211: Computer Architecture
Summer 2016

Liu Liu

Topic:
- Data Representation
- Assembly Programming
Recap

- Data representation:
  - Numerical numbers in B / O / D / H
  - Endian / Complement
  - Floating number
Today’s Topic

- Data representation:
  - Complement
  - Floating number (Review)
  - Characters

- Computer Arithmetic:
  - add / sub / mul / div (unsigned / signed)

- Assembly Programming:
  - instruction format
  - addressing
Review: 1’s 2’s complement

- unsigned integer:  6=>110   7=>111
- by adding a sign-bit:
  - 6: 0110
  - 7: 0111
  - -6: 1110
  problem:?
- 1’s complement(flipping all bits):
  - 6: 0110
  - 7: 0111
  - -6: 1001
  problem:?
- 2’s complement(flipping all bits):
  - 6: 0110
  - 7: 0111
  - -6: 1010
Review: endian

- little endian: LSB first
- big endian: MSB first

why little endian?

1 in single byte: 00000001
1 in 2-bytes: 00000000 00000001
1 in 4-bytes: 00000000 00000000 00000000 00000001
...

Floating point

Integers typically written in ordinary decimal form
  ▪ E.g., 1, 10, 100, 1000, 10000, 12456897, etc.

But, can also be written in scientific notation
  ▪ E.g., 1x10^4, 1.2456897x10^7

What about binary numbers?
  ▪ Works the same way
  ▪ 0b100 = 0b1x2^2

Scientific notation gives a natural way for thinking about floating point numbers
  ▪ 0.25 = 2.5x10^{-1} = 0b1x2^{-2}

How to represent in computers?
IEEE floating point standard

Most computers follow IEEE 754 standard

- Single precision (32 bits)
- Double precision (64 bits)
- Extended precision (80 bits)

| S | Exponent | Fraction |
Floating point in C

32 bits single precision (type float)
- 1 bit for sign, 8 bits for exponent, 23 bits for mantissa
  - Sign bit: 1 = negative numbers, 0 = positive numbers
  - Exponent is power of 2
- Have 2 zero’s
- Range is approximately $-10^{38}$ to $10^{38}$

64 bits double precision (type double)
- 1 bit for sign, 11 bits for exponent, 52 bits for mantissa
- Majority of new bits for mantissa $\rightarrow$ higher precision
- Range is $-10^{308}$ to $+10^{308}$

$1.02 \times 10^{-10}$ $1.02 \times 10^{+10}$, which one is has higher precision?
Numerical Values

3 different cases:

- 1) Normalized values
  - exponent field $\neq 0$ and exponent field $\neq 2^{k-1}$ (all 1’s)
  - exponent = binary value – Bias
    - Bias = $2^{k-1}-1$ (e.g., 127 for float)
  - mantissa = 1.(mantissa field)
  - Ex: (sign: 0, exp: 1, mantissa: 1) would give 0b1.1x2^{-126}

- 2) Denormalized values
  - exponent field = 0
  - exponent = 1 – Bias (e.g., -126 for float), why not -127?
  - Mantissa = mantissa field (no leading 1)
  - Ex: (sign: 0, exp: 0, mantissa: 10) would give 0b10x2^{-126}
Numerical Values

3 different cases:

- Special values: $+\infty$, $-\infty$, 0, and NaN
  - what is the value of 32bit all ‘0’? Is it $1.0 \times 2^{-127}$? -> 0
  - +/- $\infty$ occurs when dividing a number with 0.0.
  - What is 0.0 / 0.0? -> NaN (ill-defined value)
  - Square root of a neg value? -> NaN
# Numerical Values

<table>
<thead>
<tr>
<th>Category</th>
<th>Sign Bit</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>Anything</td>
<td>$0^8$</td>
<td>$0^{23}$</td>
</tr>
<tr>
<td>Infinity</td>
<td>Anything</td>
<td>$1^8$</td>
<td>$0^{23}$</td>
</tr>
<tr>
<td>NaN</td>
<td>Anything</td>
<td>$1^8$</td>
<td>Not $0^{23}$</td>
</tr>
<tr>
<td>Denormalized numbers</td>
<td>Anything</td>
<td>$0^8$</td>
<td>Not $0^{23}$</td>
</tr>
<tr>
<td>Normalized numbers</td>
<td>Anything</td>
<td>Not $0^8$, nor $1^8$</td>
<td>Anything</td>
</tr>
</tbody>
</table>

