Lecture 12: Assembly Part II

Computer Systems
Fall 2017
Stanford University
Computer Science Department

Reading: Chapter 3.5-3.6

Lecturers: Julie Zelenski and Chris Gregg
Today's Topics

- Reading: Chapter 3.5-3.6
- Programs from class: /afs/ir/class/cs107/samples/lect12
- More x86 Assembly Language
  - Review of what we know so far
  - The \texttt{lea} instruction
  - \texttt{pushing} and \texttt{popping} from the stack
  - Unary operations, Binary operations, Shift operations
  - Special multiplication and division
  - Control
    - Condition codes
    - Conditional branches
- Logistics
  - Midterm Comments
What did we cover last Monday?

• Registers:
  • 16 regular integer registers, $%rax$, $%rbx$, ...
  • naming is historical, and a register has four nested parts:

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
<th>%ax</th>
<th>%al</th>
</tr>
</thead>
</table>

return value

• Operand forms: lots of ways we can refer to immediate values, register values, or memory:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
<th>Operand</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%rax</td>
<td>0x100</td>
<td>$%rax</td>
<td>0x100</td>
<td>Register</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>%rcx</td>
<td>0x1</td>
<td>4($%rax)</td>
<td>0xFF</td>
<td>Address 0x100</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%rdx</td>
<td>0x3</td>
<td>9($%rax,%rdx)</td>
<td>0x11</td>
<td>Address 0x10C</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td></td>
<td></td>
<td>260(%rcx,%rdx)</td>
<td>0x13</td>
<td>Address 0x108</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0xFC(%rcx,4)</td>
<td>0xFF</td>
<td>Address 0x100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(%%rax,%rdx,4)</td>
<td>0x11</td>
<td>Address 0x10C</td>
</tr>
</tbody>
</table>
What did we cover last Monday? (continued)

- Data movement instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>S, D</td>
<td>D ← S</td>
</tr>
<tr>
<td>movb</td>
<td></td>
<td>Move byte</td>
</tr>
<tr>
<td>movw</td>
<td></td>
<td>Move word</td>
</tr>
<tr>
<td>movl</td>
<td></td>
<td>Move double word</td>
</tr>
<tr>
<td>movq</td>
<td></td>
<td>Move quad word</td>
</tr>
<tr>
<td>movabsq</td>
<td>I, R</td>
<td>R ← I</td>
</tr>
</tbody>
</table>

- Examples:

1. `movl $0x4050,%eax` Immediate–Register, 4 bytes
2. `movw %bp,%sp` Register–Register, 2 bytes
3. `movb (%rdi,%rcx),%al` Memory–Register, 1 byte
4. `movb $-17,(%rsp)` Immediate–Memory, 1 byte
5. `movq %rax,-12(%rbp)` Register–Memory, 8 bytes
The `lea` instruction

The `lea` instruction is related to the `mov` instruction. It has the form of an instruction that reads from memory to a register, but it does not reference memory at all.

- It's first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination.
- You can think of it as the "&" operator in C — it retrieves the address of a memory location:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>leaq S,D</code></td>
<td>$D \leftarrow &amp;S$</td>
<td>Load effective address</td>
</tr>
</tbody>
</table>

Examples: if `%rax` holds value $x$ and `%rcx` holds value $y$:

- `leaq 6(%rax), %rdx` : %rdx now holds $x + 6$
- `leaq (%rax,%rcx), %rdx` : %rdx now holds $x + y$
- `leaq (%rax,%rcx,4), %rdx` : %rdx now holds $x + 4y$
- `leaq 7(%rax,%rax,8), %rdx` : %rdx now holds $7 + 9x$
- `leaq 0xA(,%rcx,4), %rdx` : %rdx now holds $10 + 4y$
- `leaq 9(%rax,%rcx,2), %rdx` : %rdx now holds $9 + x + 2y$
Pushing and Popping from the Stack

- As we have seen from stack-based memory allocation in C, the stack is an important part of our program, and assembly language has two built-in operations to use the stack.
- Just like the stack ADT, they have a first-in, first-out discipline.
- By convention, we draw stacks upside down, and the stack "grows" downward.

