Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis

http://xkcd.com/138/
To-Do List

- Start-of-Course survey [Catalyst] due Thursday (3/30)
- Section 1 is this Thursday
  - Install the virtual machine (VM) before coming to section
  - Bring your computer with you to section
- Lab 0, due Monday (4/3) @ 11:59pm
- Homework 1, due Monday (4/3) @ 11:59pm
- Readings in CSAPP
Lab 0, Homework 1, and Readings

- **Lab 0, due Monday (4/3) @ 11:59pm**
  - Basic exercises to start getting familiar with C – need the VM
  - Credit/no-credit
  - Do ASAP, attending Section 1 will help

- **Homework 1, due Monday (4/3) @ 11:59pm**
  - 3 canvas quizzes, 20 tries each, you best overall score is kept
    - Course policies – you can do this one now!
    - Unsigned Number Representations – will discuss on Wed
    - Number Bases - will discuss on Wed

- **Readings:**
  - For Mon: CSPP: § 1.0-1.10 (pp. 1-28)
  - For Wed: CSPP: § 2.0-2.1.3 (pp. 31-48)
  - For Fri: CSPP: § 2.1.3-2.1.5 (pp. 42-50)
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
mpg =
c.getMPG();

Assembly language:

get_mpg:
  pushq %rbp
  movq %rsp, %rbp
  ...
  popq %rbp
  ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
11000001111111011000011111

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:

Windows 8
Mac

L02: Memory & Data I
CSE351, Spring 2017
Hardware: Logical View

CPU

Memory

Bus

Disks

Net

USB

Etc.
Hardware: Physical View
CPU executes instructions; memory stores data

To execute an instruction, the CPU must:

- fetch an instruction;
- fetch the data used by the instruction; and, finally,
- execute the instruction on the data...
- which may result in writing data back to memory
The CPU holds instructions temporarily in the **instruction cache**.
The CPU holds data temporarily in a fixed number of **registers**.
Instruction and operand fetching is **hardware-controlled**.
Data movement is **programmer-controlled** (in assembly).
We’ll learn about the instructions the CPU executes – take CSE/EE470 to find out how it actually executes them.
The CPU holds data temporarily in a fixed number of registers.

Instruction and operand fetching is hardware-controlled.

Data movement is programmer-controlled (in assembly).

We’ll learn about the instructions the CPU executes – take CSE/EE470 to find out how it actually executes them.

How are data and instructions represented?

How does a program find its data in memory?
Memory, Data, and Addressing

- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
Question 1:

How are data and instructions represented?
Binary Representations

- **Base 2 number representation**
  - A base 2 digit (0 or 1) is called a *bit*
  - Represent $351_{10}$ as $0000001010111111_2$ or $101011111_2$

- **Electronic implementation**
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires

![Diagram showing voltage levels for binary representation](image)
Review: Number Bases

- **Key terminology:** digit \((d)\) and base \((B)\)
  - In base B, each digit is one of B possible symbols
- Value of \(i\)-th digit is \(d \times B^i\) where \(i\) starts at 0 and increases from right to left
  - \(n\) digit number \(d_{n-1}d_{n-2} \ldots d_1d_0\)
  - value = \(d_{n-1} \times B^{n-1} + d_{n-2} \times B^{n-2} + \ldots + d_1 \times B^1 + d_0 \times B^0\)
  - In a **fixed-width** representation, left-most digit is called the **most-significant** and the right-most digit is called the **least-significant**
- **Notation:** Base is indicated using either a prefix or a subscript
Describing Byte Values

- **Binary** \((00000000_2 \text{ – } 11111111_2)\)
  - Byte = 8 bits (binary digits)
  
<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

- **Decimal** \((0_{10} \text{ – } 255_{10})\)

- **Hexadecimal** \((00_{16} \text{ – } FF_{16})\)
  - Byte = 2 hexadecimal (or “hex” or base 16) digits
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37B\(_{16}\) in the C language
    - as 0xFA1D37B or 0xfa1d37b

- More on specific data types later...
Question 2:

