CS 31: Intro to Systems C Programming
L07-08: ISA Assembly

Vasanta Chaganti & Kevin Webb
Swarthmore College
September 26-28, 2023
“If you can do logic gates in your head, please confirm you are not a replicant”

http://smbc-comics.com/comic/logic-gates
Reading Quiz

• Note the red border!

• 1 minute per question

• No talking, no laptops, phones during the quiz

Check your frequency:
• Iclicker2: frequency AA
• Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:
1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA
vote status light will indicate success
Agenda

• Hardware basics
  • Machine memory models
  • Digital signals
  • Logic gates
Today

• How to directly interact with hardware

• Instruction set architecture (ISA)
  – Interface between programmer and CPU
  – Established instruction format (assembly lang)

• Assembly programming (x86_64)
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources

Complex devices
Compute & I/O
Abstraction

This week: Machine Interface

Last week: Circuits, Hardware Implementation

Applications
Specific functionality

Operating system
Manage resources

Complex devices
Compute & I/O

Abstraction

Applications
Specific functionality

Operating system
Manage resources

Complex devices
Compute & I/O

This week: Machine Interface

Last week: Circuits, Hardware Implementation
CPU Game Plan

- **Fetch** instruction from memory

- **Decode** what the instruction is telling us to do
  - Tell the ALU what it should be doing
  - Find the correct operands

- **Execute** the instruction (arithmetic, etc.)

- **Store** the result
Machine Code

Binary (0’s and 1’s) Encoding of ISA Instructions
– some bits: encode the instruction (opcode bits)
– others encode operand(s)

(eg) \[ 01001010 \]  \textit{opcode} \ \textit{operands}

\[ 01 \ 001 \ 010 \]
ADD \texttt{\%r1 \ %r2}

– different bits fed through different CPU circuitry:
Hardware: Control, Storage, ALU circuitry

Program Counter (PC): Address 0

Instruction Register (IR): OP Code | Reg A | Reg B | Result

- acts on instruction bits to execute individual instructions
- PC value used to determine next instruction to execute

Register File

Let the ALU do its thing. (e.g., Add)

(Memory)

64-bit Register #0
64-bit Register #1
64-bit Register #2
64-bit Register #3

MUX

MUX
Pipelining (CPU)

CPU Stages: fetch, decode, execute, store results

1 Nanosecond

1st nanosecond: F

2nd nanosecond: F D

3rd nanosecond: F D E

4th nanosecond: F D E S

5th nanosecond: F D E S

Steady state: One instruction finishes every nanosecond! (Clock rate can be faster.)
How a computer runs a program:

- We know: How HW Executes Instructions:
- This Week: Instructions and ISA
  - Program Encoding: C code to assembly code
  - Learn IA32 Assembly programming
Compilation Steps (.c to a.out)

Usually compile to a.out in a single step: gcc –m32 p1.c

-m32 tells gcc to compile for 32-bit Intel machines

Reality is more complex: there are intermediate steps!

Executable code (a.out)

machine code instructions
Compilation Steps (.c to a.out)

- C program (p1.c)
- Compiler (gcc -S)
- Assembly program (p1.s)

You can see the results of intermediate compilation steps using different gcc flags.

Executable code (a.out)

machine code instructions
Compilation Steps (.c to a.out)

C program (p1.c) → Compiler (gcc -S)

Assembly program (p1.s) → Assembler (gcc -c (or as = gcc’s assembler))

Object code (p1.o) → Linker (gcc (or ld))

Executable code (a.out) → Library obj. code (libc.a)

You can see the results of intermediate compilation steps using different gcc flags.
Machine Code

Binary (0’s and 1’s) Encoding of ISA Instructions

– some bits: encode the instruction (opcode bits)
– others encode operand(s)

(eg) \[01001010\] opcode operands
\[01\ 001\ 010\]
ADD \%r1 \%r2

– different bits fed
through different
CPU circuitry:
Assembly Code

- C program (`p1.c`)
  - Compiler (`gcc -S`)
  - Assembly program (`p1.s`)
    - Assembler (`gcc -c (or as = gcc's assembler)`) → Object code (`p1.o`)
      - Linker (`gcc (or ld)`) → Executable code (`a.out`)

