CS31: Introduction to Computer Systems

Week 4, Class 2
ISAs and Assembly
02/13/24

Dr. Sukrit Venkatagiri
Swarthmore College
## Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td>&quot;</td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td>&quot;</td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td>&quot;</td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td>&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

### Diagram

```
  C
   ↓
 x86 Assembly
   ↓
  Binary
   ↓
 CPU / memory
   ↓
 logic gates, circuits
```

- **C**: programming language
- **x86 Assembly**: instruction set architecture
- **Binary**: logic / bits
- **CPU / memory**: logic / bits
- **logic gates, circuits**: voltage

---

- **Wk**: Week
- **Lecture**: Lecture Topic
- **Lab**: Lab Topic
Overview

• 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

Applications
Specific functionality

Operating system
Manage resources

Complex devices
Computers

This week: Machine Interface

Last week: Circuits, Hardware Implementation
Compilation Steps (.c to a.out)

C program (p1.c) -> Compiler (gcc) -> Executable code (a.out)

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

Reality is more complex: there are intermediate steps!
Compilation Steps (.c to a.out)

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

- C program (p1.c)
- Assembly program (p1.s)
- Executable code (a.out)
Assembly Code

Human-readable form of CPU instructions
  • Almost a 1-to-1 mapping to hardware instructions (Machine Code)
  • Hides some details:
    • Registers have names rather than numbers
    • Instructions have names rather than variable-size codes

We’re going to use x86_64 assembly
  • Can compile C to x86_64 assembly on our system:

  gcc -S code.c  # open code.s in an editor to view
C to Assembly

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

  a = a + b;

  return a;
}

x86_64 Assembly
push   %rbp
mov    %rsp,%rbp
movq   $10,-0x10(%rbp)
movq   $20,-0x8(%rbp)
mov    -0x8(%rbp),%rax
add    %rax,-0x10(%rbp)
mov    -0x10(%rbp),%rax
pop    %rbp
ret
Compilation Steps (.c to a.out)

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

Binary (0’s and 1’s) encoding of instructions

• Opcode bits identify the instruction
• Other bits encode operand(s), where to store the results

(ex) 01001010   opcode   operands
     01   001   010
     ADD %r1 %r2

• bits fed through different CPU circuitry:
Assembly to Machine Code

**x86_64 Assembly**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push %rbp</td>
<td>Push %rbp to the stack frame</td>
</tr>
<tr>
<td>mov %rsp,%rbp</td>
<td>Move %rbp to %rsp</td>
</tr>
<tr>
<td>movq $10,-0x10(%rbp)</td>
<td>Move $10 to offset -0x10 from %rbp</td>
</tr>
<tr>
<td>movq $20,-0x8(%rbp)</td>
<td>Move $20 to offset -0x8 from %rbp</td>
</tr>
<tr>
<td>mov -0x8(%rbp),%rax</td>
<td>Move offset -0x8 from %rbp to %rax</td>
</tr>
<tr>
<td>add %rax,-0x10(%rbp)</td>
<td>Add %rax to offset -0x10 from %rbp</td>
</tr>
<tr>
<td>mov -0x10(%rbp),%rax</td>
<td>Move offset -0x10 from %rbp to %rax</td>
</tr>
<tr>
<td>pop %rbp</td>
<td>Pop %rbp from the stack frame</td>
</tr>
<tr>
<td>ret</td>
<td>Return from the function</td>
</tr>
</tbody>
</table>

**x86_64 Machine Code (in hex)**

<table>
<thead>
<tr>
<th>Hex Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
</tr>
<tr>
<td>48 89 e5</td>
</tr>
<tr>
<td>48 c7 45 f0 0a 00 00 00</td>
</tr>
<tr>
<td>48 c7 45 f8 14 00 00 00</td>
</tr>
<tr>
<td>48 8b 45 f8</td>
</tr>
<tr>
<td>48 01 45 f0</td>
</tr>
<tr>
<td>48 8b 45 f0</td>
</tr>
<tr>
<td>5d</td>
</tr>
<tr>
<td>c3</td>
</tr>
</tbody>
</table>
Compilation Steps (.c to a.out)

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

- **Library obj. code (libc.a)**
- **Other object files (p2.o, p3.o, …)**

High-level language

Interface for speaking to CPU

CPU-specific format (011010…)
“Why should I learn Assembly?”

• Because I have to...