Decimal to IEEE Floating Point

5.625

In binary

101.101 → 1.01101 \times 2^2

Exponent field has value 2
  - add 127 to get 129

Exponent is 10000001

Mantissa is 01101

Sign bit is 0

0 10000001 01101000000000000000000
One more example

Convert 12.375 to floating point representation

Binary is 1100.011

\[ 1.100011 \times 2^3 \]

Exponent = 127 + 3 = 130 = 0b10000010
Mantissa = 100011
Sign = 0
Extended precision

80 bits used to represent a real number
1 sign bit, 15 bit exponent, 64 bit mantissa
20 decimal digits of accuracy
10^{-4932} to 10^{4932}
Not supported in C

What about characters?
ASCII

A character is stored as 1 byte according to the ASCII standard.

Originally used only 128 values (7 bits)
  - One bit could be used for error detection (will discuss later)

Subsequently extended to use all 256 values.
## ASCII table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>DLE</td>
<td>space</td>
<td>@</td>
<td>P</td>
<td>`</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>DC1</td>
<td>XON</td>
<td>!</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>DC3</td>
<td>XOFF</td>
<td>#</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>7</td>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
</tr>
<tr>
<td>C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td>E</td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
</tr>
<tr>
<td>F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>del</td>
</tr>
</tbody>
</table>
What about characters for other languages?
Unicode is a standard that defines more than 107,000 characters across 90 scripts (and more …)
Unicode can be implemented by different character encodings
Most common: UTF-8
- Variable length encoding of Unicode: 1-4 bytes for each character
- 1-byte form is reserved for ASCII for backward compatibility
Data sizes

All Information is represented in binary form but require different sizes

Characters in 1 byte, integers 2 to 4 bytes, real numbers 4 to 8 bytes

<table>
<thead>
<tr>
<th>C declaration</th>
<th>32-bit machine</th>
<th>64-bit machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Computer Arithmetic

Then, we are going to revisit our mathematic 101

Why?

- Finite number of bits
- Number representation (e.g., two’s complement)
- Develop algorithms easy to implement in hardware
Unsigned Addition and Subtraction

\[
\begin{array}{c}
5 \\
+ 6 \\
\hline
11
\end{array}
\quad \begin{array}{c}
0101 \\
0110 \\
\hline
1011
\end{array}
\]

\[
\begin{array}{c}
6 \\
- 5 \\
\hline
1
\end{array}
\quad \begin{array}{c}
0110 \\
0101 \\
\hline
0001
\end{array}
\]
Overflow

What happens to the following addition if each number is represented by only 4 bits?

\[
\begin{array}{cccc}
8 & 1000 \\
+ & 10 & 1010 \\
\hline
18 & 10010 \\
\end{array}
\]

Adding two n bits numbers may result in an n+1 bit number

No space to hold extra bit → modulo arithmetic

\[
x + y \text{ is really } (x + y) \mod 2^n, \text{ where } n = \text{number of bits}
\]

\[
(8 + 10) \mod 2^4 = 2
\]
An overflow occurs when the result cannot fit within the size limit of the data type

When executing C programs, overflows are not signaled as errors!

Programmer must figure it out

When two unsigned numbers $x$, $y$ are added

Overflow has occurred if

- $x+y < x$ or $x+y < y$
2’s Complement Addition & Subtraction

Addition = binary addition, ignore carry out

Subtraction = invert subtrahend and add

\[
\begin{array}{c}
-7 \quad 1001 \\
+ \quad 5 \quad 0101 \\
\hline
\quad -2 \quad 1110
\end{array}
\]

\[
\begin{array}{c}
-5 \quad 1011 \\
+ \quad -2 \quad 1110 \\
\hline
\quad -7 \quad 11001
\end{array}
\]

\[4 - 2 = 4 + -2\]

\[
\begin{array}{c}
4 \quad 0100 \\
+ \quad -2 \quad 1110 \\
\hline
\quad 2 \quad 0010
\end{array}
\]
2’s Complement Addition

Why does it work?

- Adding 2 positive numbers: obvious

- Adding positive and negative number?
  - |pos number| > |neg number| → Will wrap around → 0 MSB
  - |neg number| > |pos number| → Will not wrap → 1 MSB

- Adding 2 negative numbers just like adding two positive numbers
  - Will always wrap → 1 MSB is preserved
2’s Complement Overflow

What happens when overflow occurs with 2’s complement?