- HumanReadable Form of Machine Code
- Human Readable Form of Machine Code
- Machine code instructions
What is “assembly”? 

```
push %rbp
mov %rsp, %rbp
sub $16, %rsp
movl $10, -8(%rbp)
movl $20, -4(%rbp)
movl -4(%rbp), $rax
addl $rax, -8(%rbp)
movl -8(%rbp), %rax
leave
```

Assembly is the “human readable” form of the instructions a machine can understand.

```
objdump -d a.out
```
# Object / Executable / Machine Code

<table>
<thead>
<tr>
<th><strong>Assembly</strong></th>
<th><strong>Machine Code (Hexadecimal)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>push %ebp</td>
<td>55</td>
</tr>
<tr>
<td>mov %esp, %ebp</td>
<td>89 E5</td>
</tr>
<tr>
<td>sub $16, %esp</td>
<td>83 EC 10</td>
</tr>
<tr>
<td>movl $10, -8(%ebp)</td>
<td>C7 45 F8 0A 00 00 00</td>
</tr>
<tr>
<td>movl $20, -4(%ebp)</td>
<td>C7 45 FC 14 00 00 00</td>
</tr>
<tr>
<td>movl -4(%ebp), $eax</td>
<td>8B 45 FC</td>
</tr>
<tr>
<td>addl $eax, -8(%ebp)</td>
<td>01 45 F8</td>
</tr>
<tr>
<td>movl -8(%ebp), %eax</td>
<td>B8 45 F8</td>
</tr>
<tr>
<td>leave</td>
<td>C9</td>
</tr>
</tbody>
</table>

Almost a 1-to-1 mapping to Machine Code
Hides some details like num bytes in instructions
Object / Executable / Machine Code

Assembly

```assembly
push %ebp
mov %esp, %ebp
sub $16, %esp
movl $10, -8(%ebp)
movl $20, -4(%ebp)
movl -4(%ebp), $eax
addl $eax, -8(%ebp)
movl -8(%ebp), %eax
leave
```

```c
int main() {
    int a = 10;
    int b = 20;
    a = a + b;
    return a;
}
```
Compilation Steps (.c to a.out)

C program (p1.c) -> Compiler (gcc -m32 -S) -> Assembly program (p1.s) -> Assembler (gcc -c (or as)) -> Object code (p1.o) -> Linker (gcc (or ld)) -> Executable code (a.out)

- High-level language
- Interface for speaking to CPU
- CPU-specific format (011010...)

Slide 27
Instruction Set Architecture (ISA)

• ISA (or simply architecture):
  Interface between lowest software level and the hardware.

• Defines the language for controlling CPU state:
  – Defines a set of instructions and specifies their machine code format
  – Makes CPU resources (registers, flags) available to the programmer
  – Allows instructions to access main memory (potentially with limitations)
  – Provides control flow mechanisms (instructions to change what executes next)
Instruction Set Architecture (ISA)

The agreed-upon interface between all software that runs on the machine and the hardware that executes it.
Instruction Set Architecture (ISA)

The agreed-upon interface between all software that runs on the machine and the hardware that executes it.
Instruction Set Architecture (ISA)

- ISA is Interface between CPU and Compiler:
  - Compiler translates program source code to machine code of a target ISA
  - (e.g.) C program $\rightarrow$ gcc $\rightarrow$ ISA machine code (0’s and 1’s)
ISA Examples

- Intel IA-32 (80x86)
- ARM
- MIPS
- PowerPC
- IBM Cell
- Motorola 68k
- Intel x86_64
- Intel IA-64 (Itanium)
- VAX
- SPARC
- Alpha
- IBM 360
Intel x86 Family

Intel i386 (1985)
- 12 MHz - 40 MHz
- ~300,000 transistors
- Component size: 1.5 µm

Intel Core i9 9900k (2018)
- ~4,000 MHz
- ~7,000,000,000 transistors
- Component size: 14 nm

Everything in this family uses the same ISA (Same instructions)!
Instruction Set Architecture (ISA)

• ISA (or simply architecture):
  Interface between lowest software level and the hardware.