• You want to understand how computers work

• You want to learn how to write fast and efficient code

• Assembly is scary at first; eventually it will be scary good
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.
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
How many of these ISAs have you used?  (Don’t worry if you’re not sure. Try to guess based on the types of CPUs/devices you interact with.)

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

A. 0  B. 1-2  C. 3-4  D. 5-6  E. 7+
How many of these ISAs have you used?  (Don’t worry if you’re not sure.  Try to guess based on the types of CPUs/devices you interact with.)

<table>
<thead>
<tr>
<th>ISA</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel IA-32 (80x86)</td>
<td>[Intel ~&lt;2010s]</td>
</tr>
<tr>
<td>ARM</td>
<td>[Macs ~&gt; 2020, phones, routers, etc.]</td>
</tr>
<tr>
<td>MIPS</td>
<td>[routers]</td>
</tr>
<tr>
<td>PowerPC</td>
<td>[Macs &lt; 2006]</td>
</tr>
<tr>
<td>IBM Cell</td>
<td>[Sony PS3]</td>
</tr>
<tr>
<td>Motorola 68k</td>
<td></td>
</tr>
<tr>
<td>Intel x86_64</td>
<td>[Intel &amp; AMD today, PS4]</td>
</tr>
<tr>
<td>Intel IA-64 (Itanium)</td>
<td></td>
</tr>
<tr>
<td>VAX</td>
<td></td>
</tr>
<tr>
<td>SPARC</td>
<td></td>
</tr>
<tr>
<td>Alpha</td>
<td></td>
</tr>
<tr>
<td>IBM 360</td>
<td></td>
</tr>
</tbody>
</table>

A. 0  
B. 1-2  
C. 3-4  
D. 5-6  
E. 7+
ISA Characteristics

• Above ISA: High-level language (C, Python, ...)
  • Hides ISA from users
  • Allows a program to run on any machine
    (after translation by human and/or compiler)

• Below ISA: Hardware implementing ISA can change (faster, smaller, ...)
  • ISA is like a CPU “family”
ISA Characteristics

• Above ISA: High-level language (C, Python, ...)
  • Hides ISA from users
  • Allows a program to run on any machine
    (after translation by human and/or compiler)

• Below ISA: Hardware implementing ISA can change (faster, smaller, ...)
  • ISA is like a CPU “family”
**Instruction Translation**

**sum.c (High-level C)**

```c
long sum(long x, long y) {
    long result;
    result = x + y;
    return result;
}
```

**sum.s (Assembly)**

```assembly
push   %rbp
mov    %rsp,%rbp
mov    %rdi,-0x18(%rbp)
mov    %rsi,-0x20(%rbp)
mov    -0x18(%rbp),%rdx
mov    -0x20(%rbp),%rax
add    %rdx,%rax
mov    %rax,-0x8(%rbp)
mov    -0x8(%rbp),%rax
pop    %rbp
ret
```

**sum.s from sum.c:**

```bash
gcc -S sum.c
```

- Instructions to set up the stack frame and get argument values
- An add instruction to compute sum
- Instructions to return from function
Instruction Translation

sum.c (High-level C)

```c
long sum(long x, long y) {
    long result;
    result = x + y;
    return result;
}
```

sum.s from sum.c:

```assembly
gcc -S sum.c
```

- What should these instructions do?
- What is/isn’t allowed by hardware?
- How complex should they be?

Example: supporting multiplication
Questions?
Multiplexor: Chooses an input value

**Inputs:** $2^N$ data inputs, N signal bits  
**Output:** is one of the $2^N$ input values

- Control signal c, chooses the input for output
  - When c is 1: choose a, when c is 0: choose b

\[\text{out} = (c \land a) \lor (\neg c \land b)\]
N-Way Multiplexor

Choose one of N inputs, need $\log_2 N$ select bits

<table>
<thead>
<tr>
<th>$c_1$</th>
<th>$c_2$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>D0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>D2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>D3</td>
</tr>
</tbody>
</table>

4-Way Multiplexor

C Input to choose D0
Two multiplexors in CPU:
Two multiplexors in CPU:
Two multiplexors in CPU:
Two multiplexors in CPU:
Two multiplexors in CPU:

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

MUX

ALU

Register File
R-S Latch: Stores Value Q

When R and S are both 1: Maintain a value
R and S are never both simultaneously 0