- Need one extra bit so sign bit will be wrong

\[
\begin{array}{c}
6 & 0110 \\
+ & 5 & 0101 \\
\hline
-5 & 1011
\end{array}
\quad
\begin{array}{c}
-6 & 1010 \\
+ & -6 & 1010 \\
\hline
4 & 0100
\end{array}
\]

- How to detect?
  - Adding 2 positive numbers \(\rightarrow\) negative result
  - Adding 2 negative numbers \(\rightarrow\) positive result
Unsigned Multiply

How can we compute $n \times m$?

$$\sum_{i=1}^{m} n$$

Not very efficient …
Unsigned Multiplication

\[
\begin{array}{c}
54 \\
\times 67 \\
\hline
378 \\
324 \\
\hline
3618
\end{array}
\]

\[
\begin{array}{c}
1011 \\
\times 101 \\
\hline
1011 \\
0000 \\
1011 \\
\hline
110111
\end{array}
\]

Notice anything about the binary form?
Algorithm

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
\times & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
\hline
1 & 1 & 0 & 1 & 1 & 1
\end{array}
\]

(multiplicand x 1) shift left 0
(multiplicand x 0) shift left 1
(multiplicand x 1) shift left 2
Algorithm

1. result = 0
2. If LSB of multiplier = 1, add multiplicand to result
   else, add 0 to result
3. Shift current result left by 1 bit (fill LSB with 0)
4. Shift multiplier right by 1 bit (fill MSB with 0)
5. If multiplier > 0, go to 1
6. Stop

We only need to know how to add and shift in order to multiply
Overflow

The result of multiplying two n-bits long numbers can be up to 2n-bits long

What to do with the extra bits?

- Discard n most significant bits $\rightarrow$ modulo arithmetic
- Some processors leave results in two registers so programmer can have entire result if desired
Signed Multiplication

When multiplying two n-bits signed number, if you just want the n least significant bits of the result, then a very slightly modified algorithm works

- Book develops a mathematical proof

If want all 2n bits of result, then what?

- Convert both multiplicand and multiplier to positive numbers
- Multiply
- Apply correct sign
  - Same signs before: positive
  - Different signs before: negative

\[
\begin{array}{c}
111 \\
\times 111 \\
\hline
111 \\
111 \\
111 \\
\hline
110001
\end{array}
\]
Algorithm

1. result = 0
2. If LSB of multiplier = 1, add multiplicand to result else, add 0 to result
3. Shift multiplicand left by 1 bit (fill LSB with 0)
4. Shift multiplier right by 1 bit (fill MSB with sign bit)
5. If multiplier > 0, go to 1
6. Stop
## Sign extension

<table>
<thead>
<tr>
<th>Signed integer</th>
<th>4-bit representation</th>
<th>8-bit representation</th>
<th>16-bit representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>0001</td>
<td>00000001</td>
<td>000000000000000001</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
<td>11111111</td>
<td>111111111111111111</td>
</tr>
</tbody>
</table>
Unsigned Division

How can we compute n / m?

- Count how many times we can subtract n from m until the remainder r is less than m
- Call this count q

\[
\frac{n}{m} = q + r
\]

Again, not very efficient …
Unsigned Division

53 ÷ 7 = 7 rem 4
Algorithm

1. Shift divisor left, filling LSB with 0, until same number of bits as dividend
2. quotient = 0, remainder = dividend
3. If remainder < original divisor, stop
4. If remainder < divisor, shift quotient left, fill LSB with 0
5. If remainder \( \geq \) divisor, shift quotient left, fill LSB with 1
6. remainder = remainder – divisor
7. Shift divisor right, fill MSB with 0
8. Go to 3

Only need subtract and shift to implement division
Signed Division?

Change any negative number to positive number
Perform division
Compute signs of quotient and remainder
Negate quotient and/or remainder if necessary
What About Floating Point?

Addition and Subtraction
- Scientific form: have to align “.” so exponents are the same
- Sign/magnitude representation
  - Need both addition and subtraction
    » Can convert magnitude to 2’s complement to avoid subtraction
  - Set sign correctly after compute new mantissa
- Reset to scientific form

Multiplication (Division?)
- Multiply mantissas (keep track of “.”)
- Add exponents
- Set sign correctly
- Reset to scientific form
Break
Assembly Programming

Brief tour through assembly language programming

Why?

