Memory, Data, & Addressing I
CSE 351 Spring 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Rehaan Bhimani
Callum Walker
Chin Yeoh
Diya Joy
Eric Fan
Edan Sneh
Jonathan Chen
Jeffery Tian
Millicent Li
Melissa Birchfield
Porter Jones
Joseph Schafer
Connie Wang
Eddy (Tianyi) Zhou

http://xkcd.com/953/
Administrivia

- Assignments – Nothing **Due** this week!
- Pre-Course Survey, hw0, hw1, hw2 due Mon (4/06) – 11:59pm
- Lab 0 due Tuesday (4/07) – 11:59pm
  - This lab is *exploratory* and looks like a hw; the other labs will look a lot different (involve writing code etc.)
  - Don’t worry if everything in Lab 0 doesn’t make perfect sense right now! We will cover all of these topics in more detail later in the course.
  - Lab 0 is about getting you used to modifying C code and running it to see what the outcome is – a powerful tool for understanding the concepts in this course!
- hw3 due Wednesday (4/08) – **11am**
Q & A Since Monday (1 of 2)

- **CSE391 – Unix Tools**
  - *Anyone* taking this course, including non-CSE majors
    - Tuesday 1:30-2:20pm
    - [https://courses.cs.washington.edu/courses/cse391/20sp/](https://courses.cs.washington.edu/courses/cse391/20sp/)
  - 391 Instructors trying to increase enrollment cap, if not I can post some old 391 videos

- **Textbook**
  - Normally I have a copy of our text on reserve at the Engineering library, this quarter I have requested that an electronic version be on reserve but have not heard back about that.
  - There is a [180 day digital rental](https://www.uwbookstore.com/product/9781108762376) available for $34.99 from the UW Bookstore.
Q & A Since Monday (2 of 2)

- **PollEverywhere participation**
  - You may attend either lecture section – your pollEverywhere responses will be counted for credit in either lecture.
  - **This week** PollEverywhere use is just to get folks used to the technology, not for Participation credit
  - **Starting next week** we will have a mechanism in place for students who cannot attend lecture synchronously to be able to answer pollEverywhere questions. This will likely be outside of pollEverywhere (maybe canvas), and will likely be due before the next lecture meeting. (We cannot simply leave pollEverywhere open between classes in the case that we have multiple questions per class (likely) – I can only have one question open at a time. I am trying to leave a question open after lecture this week just to let folks try out the technology.)
**Roadmap**

