CS 107
Lecture 18:
Assembly Part IV

Wed., February 21st, 2024

Computer Systems
Winter 2024
Stanford University
Computer Science Department

Reading: Course Reader: x86-64 Assembly Language, Textbook: Chapter 3.1-3.4

Lecturer: Chris Gregg

```c
// Type your code here, or load an example.
void while_loop()
{
    int i=100;
    int total;
    while (i >=0) {
        total += i;
        i--;
    }
}
```

11010 | .LX0: | .text | //

```
while_loop():
  mov eax, 100
  jmp .L2
.L3:
  sub eax, 1
.L2:
  test eax, eax
  jns .L3
  rep ret
```

Today's Topics

- Logistics
- Reading: Course Reader: x86-64 Assembly Language;Textbook, Chapter 3.1-3.4
- Programs from class: `/afs/ir/class/cs107/samples/lect18`
- PDF handout in class: https://web.stanford.edu/class/cs107/lectures/15/asm-intro.pdf
- x86 reference sheet handout: https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf
- Introduction to x86 Assembly Language
  - Overview of assembly code and the weirdness of x86 (primarily historical)
    - First example: HelloWorld, gcc -S, gdbtui
    - Second Example: Looper
  - Registers
  - Data formats
  - Addressing Modes
  - The **`mov`** instruction
  - Access to variables of various types

Control

- So far, we have only been discussing "straight-line" code, where one instruction happens directly after the previous instruction.
- However, it is often necessary to perform one instruction or another instruction based on the logic in our programs, and assembly code gives us tools to do this.
- We can alter the flow of code using a "jump" instruction, which indicates that the next instruction will be somewhere else in the program (this is called a *branch*)
- We will start by discussing "condition codes" that are set when we do arithmetic (and other operations), and then we will talk about jump instructions to change control flow.

Condition Codes

- Besides the registers we have already discussed, the CPU has a separate set of single-bit *condition code* registers describing attributes of recent operations.
- We can use these registers (by testing them) to perform branches in the code.
- These are the most useful condition code registers:
  - **CF**: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
  - **ZF**: Zero flag. The most recent operation yielded zero.
  - **SF**: Sign flag. The most operation yielded a negative value.
  - **OF**: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

Condition Codes: Examples

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant b/t. Used to detect overflow for unsigned operations.
- **ZF**: Zero flag. The most recent operation yielded zero.
- **SF**: Sign flag. The most operation yielded a negative value.
- **OF**: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

```
int a = 5;
int b = -5;
int t = a + b;
```

Which flag above would be set?

The **ZF** flag.

Condition Codes: Examples

- **CF**: Carry flag. The most recent operation generated a carry out of the most significant b/t. Used to detect overflow for unsigned operations.
- **ZF**: Zero flag. The most recent operation yielded zero.
- **SF**: Sign flag. The most operation yielded a negative value.
- **OF**: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

```
int a = 5;
int b = -5;
int t = a + b;
```

Which flag above would be set?

The **ZF** flag.

```
int a = 5;
int b = -20;
int t = a + b;
```

Which flag above would be set?

The **SF** flag.

Condition Codes

- The **`leaq`** instruction does not set any condition codes (because it is intended for address computations), but the other arithmetic instructions we talked about do set them (**`inc`**, **`dec`**, **`neg`**, **`not`**, **`add`**, **`sub`**, **`imul`**, **`xor`**, **`or`**, **`and`**, **`shl`**, **`sar`**, **`shr`**, etc.)
- For logical operations (e.g., **`xor`**), the carry and overflow flags are set to 0.
- For shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero.
- **`inc`** and **`dec`** set the overflow and zero flags, but leave the carry flag unchanged (see here about a potential reason why).