- Machine interface: where software meets hardware
- To understand how the hardware works, we have to understand the interface that it exports

Why not binary language?

- Much easier for humans to read and reason about
- Major differences:
  - Human readable language instead of binary sequences
  - Relative instead of absolute addresses
# Programming Meets Hardware

## High-Level Language Program

```c
#include <stdio.h>
int main() {
    int x, y, temp;
    x=1; y=2;
    temp =x; x=y; y=temp;
    printf("%d %d %d\n",x,y,temp);
}
```

## Assembly Language Program

```assembly
movl $1, -8(%ebp)
movl $2, -12(%ebp)
movl -8(%ebp), %eax
movl %eax, -16(%ebp)
movl -12(%ebp), %eax
movl %eax, -8(%ebp)
movl -16(%ebp), %eax
movl %eax, 12(%esp)
movl -12(%ebp), %eax
movl %eax, 8(%esp)
movl %eax, 4(%esp)
```

## Machine Language Program

```
7f 45 4c 46 01 01 01
00 00 00 00 00 00 00
00 00 02 00 03 00 01
00 00 00 f0 82 04 08
34 00 00 00 c4 0c 00
00 00 00 00 00 34 00
```
There are many different assembly languages because they are processor-specific

- **IA32 (x86)**
  - x86-64 for new 64-bit processors
  - IA-64 radically different for Itanium processors
- **PowerPC**
- **MIPS**

We will focus on IA32 because you can generate and run on iLab machines (as well as your own PC/laptop)

- IA32 is also dominant in the market although smart phone, eBook readers, etc. are changing this
Aside About Implementation of x86

About 30 years ago, the instruction set actually reflected the processor hardware
- E.g., the set of registers in the instruction set is actually what was present in the processor

As hardware advanced, industry faced with choice
- Change the instruction set: bad for backward compatibility
- Keep the instruction set: harder to exploit hardware advances
  - Example: many more registers but only small set introduced circa 1980

Starting with the P6 (PentiumPro), IA32 actually got implemented by Intel using an “interpreter” that translates IA32 instructions into a simpler “micro” instruction set
P6 Decoder/Interpreter
Assembly Programmer’s View

CPU

ALU
Condition Codes
Control Logic
PC
Registers

Memory

(OS code & data)
Object Code
Program Data

Addresses
Data
Instructions
Memory

CPU

Address

Command: R/W

Data

Storage

Memory

Addresses

00 00101100
01 10001000
02 11111111
03 01010101
04 00000000
05 11000001
06 00000000
07 11111101
08 11111100
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011101
0F 00000000
# Memory Access: Read

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00101100</td>
</tr>
<tr>
<td>01</td>
<td>10001000</td>
</tr>
<tr>
<td>02</td>
<td>11111111</td>
</tr>
<tr>
<td>03</td>
<td>01010101</td>
</tr>
<tr>
<td>04</td>
<td>00000000</td>
</tr>
<tr>
<td>05</td>
<td>11000001</td>
</tr>
<tr>
<td>06</td>
<td>00000000</td>
</tr>
<tr>
<td>07</td>
<td>11111001</td>
</tr>
<tr>
<td>08</td>
<td>11111000</td>
</tr>
<tr>
<td>09</td>
<td>00110000</td>
</tr>
<tr>
<td>0A</td>
<td>00000000</td>
</tr>
<tr>
<td>0B</td>
<td>00000000</td>
</tr>
<tr>
<td>0C</td>
<td>00000000</td>
</tr>
<tr>
<td>0D</td>
<td>11000011</td>
</tr>
<tr>
<td>0E</td>
<td>00011001</td>
</tr>
<tr>
<td>0F</td>
<td>00000000</td>
</tr>
</tbody>
</table>
Memory Access: Write

CPU

Memory

Addresses

00 00101100
01 10001000
02 11111111
03 01010101
04 00000000
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000
Memory Access: Write

The diagram illustrates the memory access process during a write operation. The CPU's address bus sends the address `01010101` to the memory. The memory then transfers data to the CPU, as indicated by the arrow labeled `W`. The memory locations are organized in hexadecimal format, with addresses ranging from `00` to `0F`. The data stored at these addresses is also shown.
Processor: ALU & Registers

**ALU**

\[ C = F_S(A, B) \]

