The Hardware/Software Interface
CSE351 Winter 2013

Memory, Data & Addressing

Announcements
- Lab 0 is due Friday (no late days)
- Section 1 tomorrow
  - If possible, bring your laptop
- Visit the website and use:
  - The link to the CSE home VM
  - The speedometer
  - The anonymous feedback link
  - The discussion board!
- Visit office hours
- Lab 1 posted today, due next Friday

Today’s Topics
- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations
There are a fixed number of registers in the CPU
- Registers hold data

There is an L1 cache in the CPU that holds recently fetched instructions
- If you execute a loop that fits in the cache, the CPU goes to memory for those instructions only once, then executes it out of its cache

This slide is just an introduction.
We’ll see a fuller explanation later in the course.
Performance: It’s Not Just CPU Speed

- Data and instructions reside in memory
  - To execute an instruction, it must be fetched into the CPU
  - Next, the data the instruction operates on must be fetched into the CPU
- CPU ↔ Memory bandwidth can limit performance
  - Improving performance 1: hardware improvements to increase memory bandwidth (e.g., DDR → DDR2 → DDR3)
  - Improving performance 2: move less data into/out of the CPU
    - Put some “memory” in the CPU chip itself (this is “cache” memory)

Binary Representations

- Base 2 number representation
  - Represent 351₁₀ as 0000000101011111₂ or 10101111₁₂
- Electronic implementation
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires

Encoding Byte Values

- Binary 00000000₂ -- 11111111₂
  - Byte = 8 bits (binary digits)
- Decimal 0₁₀ -- 255₁₀
- Hexadecimal 00₁₆ -- FF₁₆
  - Byte = 2 hexadecimal (hex) or base 16 digits
  - Base-16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write FA1D37B₁₆ in C code as:
    - 0xFA1D37B or 0xfa1d37b

How is memory organized?

- How do we find data in memory?
Byte-Oriented Memory Organization

- Programs refer to addresses
  - Conceptually, a very large array of bytes, each with an address (index)
  - Operating system provides an address space private to each “process”
    - Process = program being executed + its data + its “state”
    - Program can modify its own data, but not that of others
    - Clobbering code or “state” often leads to crashes (or security holes)
- Compiler + run-time system control memory allocation
  - Where different program objects should be stored
  - All allocation within a single address space

Machine Words

- Machine has a “word size”
  - Nominal size of integer-valued data
    - Including addresses
  - Until recently, most machines used 32 bit (4 byte) words
    - Limits addresses to 4GB
    - Became too small for memory-intensive applications
  - Most current x86 systems use 64 bit (8 byte) words
    - Potential address space: $2^{64} = 1.8 \times 10^{19}$ bytes (18 EB – exabytes)
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always a power-of-2 number of bytes: 1, 2, 4, 8, ...

Word-Oriented Memory Organization

- Addresses specify locations of bytes in memory
  - Address of first byte in word
  - Addresses of successive words differ by 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>0000</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0003</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0006</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0007</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0009</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0013</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0015</td>
<td></td>
</tr>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Addresses and Pointers

- **Address** is a *location* in memory
- **Pointer** is a data object that *contains an address*
- **Address 0004** stores the value 351 (or 15F₁₆)
  - Pointer to address 0004 stored at address 001C
- **Address 0014** stores the value 12
  - Is it a pointer?

**Memory**
Data Representations

- Sizes of objects (in bytes)
  - Java data type  
    - boolean: ```bool```  
    - byte: ```char```  
    - char:  
    - short: ```short int```  
    - int: ```int```  
    - float: ```float```  
    - long: ```long int```  
    - double: ```double```  
    - (reference) ```pointer *```  
  - C data type  
  - Typical 32-bit: ```1```  
  - x86-64: ```1```  
  - boolean: ```1```  
  - byte: ```1```  
  - char: ```2```  
  - short: ```2```  
  - int: ```4```  
  - float: ```4```  
  - long: ```4```  
  - double: ```8```  
  - long double: ```8```  

Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Say you want to store the 4-byte word ```0xaabbccdd```  
  - What order will the bytes be stored?