`cmp` and `test`

- There are two types of instructions we can use that set the condition codes **without altering any other registers**, the `cmp` and `test` instructions:

Instruction		Based on	Description
`CMP`	S_1, S_2	S_2 - S_1	Compare
`cmpb`			Compare byte
`cmpw`			Compare word
`cmpl`			Compare double word
`cmpq`			Compare quad word
`TEST`	S_1, S_2	S_2 & S_1	Test
`testb`			Test byte
`testw`			Test word
`testl`			Test double word
`testq`			Test quad word

- By setting the condition codes, we can set up for a jump or other logic, based on some condition (e.g., whether a register has reached a certain value.
- Be careful! The operands for **`cmp`** are listed in reverse order! (**`cmp`** is based on the **`sub`** instruction)
- Often, we use **`testq %rax, %rax`** to see whether **`%rax`** is negative, zero, or positive.

Accessing the Condition Codes

- There are three common ways to use the condition codes:
  1. We can set a single byte to 0 or 1 depending on some combination of the condition codes.
  2. We can conditionally jump to some other part of the program.
  3. We can conditionally transfer data.

Accessing the Condition Codes

- There are three common ways to use the condition codes:
  1. We can set a single byte to 0 or 1 depending on some combination of the condition codes.
  2. We can conditionally jump to some other part of the program.
  3. We can conditionally transfer data.

Instruction	Synonym	Set Condition
`sete D`	`setz`	Equal / zero
`setne D`	`setnz`	Not equal / not zero
`sets D`		Negative
`setns D`		Nonnegative
`setg D`	`setnle`	Greater (signed >)
`setge D`	`setnl`	Greater or equal (signed >=)
`setl D`	`setnge`	Less (signed <)
`setle D`	`setng`	Less or equal (signed <=)
`seta D`	`setnbe`	Above (unsigned >)
`setae D`	`setnb`	Above or equal (unsigned >=)
`setb D`	`setnae`	Below (unsigned <)
`setbe D`	`setna`	Below or equal (unsigned <=)

Example: `a < b`

```
int comp(data_t a, data_t b)
a in %rdi, b in %rsi

comp:
cmpq %rsi, %rdi   # Compare a:b
setl %al          # Set low-order byte of
                  # %eax to 0 or 1
movzbl %al, %eax  # Clear rest of %eax
                  # (and rest of %rax)
ret
```

Accessing the Condition Codes

- There are three common ways to use the condition codes:
  1. We can set a single byte to 0 or 1 depending on some combination of the condition codes.
  2. We can conditionally jump to some other part of the program.
  3. We can conditionally transfer data.

Instruction	Synonym	Set Condition
`jmp` *`Label`*		Direct jump
`jmp` *`*Operand`*		Indirect jump
`je` *`Label`*	`jz`	Equal / zero (ZF=1)
`jne` *`Label`*	`jnz`	Not equal / not zero (ZF=0)
`js` *`Label`*		Negative (SF=1)
`jns` *`Label`*		Nonnegative (SF=0)
`jg` *`Label`*	`jnle`	Greater (signed >) (ZF=0 and SF=OF)
`jge` *`Label`*	`jnl`	Greater or equal (signed >=) (SF=OF)
`jl` *`Label`*	`jnge`	Less (signed <) (SF != OF)
`jle` *`Label`*	`jng`	Less or equal (signed <=) (ZF=1 or SF!=OF)
`ja` *`Label`*	`jnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`jae` *`Label`*	`jnb`	Above or equal (unsigned >=) (CF = 0)
`jb Label`	`jnae`	Below (unsigned <) (CF = 1)
`jbe Label`	`jna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

- Jump instructions jump to *labels* in assembly code, and those labels are changed to addresses (most often relative)
- **`jmp`** is an *unconditional* jump, meaning that the jump is always taken.
- Unconditional jumps can be direct *or* indirect
- Conditional jumps must be direct.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

Set `%eax` to 0

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

```
%rax: 0
```

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 0`

compare `%eax` to `0x63 (99d)` by subtracting `%eax - 0x63`, setting the Sign Flag (SF) because the result is negative.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 0`

`jle` is "jump less than or equal." The Sign Flag indicates that the result was negative (less than), so we jump to `0x7`.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 1`