![](image)

Stack "bottom"

Stack "top"

Increasing address

0x108
The push and pop operations write and read from the stack, and they also modify the stack pointer, `%rsp`:

<table>
<thead>
<tr>
<th>Instruct</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushq</td>
<td>R[<code>%rsp</code>] ← R[<code>%rsp</code>]-8; M[R[<code>%rsp</code>]] ← S</td>
<td>Push quad word</td>
</tr>
<tr>
<td>popq</td>
<td>D ← M[R[<code>%rsp</code>]]; R[<code>%rsp</code>] ← R[<code>%rsp</code>]+8</td>
<td>Push quad word</td>
</tr>
</tbody>
</table>

![Stack diagram](image)

- Stack "top"
- Stack "bottom"
- Increasing address
- 0x108
Pushing and Popping from the Stack

• Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0x123</td>
</tr>
<tr>
<td>%rdx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"

Increasing address

0x108

Stack "top"
• Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th>pushq %rax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax 0x123</td>
<td>%rax 0x123</td>
</tr>
<tr>
<td>%rdx 0</td>
<td>%rdx 0</td>
</tr>
<tr>
<td>%rsp 0x108</td>
<td>%rsp 0x100</td>
</tr>
</tbody>
</table>

Stack "bottom"  
0x108  
Increasing address  
Stack "top"  
0x108  

Stack "bottom"  
0x100  
Increasing address  
0x123  
Stack "top"
Pushing and Popping from the Stack

- Example:

<table>
<thead>
<tr>
<th>Initially</th>
<th>pushq $rax</th>
<th>popq $rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>$rax 0x123</td>
<td>$rax 0x123</td>
<td>$rax 0x123</td>
</tr>
<tr>
<td>$rdx 0</td>
<td>$rdx 0</td>
<td>$rdx 0x123</td>
</tr>
<tr>
<td>$rsp 0x108</td>
<td>$rsp 0x100</td>
<td>$rsp 0x108</td>
</tr>
</tbody>
</table>

Stack "bottom"  Stack "top"  Stack "bottom"

Increasing address  Increasing address  Increasing address

0x108  0x100  0x100

0x123
Pushing and Popping from the Stack

- As you can tell, pushing a quad word onto the stack involves first decrementing the stack pointer by 8, and then writing the value at the new top-of-stack address.
- Therefore, the behavior of the instruction `pushq %rbp` is equivalent to the pair of instructions:
  - `subq $8, %rsp` (subq is a subtraction, and this decrements the stack pointer)
  - `movq %rbp,(%rsp)` (Store %rbp on the stack)
- The behavior of the instruction `popq %rax` is equivalent to the pair of instructions:
  - `movq (%rsp), %rax` (Read %rax from the stack)
  - `addq $8,%rsp` (Increment the stack pointer)
## Unary Operations

The following instructions act on a single operand (register or memory):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc D</td>
<td>$D \leftarrow D + 1$</td>
<td>Increment</td>
</tr>
<tr>
<td>dec D</td>
<td>$D \leftarrow D - 1$</td>
<td>Decrement</td>
</tr>
<tr>
<td>neg D</td>
<td>$D \leftarrow -D$</td>
<td>Negate</td>
</tr>
<tr>
<td>not D</td>
<td>$D \leftarrow \neg D$</td>
<td>Complement</td>
</tr>
</tbody>
</table>

*inc D* is reminiscent of C's `++` operator, and *neg D* is reminiscent of C's `--` operator.

Examples:  
- `incq 16(%rax)`  
- `decq %rdx`  
- `not %rcx`
Binary Operations

The following instructions act on a two operands (register or memory, but not both):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add S, D</td>
<td>$D \leftarrow D + S$</td>
<td>Add</td>
</tr>
<tr>
<td>sub S, D</td>
<td>$D \leftarrow D - S$</td>
<td>Subtract</td>
</tr>
<tr>
<td>imul S, D</td>
<td>$D \leftarrow D \times S$</td>
<td>Multiply</td>
</tr>
<tr>
<td>xor S, D</td>
<td>$D \leftarrow D \oplus S$</td>
<td>Exclusive-or</td>
</tr>
<tr>
<td>or S, D</td>
<td>$D \leftarrow D \lor S$</td>
<td>Or</td>
</tr>
<tr>
<td>and S, D</td>
<td>$D \leftarrow D \land S$</td>
<td>And</td>
</tr>
</tbody>
</table>