How does a program find its data in memory?
Byte-Oriented Memory Organization

- Conceptually, memory is a single, large array of bytes, each with a unique address (index)
- The value of each byte in memory can be read and written
- Programs refer to bytes in memory by their addresses
  - Domain of possible addresses = address space
- But not all values (e.g., 351) fit in a single byte...
  - Store addresses to “remember” where other data is in memory
  - How much memory can we address with 1-byte (8-bit) addresses?
- Many operations actually use multi-byte values
Machine Words

- Word size = address size = register size
- Word size bounds the size of the address space and memory
  - word size = \( w \) bits \( \rightarrow \) \( 2^w \) addresses
- Current x86 systems use 64-bit (8-byte) words
  - Potential address space: \( 2^{64} \) addresses
    \( 2^{64} \) bytes \( \approx 1.8 \times 10^{19} \) bytes
    = 18 billion billion bytes
    = 18 EB (exabytes) = 16 EiB (exbibytes)
  - Actual physical address space: 48 bits
Aside: Units and Prefixes

- Here focusing on large numbers (exponents > 0)
- Note that $10^3 \approx 2^{10}$
- SI prefixes are ambiguous if base 10 or 2
- IEC prefixes are unambiguously base 2

### SIZE PREFIXES (10$^X$ for Disk, Communication; 2$^X$ for Memory)

<table>
<thead>
<tr>
<th>SI Size</th>
<th>Prefix</th>
<th>Symbol</th>
<th>IEC Size</th>
<th>Prefix</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>Kilo-</td>
<td>K</td>
<td>$2^{10}$</td>
<td>Kibi-</td>
<td>Ki</td>
</tr>
<tr>
<td>$10^6$</td>
<td>Mega-</td>
<td>M</td>
<td>$2^{20}$</td>
<td>Mebi-</td>
<td>Mi</td>
</tr>
<tr>
<td>$10^9$</td>
<td>Giga-</td>
<td>G</td>
<td>$2^{30}$</td>
<td>Gibi-</td>
<td>Gi</td>
</tr>
<tr>
<td>$10^{12}$</td>
<td>Tera-</td>
<td>T</td>
<td>$2^{40}$</td>
<td>Tebi-</td>
<td>Ti</td>
</tr>
<tr>
<td>$10^{15}$</td>
<td>Peta-</td>
<td>P</td>
<td>$2^{50}$</td>
<td>Pebi-</td>
<td>Pi</td>
</tr>
<tr>
<td>$10^{18}$</td>
<td>Exa-</td>
<td>E</td>
<td>$2^{60}$</td>
<td>Exbi-</td>
<td>Ei</td>
</tr>
<tr>
<td>$10^{21}$</td>
<td>Zetta-</td>
<td>Z</td>
<td>$2^{70}$</td>
<td>Zebi-</td>
<td>Zi</td>
</tr>
<tr>
<td>$10^{24}$</td>
<td>Yotta-</td>
<td>Y</td>
<td>$2^{80}$</td>
<td>Yobi-</td>
<td>Yi</td>
</tr>
</tbody>
</table>
Word-Oriented Memory Organization

- Addresses specify locations of bytes in memory
  - Address of word = address of first byte in word
  - Addresses of successive words differ by word size (in bytes): e.g., 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, ... 10?

<table>
<thead>
<tr>
<th>64-bit Words</th>
<th>32-bit Words</th>
<th>Bytes</th>
<th>Addr. (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x01</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x02</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x04</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x05</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x06</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x07</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x08</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x09</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x0A</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x0B</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x0C</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x0D</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x0E</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0x0F</td>
</tr>
</tbody>
</table>
Word-Oriented Memory Organization

Addresses still specify locations of *bytes* in memory

- Address of word = address of first byte in word
- Addresses of successive words differ by word size (in bytes): *e.g.*, 4 (32-bit) or 8 (64-bit)
- Address of word 0, 1, ... 10?
- *Alignment*
A Picture of Memory (64-bit view)

- A “64-bit (8-byte) word-aligned” view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - A 64-bit pointer will fit on one row
A Picture of Memory (64-bit view)

- A “64-bit (8-byte) word-aligned” view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - A 64-bit pointer will fit on one row
Addresses and Pointers

- An **address** is a location in memory
- A **pointer** is a data object that holds an address
  - Address can point to *any* data
- Value 351 stored at address **0x08**
  - \(351_{10} = 15F_{16}\)
  - \(= 0x\ 00\ 00\ 01\ 5F\)
- Pointer stored at **0x38** points to address **0x08**
Addresses and Pointers

- An **address** is a location in memory

- A **pointer** is a data object that holds an address
  - Address can point to *any* data

- Pointer stored at **0x48** points to address **0x38**
  - Pointer to a pointer!