- Endianness: big endian vs. little endian  
  - Two different conventions, used by different architectures  
  - Origin: *Gulliver’s Travels* (see textbook, section 2.1)

ByteOrderingExample

- Big endian (PowerPC, Sun, Internet)  
  - Big end first: most-significant byte has lowest address
- Little endian (x86)  
  - Little end first: least-significant byte has lowest address

Example

- Variable has 4-byte representation ```0x01234567```  
- Address of variable is ```0x100```  

Representing Integers

- int A = 12345;  
- int B = -12345;  
- long int C = 12345;

- Decimal: ```12345```  
- Binary: ```0011 0000 0011 1001```  
- Hex: ```3      0      3      9       -> 0x00003039```  

Two’s complement representation for negative integers (next lecture)
Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **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

- **Pointer declarations use ***
  - int *ptr; int x, y;  ptr = &x;
  - Declares a variable ptr that is a pointer to a data item that is an integer
  - Declares integer values named x and y
  - Assigns ptr to point to the address where x is stored
- **To use the value pointed to by a pointer we use dereference**
  - If ptr = &x: then y = *ptr + 1 is the same as y = x + 1
  - If ptr = &y: then y = *ptr + 1 is the same as y = y + 1
  - *ptr is the value stored at the location to which the pointer ptr is pointing
  - What is *(&x) equivalent to?
- **We can do arithmetic on pointers**
  - ptr = ptr + 1; // really adds 4: type of ptr is int*, and an int uses 4 bytes!
  - Changes the value of the pointer so that it now points to the next data item in memory (that may be y, or it may not – this is dangerous!)

Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory location (a variable)
  - RHS must evaluate to a value (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0xC02700
  - x = y + 3; //get value at y, add 3, put it in x

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0004</td>
<td>0008</td>
<td>000C</td>
</tr>
<tr>
<td>0010</td>
<td>0014</td>
<td>0018</td>
<td>001C</td>
</tr>
<tr>
<td>0020</td>
<td>0024</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0004</td>
<td>0008</td>
<td>000C</td>
</tr>
<tr>
<td>0010</td>
<td>0014</td>
<td>0018</td>
<td>001C</td>
</tr>
<tr>
<td>0020</td>
<td>0024</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Assignment in C

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

- **E.g., x at location 0x04, y at 0x18**
  - `x = y + 3; //get value at y, add 3, put it in x`

<table>
<thead>
<tr>
<th>0000</th>
<th>0004</th>
<th>0008</th>
<th>000C</th>
<th>0010</th>
<th>0014</th>
<th>0018</th>
<th>0020</th>
<th>0024</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
<td>00</td>
<td>04</td>
<td>08</td>
<td>0C</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>014</td>
<td>018</td>
<td>020</td>
<td>024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **E.g., x at location 0x04, y at 0x18**
  - `x = &y + 3; // get address of y, add 3`
  - `*x = y; // value of y copied to location to which x points`

<table>
<thead>
<tr>
<th>0000</th>
<th>0004</th>
<th>0008</th>
<th>000C</th>
<th>0010</th>
<th>0014</th>
<th>0018</th>
<th>0020</th>
<th>0024</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>04</td>
<td>08</td>
<td>0C</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>00</td>
<td>07</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
</tbody>
</table>
Assignment in C

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory location (a variable)
  - RHS must evaluate to a value (could be an address!)
- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0xC0D27000
  - int *x; int y;
    - `x = &y + 3; // get address of y, add 12`  
      - `0x0018 + 0x0000C = 0x0024`
  - `*x = y; // value copied to location to which x points`

Arrays

- **Arrays represent adjacent locations in memory storing the same type of data object**
  - e.g., int big_array[128]; allocates 512 adjacent bytes in memory starting at 0x0FF0000
- **Pointer arithmetic can be used for array indexing in C (if pointer and array have the same type!):**
  - `int *array_ptr;`  
    - `array_ptr = &big_array[0]; 0x0FF0000`
    - `array_ptr = &big_array[3]; 0x0FF000C`  
      - `(adds 3 * int)`
    - `array_ptr = &big_array[0] + 3; 0x0FF000C`  
      - `(adds 3 * int)`
    - `*array_ptr = *array_ptr + 1; 0x0FF000C`  
      - `(but big_array[3] is incremented)`
    - `array_ptr = &big_array[130]; 0x0FF00208`  
      - `(out of bounds, C doesn’t check)`
  - In general: &big_array[i] is the same as (big_array + i), 
    which implicitly computes: &big_array[0] + i * sizeof(big_array[0]);

Representing strings

- **A C-style string is represented by an array of bytes.**
  - Elements are one-byte ASCII codes for each character.
  - A 0 byte marks the end of the array.

