Computer Systems
CSE 410 Spring 2012
2 – Memory and its Data
Today’s (and Friday’s) topics

- Memory and its bits, bytes, and integers
- Representing information as bits
- Bit-level manipulations
  - Boolean algebra
  - Boolean algebra in C

- Reading: Bryant/O’Hallaron sec. 2.1
Hardware: Logical View

- CPU
- Memory
- Disks
- Net
- USB
- Etc.

Bus
Hardware: Semi-Logical View

Intel® P45 Express Chipset Block Diagram
Hardware: Physical View
There are a fixed number of registers in the CPU
- Registers hold data

There is an I-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 $351_{10}$ as $0000000101011111_2$ or $101011111_2$

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

- **Binary**
  
  - $00000000_2$ -- $11111111_2$
  
  - Byte = 8 bits (binary digits)

- **Decimal**
  
  - $0_{10}$ -- $255_{10}$

- **Hexadecimal**
  
  - $00_{16}$ -- $FF_{16}$
  
  - Byte = 2 hexadecimal (hex) or base 16 digits
  
  - Base-16 number representation
  
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  
  - Write FA1D37B\textsubscript{16} in C
    
    - as \texttt{0xFA1D37B} or \texttt{0xfal1d37b}

\begin{tabular}{|c|c|c|}
  \hline
  Hex & Decimal & Binary \\
  \hline
  0 & 0 & 0000 \\
  1 & 1 & 0001 \\
  2 & 2 & 0010 \\
  3 & 3 & 0011 \\
  4 & 4 & 0100 \\
  5 & 5 & 0101 \\
  6 & 6 & 0110 \\
  7 & 7 & 0111 \\
  8 & 8 & 1000 \\
  9 & 9 & 1001 \\
  A & 10 & 1010 \\
  B & 11 & 1011 \\
  C & 12 & 1100 \\
  D & 13 & 1101 \\
  E & 14 & 1110 \\
  F & 15 & 1111 \\
  \hline
\end{tabular}
What is memory, really?

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

- Programs refer to addresses
  - Conceptually, a very large array of bytes
  - System provides an address space private to each “process”
    - Process = program being executed + its data + its “state”
    - Program can clobber 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 bits (4 bytes) words
    - Limits addresses to 4GB
    - Became too small for memory-intensive applications
  - More recent and high-end systems use 64 bits (8 bytes) words
    - Potential address space $\approx 1.8 \times 10^{19}$ bytes (18 EB – exabytes)
    - x86-64 supports 48-bit physical addresses: 256 TB (terabytes)
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral (actually 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>0000</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0012</td>
<td>0012</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td>Addr = ??</td>
<td>Addr = ??</td>
<td>0015</td>
<td>0015</td>
</tr>
</tbody>
</table>
**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 = 0000</td>
<td>Addr = 0000</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0008</td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td></td>
<td>0015</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 $15F_{16}$)

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

Memory
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 $15F_{16}$)
- Pointer to address 0004 stored at address 001C

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td></td>
</tr>
<tr>
<td>0018</td>
<td></td>
</tr>
<tr>
<td>001C</td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td></td>
</tr>
<tr>
<td>0024</td>
<td></td>
</tr>
<tr>
<td>0000 00 01 5F</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 $15F_{16}$)
- Pointer to address 0004 stored at address 001C
- Pointer to a pointer in 0024
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 $15F_{16}$)
- Pointer to address 0004 stored at address 001C
- Pointer to a pointer in 0024
- Address 0014 stores the value 12
  - Is it a pointer?
## Data Representations

### Sizes of objects (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>
Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
  - Peanut butter or chocolate first?

- Say you want to store 0xaabbccdd
  - What order will the bytes be stored?
Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
  - Peanut butter or chocolate first?

- Say you want to store 0xaabbccdd
  - 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 Example

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

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

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

```
Big Endian

0x100 0x101 0x102 0x103

0x01 0x23 0x45 0x67

Little Endian

0x100 0x101 0x102 0x103

0x67 0x45 0x23 0x01
```
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>
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 **
  - \texttt{int * ptr; int x, y; ptr = &x;}
  - Declares a variable \texttt{ptr} that is a pointer to a data item that is an integer
  - Declares integer values named \texttt{x} and \texttt{y}
  - Assigns \texttt{ptr} to point to the address where \texttt{x} is stored

- **We can do arithmetic on pointers**
  - \texttt{ptr = ptr + 1;} \hspace{1em} // really adds 4 (because an integer uses 4 bytes?)
  - Changes the value of the pointer so that it now points to the next data item in memory (that may be \texttt{y}, or it may not – this is dangerous!)

