7.4.3. Loops in Assembly

Like `if` statements, loops in assembly are also implemented using jump instructions. However, loops enable instructions to be *revisited* based on the result of an evaluated condition.

The `sumUp` function shown in the following example sums up all the positive integers from 1 to a user-defined integer. This code is intentionally written suboptimally to illustrate a `while` loop in C.

C

```c
int sumUp(int n) {
    //initialize total and i
    int total = 0;
    int i = 1;

    while (i <= n) {  //while i is less than or equal to n
        total += i;   //add i to total
        i++;          //increment i by 1
    }
    return total;
}
```

Compiling this code and disassembling it using GDB yields the following assembly code:

```
Dump of assembler code for function sumUp:
0x400526 <+0>:   push   %rbp
0x400527 <+1>:   mov    %rsp,%rbp
0x40052a <+4>:   mov    %edi,-0x14(%rbp)
0x40052d <+7>:   mov    $0x0,-0x8(%rbp)
0x400534 <+14>:  mov    $0x1,-0x4(%rbp)
0x40053b <+21>:  jmp    0x400547 <sumUp+33>
0x40053d <+23>:  mov    -0x4(%rbp),%eax
0x400540 <+26>:  add    %eax,-0x8(%rbp)
0x400543 <+29>:  add    $0x1,-0x4(%rbp)
0x400547 <+33>:  mov    -0x4(%rbp),%eax
0x40054a <+36>:  cmp    -0x14(%rbp),%eax
0x40054d <+39>:  jle    0x40053d <sumUp+23>
0x40054f <+41>:  mov    -0x8(%rbp),%eax
0x400552 <+44>:  pop    %rbp
0x400553 <+45>:  retq
```

Again, we will not draw out the stack explicitly in this example. However, we encourage readers to draw the stack out themselves.

The First Five Instructions

The first five instructions of this function set the stack up for function execution and set up temporary values for function execution:

```
0x400526 <+0>:  push %rbp                # save %rbp onto the stack
0x400527 <+1>:  mov  %rsp,%rbp           # update the value of %rbp (new
frame)
0x40052a <+4>:  mov  %edi,-0x14(%rbp)  # copy n to %rbp-0x14
0x40052d <+7>:  mov  $0x0,-0x8(%rbp)   # copy 0 to %rbp-0x8 (total)
0x400534 <+14>: mov  $0x1,-0x4(%rbp)   # copy 1 to %rbp-0x4 (i)
```

Recall that stack locations store *temporary variables* in a function. For simplicity we will refer to the location marked by `%rbp-0x8` as `total` and `%rbp - 0x4` as `i`. The input parameter to `sumUp` (`n`) is moved to stack location `%rbp-0x14`. Despite the placement of temporary variables on the stack, keep in mind that the stack pointer has not changed after the execution of the first instruction (i.e., `push %rbp`).

The Heart of the Loop

The next seven instructions in the `sumUp` function represent the heart of the loop:

```
0x40053b <+21>:  jmp    0x400547 <sumUp+33>  # goto <sumUp+33>
0x40053d <+23>:  mov    -0x4(%rbp),%eax      # copy i to %eax
0x400540 <+26>:  add    %eax,-0x8(%rbp)      # add i to total (total += i)
0x400543 <+29>:  add    $0x1,-0x4(%rbp)      # add 1 to i (i += 1)
0x400547 <+33>:  mov    -0x4(%rbp),%eax      # copy i to %eax
0x40054a <+36>:  cmp    -0x14(%rbp),%eax     # compare i to n
0x40054d <+39>:  jle    0x40053d <sumUp+23>  # if (i <= n) goto <sumUp+23>
```

- The first instruction is a direct jump to `<sumUp+33>`, which sets the instruction pointer (`%rip`) to address 0x400547.
- The next instruction that executes is `mov -0x4(%rbp),%eax`, which places the value of `i` in register `%eax`. Register `%rip` is updated to 0x40054a.
- The `cmp` instruction at `<sumUp+36>` compares `i` to `n` and sets the appropriate condition code registers. Register `%rip` is set to 0x40054d.

The `jle` instruction then executes. The instructions that execute next depend on whether or not the branch is taken.

Suppose that the branch *is* taken (i.e., `i <= n` is true). Then the instruction pointer is set to 0x40053d and program execution jumps to `<sumUp+23>`. The following instructions then execute in sequence:

- The `mov` instruction at `<sumUp+23>` copies `i` to register `%eax`.
- The `add %eax,-0x8(%rbp)` adds `i` to `total` (i.e., `total += i`).
- The `add` instruction at `<sumUp+29>` then adds 1 to `i` (i.e., `i += 1`).
- The `mov` instruction at `<sumUp+33>` copies the updated value of `i` to register `%eax`.
- The `cmp` instruction then compares `i` to `n` and sets the appropriate condition code registers.
- Next, `jle` executes. If `i` is less than or equal to `n`, program execution once again jumps to `<sumUp+23>` and the loop (defined between `<sumUp+23>` and `<sumUp+39>`) repeats.