To write a new value:
- Set S to 0 momentarily (R stays at 1): to write a 1
- Set R to 0 momentarily (S stays at 1): to write a 0

\[
\begin{array}{cccc}
S & R & Q/\sim Q & \sim(S\&a) = Q \\
0 & 0 & ND & ND \\
0 & 1 & 1/0 & 1 \\
1 & 0 & 1/0 & 0 \\
1 & 1 & 1/0 & 1 \\
\ldots & \ldots & 0/1 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccc}
R & S & Q/\sim Q & \sim(R\&a) = \sim Q \\
0 & 0 & ND & ND \\
0 & 1 & 1/0 & 0 \\
1 & 0 & 1/0 & 1 \\
1 & 1 & 1/0 & 0 \\
\ldots & \ldots & 0/1 & \ldots \\
\end{array}
\]
Gated D Latch

Controls S-R latch writing, ensures S & R never both 0

D: into top NAND, ~D into bottom NAND
WE: write-enabled, when set, latch is set to value of D

Latches used in registers (up next) and SRAM (caches, later)
Fast, not very dense, expensive

DRAM: capacitor-based:
An N-bit Register

- Fixed-size storage (8-bit, 32-bit, 64-bit, etc.)
- One gated D latch lets us store one bit
  - Connect N of them to the same write-enable wire!
Where are we?

<table>
<thead>
<tr>
<th>Wk</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td></td>
</tr>
</tbody>
</table>

C: programming language

x86 Assembly: instruction set architecture

Binary: logic / bits

CPU / memory: logic / bits

logic gates, circuits: voltage
Compilation Steps (.c to a.out)

- **text**
- **C program (**p1.c**)**
- **Compiler (**gcc**)**
- **Executable code (**a.out**)

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

Reality is more complex: there are intermediate steps!
Compilation Steps (.c to a.out)

- **C program (p1.c)**
- **Assembly program (p1.s)**
- **Object code (p1.o)**
- **Executable code (a.out)**
- **Library obj. code (libc.a)**

**Compiler (gcc -S)**

**Assembler (gcc -c (or as))**

**Linker (gcc (or ld))**

**Text**

- High-level language
- Interface for speaking to CPU

**Binary**

- CPU-specific format (011010...)

**Executable**

- Other object files (p2.o, p3.o, ...)

**Other object files**

- Library obj. code
Assembly Code

Human-readable form of CPU instructions
  • Almost a 1-to-1 mapping to hardware instructions (Machine Code)
  • Hides some details:
    • Registers have names rather than numbers
    • Instructions have names rather than variable-size codes

We’re going to use x86_64 assembly
  • Can compile C to x86_64 assembly on our system:
    
    ```
gcc -S code.c  # open code.s in an editor to view
    ```
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)
“Why should I learn Assembly?”

• Because I have to...

• You want to understand how computers work

• You want to learn how to write fast and efficient code

• Assembly is scary at first; eventually it will be scary good
Instruction Set Architecture (ISA)

- The agreed-upon interface between all software that runs on the machine and the hardware that executes it.
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)!
C statement: \( A = A \times B \)

**Simple instructions:**
- LOAD \( A \), \( R1 \)
- LOAD \( B \), \( R2 \)
- PROD \( R1 \), \( R2 \)
- STORE \( R2 \), \( A \)

**Powerful instructions:**
- MULT \( B \), \( A \)

**Translation:**
Load the values ‘\( A \)’ and ‘\( B \)’ from memory into registers (\( R1 \) and \( R2 \)), compute the product, store the result in memory where ‘\( A \)’ was.
RISC versus CISC (Historically)

• Complex Instruction Set Computing (CISC)
  • Large, rich instruction set
  • More complicated instructions built into hardware
  • Multiple clock cycles per instruction
  • Easier for humans to reason about

• Reduced Instruction Set Computing (RISC)
  • Small, highly optimized set of instructions
  • Memory accesses are specific instructions
  • One instruction per clock cycle
  • Compiler: more work, more potential optimization
So . . . Which System “Won”? 

• Most ISAs (after mid/late 1980’s) are RISC

• The ubiquitous Intel x86 is CISC
  Tablets and smartphones (ARM) taking over?