• Defines the language for controlling CPU state:
  – Defines a set of instructions and specifies their machine code format
  – Makes CPU resources (registers, flags) available to the programmer
  – Allows instructions to access main memory (potentially with limitations)
  – Provides control flow mechanisms (instructions to change what executes next)
What are registers? Do we even need them?

A. Registers are small and fast memory used as scratch space (to store temporary variables) to perform operations on the ALU.

B. Registers are on the same chip as the ALU.

C. We can move data and instructions from main memory to registers, through a bus (group of wires) connecting main memory to the register file.

D. General purpose registers are accessed via %rax - %ebp. Special purpose registers like the program counter reference the location of the next instruction in main memory.

E. All of the above
Processor State in Registers

Working memory for currently executing program
- Temporary data: \%rax - \%r15
- Current stack frame
  - \%rbp: base pointer
  - \%rsp: stack pointer
- Address of next instruction to execute: \%rip
- Status of recent ALU tests (CF, ZF, SF, OF)
Component Registers

- Registers starting with “r” are 64-bit registers
  - %rax, %rbx, ..., %rsi, %rdi

- Sometimes, you might only want to store 32 bits (e.g., int variable)
  - You can access the lower 32 bits of a register with prefix e:
    - %eax, %ebx, ..., %esi, %edi
  - with a suffix of d for registers %r8 to %r15
    - %r8d, %r9d, ..., %r15d

General purpose registers

Current stack top

Current stack frame

Program Counter (PC)

Condition codes (flags)
Assembly Programmer’s View of State

Registers:
- **PC**: Program counter (%rip)
- **Condition codes** (%RFLAGS)
- **General Purpose** (%rax - %r15)

Memory:
- Byte addressable array
- Program code and data
- Execution stack

### CPU

<table>
<thead>
<tr>
<th>64-bit (8 byte) Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td><strong>value</strong></td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
</tr>
<tr>
<td>%rip</td>
<td>next instr addr (PC)</td>
</tr>
<tr>
<td>%RFLAGS</td>
<td>cond. codes</td>
</tr>
</tbody>
</table>

### BUS

Addresses

Data

Instructions

Memory:
- • Byte addressable array
- • Program code and data
- • Execution stack
Types of assembly instructions

• Data movement
  – Move values between registers and memory
  – Examples: `mov`, `movl`, `movq`

• Load: move data from memory to register

• Store: move data from register to memory

The suffix letters specify how many bytes to move (not always necessary, depending on context).

- `l` -> 32 bits
- `q` -> 64 bits
Data Movement

Move values between memory and registers or between two registers.

**Program Counter (PC):** Memory address of next instr

**Instruction Register (IR):** Instruction contents (bits)

- Data in
- WE
- Data in
- WE
- Data in
- WE
- Data in
- WE
- Data in
- WE

64-bit Register #0

64-bit Register #1

64-bit Register #2

64-bit Register #3

... 

Register File
Types of assembly instructions

• Data movement
  – Move values between registers and memory

• Arithmetic
  – Uses ALU to compute a value
  – Examples: add, addl, addq, sub, subl, subq...
Arithmetic

Use ALU to compute a value, store result in register / memory.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)

Register File

Data in
WE
Data in
WE
Data in
WE
Data in
WE

64-bit Register #0

MUX

64-bit Register #1

MUX

64-bit Register #2

MUX

64-bit Register #3

...
Types of assembly instructions

• Data movement
  – Move values between registers and memory

• Arithmetic
  – Uses ALU to compute a value

• Control
  – Change PC based on ALU condition code state
  – Example: jmp
Control

Change PC based on ALU condition code state.