Add 1 to `%eax`

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>:  cmp    $0x63,%eax
   0x000000000000000d <+13>:  jle    0x7 <loop+7>
   0x000000000000000f <+15>:  repz retq
End of assembler dump.
```

`%rax: 1`

Compare `%eax` to `0x63 (99d)` by subtracting `%eax - 0x63`. When `%rax` is 0, what flags change based on the the comparison? (We care about **Zero Flag**, **Sign Flag**, **Carry Flag**, and **Overflow Flag**): 0 - 99, so **SF** and **CF**

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:  mov    $0x0,%eax
   0x0000000000000005 <+5>:  jmp    0xa <loop+10>
   0x0000000000000007 <+7>:  add    $0x1,%eax
   0x000000000000000a <+10>: cmp    $0x63,%eax
   0x000000000000000d <+13>: jle    0x7 <loop+7>
   0x000000000000000f <+15>: repz retq
End of assembler dump.
```

`%rax: 1`

Eventually, this will become positive (when `%eax` is 100), and the loop will end.

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x0,%eax
   0x0000000000000005 <+5>:   jmp    0xa <loop+10>
   0x0000000000000007 <+7>:   add    $0x1,%eax
   0x000000000000000a <+10>: cmp    $0x63,%eax
   0x000000000000000d <+13>: jle    0x7 <loop+7>
   0x000000000000000f <+15>: repz retq
End of assembler dump.
```

Could the compiler have done better with this loop?

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

~~`gcc -c -Og while_loop.c`~~

`gcc -c -O1 while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x64,%eax
   0x0000000000000005 <+5>:   sub    $0x1,%eax
   0x0000000000000008 <+8>:   jne    0x5 <loop+5>
   0x000000000000000a <+10>:  repz retq
End of assembler dump.
```

Fewer lines, less jumping!

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

~~`gcc -c -Og while_loop.c`~~

`gcc -c -O1 while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   mov    $0x64,%eax
   0x0000000000000005 <+5>:   sub    $0x1,%eax
   0x0000000000000008 <+8>:   jne    0x5 <loop+5>
   0x000000000000000a <+10>:  repz retq
End of assembler dump.
```

Could we do better?

Jump instructions example

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

~~`gcc -c -Og while_loop.c`~~

~~`gcc -c -O1 while_loop.c`~~

`gcc -c -O2 while_loop.c`

```
$ gdb while_loop.o
The target architecture is assumed to be i386:x86-64
Reading symbols from while_loop.o...done.
(gdb) disas loop
Dump of assembler code for function loop:
   0x0000000000000000 <+0>:   repz retq
End of assembler dump.
```

Sure! As the optimization level goes up, **gcc** gets smarter! The compiler realized that this loop is not doing anything, so it completely optimized it out!

Practice: Reverse-engineer Assembly to C

- Take the following function:

```
long test(long x, long y, long z) {
    long val = __________;
    if (__________) {
        if (__________)
            val = __________;
        else
            val = __________;
    } else if (__________)
        val = __________;
    return val;
}
```

```
# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
```

Practice: Reverse-engineer Assembly to C

- Take the following function:

```c
long test(long x, long y, long z) {
    long val = x + y + z;
    if (x < -3) {
        if (y < z)
            val = x * y;
        else
            val = y * z;
    } else if (x > 2)
        val = x * z;
    return val;
}
```

```
# x in %rdi, y in %rsi, z in %rdx
test:
    leaq (%rdi, %rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    ret
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    rep; ret
```

Conditional Moves

- The x86 processor provides a set of "conditional move" instructions that move memory based on the result of the condition codes, and that are completely analogous to the jump instructions:

Instruction	Synonym	Move Condition
`cmove S,R`	`cmovz`	Equal / zero (ZF=1)
`cmovne S,R`	`cmovnz`	Not equal / not zero (ZF=0)
`cmovs S,R`		Negative (SF=1)
`cmovns S,R`		Nonnegative (SF=0)
`cmovg S,R`	`cmovnle`	Greater (signed >) (SF=0 and SF=OF)
`cmovge S,R`	`cmovnl`	Greater or equal (signed >=) (SF=OF)
`cmovl S,R`	`cmovnge`	Less (signed <) (SF != OF)
`cmovle S,R`	`cmovng`	Less or equal (signed <=) (ZF=1 or SF!=OF)
`cmova S,R`	`cmovnbe`	Above (unsigned >) (CF = 0 and ZF = 0)
`cmovae S,R`	`cmovnb`	Above or equal (unsigned >=) (CF = 0)
`cmovb S,R`	`cmovnae`	Below (unsigned <) (CF = 1)
`cmovbe S,R`	`cmovna`	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

- With these instructions, we can sometimes eliminate branches, which are particularly inefficient on modern computer hardware.

Jumps -vs- Conditional Move