If the branch is *not* taken (i.e., `i` is *not* less than or equal to `n`), the following instructions execute:

```
0x40054f <+41>:  mov    -0x8(%rbp),%eax     # copy total to %eax
0x400552 <+44>:  pop    %rbp                # restore rbp
0x400553 <+45>:  retq                       # return (total)
```

These instructions copy `total` to register `%eax`, restore `%rbp` to its original value, and exit the function. Thus, the function returns `total` upon exit.

Table 1 shows the assembly and C goto forms of the `sumUp` function:

Table 1. Translating sumUp into goto C form.

Assembly	Translated goto Form

Assembly

```
<sumUp>:
   <+0>:   push   %rbp
   <+1>:   mov    %rsp,%rbp
   <+4>:   mov    %edi,-0x14(%rbp)
   <+7>:   mov    $0x0,-0x8(%rbp)
   <+14>:  mov    $0x1,-0x4(%rbp)
   <+21>:  jmp    0x400547
<sumUp+33>
   <+23>:  mov    -0x4(%rbp),%eax
   <+26>:  add    %eax,-0x8(%rbp)
   <+29>:  add    $0x1,-0x4(%rbp)
   <+33>:  mov    -0x4(%rbp),%eax
   <+36>:  cmp    -0x14(%rbp),%eax
   <+39>:  jle    0x40053d
<sumUp+23>
   <+41>:  mov    -0x8(%rbp),%eax
   <+44>:  pop    %rbp
   <+45>:  retq
```

Translated goto Form

C

```c
int sumUp(int n) {
    int total = 0;
    int i = 1;
    goto start;
body:
    total += i;
    i += 1;
start:
    if (i <= n) {
        goto body;
    }
    return total;
}
```

The preceding code is also equivalent to the following C code without `goto` statements:

C

```c
int sumUp(int n) {
    int total = 0;
    int i = 1;
    while (i <= n) {
        total += i;
        i += 1;
    }
    return total;
}
```

for Loops in Assembly

The primary loop in the `sumUp` function can also be written as a `for` loop:

C

```c
int sumUp2(int n) {
    int total = 0;             //initialize total to 0
    int i;
    for (i = 1; i <= n; i++) { //initialize i to 1, increment by 1 while
i<=n
        total += i;            //updates total by i
    }
    return total;
}
```

This version yields assembly code identical to our `while` loop example. We repeat the assembly code below and annotate each line with its English translation:

```
Dump of assembler code for function sumUp2:
0x400554 <+0>:   push   %rbp                   #save %rbp
0x400555 <+1>:   mov    %rsp,%rbp              #update %rpb (new stack
frame)
0x400558 <+4>:   mov    %edi,-0x14(%rbp)       #copy %edi to %rbp-0x14 (n)
0x40055b <+7>:   movl   $0x0,-0x8(%rbp)        #copy 0 to %rbp-0x8 (total)
0x400562 <+14>:  movl   $0x1,-0x4(%rbp)        #copy 1 to %rbp-0x4 (i)
0x400569 <+21>:  jmp    0x400575 <sumUp2+33>   #goto <sumUp2+33>
0x40056b <+23>:  mov    -0x4(%rbp),%eax        #copy i to %eax [loop]
0x40056e <+26>:  add    %eax,-0x8(%rbp)        #add i to total (total+=i)
0x400571 <+29>:  addl   $0x1,-0x4(%rbp)        #add 1 to i (i++)
0x400575 <+33>:  mov    -0x4(%rbp),%eax        #copy i to %eax [start]
0x400578 <+36>:  cmp    -0x14(%rbp),%eax       #compare i with n
0x40057b <+39>:  jle    0x40056b <sumUp2+23>   #if (i <= n) goto loop
0x40057d <+41>:  mov    -0x8(%rbp),%eax        #copy total to %eax
0x400580 <+44>:  pop    %rbp                   #prepare to leave the
function
0x400581 <+45>:  retq                          #return total
```

To understand why the `for` loop version of this code results in identical assembly to the `while` loop version of the code, recall that the `for` loop has the following representation:

```
for ( <initialization>; <boolean expression>; <step> ){
    <body>
}
```

and is equivalent to the following `while` loop representation:

```C
<initialization>
while (<boolean expression>) {
    <body>
    <step>
}
```

Since every `for` loop can be represented by a `while` loop, the following two C programs are equivalent representations for the previous assembly:

Table 2. Equivalent ways to write the sumUp function.

<table>
<tr><th>For loop</th><th>While loop</th></tr>
<tr><td><pre>int sumUp2(int n) {
    int total = 0;
    int i = 1;
    for (i; i <= n; i++) {
        total += i;
    }
    return total;
}</pre></td><td><pre>int sumUp(int n){
    int total = 0;
    int i = 1;
    while (i <= n) {
        total += i;
        i += 1;
    }
    return total;
}</pre></td></tr>
</table>

Contents

7.4.3. Loops in Assembly