Program Counter (PC): Memory address of next instr

Instruction Register (IR): Instruction contents (bits)
Types of assembly instructions

• Data movement
  – Move values between registers and memory

• Arithmetic
  – Uses ALU to compute a value

• Control
  – Change PC based on ALU condition code state

• Stack / Function call  (We’ll cover these in detail later)
  – Shortcut instructions for common operations
Addressing Modes

• Instructions need to be told where to get operands or store results

• Variety of options for how to address those locations

• A location might be:
  – A register
  – A location in memory

• In x86_64, an instruction can access at most one memory location
Addressing Modes

• Instructions can refer to:
  – the name of a register (%rax, %rbx, etc)
  – to a constant or “literal” value, starts with $%
  – (%rax) : accessing memory
    • treat the value in %rax as a memory address,
Addressing Mode: Memory

\textbf{movl (rcx), rax}

- Use the address in register \( \texttt{rcx} \) to access memory,
- then, \textit{store result at that memory address} in register \( \texttt{rax} \)

1. Index into memory using the address in \( \texttt{rcx} \).
Addressing Mode: Memory

\[
\text{movl} \ (%rcx), \ %rax
\]

– Use the address in register \%rcx to access memory,
– then, \textit{store result at that memory address} in register \%rax

1. Index into memory using the address in \%rcx.
2. Copy value at that address to \%rax.
Addressing Mode: Register

• Instructions can refer to the name of a register

• Examples:
  – `mov %rax, %r15`
    (Copy the contents of %rax into %r15 -- overwrites %r15, no change to %rax)

  – `add %r9, %rdx`
    (Add the contents of %r9 and %rdx, store the result in %rdx, no change to %r9)
Addressing Mode: Immediate

• Refers to a constant or “literal” value, starts with $  
  
• Allows programmer to hard-code a number  

• Can be either decimal (no prefix) or hexadecimal (0x prefix)  

\textbf{mov $10$, %rax}
  – Put the constant value 10 in register rax.

\textbf{add $0xF$, %rdx}
  – Add 15 (0xF) to %rdx and store the result in %rdx.
Addressing Mode: Memory

- Accessing memory requires you to specify which address you want.
  - Put the address in a register.
  - Access the register with ( ) around the register’s name.

```
mov (%rcx), %rax
```
- Use the address in register %rcx to access memory, store result in register %rax
Addressing Mode: Displacement

- Like memory mode, but with a constant offset
  - Offset is often negative, relative to `%rbp`

```assembly
movl  -16(%rbp), %rax
```
- Take the address in `%rbp`, subtract 16 from it, index into memory and store the result in `%rax`. 
Addressing Mode: Displacement

\texttt{movl -16(\%rbp), \%rax}

- Take the address in \%rbp, subtract 16 from it, index into memory and store the result in \%rax.

\begin{tabular}{|c|c|}
  \hline
  name & value \\
  \hline
  \%rax & 0 \\
  \%rcx & 0x1A68 \\
  \%rbp & 0x1A70 \\
  \hline
\end{tabular}

1. Access address: 0x1A70 – 16 => 0x1A60
Addressing Mode: Displacement

```asm
movl -16(%rbp), %rax
```
– Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.

**CPU Registers**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>11</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1A70</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Access address: 0x1A70 – 16 => 0x1A60

2. Copy value at that address to rax.
Let’s try a few examples...
What will the state of registers and memory look like after executing these instructions?

```
sub $16, %rsp
movq $3, -8(%rbp)
mov $10, %rax
sal $1, %rax
add -8(%rbp), %rax
movq %rax, -16(%rbp)
add $16, %rsp
```

x is stored at rbp-8
y is stored at rbp-16

<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>0x1FFF000AD0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0x1FFF000AD8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0x1FFF000AE0</td>
<td>0x1FFF000AF0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
What will the state of registers and memory look like after executing these instructions?