• x86 breaks down CISC assembly into multiple, RISC-like, machine language instructions

• Distinction between RISC and CISC is less clear
  • Some RISC instruction sets have more instructions than some CISC sets
ISA Examples

- Intel IA-32 (CISC)
- ARM (RISC)
- MIPS (RISC)
- PowerPC (RISC)
- IBM Cell (RISC)
- Motorola 68k (CISC)
- Intel x86_64 (CISC)
- Intel IA-64 (Neither, VLIW)
- VAX (CISC)
- SPARC (RISC)
- Alpha (RISC)
- IBM 360 (CISC)
ISA Characteristics

• Above ISA: High-level language (C, Python, ...)
  • Hides ISA from users
  • Allows a program to run on any machine
    (after translation by human and/or compiler)

• Below ISA: Hardware implementing ISA can change (faster, smaller, ...)
  • ISA is like a CPU “family”
Recall: 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)
Processor State in Registers

- Working memory for currently executing program
  - Temporary data ( %rax - %r15 )

- Location of runtime stack (%rbp, %rsp )

- Address of next instruction to execute ( %rip )

- Status of recent ALU tests ( CF, ZF, SF, OF )

![Diagram showing processor state in registers]

- General purpose registers
  - %rax, %rbx, %rcx, %rdx, %rsi, %rdi, %r8, %r9, %r10, %r11, %r12, %r13, %r14, %r15

- Current stack top
  - %rsp

- Current stack frame
  - %rbp

- Program Counter (PC)
  - %rip

- Condition codes (flags)
  - CF, ZF, SF, OF
Component Registers

• Registers starting with “r” are 64-bit registers

• Sometimes, you might only want to store 32 bits (e.g., int variable)

• You can access the lower 32 bits of a register:
  • with a prefix of e rather than r for registers %rax - %rdi (e.g., %eax, %ebx, ..., %esi, %edi)
  • with a suffix of d for registers %r8 - %r15 (e.g., %r8d, %r9d, ..., %r15d)

%rax  %r8  %r14
%rbx  %r9  %r15
%rcx  %r10
%rdx  %r11
%rsi  %r12
%rdi  %r13
%rsp
%rbp
%rip

General purpose registers
Current stack top
Current stack frame
Program Counter (PC)
Condition codes (flags)
Assembly Programmer’s View of State

**CPU**

<table>
<thead>
<tr>
<th>Registers</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rip</td>
<td>next instr addr (PC)</td>
<td></td>
</tr>
<tr>
<td>%EFLAGS</td>
<td>cond. codes</td>
<td></td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Program:**
- data
- instrs
- stack

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

**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)

Move values between memory and registers or between two registers.
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)

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

MUX

MUX

ALU

(Memory)
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)

Data in
Data in
Data in
Data in

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

MUX
MUX

ALU

Register File
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 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) 

mov $10, %rax 
  • Put the constant value 10 in register rax. 

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.

\texttt{mov (\%rcx), \%rax}

• Use the address in register \%rcx to access memory, store result in register \%rax
Addressing Mode: Memory

```bash
mov (%rcx), %rax
```

- Use the address in register `%rcx` to access memory, store result in register `%rax`

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>0</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Addressing Mode: Memory

mov (%rcx), %rax

- Use the address in register %rcx to access memory, store result in register %rax

1. Index into memory using the address in rcx.
Addressing Mode: Memory

\textbf{mov (\%rcx), \%rax}

- Use the address in register \%rcx to access memory, store result in register \%rax

1. Index into memory using the address in \%rcx.

2. Copy value at that address to \%rax.
Addressing Mode: Displacement

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

\texttt{mov -24(}%rbp\texttt{), }%rax

• Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.
Addressing Mode: Displacement

`mov -24(%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>0</td>
</tr>
<tr>
<td>%rcx</td>
<td>0x1A68</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1A78</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1. Access address: 0x1A78 – 24 => 0x1A60
Addressing Mode: Displacement

`mov -24(\%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>0x1A78</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

1. Access address: `0x1A78 - 24 => 0x1A60`
2. Copy value at that address to rax.
Welcome! *Discuss now with your neighbor:* 

In the reading, we learned about how Toyota didn’t properly protect memory from stack overflow (or properly test its code)—resulting in unintended acceleration of cars.

As future software engineers, how sure would you need to be about the safety of your code before you shipped it?
Announcements

• Lab 3 due Thur 11:59pm
  • Finish **feedback + partner survey before class** on Thursday
  • YOU get to pick your lab partner (**both** have to list each other)
  • Otherwise, randomly assigned

• Lab 4: **watch video** and **complete #1-6 on In-Lab Exercise #5** before lab

• HW grades out
  • Solutions: printed and by my door

• HW 4 due Feb 27\textsuperscript{th}, 11:59pm

• Exam syllabus: first day of class to Feb 29\textsuperscript{th}

• Edstem -> example for how to convert truth table into Boolean expression
Questions?
Where are we?