```
long absdiff(long x, long y)
{
    long result;
    if (x < y)
        result = y - x;
    else
        result = x - y;
    return result;
}
```

```
long cmovdiff(long x, long y)
{
    long rval = y-x;
    long eval = x-y;
    long ntest = x >= y;
    if (ntest) rval = eval;
    return rval;
}
```

```
# x in %rdi, y in %rsi
absdiff:
    cmpq %rsi, %rdi
    jge .L2
    movq %rsi, %rax
    subq %rdi, %rax
    ret
.L2:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

Which is faster?
Let's test!

```
# x in %rdi, y in %rsi
cmovdiff:
    movq %rsi, %rax
    subq %rdi, %rax
    movq %rdi, %rdx
    subq %rsi, %rdx
    cmpq %rsi, %rdi
    cmovge %rdx, %rax
    ret
```

Extra Slides

Digging Deeper: Jump Instruction Encodings

- As we have mentioned before, assembly language is still one step higher than machine code.
- It is instructive in this case to look at the machine code for some jump instructions, just to see how the underlying machine is referencing where to jump.
- Remember, `%rip` is the *instruction pointer*, which has an address of the current instruction.
  - Well…kind of. On older x86 machines, when an instruction was executing, the first thing that happened was that `%rip` is changed to point to the next instruction. The instruction set has retained this behavior.
  - Jump instructions are often encoded to jump *relative to* `%rip`. Let's see what that means in practice…

Digging Deeper: Jump Instruction Encodings

- Let's look at our while loop again:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

Run the objdump program:

`objdump -d while_loop.o`

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

0-based addresses for each instruction (will be replaced with real addresses when a full program is created)

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

Machine code for the instructions. Instructions are "variable length" — the **`mov`** instruction is 5 bytes, the tmp is 3 bytes, etc.

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

`gcc -c -Og while_loop.c`

Run the objdump program:

`objdump -d while_loop.o`

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

The `jmp` instruction. "`eb`" means that this is a `jmp`, and `03` is the number of instructions to jump, relative to `%rip`. When the instruction is executing, `%rip` is set to the next instruction (`7` in this case). So…`7 + 3` is 0xa, so this instruction jumps to `0xa`.

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00          mov    $0x0,%eax
   5: eb 03                   jmp    a <loop+0xa>
   7: 83 c0 01                add    $0x1,%eax
   a: 83 f8 63                cmp    $0x63,%eax
   d: 7e f8                   jle    7 <loop+0x7>
   f: f3 c3                   repz retq
```

The **cmp** instruction. Notice that the 0x63 is embedded into the machine code, because it is an immediate value.

Digging Deeper: Jump Instruction Encodings

- Take the following function:

```
void loop()
{
    int i = 0;
    while (i < 100) {
        ++i;
    }
}
```

Compile to an object file:

```
gcc -c -Og while_loop.c
```

Run the objdump program:

```
objdump -d while_loop.o
```

```
Disassembly of section .text:

0000000000000000 <loop>:
   0: b8 00 00 00 00         mov    $0x0,%eax
   5: eb 03                  jmp    a <loop+0xa>
   7: 83 c0 01               add    $0x1,%eax
   a: 83 f8 63               cmp    $0x63,%eax
   d: 7e f8                  jle    7 <loop+0x7>
   f: f3 c3                  repz retq
```

The `jle` instruction. "`7e`" means that this is a `jle` (jump if less than), and `f8` is the number of instructions to jump (in two's complement! So, it means -8), relative to `%rip`, which is at `0xf` when the instruction is running. So, `0xf - 8` is 0xa, so this instruction jumps to `0x7`.