A. 

```
sub $16, %rsp
movq $3, -8(%rbp)
mov $10, %rax
sal $1, %rax
add -8(%rbp), %rax
movq %rax, -16(%rbp)
add $16, %rsp
```

x is stored at rbp-8
y is stored at rbp-16
sub $16, %rsp
movq $3, -8(%rbp)
mov $10, %rax
sal $1, %rax
add -8(%rbp), %rax
movq %rax, -16(%rbp)
add $16, %rsp

x is stored at rbp-8
y is stored at rbp-16
Assembly Visualization Tool

• The authors of Dive into Systems, including Swarthmore faculty with help from Swarthmore students, have developed a tool to help visualize assembly code execution:

  • https://asm.diveintosystems.org

• For this example, use the arithmetic mode.

  sub $16, %rsp
  movq $3, -8(%rbp)
  mov $10, %rax
  sal $1, %rax
  add -8(%rbp), %rax
  movq %rax, -16(%rbp)
  add $16, %rsp

  x is stored at rbp-8
  y is stored at rbp-16
Solution

```c
x = 3;
y = x + (10 << 1);
```

```
sub $16, %rsp
movq $3, -8(%rbp)
mov $10, %rax
sal $1, %rax
add -8(%rbp), %rax
movq %rax, -16(%rbp)
add $16, %rsp
```

Subtract constant 16 from %rsp
Move constant 3 to address %rbp-8
Move constant 10 to register %rax
Shift the value in %rax left by 1 bit
Add the value at address %rbp-8 to %rax
Store the value in %rax at address rbp-16
Add constant 16 to %rsp

|x| y|
---|---|
| is stored at rbp-8 | is stored at rbp-16 |

### Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>23</td>
</tr>
<tr>
<td>%rsp</td>
<td>...AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>...AE0</td>
</tr>
</tbody>
</table>

### Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1FFF000AD0</td>
<td>23</td>
</tr>
<tr>
<td>0x1FFF000AD8</td>
<td>3</td>
</tr>
<tr>
<td>0x1FFF000AE0</td>
<td>0x1FFF000AF0</td>
</tr>
</tbody>
</table>
What will the state of registers and memory look like after executing these instructions?

... 

mov  %rbp, %rcx
sub  $8, %rcx
movq (%rcx), %rax
or   %rax, -16(%rbp)
neg  %rax

<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0</td>
</tr>
<tr>
<td>%rcx</td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</td>
</tr>
</tbody>
</table>

| Memory |
|--------|-------|
| Address | Value |
| ...    | ...   |
| 0x1FFF000AD0 | 8    |
| 0x1FFF000AD8  | 5    |
| 0x1FFF000AE0  | 0x1FFF000AF0 |
| ...    | ...   |
How might you implement the following C code in assembly?

\[
z = x \wedge y
\]

- \(x\) is stored at \%rbp-8
- \(y\) is stored at \%rbp-16
- \(z\) is stored at \%rbp-24

A:

\[
\begin{align*}
\text{movq} & \ -8(%rbp), \ %rax \\
\text{movq} & \ -16(%rbp), \ %rdx \\
\text{xor} & \ %rax, \ %rdx \\
\text{movq} & \ %rax, \ -24(%rbp)
\end{align*}
\]

B:

\[
\begin{align*}
\text{movq} & \ -8(%rbp), \ %rax \\
\text{movq} & \ -16(%rbp), \ %rdx \\
\text{xor} & \ %rdx, \ %rax \\
\text{movq} & \ %rax, \ -24(%rbp)
\end{align*}
\]

C:

\[
\begin{align*}
\text{movq} & \ -8(%rbp), \ %rax \\
\text{movq} & \ -16(%rbp), \ %rdx \\
\text{xor} & \ %rax, \ %rdx \\
\text{movq} & \ %rax, \ -8(%rbp)
\end{align*}
\]

D:

\[
\begin{align*}
\text{movq} & \ -24(%rbp), \ %rax \\
\text{movq} & \ -16(%rbp), \ %rdx \\
\text{xor} & \ %rdx, \ %rax \\
\text{movq} & \ %rax, \ -8(%rbp)
\end{align*}
\]
How might you implement the following C code in assembly?

\[ x = y \gg 3 \mid x \times 8 \]

- \( x \) is stored at \%rbp-8
- \( y \) is stored at \%rbp-16
- \( z \) is stored at \%rbp-24
Solutions (other instruction sequences can work too!)