<table>
<thead>
<tr>
<th>Week</th>
<th>Lecture</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intro to C</td>
<td>C Arrays, Sorting</td>
</tr>
<tr>
<td>2</td>
<td>Binary Representation, Arithmetic</td>
<td>Data Rep. &amp; Conversion</td>
</tr>
<tr>
<td>3</td>
<td>Digital Circuits</td>
<td>Circuit Design</td>
</tr>
<tr>
<td>4</td>
<td>ISAs &amp; Assembly Language</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Pointers and Memory</td>
<td>Pointers and Assembly</td>
</tr>
<tr>
<td>6</td>
<td>Functions and the Stack</td>
<td>Binary Maze</td>
</tr>
<tr>
<td>7</td>
<td>Arrays, Structures &amp; Pointers</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Storage and Memory Hierarchy</td>
<td>Game of Life</td>
</tr>
<tr>
<td>9</td>
<td>Caching</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Operating System, Processing</td>
<td>Strings</td>
</tr>
<tr>
<td>11</td>
<td>Virtual Memory</td>
<td>Unix Shell</td>
</tr>
<tr>
<td>12</td>
<td>Parallel Applications, Threading</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Threading</td>
<td>pthreads Game of Life</td>
</tr>
<tr>
<td>14</td>
<td>Threading</td>
<td></td>
</tr>
</tbody>
</table>

The diagram illustrates the relationship between programming language (C), instruction set architecture (x86 Assembly), and hardware implementation (CPU/mem, logic gates, circuits, voltage). The process of compiling (C) to x86 Assembly and then assembled to binary (0s and 1s) is depicted, followed by logic gates and circuits to voltage.
Recall: Assembly Programmer’s View

**CPU**

**Registers**

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<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>%EFLAGS</td>
<td>cond. codes</td>
</tr>
</tbody>
</table>

**Main Memory**

**Addresses**

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000001</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Data**

Program: data instrs stack

**Instructions**

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

**Main Memory:**
- Byte addressable array
- Program code and data
- Execution stack
Recall: Assembly Programmer’s View

CPU

64 bits (8 bytes)

rax
rax: all 8 bytes
⇒ movq: all 8 bytes

rbx
rax: bytes 0-3
⇒ movl: bytes 0-3

r15
ax: bytes 0-1
⇒ movw: bytes 0-1

al: byte 0
⇒ movb: byte 0
Recall: Assembly Programmer’s View

**64 bits (8 bytes)**

- **CPU**
  - **rax**
    - All 8 bytes
  - **rbx**
    - 0x0
  - **r15**
    - Register

**Main Memory**

- **Hex address** (64 bits / 16 hex digits)
  - 0x0000000000000000
  - 01000001
  - 10101010

- **“byte addressable memory”**
  - 0x0

- **Largest possible memory address**
  - = largest possible address a register can hold
  - = $2^{64} - 1 = 18,446,744,073,709,551,616 - 1$

**Example Mov Instructions**

- **movq** $0x0$, %rbx
- **movb** (%rbx), %al

**Diagram Highlights**

- **movq**: All 8 bytes
- **movl**: Bytes 0-3
- **movw**: Bytes 0-1
- **movb**: Byte 0
Recall: Assembly Programmer’s View

CPU

Main Memory

- **rax**: all 8 bytes
- **ebx**: bytes 0-3
- **ax**: bytes 0-1
- **al**: byte 0

- **movq**: all 8 bytes
- **movl**: bytes 0-3
- **movw**: bytes 0-1
- **movb**: byte 0

**largest possible memory address**

= largest possible address a register can hold

= $2^{64} - 1 = 18,446,744,073,709,551,616 - 1$
Recall: Assembly Programmer’s View

CPU

r15
register

rax
register

rbx
register

64 bits (8 bytes)

Main Memory

hex address (64 bits / 16 hex digits)

0x0000000000000000
0x0000000000000001
0x0000000000000010
0x0000000000000011
0x0000000000000017
0xFFFFFFFFFFFFFFFF

largest possible memory address
= largest possible address a register can hold
= $2^{64} - 1 = 18,446,744,073,709,551,616 - 1$
Recall: Assembly Programmer’s View

**CPU**

- **rax**: 64 bits (8 bytes) as a register
- **rbx**: 0x10
- **r15**: register

**Main Memory**

- Hex address (64 bits / 16 hex digits)
  - **0x0000000000000000**
  - **0x0000000000000001**

- Largest possible memory address

- LARGEST POSSIBLE MEMORY ADDRESS
  - Largest possible address a register can hold
  - \(2^{64} - 1 = 18,446,744,073,709,551,616 - 1\)
  - Largest memory storage size
  - \(2^{64} = 17,179,869,184,000,000,000,000\) bytes
  - \(17,179,869,184,000,000,000,000\) gigabytes (approx.)

