`.
- `smallest = x` is the `else_statement`.
- The line `smallest = y` is the `then_statement`.
- The last line in the function is `return smallest`.

Rewriting the original version of the function with the preceding annotations yields:

```c
int getSmallest(int x, int y) {
    int smallest;
    if (x > y) {     //!(x <= y)
        smallest = y; //then_statement
    }
    else {
        smallest = x; //else_statement
```

```
    }
    return smallest;
}
```

This version is identical to the original `getSmallest` function. Keep in mind that a function written in different ways at the C code level can translate to the same set of assembly instructions.

The cmov Instructions

The last set of conditional instructions we cover are **conditional move** (`cmov`) instructions. The `cmp` , `test` , and `jmp` instructions implement a *conditional transfer of control* in a program. In other words, the execution of the program branches in many directions. This can be very problematic for optimizing code, because these branches are very expensive.

In contrast, the `cmov` instruction implements a *conditional transfer of data*. In other words, both the `then_statement` and `else_statement` of the conditional are executed, and the data is placed in the appropriate register based on the result of the condition.

The use of C's **ternary expression** often results in the compiler generating a `cmov` instruction in place of jumps. For the standard if-then-else statement, the ternary expression has the form:

```c
result = (condition) ? then_statement : else_statement;
```

Let's use this format to rewrite the `getSmallest` function as a ternary expression. Keep in mind that this new version of the function behaves exactly as the original `getSmallest` function:

```c
int getSmallest_cmov(int x, int y) {
    return x > y ? y : x;
}
```

Although this may not seem like a big change, let's look at the resulting assembly. Recall that the first and second parameters (`x` and `y`) are stored in registers `%edi` and `%esi` , respectively.

```
0x4005d7 <+0>:   push   %rbp              #save %rbp
0x4005d8 <+1>:   mov    %rsp,%rbp         #update %rbp
0x4005db <+4>:   mov    %edi,-0x4(%rbp)   #copy x to %rbp-0x4
0x4005de <+7>:   mov    %esi,-0x8(%rbp)   #copy y to %rbp-0x8
0x4005e1 <+10>:  mov    -0x8(%rbp),%eax   #copy y to %eax
0x4005e4 <+13>:  cmp    %eax,-0x4(%rbp)   #compare x and y
0x4005e7 <+16>:  cmovle -0x4(%rbp),%eax   #if (x <=y) copy x to %eax
```

```
0x4005eb <+20>:  pop    %rbp              #restore %rbp
0x4005ec <+21>:  retq                     #return %eax
```

This assembly code has no jumps. After the comparison of `x` and `y`, `x` moves into the return register only if `x` is less than or equal to `y`. Like the jump instructions, the suffix of the `cmov` instructions indicates the condition on which the conditional move occurs. Table 3 lists the set of conditional move instructions.

Table 3. The cmov Instructions.

Signed	Unsigned	Description
`cmove` (`cmovz`)		move if equal (==)
`cmovne` (`cmovnz`)		move if not equal (!=)
`cmovs`		move if negative
`cmovns`		move if non-negative
`cmovg` (`cmovnle`)	`cmova` (`cmovnbe`)	move if greater (>)
`cmovge` (`cmovnl`)	`cmovae` (`cmovnb`)	move if greater than or equal (>=)
`cmovl` (`cmovnge`)	`cmovb` (`cmovnae`)	move if less (<)
`cmovle` (`cmovng`)	`cmovbe` (`cmovna`)	move if less than or equal (<=)

In the case of the original `getSmallest` function, the compiler's internal optimizer (see chapter 12) will replace the jump instructions with a `cmov` instruction if level 1 optimizations are turned on (i.e., `-O1`):

```
#compiled with: gcc -O1 -o getSmallest getSmallest.c
<getSmallest>:
   0x400546 <+0>: cmp    %esi,%edi      #compare x and y
   0x400548 <+2>: mov    %esi,%eax      #copy y to %eax
   0x40054a <+4>: cmovle %edi,%eax      #if (x<=y) copy x to %eax
   0x40054d <+7>: retq                  #return %eax
```

In general, the compiler is very cautious about optimizing jump instructions into `cmov` instructions, especially in cases where side effects and pointer values are involved. Table 4 shows two equivalent ways of writing a function, `incrementX`:

Table 4. Two functions that attempt to increment the value of integer x.

C code

```c
int incrementX(int *x) {
    if (x != NULL) { //if x is not NULL
        return (*x)++; //increment x
    }
    else { //if x is NULL
        return 1; //return 1
    }
}
```

C ternary form

```c
int incrementX2(int *x){
    return x ? (*x)++ : 1;
}
```

Each function takes a pointer to an integer as input and checks if it is `NULL`. If `x` is not `NULL`, the function increments and returns the dereferenced value of `x`. Otherwise, the function returns the value 1.

It is tempting to think that `incrementX2` uses a `cmov` instruction since it uses a ternary expression. However, both functions yield the exact same assembly code:

```
0x4005ed <+0>:   push   %rbp
0x4005ee <+1>:   mov    %rsp,%rbp
0x4005f1 <+4>:   mov    %rdi,-0x8(%rbp)
0x4005f5 <+8>:   cmpq   $0x0,-0x8(%rbp)
0x4005fa <+13>:  je     0x40060d <incrementX+32>
0x4005fc <+15>:  mov    -0x8(%rbp),%rax
0x400600 <+19>:  mov    (%rax),%eax
0x400602 <+21>:  lea    0x1(%rax),%ecx
0x400605 <+24>:  mov    -0x8(%rbp),%rdx
0x400609 <+28>:  mov    %ecx,(%rdx)
0x40060b <+30>:  jmp    0x400612 <incrementX+37>
0x40060d <+32>:  mov    $0x1,%eax
0x400612 <+37>:  pop    %rbp
0x400613 <+38>:  retq
```

Recall that the `cmov` instruction *executes both branches of the conditional*. In other words, `x` gets dereferenced no matter what. Consider the case where `x` is a null pointer. Recall that dereferencing a null pointer leads to a null pointer exception in the code, causing a segmentation fault. To prevent any chance of this happening, the compiler takes the safe road and uses jumps.