- $z = x^y$

  ```assembly
  movq -8(%rbp), %rax
  movq -16(%rbp), %rdx
  xor %rdx, %rax
  movq %rax, -24(%rbp)
  ```

- $x = y >> 3 \mid x \ast 8$

  ```assembly
  mov -8(%rbp), %rax
  imul $8, %rax
  movq -16(%rbp), %rdx
  sar $3, %rdx
  or %rax, %rdx
  movq %rdx, -8(%rbp)
  ```
Recall Memory Operands

- displacement(\%reg)
  - e.g., add \%rax, -8(\%rbp)

- x86_64 allows a memory operand as the source or destination, but NOT BOTH!
  - One of the operands must be a register

- This would not be allowed:
  - add -8(\%rbp), -16(\%rbp)
  - If you wanted this, movq one value into a register first
Control Flow

- Previous examples focused on:
  - data movement (mov, movq)
  - arithmetic (add, sub, or, neg, sal, etc.)

- Up next: Jumping!

(Changing which instruction we execute next.)
Relevant XKCD

I could restructure the program's flow
or use one little 'goto' instead.

Eh, screw good practice.
How bad can it be?

goto main_sub3;

*compile*

xkcd #292
Unconditional Jumping / Goto

A label is a place you might jump to.

Labels ignored except for goto/jumps.

(Skipped over if encountered)

```c
int main(void) {
    long a = 10;
    long b = 20;

    goto label1;
    a = a + b;

    label1:
    return;
}
```

```c
int x = 20;
L1:
    int y = x + 30;
L2:
    printf("%d, %d\n", x, y);
```
Unconditional Jumping / Goto

int main(void) {
    long a = 10;
    long b = 20;
    goto label1;
    a = a + b;

    label1:
    return;
}

pushq %rbp
mov %rsp, %rbp
sub $16, %rsp
movq $10, -16(%ebp)
movq $20, -8(%ebp)
jmp label1

these instructions
are never
executed in this
code
Unconditional Jumping / Goto

Usage besides goto?
- infinite loop
- break;
- continue;
- functions (handled differently)

- Often, we only want to jump when some condition is true / false.

- We need some way to compare values, jump based on comparison results.

```assembly
pushq %rbp
mov %rsp, %rbp
sub $16, %rsp
movq $10, -16(%ebp)
movq $20, -8(%ebp)
jmp label1
movq -8(%rbp), $rax
add $rax, -16(%rbp)
movq -16(%rbp), %rax

label1:
leave
```
Condition Codes (or Flags)

- Set in two ways:
  1. As “side effects” produced by ALU
  2. In response to explicit comparison instructions

- x86_64 condition codes tell you:
  - If the result is zero (**ZF**)
  - If the result’s **first bit is set** (negative if signed) (**SF**)
  - If the result **overflowed** (assuming unsigned) (**CF**)
  - If the result **overflowed** (assuming signed) (**OF**)
Processor State in Registers

Working memory for currently executing program
- Temporary data: %rax - %r15
- Current stack frame
  - %rbp: base pointer
  - %rsp: stack pointer
- Address of next instruction to execute: %rip
- Status of recent ALU tests (CF, ZF, SF, OF)
Instructions that set condition codes

1. Arithmetic/logic side effects (add, sub, or, etc.)

2. CMP and TEST:

   `cmp b, a` like computing `a - b` without storing result
   - Sets OF if overflow, Sets CF if carry-out,
     Sets ZF if result zero, Sets SF if result is negative

   `test b, a` like computing `a & b` without storing result
   - Sets ZF if result zero, sets SF if `a & b < 0`
     OF and CF flags are zero (there is no overflow with &
Which flags would this sub set?

