Memory, Data, & Addressing I
CSE 351 Autumn 2022

Instructor:
Justin Hsia

Teaching Assistants:
Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao

http://xkcd.com/953/
Relevant Course Information

❖ Everything not a reading or lecture lesson due @ 11:59 pm
  ▪ Pre-Course Survey and HW0 due tonight
  ▪ HW1 due Monday (10/3)
  ▪ Lab 0 due Monday (10/3)
    • This lab is exploratory and looks like a HW; the other labs will look a lot different

❖ Ed Discussion etiquette
  ▪ For anything that doesn’t involve sensitive information or a solution, post publicly (you can post anonymously!)
  ▪ If you feel like you question has been sufficiently answered, make sure that a response has a checkmark
EPA

❖ Encourage class-wide learning!

❖ Effort
  ▪ Attending office hours, completing all assignments
  ▪ Keeping up with Ed Discussion activity

❖ Participation
  ▪ Making the class more interactive by asking questions in lecture, section, office hours, and on Ed Discussion
  ▪ Lecture question voting

❖ Altruism
  ▪ Helping others in section, office hours, and on Ed Discussion
The Hardware/Software Interface

- **Topic Group 1: Data**
  - Memory, Data, Integers, Floating Point, Arrays, Structs

- **Topic Group 2: Programs**
  - x86-64 Assembly, Procedures, Stacks, Executables

- **Topic Group 3: Scale & Coherence**
  - Caches, Processes, Virtual Memory, Memory Allocation
The Hardware/Software Interface

- Topic Group 1: Data
  - Memory, Data, Integers, Floating Point, Arrays, Structs

- How do we store information for other parts of the house of computing to access?
  - How do we represent data and what limitations exist?
  - What design decisions and priorities went into these encodings?
Hardware: Physical View

- CPU (empty slot)
- USB...
- Memory
- I/O controller
- Storage connections
- Bus connections

Diagram of a motherboard showing various components and connections.
Hardware: Logical View
The CPU **executes** instructions

Memory **stores** data

**Binary encoding!**
- Instructions *are* just data

**Q1:** 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 hot @UW
  ▪ Quantum computing
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
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

Addresses!
- Can be stored in pointers

Q2: How does a program find its data in memory?
Reading Review

❖ Terminology:
   - word size, byte-oriented memory
   - address, address space
   - most-significant bit (MSB), least-significant bit (LSB)
   - big-endian, little-endian
   - pointer

❖ Questions from the Reading?
Review Questions

❖ By looking at the bits stored in memory, I can tell what a particular 4 bytes is being used to represent.

A. True    B. False

❖ We can fetch a piece of data from memory as long as we have its address.

A. True    B. False

❖ Which of the following bytes have a most-significant bit (MSB) of 1?

A. 0x63    B. 0x90    C. 0xCA    D. 0xF
Fixed-Length Binary (Review)

- 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

  Most Significant Bit (MSB)

  Least Significant Bit (LSB)
1 byte = 8 bits

\( n \) bits can represent up to \( 2^n \) things
- Sometimes (oftentimes?) those “things” are bytes!

If an addresses are \( a \)-bits wide, how many distinct addresses are there?

What does each address refer to?
Machine “Words” (Review)

❖ 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 x \( 10^{19} \) bytes
    = 18 billion billion bytes = 18 EB (exabytes)
  ▪ Actual physical address space: 48 bits
## Data Representations

- Sizes of data types (in bytes)

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>IA-32 (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</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>long</td>
<td>long double</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

*(reference) pointer * 4 8

To use "bool" in C, you must `#include <stdbool.h>`

**address size = word size**
Discussion Question

❖ Over time, computers have grown in word size:

<table>
<thead>
<tr>
<th>Word size</th>
<th>Instruction Set Architecture</th>
<th>First? Intel CPU</th>
<th>Year Introduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>??? (Poor &amp; Pyle)</td>
<td>Intel 8008</td>
<td>1972</td>
</tr>
<tr>
<td>16-bit</td>
<td>x86</td>
<td>Intel 8086</td>
<td>1978</td>
</tr>
<tr>
<td>32-bit</td>
<td>IA-32</td>
<td>Intel 386</td>
<td>1985</td>
</tr>
<tr>
<td>64-bit</td>
<td>IA-64</td>
<td>Itanium (Merced)</td>
<td>2001</td>
</tr>
<tr>
<td>64-bit</td>
<td>x86-64</td>
<td>Xeon (Nocona)</td>
<td>2004</td>
</tr>
</tbody>
</table>

❖ What do you think were some of the causes, advantages, and disadvantages of this trend?
Address of Multibyte Data (Review)

- Addresses still specify locations of bytes in memory, but we can choose to view memory as a series of chunks of fixed-sized data instead.
  - Addresses of successive chunks differ by data 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.

<table>
<thead>
<tr>
<th>64-bit data</th>
<th>32-bit data</th>
<th>Bytes</th>
<th>Addr (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>0x00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td>0x02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>0x04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td>0x05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td>0x06</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td>0x07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>0x08</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td>0x09</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td>0x0A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td>0x0B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td>0x0C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td>0x0D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td>0x0E</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</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 as a word at addr **0x08**
  ▪ $504_{10} = 1F8_{16}$
    = 0x 00 ... 00 01 F8
❖ Pointer stored at **0x38** points to address **0x08**

<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></td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>08</td>
</tr>
</tbody>
</table>

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)

Big-endian
Byte Ordering (Review)

❖ 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>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A1</td>
<td>B2</td>
<td>C3</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>
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 in Ed Lessons

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

❖ Pointers are data objects that hold addresses
  ▪ Type of pointer determines size of thing being pointed at, which could be another pointer

❖ Endianness determines memory storage order for multi-byte data
  ▪ Least significant byte in lowest (little-endian) or highest (big-endian) address of memory chunk