Announcements

- **On the website: cs.uw.edu/351**
  - Speedometer!
  - Anonymous feedback form
  - Make sure you are subscribed to the mailing list.
  - Lecture slides on the web schedule
  - Lab 0, having fun?
  - Discussion boards
  - Videos for *optional* reference.
    - Tips for C, debugging, etc.
    - Lecture content.

- **Office hours posted: if they don’t work for you, let us know.**
- **Any non-CSE majors not yet enrolled?**
- **Regular section meetings Thursday 9:40-10:40am in LOW 105**
- **(?) Section meeting next week Wednesday 2-3pm (?)**
Hardware: Logical View

CPU

Memory

Disks

Net

USB

Etc.

Bus
Hardware: Semi-Logical View

Intel® P45 Express Chipset Block Diagram
Hardware: Physical View

- Bus connections
- USB...
- I/O controller
- Storage connections
- Memory
- CPU
- PCI Slots
- Back Panel Connectors
- PCI-Express Slots
  - 1 PCI-E X16, 2 PCI-E X1
- Socket 775
  - Core2 Quad/ Core2 Extreme Ready
- Intel P45 Chipset
- DDR2 1066+MHz Dual Channel Memory Slots
- Serial ATA Headers
- Intel ICH10 Chipset
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 fetching is HW-controlled.
- Data movement is programmer-controlled.
- We’ll learn about the instructions the CPU executes. Take 352 to find out how it executes them.
The CPU holds instructions temporarily in the instruction cache. Instruction fetching is HW-controlled. Data movement is programmer-controlled.

How are data and instructions represented?

- The CPU holds data temporarily in registers.
- How does a program find its data in memory?
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);
float mpg =
c.getMPG();

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

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
11000001111110101000011111

Computer system:

Memory:
Data & addressing
Integers & floats
Machine code & C
x86 assembly
programming
Procedures &
stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Memory, Data, and Addressing

- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations
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
Describing Byte Values

- **Binary** \(00000000_2 \rightarrow 11111111_2\)
  - Byte = 8 bits (binary digits)
- **Decimal** \(0_{10} \rightarrow 255_{10}\)
- **Hexadecimal** \(00_{16} \rightarrow FF_{16}\)
  - 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 C
    - as \(0xFA1D37B\) or \(0xfa1d37b\)
- **More on specific data types later...**

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

- Conceptually, memory is a single, large array of bytes, each with an 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

- fixed number of contiguous bytes in memory, chosen by HW
- the largest unit of data a machine instruction can use
- 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
  - Until recently, most machines used 32-bit (4-byte) words.
    - Potential address space: \( 2^{32} \) addresses
    - \( 2^{32} \) bytes \( \approx 4 \times 10^9 \) bytes = 4 billion bytes = 4GB
      - (living humans / addressable bytes \( \approx 1.8 \))
    - Became too small for memory-intensive applications
  - 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)
      - (possible living acquaintances / addressable bytes \( \approx 2.8 \))
## 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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td></td>
<td>0015</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**

![Diagram of 64-bit and 32-bit words with addresses and byte positions]
Memory Alignment

- Data of size $n$ only stored at addresses $a$ where $a \mod n = 0$
  - Convention or rule, depending on platform.
  - $n$ is usually a power of 2.

- A 32-bit (4-byte) word-aligned view of memory:
  - Each row is a word composed of 4 bytes.
  - Cells in a row are the word’s bytes.

More about alignment later in the course.
Addresses and Pointers

- An **address** is a location in memory.
- A **pointer** is a data object that holds an address.
- The value 351 is stored at address **0x04**.
  
  - $351_{10} = 15F_{16} = 0x00\ 00\ 01\ 5F$

```
 00 00 01 5F
0x00
0x04
0x08
0x0C
0x10
0x14
0x18
0x1C
0x20
0x24
```
Addresses and Pointers

- An **address** is a location in memory.
- A **pointer** is a data object that holds an address.
- The value 351 is stored at address **0x04**.
  - \(351_{10} = 15F_{16} = 0x00\ 00\ 01\ 5F\)
- A pointer stored at address **0x1C** points to address **0x04**.

```
  00 00 01 5F
  00 00 00 04
  0x00
  0x04
  0x08
  0x0C
  0x10
  0x14
  0x18
  0x1C
  0x20
  0x24
```
Addresses and Pointers

- An *address* is a location in memory.
- A *pointer* is a data object that holds an address.
- The value 351 is stored at address *0x04*.
  - $351_{10} = 15F_{16} = 0x00\ 00\ 01\ 5F$
- A pointer stored at address *0x1C* points to address *0x04*.
- A pointer to a pointer is stored at address *0x24*. 
Addresses and Pointers

- An **address** is a location in memory.
- A **pointer** is a data object that holds an address.
- The value 351 is stored at address **0x04**.
  - \(351_{10} = 15F_{16} = 0x00 \ 00 \ 01 \ 5F\)
- A pointer stored at address **0x1C** points to address **0x04**.
- A pointer to a pointer is stored at address **0x24**.
- The value 12 is stored at address **0x14**.
  - Is it a pointer?
# Data Representations

## Sizes of data types (in bytes)

<table>
<thead>
<tr>
<th>Java Data Type</th>
<th>C Data Type</th>
<th>Typical 32-bit</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>

**address size = word size**
Byte Ordering

- How should bytes within a word be ordered in memory?
- Example: Store the 4-byte word 0xa1 b2 c3 d4.
  - In what order will the bytes be stored?