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

**Java:**
```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

**Assembly language:**
```
get_mpg:
    pushq  %rbp
    movq  %rsp, %rbp
    ...
    popq  %rbp
    ret
```

**Machine code:**
```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111
```

**OS:**
- Windows 10
- OS X Yosemite

**Memory & data**
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C
Memory, Data, and Addressing

- Hardware - High Level Overview
- Representing information as bits and bytes
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
- Organizing and addressing data in memory
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations
Hardware: Physical View

- CPU (empty slot)
- USB...
- Memory
- I/O controller
- Storage connections
- Bus connections
- PCI-Express Slots
- 1 PCI-E X16, 2 PCI-E X1
- Back Panel Connectors
- PCI Slots
- Intel ICH10 Chipset
- Serial ATA Headers
- DDR2 1066+MHz Dual Channel Memory Slots
- Socket 775 Core2 Quad/Core2 Extreme Ready
- Intel P45 Chipset
Hardware: Logical View
Hardware: 351 View (version 0)

- The CPU executes instructions
- Memory stores data
- Binary encoding!
  - Instructions are just data

How are data and instructions represented?
Aside: Why Base 2?

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

- Other bases possible, but not yet viable:
  - DNA data storage (base 4: A, C, G, T) is a hot topic
  - Quantum computing
Binary Encoding Additional Details

- Because storage is finite in reality, everything is stored as “fixed” length
  - Data is moved and manipulated in fixed-length chunks
  - Multiple fixed lengths (e.g. 1 byte, 4 bytes, 8 bytes)
  - Leading zeros now must be included up to “fill out” the fixed length

- Example: the “eight-bit” representation of the number 4 is 0b00000100
To execute an instruction, the CPU must:

1) Fetch the instruction
2) (if applicable) Fetch data needed by the instruction
3) Perform the specified computation
4) (if applicable) Write the result back to memory
Hardware: 351 View (version 1)

- More CPU details:
  - Instructions are held temporarily in the instruction cache
  - Other data are held temporarily in registers
- Instruction fetching is hardware-controlled
- Data movement is programmer-controlled (assembly)
We will start by learning about Memory.

How does a program find its data in memory?
An Address Refers to a Byte of Memory

- Conceptually, memory is a single, large array of bytes, each with a unique *address* (index)
  - Each address is just a number represented in *fixed-length* binary

- Programs refer to bytes in memory by their *addresses*
  - Domain of possible addresses = *address space*
  - We can store addresses as data to “remember” where other data is in memory

- But not all values fit in a single byte... *(e.g. 351)*
  - Many operations actually use multi-byte values
Polling Question

- If we choose to use 4-bit addresses, how big is our address space?
  - *i.e.* How much space can we “refer to” using our addresses?
  - Vote at [http://PollEv.com/rea](http://PollEv.com/rea)

A. 16 bits
B. 16 bytes
C. 4 bits
D. 4 bytes
E. We’re lost...
Machine “Words”

- Instructions encoded into machine code (0’s and 1’s)
  - Historically (still true in some assembly languages), all instructions were exactly the size of a word

- We have chosen to tie word size to address size/width
  - word size = address size = register size
  - 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)
  - Actual physical address space: 48 bits
Word-Oriented View of Memory

Addresses still specify locations of bytes in memory, but we can choose to view memory as a series of word-sized chunks of data instead:

- Addresses of successive words differ by word size
- Which byte’s address should we use for each word?

<table>
<thead>
<tr>
<th>64-bit Words</th>
<th>32-bit Words</th>
<th>Bytes (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x00</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x01</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x02</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x03</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x04</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x05</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x06</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x07</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x08</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x09</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x0A</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x0B</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x0C</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x0D</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x0E</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0x0F</td>
</tr>
</tbody>
</table>
Address of a Word = Address of First Byte in the Word

- Addresses still specify locations of bytes in memory, but we can choose to view memory as a series of word-sized chunks of data instead.
  - Addresses of successive words differ by word size.
  - Which byte’s address should we use for each word?

- The address of any chunk of memory is given by the address of the first byte.
  - To specify a chunk of memory, need both its address and its size.
## Alignment

- The address of a chunk of memory is considered *aligned* if its address is a multiple of its size
  - View memory as a series of consecutive chunks of this particular size and see if your chunk doesn’t cross a boundary

<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 = 0000</td>
<td>Addr = 0000</td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0004</td>
<td></td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td>Addr = 0008</td>
<td></td>
<td>0x02</td>
</tr>
<tr>
<td></td>
<td>Addr = 0012</td>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x04</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x06</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x07</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x09</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x0F</td>
</tr>
</tbody>
</table>
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
  - An aligned, 64-bit chunk of data 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
  - An aligned, 64-bit chunk of data will fit on one row
Addresses and Pointers

- An address refers to a location in memory
- A pointer is a data object that holds an address
  - Address can point to any data
- Value 504 stored at address 0x08
  - $504_{10} = 1F8_{16}$
  - = 0x 00 ... 00 01 F8
- Pointer stored at 0x38 points to address 0x08

64-bit example (pointers are 64-bits wide)
big-endian
## Addresses and Pointers

- **An address** refers to 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>Address</th>
<th>0x00</th>
<th>0x08</th>
<th>0x10</th>
<th>0x18</th>
<th>0x20</th>
<th>0x28</th>
<th>0x30</th>
<th>0x38</th>
<th>0x40</th>
<th>0x48</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>F8</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
<td></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></td>
<td>long int</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 long</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>long double</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>`

**address size = word size**
Memory Alignment Revisited

- A primitive object of $K$ bytes must have an address that is a multiple of $K$ to be considered *aligned*

<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, pointers</td>
</tr>
</tbody>
</table>

- For good memory system performance, Intel (x86) recommends data be aligned
  - However the x86-64 hardware will work correctly otherwise
    - Design choice: x86-64 instructions are *variable* bytes long
Byte Ordering

- How should bytes within a word be ordered in memory?
  - Want to keep consecutive bytes in consecutive addresses
  - **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**
    - In which address does the least significant byte go?
    - 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>Little-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100 0x101 0x102 0x103</td>
<td>0x100 0x101 0x102 0x103</td>
</tr>
</tbody>
</table>
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

- **Big-Endian**
  - 0x100: a1
  - 0x101: b2
  - 0x102: c3
  - 0x103: d4

- **Little-Endian**
  - 0x100: d4
  - 0x101: c3
  - 0x102: b2
  - 0x103: a1
**Byte Ordering Examples**

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

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

(A `long int` is the size of a word)
Polling Question

- We store the value `0x01 02 03 04` as a *word* at address `0x100` in a big-endian, 64-bit machine.
- What is the *byte of data* stored at address `0x104`?
  - Vote at [http://pollev.com/rea](http://pollev.com/rea)

A. 0x04
B. 0x40
C. 0x01
D. 0x10
E. We’re lost...
Endianness

- **Endianness only applies to memory storage**

- 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`)*
  - Need to know exact values to debug memory errors
  - Manual translation to and from machine code *(in 351)*
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 $K$

- Pointers are data objects that hold addresses

- Endianness determines memory storage order for multi-byte data