Suppose %rax holds 5, %rcx holds 7

\texttt{sub } \$5, \%rax

If the result is zero (ZF)
If the result’s first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF  
B. SF  
C. CF and ZF  
D. CF and SF  
E. CF, SF, and CF
Which flags would this \texttt{sub} set?

Suppose \%rax holds 5, \%rcx holds 7

\texttt{sub \$5, \%rax}

- If the result is zero (ZF)
- If the result’s first bit is set (negative if signed) (SF)
- If the result overflowed (assuming unsigned) (CF)
- If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF
Which flags would this `cmp` set?

Suppose `%rax` holds 5, `%rcx` holds 7

```assembly
cmp %rcx, %rax
```

- If the result is zero (ZF)
- If the result’s first bit is set (negative if signed) (SF)
- If the result overflowed (assuming unsigned) (CF)
- If the result overflowed (assuming signed) (OF)

A. ZF
B. SF
C. CF and ZF
D. CF and SF
E. CF, SF, and CF
Which flags would this `cmp` set?

Suppose `%rax` holds 5, `%rcx` holds 7

```
cmp %rcx, %rax
```

If the result is zero (ZF)
If the result’s first bit is set (negative if signed) (SF)
If the result overflowed (assuming unsigned) (CF)
If the result overflowed (assuming signed) (OF)

A. ZF  
B. SF  
C. CF and ZF  
D. CF and SF  
E. CF, SF, and CF
# Conditional Jumping

Jump based on which condition codes are set

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>jmp</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>je</strong></td>
<td>ZF</td>
</tr>
<tr>
<td><strong>jne</strong></td>
<td>~ZF</td>
</tr>
<tr>
<td><strong>js</strong></td>
<td>SF</td>
</tr>
<tr>
<td><strong>jns</strong></td>
<td>~SF</td>
</tr>
<tr>
<td><strong>jg</strong></td>
<td>~(SF^OF) &amp;~ZF</td>
</tr>
<tr>
<td><strong>jge</strong></td>
<td>~(SF^OF)</td>
</tr>
<tr>
<td><strong>jl</strong></td>
<td>(SF^OF)</td>
</tr>
<tr>
<td><strong>jle</strong></td>
<td>(SF^OF)</td>
</tr>
<tr>
<td><strong>ja</strong></td>
<td>~CF&amp;~ZF</td>
</tr>
<tr>
<td><strong>jb</strong></td>
<td>CF</td>
</tr>
</tbody>
</table>

**Jump Instructions:**
(See book section 7.4.1)

You do not need to memorize these!
Example Scenario

```c
long userval;
scanf("%ld", &userval);
if (userval == 42) {
    userval = userval + 5;
} else {
    userval = userval - 10;
}
```

- Suppose user gives us a value via `scanf`
- We want to check to see if it equals 42
  - If so, add 5
  - If not, subtract 10
How would we use jumps/CCs for this?

```c
long userval;
scanf("%ld", &userval);

if (userval == 42) {
    userval = userval + 5;
} else {
    userval = userval - 10;
}
```

Assume userval is stored in %rax at this point.
How could we use jumps/CCs to implement this C code?

```c
long userval;
scanf("%ld", &userval);
if (userval == 42) {
    userval = userval + 5;
} else {
    userval = userval - 10;
}
```

(A) `cmp $42, %rax`  
    `je L2`  
    `L1:`  
    `sub $10, %rax`  
    `jmp DONE`  
    `L2:`  
    `add $5, %rax`  
    `DONE:`

(B) `cmp $42, %rax`  
    `jne L2`  
    `L1:`  
    `add $5, %rax`  
    `jmp DONE`  
    `L2:`  
    `add $5, %rax`  
    `DONE:`

(C) `cmp $42, %rax`  
    `jne L2`  
    `L1:`  
    `sub $10, %rax`  
    `jmp DONE`  
    `L2:`  
    `sub $10, %rax`  
    `DONE:`
How could we use jumps/CCs to implement this C code?

```c
long userval;
scanf("%ld", &userval);
if (userval == 42) {
    userval = userval + 5;
} else {
    userval = userval - 10;
}
```

Assume userval is stored in %rax at this point.