Conventions!
- Big-endian, Little-endian
- Based on *Gulliver’s Travels*: tribes cut eggs on different sides (big, little)
# Byte Ordering

- **Big-Endian** (PowerPC, SPARC, The Internet)
  - Least significant byte has highest address

- **Little-Endian** (x86)
  - Least significant byte has lowest address

### Example
- Variable has 4-byte representation **0xa1b2c3d4**
- Address of variable is **0x100**

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

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

```c
int x = 12345;
// long int = word
long int y = 12345;
```

Decimal: 12345
Binary: 0011 0000 0011 1001
Hex: 3 0 3 9

IA32, x86-64 x

SPARC x

39
30
00
00

39
30
00
00

32-bit SPARC y

64-bit SPARC y

00
00
00
00

00
00
00
00

00
00
00
00
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 CPU’s memory)

<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>
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 CPU’s memory)*

<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
Addresses and Pointers in C

\[ \text{int* } \text{ptr}; \]

Declares a variable, \text{ptr}, that is a pointer to (i.e., holds the address of) an int in memory.

\[ \text{int } \text{x} = 5; \]
\[ \text{int } \text{y} = 2; \]

Declares two variables, \text{x} and \text{y}, that hold ints, and sets them to 5 and 2, respectively.

\[ \text{ptr} = &\text{x}; \]

Sets \text{ptr} to the address of \text{x}. Now, “\text{ptr points to } \text{x}.”

“Dereference \text{ptr}.”

\[ \text{y} = 1 + *\text{ptr}; \]

Sets \text{y} to 1 plus the value at the address held by \text{ptr}. Because \text{ptr} points to \text{x}, this is equivalent to \text{y}=1+\text{x};
Assignment in C

- A variable is represented by a memory location.
- Initially, it may hold any value.
- int x, y;
  - // x is at location 0x04, y is at 0x18.

\& = ‘address of’
* = ‘value at address’
or ‘dereference’

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x04</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7</td>
<td>00</td>
<td>32</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>29</td>
</tr>
<tr>
<td>EE</td>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>FA</td>
<td>CE</td>
<td>CA</td>
</tr>
<tr>
<td>26</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>FF</td>
<td>00</td>
<td>F4</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>42</td>
<td>17</td>
</tr>
</tbody>
</table>

| 0x08 | 0x0C |  
|------|------|---|
| EE   | EE   | EE |
| FA   | CE   | CA |
| 00   | 00   | 00 |
| 00   | 10   | 00 |
| 00   | 00   | 00 |
| 00   | 00   | 00 |

<table>
<thead>
<tr>
<th>0x10</th>
<th>0x14</th>
<th>0x18</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>FF</td>
<td>00</td>
<td>F4</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x1C</th>
<th>0x20</th>
<th>0x24</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>17</td>
<td>34</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>96</td>
</tr>
</tbody>
</table>
Assignment in C

- A variable is represented by a memory location.
- Initially, it may hold any value.
- `int x, y;`
  - // x is at location 0x04, y is at 0x18.

\& = ‘address of’
* = ‘value at address’
or ‘dereference’


Assignment in C

- **Left-hand-side = right-hand-side;**
  - LHS must evaluate to a memory *location*.
  - RHS must evaluate to a *value*. (could be an address!)
  - Store RHS value at LHS location.

- `int x, y;`
- `x = 0;`

& = ‘address of’
* = ‘value at address’ or ‘dereference’
Assignment in C

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

- int x, y;
- x = 0;
- y = 0x3CD02700;

& = ‘address of’
* = ‘value at address’ or ‘dereference’

little endian!
Assignment in C

- **Left-hand-side = right-hand-side;**
  - LHS must evaluate to a memory *location*.
  - RHS must evaluate to a *value*. (could be an address!)
  - Store RHS value at LHS location.

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - // Get value at y, add 3, put it in x.
Assignment in C

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - // Get value at y, add 3, put it in x.
- int* z

\& = 'address of'
* = 'value at address' or 'dereference'

- `int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
int* z`
Assignment in C

- **Left-hand-side = right-hand-side;**
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - // Get value at y, add 3, put it in x.
- `int* z = &y + 3;`
  - // Get address of y, add ???, put it in z.

\& = ‘address of’
asterisk = ‘value at address’
or ‘dereference’
Assignment in C

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - // Get value at y, add 3, put it in x.
- `int* z = &y + 3;`
  - // Get address of y, add 12, put it in z.

& = 'address of'
* = 'value at address' or 'dereference'

Pointer arithmetic can be dangerous.

Pointer arithmetic is scaled by size of target type.
Assignment in C

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - // Get value at y, add 3, put it in x.
- int* z = &y + 3;
  - // Get address of y, add 12, put it in z.
- *z = y;
  - // What does this do?

& = ‘address of’
* = ‘value at address’ or ‘dereference’
Assignment in C

- **Left-hand-side = right-hand-side;**
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

```c
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
// Get value of y, add 3, put it in x.
int* z = &y + 3;
// Get address of y, add 12, put it in z.
*z = y;
// Get value of y, put it at the address stored in z.
```

The target of a pointer is also a memory location.

& = ‘address of’
* = ‘value at address’ or ‘dereference’