Reading the syntax is a bit tricky — e.g., `subq %rax,%rdx` decrements `%rdx` by `%rax`, and can be read as "Subtract `%rax` from `%rdx`"

Examples:
- `addq %rcx,(%rax)`
- `imulq $16,(%rax,%rdx,8)`
- `subq %rdx,8(%rax)`
Shift Operations

The following instructions perform shifts. The first operator can be either an immediate value or the byte `%cl` (and only that register!)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sal k, D</td>
<td>(D \leftarrow D \ll k)</td>
<td>Left shift</td>
</tr>
<tr>
<td>shl k, D</td>
<td>(D \leftarrow D \ll k)</td>
<td>Left shift (same as sal)</td>
</tr>
<tr>
<td>sar k, D</td>
<td>(D \leftarrow D \gg_A k)</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td>shr k, D</td>
<td>(D \leftarrow D \gg_L k)</td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>

Technically, you could shift up to 255 with `%cl`, but the data width operation determines how many bits are shifted, and the high order bits are ignored. For example, if `%cl` has a value of 0xFF, then `shlb` shifts by 7 (ignoring the upper bits), `shlw` shifts by 15, `shll` would shift by 31, and `shll` would shift by 63.

Examples:

- `shll $3,(%rax)`
- `shr %cl,(%rax,%rdx,8)`
- `sar $4,8(%rax)`
Recall that multiplying two 64-bit numbers can produce a 128-bit result. The x86-64 instruction set supports 128-bit numbers with the "oct" (16-byte) size.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imulq S</td>
<td>R[%rdx]:R[%rax] ← S × R[%rax]</td>
<td>Signed full multiply</td>
</tr>
<tr>
<td>mulq S</td>
<td>R[%rdx]:R[%rax] ← S × R[%rax]</td>
<td>Unsigned full multiply</td>
</tr>
</tbody>
</table>

The *imulq* instruction has two forms. One, shown on slide 11, takes two operands and leaves the result in a single 64-bit register, truncating if necessary (and acts the same on signed and unsigned numbers). Example: *imulq %rbx, %rcx*

The second form (shown above) multiplies the source by %rax, and puts the product into the 128-bit %rdx (upper 64 bits) and %rax (lower 64 bits).
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod(uint128_t *dest, uint64_t x, uint64_t y)
{
    *dest = x * (uint128_t) y;
}

int main()
{
    uint64_t x = 2000000000000; // 2 trillion
    uint64_t y = 3000000000000; // 3 trillion
    uint128_t z;
    store_uprod(&z, x, y);
    print_uint128(z); // see lect12 code
    // for function definition
    return 0;
}
Slide 11 did not list a division instruction or modulus instruction. There are single-operand divide instructions (shown below):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cqto</td>
<td>R[%rdx]:R[%rax] ← SignExtend(R[%rax])</td>
<td>Convert to oct word</td>
</tr>
<tr>
<td>idivq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Signed divide</td>
</tr>
<tr>
<td>divq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Unsigned divide</td>
</tr>
</tbody>
</table>

The dividend for the `idivq` and `divq` instructions is the 128-bit quantity in registers `%rdx` (high-order 64-bits) and `%rax` (low-order 64-bits). The divisor is the operand source. The quotient from the division is stored in `%rax`, and the remainder is stored in `%rdx`.

For most division, the dividend is just in `%rax`, and `%rdx` is either all zeros (for unsigned, or the sign bit of `%rax` (for signed arithmetic). The ctqo instruction can be used to accomplish this.
void remdiv(long x, long y,  
    long *qp, long *rp)
{
    long q = x / y;
    long r = x % y;
    *qp = q;
    *rp = r;
}

mov   %rdx,%r8    # copy qp
mov   %rdi,%rax   # Move x to lower 8 bytes of dividend
cqto  # Sign-extend to upper 8 bytes of dividend
idiv  %rsi        # Divide by y
mov   %rax,(%r8)  # Store quotient at qp
mov   %rdx,(%rcx) # Store remainder at rp
retq

Note: %rdi is 1st argument  
%rsi is 2nd argument  
%rdx is 3rd argument  
%rcx is 4th argument

gcc is clever enough to see that only one division is needed!
• 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 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.
**Condition Codes: Examples**

- **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.

```c
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 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.

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

*Which flag above would be set?*

The **ZF** flag.