**F includes**
- **Arithmetic**: +, -, *, /, ~, etc.
- **Logical**: <, >, =, etc.

**Registers**

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>00101100</td>
</tr>
<tr>
<td>R1</td>
<td>10001000</td>
</tr>
<tr>
<td>R2</td>
<td>11111111</td>
</tr>
<tr>
<td>R3</td>
<td>01010101</td>
</tr>
</tbody>
</table>
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

ALU

Control Logic

Registers

Memory

Storage

Addresses

00  00101100
01  10001000
02  11111111
03  01010101
04  01010101
05  11000001
06  00000000
07  11111001
08  11111000
09  00110000
0A  00000000
0B  00000000
0C  00000000
0D  11000011
0E  00011001
0F  00000000
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

1

ALU

Control Logic

Registers

Memory

Storage

Addresses

00 00101100
01 10001000
02 11111111
03 01010101
04 01010101
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

ALU

Control Logic

Registers

Memory

Storage

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00101100</td>
</tr>
<tr>
<td>01</td>
<td>10001000</td>
</tr>
<tr>
<td>02</td>
<td>11111111</td>
</tr>
<tr>
<td>03</td>
<td>01010101</td>
</tr>
<tr>
<td>04</td>
<td>01010101</td>
</tr>
<tr>
<td>05</td>
<td>11000001</td>
</tr>
<tr>
<td>06</td>
<td>00000000</td>
</tr>
<tr>
<td>07</td>
<td>11111001</td>
</tr>
<tr>
<td>08</td>
<td>11111000</td>
</tr>
<tr>
<td>09</td>
<td>00110000</td>
</tr>
<tr>
<td>0A</td>
<td>00000000</td>
</tr>
<tr>
<td>0B</td>
<td>00000000</td>
</tr>
<tr>
<td>0C</td>
<td>00000000</td>
</tr>
<tr>
<td>0D</td>
<td>11000011</td>
</tr>
<tr>
<td>0E</td>
<td>00011001</td>
</tr>
<tr>
<td>0F</td>
<td>00000000</td>
</tr>
</tbody>
</table>
Putting It All Together

CPU

Condition Codes

Program Counter (PC)

ALU

Control Logic

Registers

R0: x

R1: y

Memory

Addresses

Storage

00 00101100
01 10001000
02 11111111
03 01010101
04 01010101
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000
Putting It All Together

ALU

Control Logic

Program Counter (PC)

Condition Codes

Registers

Memory

Storage

Addresses

00 00101100
01 10001000
02 11111111
03 01010101
04 01010101
05 11000001
06 00000000
07 11111001
08 11111000
09 00110000
0A 00000000
0B 00000000
0C 00000000
0D 11000011
0E 00011001
0F 00000000
Putting It All Together
Basic CPU Function

- FETCH[PC++]
- DECODE
- EXECUTE

Arithmetic: +, -, *, /
Logic: bre, jmp
Assembly Characteristics

Sequence of simple instructions

Minimal Data Types

- “Integer” data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

No type checking

- Interpretation of data format depends on instruction
- No protection against misinterpretation of data
Assembly Characteristics

Primitive Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
x86 Characteristics

Variable length instructions: 1-15 bytes
Can address memory directly in most instructions
Uses Little-Endian format
Instruction Format

General format:

```
opcode operands
```

Opcode:
- Short mnemonic for instruction’s purpose
  - `movb, addl, etc.`

Operands:
- Immediate, register, or memory
- Number of operands command-dependent

Example:
- `movl %ebx, (%ecx)`
Machine Representation

Remember, each assembly instruction translated to a sequence of 1-15 bytes

| opcode byte | addressing mode byte | other bytes |

First byte holds the binary representation of the opcode

Second byte specifies the addressing mode
  - The type of operands (registers or register and memory)
  - How to interpret the operands

Some instructions can be single-byte because operands and addressing mode are implicitly specified by the instruction
  - E.g., pushl
MOV instruction

Most common instruction is data transfer instruction

- mov S, D
  - Copy value at S to D

Used to copy data from: (what’s missing?)

- Memory to register
- Register to memory
- Register to register
- Constant to register
Data Formats

Byte: 8 bits
  - E.g., char

Word: 16 bits (2 bytes)
  - E.g., short int

Double Word: 32 bits (4 bytes)
  - E.g., int, float

Quad Word: 64 bits (8 bytes)
  - E.g., double

Instructions can operate on any data size
  - movl, movw, movb
    - Move double word, word, byte, respectively
  - End character specifies what data size to be used