**Move Instructions**

- **movq**: all 8 bytes
- **movl**: bytes 0-3
- **movw**: bytes 0-1
- **movb**: byte 0

- **movq $0x10, %rbx**
- **movq (%rbx), %rax**

- **START**: 17th byte
- **18th byte**
- **24th byte**

- 8 bytes (64 bits)
- 1 byte (8 bits)
Recall: Component Registers

- Registers starting with “r” are 64-bit registers

- Sometimes, you might only want to store 32 bits (e.g., int variable)

- You can access the lower 32 bits of a register:
  - with a prefix of e rather than r for registers %rax - %rdi
    (e.g., %eax, %ebx, ..., %esi, %edi)
  - with a suffix of d for registers %r8 - %r15
    (e.g., %r8d, %r9d, ..., %r15d)
Recall: 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
Recall: Addressing Modes

• Instructions need to be told where to get operands or store results. Variety of options for how to address those locations

• Four different addressing modes:
  • A register: %rax  %r15
  • An immediate value: $10  $0x1F  $0b000111
  • A location in memory: (%rax)
  • A location in memory with displacement: -16(%rbp)

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

A. Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>2</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1FFF000AD0</td>
<td>3</td>
</tr>
<tr>
<td>0x1FFF000AD8</td>
<td>10</td>
</tr>
<tr>
<td>0x1FFF000AE0</td>
<td>0x1FFF000AF0</td>
</tr>
</tbody>
</table>

B. Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>10</td>
</tr>
<tr>
<td>%rsp</td>
<td>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</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>10</td>
</tr>
<tr>
<td>0x1FFF000AE0</td>
<td>0x1FFF000AF0</td>
</tr>
</tbody>
</table>

C. 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>0x1FFF000AE0</td>
</tr>
<tr>
<td>%rbp</td>
<td>0x1FFF000AE0</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>
Solution

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

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

C code equivalent:

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

Subtract 16 from %rsp, %rsp <- 0x...AD0

Move constant 3 to value at 0x...AD8 (x)

Move constant 10 to register %rax

Shift the value in %rax left by 1 bit

Add the value at 0x...AD8 (x) to %rax

Store the value in %rax at 0x...AD0 (y)

Add 16 to %rsp, %rsp <- 0x...AE0

---

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Value</td>
</tr>
<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>
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.

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

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

label1:
    return;
}

A label is a place you might jump to.

Labels ignored except for goto/jumps.

(Skipped over if encountered)

int x = 20;

L1:
    int y = x + 30;

L2:
    printf("%d, %d\n", x, y);
Unconditional Jumping / Goto

```c
int main(void) {
    long a = 10;
    long b = 20;
    goto label1;
    a = a + b;
    label1:
    return;
}
```

```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
```
Unconditional Jumping / Goto

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

- Often, we only want to jump when *something* is true / false

- 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 (e.g., cmp, test)

• x86_64 condition codes tell you:
  • ZF — zero flag — if the result is zero
  • SF — sign flag — if the result’s first bit is set (negative if signed)
  • CF — carry flag — if the result overflowed (assuming unsigned) [“carried”]
  • OF — overflow flag — if the result overflowed (assuming signed)
Processor State in Registers

- Working memory for currently executing program
  - Temporary data (%rax - %r15)
- Location of runtime stack (%rbp, %rsp)
- 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: Does not change state of registers, only condition codes

   `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 results 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 &)


Conditional Jumping

• Jump based on which condition codes are set

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

You do not need to memorize these!

<table>
<thead>
<tr>
<th></th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF&amp;~ZF</td>
<td>Above (unsigned jg)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Example Scenario

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

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

- Suppose user gives us a value via scanf (don’t know value in advance)

- We want to check to see if it equals 42
  - If so, add 5
  - If not, subtract 10
Assembly Visualization Demo: Jump

• Try this in **arithmetic** mode:

https://asm.diveintosystems.org

Change the value 3 to 42 to alter the behavior.

```assembly
# Initialize rax
mov $3, %rax

cmp $42, %rax
je L2

L1:
sub $10, %rax
jmp DONE

L2:
add $5, %rax
DONE:
```
Loops

• We’ll look at 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)