```plaintext
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.
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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Based on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP</strong></td>
<td>( S_1, S_2 )</td>
<td>( S_2 - S_1 ) Compare</td>
</tr>
<tr>
<td>( \text{cmpb} )</td>
<td></td>
<td>Compare byte</td>
</tr>
<tr>
<td>( \text{cmpw} )</td>
<td></td>
<td>Compare word</td>
</tr>
<tr>
<td>( \text{cmpl} )</td>
<td></td>
<td>Compare double word</td>
</tr>
<tr>
<td>( \text{cmpq} )</td>
<td></td>
<td>Compare quad word</td>
</tr>
<tr>
<td><strong>TEST</strong></td>
<td>( S_1, S_2 )</td>
<td>( S_2 &amp; S_1 ) Test</td>
</tr>
<tr>
<td>( \text{testb} )</td>
<td></td>
<td>Test byte</td>
</tr>
<tr>
<td>( \text{testw} )</td>
<td></td>
<td>Test word</td>
</tr>
<tr>
<td>( \text{testl} )</td>
<td></td>
<td>Test double word</td>
</tr>
<tr>
<td>( \text{testq} )</td>
<td></td>
<td>Test quad word</td>
</tr>
</tbody>
</table>

- Be careful! The operands for `cmp` are listed in reverse order!

- Often, we use `testq %rax, %rax` to see whether `%rax` is negative, zero, or positive.
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.
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.

Example: \( a < b \)

```c
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
    movzbl %al, %eax # %eax to 0 or 1
    setbe %al, %eax # Clear rest of %eax
    ret
```
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.

### Jump Instructions

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

Jump instructions jump to labels in assembly code, and those labels are changed to addresses (most often relative). They are unconditional or conditional.

- **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

```c
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

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

Compile to an object file:

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

Run GDB:

```
$ 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
```

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
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.

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

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

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

Compile to an object file:
gcc -c -Og while_loop.c

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

```c
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?
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

```c
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!
3 minute break
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...
Let's look at our while loop again:

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

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
```

Compile to an object file:
```
gcc -c -Og while_loop.c
```
Digging Deeper: Jump Instruction Encodings

- Take the following function:

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

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: 7e f8                jle    7 <loop+0x7>
  f: f3 c3                repz retq
```

Compile to an object file:
```
gcc -c -Og while_loop.c
```

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:

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

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
da: 83 f8 63            cmp    $0x63,%eax
d: 7e f8                jle    7 <loop+0x7>
f: f3 c3                repz retq
```

Compile to an object file:
```
gcc -c -Og while_loop.c
```

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:

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

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`.

Compile to an object file:
```
gcc -c -Og while_loop.c
```
Digging Deeper: Jump Instruction Encodings

- Take the following function:

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

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.

Compile to an object file:
```
gcc -c -Og while_loop.c
```
Digging Deeper: Jump Instruction Encodings

- Take the following function:

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

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: 7e f8                jle    7 <loop+0x7>
   d: 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`.

Compile to an object file:
```
gcc -c -Og while_loop.c
```
Practice: Reverse-engineer Assembly to C

• Take the following function:

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

```assembly
# 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
```
• 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;
}
```

```assembly
# 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:

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

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

Which is faster? Let's test!
Midterm Comments

• Challenging exam with a wide spread of grades
• If you didn't do as well as you had hoped, review the exam and try and assess where you had misconceptions.
  • Was there a time issue?
  • Was there a misconception issue?
• For those who want to do better on the final exam: really understanding the homework and labs is key.
• We will have more conceptual office hours going forward.
• References:
  • Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  • CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  • gdbtui: https://beej.us/guide/bggdb/
  • More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  • Compiler explorer: https://gcc.godbolt.org
• Advanced Reading:
  • history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  • x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64