(A) ```
cmp $42, %rax
je L2
L1:
    sub $10, %rax
    jmp DONE
L2:
    add $5, %rax
DONE:
```

(B) ```
cmp $42, %rax
jne L2
L1:
    sub $10, %rax
    jmp DONE
L2:
    add $5, %rax
DONE:
```

(C) ```
cmp $42, %rax
jne L2
L1:
    add $5, %rax
    jmp DONE
L2:
    sub $10, %rax
DONE:
```
Visualization demo

Try this in arithmetic mode:

https://asm.diveintosystems.org

Change the value 3 to 42 to alter the behavior.

# Initialize rax
mov $3, %rax

cmp $42, %rax
je L2
L1:
sub $10, %rax
jmp DONE
L2:
add $5, %rax
DONE:
### C Loops to x86_64

<table>
<thead>
<tr>
<th>C Loop Type</th>
<th>C Gotot Translations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>do-while:</strong></td>
<td>do { loop body } while (cond); if(!cond) goto done</td>
</tr>
<tr>
<td><strong>while:</strong></td>
<td>while(condition) { loop body } if(!condition) goto done</td>
</tr>
<tr>
<td><strong>for:</strong></td>
<td>for(init; cond; step){ loop body } if(!condition) goto done</td>
</tr>
</tbody>
</table>
 Convert to C goto:

```c
int x = 0;
for(i=0; i < 10; i++) {
    x = x + 1;
}
z = x * 3;
```

<table>
<thead>
<tr>
<th>for:</th>
<th>init code</th>
</tr>
</thead>
</table>
| for(init; cond; step){
    loop body
} | <fill in your answer here> |
Convert to C goto:

```c
x = 0;
for(i=0; i < 10; i++) {
    x = x + 1;
}
z = x * 3;
```

```
for:
for(init; cond; step){
    loop body
}
```

```
init code
if(!cond) goto done
loop:
    loop body
    step
    if(cond) goto loop
done:
```
Using Jump Instructions

- `jmp label`  # unconditional jump (ex. `jmp .L2`)  
- `jge label`  # conditional jump (ex. if $\geq$) (je, jne, js, jg, ...)

(A label is a place you might jump to. Labels ignored except for goto/jumps)

Try out this code: what does it do?

```assembly
movl $0, %rax
movl $4, %rbx
movl $0, %rdx
jmp .L2
.L1:
    addl $1, %rax
.L2:
    addl %rax, %rdx
    cmp %rax, %rbx  # R[ebx] - R[eax]
jge .L1
```
Using Jump Instructions

- `jmp label`  # unconditional jump  (ex. `jmp .L2`)
- `jge label`  # conditional jump (ex. if >=)  (je, jne, js, jg, ...)

(A label is a place you might jump to.  Labels ignored except for goto/jumps)

Try out this code: what does it do?

```assembly
movq $0, %rax
movq $4, %rbx
movq $0, %rdx
jmp .L2
.L1:
    addq $1, %rax
.L2:
    addq %rax, %rdx
cmp %rax, %rbx  # R[%rbx] - R[%rax]
jge .L1
```

CPU Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0 1</td>
</tr>
<tr>
<td>%rdx</td>
<td>0 0</td>
</tr>
<tr>
<td>%rbx</td>
<td>4</td>
</tr>
</tbody>
</table>
Loops

- We will look at more of these in the lab!
Summary

• ISA defines what programmer can do on hardware
  – Which instructions are available
  – How to access state (registers, memory, etc.)
  – This is the architecture’s assembly language

• In this course, we’ll be using x86_64
  – Instructions for:
    • moving data (mov, movl, movq)
    • arithmetic (add, sub, imul, or, sal, etc.)
    • control (jmp, je, jne, etc.)
  – Condition codes for making control decisions
    • If the result is zero (ZF)
    • If the result’s first bit is set (negative if signed) (SF)
    • If the result overflowed (assuming unsigned) (CF)
    • If the result overflowed (assuming signed) (OF)