Null-terminated strings

- **For example, “Harry Potter” can be stored as a 13-byte array.**
  - ![Array representation of "Harry Potter"](image)
  - Why do we put a 0, or null zero, at the end of the string? 
    - Note the special symbol: `string[12] = "\0"`
  - **How do we compute the string length?**
Compatibility

char S[6] = "12345";

<table>
<thead>
<tr>
<th>IA32, x86-64 $</th>
<th>Sun $</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

- Byte ordering (endianness) is not an issue for standard C strings (char arrays)
- Unicode characters – up to 4 bytes/character
  - ASCII codes still work (just add leading 0 bits) but can support the many characters in all languages in the world
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)

Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char

```c
typedef char byte; //size of char == 1 byte
void show_bytes(byte *start, int len) {
  int i;
  for (i = 0; i < len; i++)
    printf("%p %x\n", start+i, *start+i);
}

void show_int(int x) {
    show_bytes( (byte *) &x, sizeof(int));
}
```

printf directives:
- `%p` Print pointer
- `	` Tab
- `%x` Print value as hex
- `\n` New line

Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0
  - AND: A&B = 1 when both A is 1 and B is 1
  - OR: A|B = 1 when either A is 1 or B is 1
  - XOR: A^B = 1 when either A is 1 or B is 1, but not both
  - NOT: ~A = 1 when A is 0 and vice-versa
  - DeMorgan’s Law: ~(A | B) = ~A & ~B

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0 1</th>
<th>0 1</th>
<th>^</th>
<th>0 1</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
<td>0 1</td>
<td>0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0</td>
<td>1 1</td>
<td>1</td>
<td>1 0</td>
<td>1 0</td>
</tr>
</tbody>
</table>

Result (Linux on attu):

```c
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((byte *) &a, sizeof(int));
```

Result (Linux on attu):

```c
int a = 12345;
0x7fff6f330dcc  0x39
0x7fff6f330dcd  0x30
0x7fff6f330dce  0x00
0x7fff6f330dcf  0x00
```
Manipulating Bits

- Boolean operators can be applied to **bit vectors**: operations are applied bitwise

\[
\begin{array}{c|c|c|c|c|c}
& 01101001 & 01101001 & 01101001 & 01101001 \\
\& & 01101010 1 & 01101010 1 & 01101010 1 & 01101010 1 \\
\end{array}
\]

Bit-Level Operations in C

- Bitwise operators &, |, ^, ~ are available in C
  - Apply to any “integral” data type
    - long, int, short, char
  - Arguments are treated as bit vectors
  - Operations applied bitwise

Examples (char data type)

- ~0x41 → 0xBE
- ~0x00 → 0xFF
- 0x69 & 0x55 → 0x41
- 0x00 && 0x55 → 0x01
- 0x69 | 0x55 → 0x7D
- 0x00 || 0x55 → 0x55

Contrast: Logic Operations in C

- Logical operators in C: &&, ||, !
  - Behavior:
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination (&& and ||)

Examples (char data type)

- !0x41 → 0x00
- !0x00 → 0x01
- 0x69 && 0x55 → 0x01
- 0x00 && 0x55 → 0x00
- 0x69 || 0x55 → 0x01
- p && *p++ (avoids null pointer access: null pointer = 0x00000000)
  - short for: if (p) { *p++; }

Representing & Manipulating Sets

- Bit vectors can be used to represent sets
  - Width w bit vector represents subsets of \{0, ..., w-1\}
  - a_j = 1 if \( j \in A \) — each bit in the vector represents the absence (0) or presence (1) of an element in the set

\[
\begin{array}{c|c}
& 01101001 \\
\& & 01101010 1 \\
\end{array}
\]

- Operations
  - & Intersection
  - | Union
  - ^ Symmetric difference
  - ~ Complement