- Is the data stored at **0x08** a pointer?
  - Could be, depending on how you use it

### 64-bit example (pointers are 64-bits wide)

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>01</th>
<th>5F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address 0x00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address 0x48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Representations

- Sizes of data types (in bytes)

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>32-bit (old)</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>short</td>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>(reference)</td>
<td>pointer *</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

To use “bool” in C, you must #include <stdbool.h>
More on Memory Alignment in x86-64

- For good memory system performance, Intel recommends data be aligned
  - However the x86-64 hardware will work correctly regardless of alignment of data
  - Design choice: x86-64 instructions are *variable* bytes long

- **Aligned**: Primitive object of $K$ bytes must have an address that is a multiple of $K$
  - More about alignment later in the course

<table>
<thead>
<tr>
<th>$K$</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, <strong>pointers</strong></td>
</tr>
</tbody>
</table>
Byte Ordering

- How should bytes within a word be ordered in memory?
  - **Example:** store the 4-byte (32-bit) int: 0x a1 b2 c3 d4

- By convention, ordering of bytes called *endianness*
  - The two options are big-endian and little-endian
  - Based on *Gulliver’s Travels:* tribes cut eggs on different sides (big, little)
Byte Ordering

- **Big-endian (SPARC, z/Architecture)**
  - Least significant byte has highest address

- **Little-endian (x86, x86-64)**
  - Least significant byte has lowest address

- **Bi-endian (ARM, PowerPC)**
  - Endianness can be specified as big or little

**Example:** 4-byte data 0xa1b2c3d4 at address 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a1</td>
<td>b2</td>
<td>c3</td>
<td>d4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d4</td>
<td>c3</td>
<td>b2</td>
<td>a1</td>
</tr>
</tbody>
</table>
Byte Ordering Examples

int x = 12345;
// or x = 0x3039;

long int y = 12345;
// or y = 0x3039;

(A long int is the size of a word)
Endianness

- Often programmer can ignore endianness because it is handled for you
  - Bytes wired into correct place when reading or storing from memory (hardware)
  - Compiler and assembler generate correct behavior (software)

- Endianness still shows up:
  - Logical issues: accessing different amount of data than how you stored it (e.g. store int, access byte as a char)
  - When running down memory errors, need to know exact values
  - Manual translation to and from machine code (in 351)
Reading Byte-Reversed Listings

- Disassembly
  - Take binary machine code and generate an assembly code version
  - Does the reverse of the assembler

- Example instruction in memory
  - add value 0x12ab to register ‘ebx’ (*a special location in the CPU*)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
</tbody>
</table>

Deciphering numbers
Reading Byte-Reversed Listings

- Disassembly
  - Take binary machine code and generate an assembly code version
  - Does the reverse of the assembler

- Example instruction in memory
  - add value 0x12ab to register ‘ebx’ (a special location in the CPU)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
</tbody>
</table>

Deciphering numbers

- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse (little-endian): ab 12 00 00
Question:

- We store the value `0x00 01 02 03` as a `long int` (size of `long int` = size of pointer) at address `0x100` and then get back `0x00` when we read a `byte` at address `0x102`

- What machine setup are we using?

- (A) 32-bit, big-endian
- (B) 32-bit, little-endian
- (C) 64-bit, big-endian
- (D) 64-bit, little-endian
Summary

- Memory is a long, *byte-addressed array*
  - Word size bounds the size of the *address space* and memory
  - Different data types use different number of bytes
  - Address of chunk of memory given by address of lowest byte in chunk
  - Object of $K$ bytes is *aligned* if it has an address that is a multiple of $K$

- IEC prefixes refer to powers of $2^{10}$
- Pointers are data objects that holds addresses
- Endianness determines storage order for multi-byte objects