- **To use the value pointed to by a pointer we use de-reference**
  - \texttt{y = *ptr + 1;} is the same as \texttt{y = x + 1;}
  - But, if \texttt{ptr = &y} then \texttt{y = *ptr + 1;} is the same as \texttt{y = y + 1;}
  - \texttt{*ptr} is the value stored at the location to which the pointer \texttt{ptr} is pointing
Arrays

- Arrays represent adjacent locations in memory storing the same type of data object
  - e.g., int big_array[128]; allocated 512 adjacent locations in memory starting at 0x00ff0000

- Pointers to arrays point to a certain type of object
  - e.g., int * array_ptr;
    array_ptr = big_array;
    array_ptr = &big_array[0];
    array_ptr = &big_array[3];
    array_ptr = &big_array[0] + 3;
    array_ptr = big_array + 3;
    *array_ptr = *array_ptr + 1;
    array_ptr = &big_array[130];
  - In general: &big_array[i] is the same as (big_array + i)
    - which implicitly computes: &bigarray[0] + i*sizeof(bigarray[0]);
Arrays

- Arrays represent adjacent locations in memory storing the same type of data object
  - E.g., int big_array[128];
    allocated 512 adjacent locations in memory starting at 0x00ff0000

- Pointers to arrays point to a certain type of object
  - E.g., int * array_ptr;
    array_ptr = big_array; 0x00ff0000
    array_ptr = &big_array[0]; 0x00ff0000
    array_ptr = &big_array[3]; 0x00ff000c
    array_ptr = &big_array[0] + 3; 0x00ff000c (adds 3 * size of int)
    array_ptr = big_array + 3; 0x00ff000c (adds 3 * size of int)
    *array_ptr = *array_ptr + 1; 0x00ff000c (but big_array[3] is incremented)
    array_ptr = &big_array[130]; 0x00ff0208 (out of bounds, C doesn’t check)

  - In general: &big_array[i] is the same as (big_array + i)
    which implicitly computes: &bigarray[0] + i*sizeof(bigarray[0]);
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x0027D03C

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>27</td>
</tr>
<tr>
<td>00</td>
<td>D0</td>
</tr>
<tr>
<td>00</td>
<td>3C</td>
</tr>
<tr>
<td>00</td>
<td>04</td>
</tr>
<tr>
<td>00</td>
<td>08</td>
</tr>
<tr>
<td>00</td>
<td>0C</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td>14</td>
</tr>
<tr>
<td>00</td>
<td>18</td>
</tr>
<tr>
<td>00</td>
<td>1C</td>
</tr>
<tr>
<td>00</td>
<td>20</td>
</tr>
<tr>
<td>00</td>
<td>24</td>
</tr>
</tbody>
</table>
```
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x0027D03C
  - int x, y;
    - x = y; // get value at y and put it in x

```
Memory

00 00 00 00 0000
00 00 00 00 0004
00 00 00 00 0008
00 00 00 00 000C
00 00 00 00 0010
00 00 00 00 0014
00 27 D0 3C 0018
00 00 00 00 001C
00 00 00 00 0020
00 00 00 00 0024
```
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x0027D03C
  - int x, y;
  - x = y; // get value at y and put it in x

```
00 27 D0 3C
0018 001C 0020 0024
0010 0014 0008 000C
0000 0004 0008 000C
```

Memory
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x0027D03C
  - int x, y;
    x = y;  // get value at y and put it in x
  - int * x; int y;
    x = &y + 3;  // get address of y add 12
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x0027D03C
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + 3; // get address of y add 12

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>00</td>
<td>18</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>1C</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>20</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>24</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

Memory
General rules for C (assignments)

- **Left-hand-side = right-hand-side**
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)

- **E.g., x at location 0x04, y at 0x18**
  - x originally 0x0, y originally 0x0027D03C
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + 3; // get address of y add 12
  - int * x; int y;
    *x = y; // value of y copied to
    // location to which x points

```
+----------------+-----------------+-----------------+-----------------+-----------------+
|  24  00  00  00 |      0000       |
|  0010           |      0010       |
|  0014           |      0014       |
|  0018           |      0018       |
|  0020           |      0020       |
|  0024           |      0024       |
+----------------+-----------------+-----------------+-----------------+-----------------+
```

Memory
General rules for C (assignments)

- Left-hand-side = right-hand-side
  - LHS must evaluate to a memory LOCATION
  - RHS must evaluate to a VALUE (could be an address)

- E.g., x at location 0x04, y at 0x18
  - x originally 0x0, y originally 0x0027D03C
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + 3; // get address of y add 12
  - int * x; int y;
    *x = y; // value of y copied to
    // location to which x points

Memory

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

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

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

Two’s complement representation for negative integers (covered later)
Representing Pointers

- `int B = -12345;`
- `int *P = &B;`

Different compilers & machines assign different locations to objects
Examining Data Representations

- Code to print byte representation of data
  - Casting pointer to unsigned char * creates byte array

```c
typedef unsigned char * pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p\t0x%.2x\n", start+i, start[i]);
    printf("\n");
}

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

Some printf directives:
- %p: Print pointer
- %x: Print hexadecimal
- "\n": New line
show_bytes Execution Example

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

Result (Linux):

```c
int a = 12345;
0x11ffffffcb8 0x39
0x11ffffffcb9 0x30
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```
Representing strings

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

- For example, “Harry Potter” can be stored as a 13-byte array.

```
72  97  114  114  121  32  80  111  116  116  101  114  0
H a r r y  P o t t e r \0
```

- Why do we put a 0, or null, at the end of the string?

- Computing string length?
### Compatibility

```c
char S[6] = "12345";
```

<table>
<thead>
<tr>
<th>Linux/Alpha S</th>
<th>Sun S</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 not an issue**
- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (leading 0 bit) 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)
### 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: $A \mid B = 1$ when either $A$ is 1 or $B$ is 1
  - XOR: $A \wedge B = 1$ when either $A$ is 1 or $B$ is 1, but not both
  - NOT: $\sim A = 1$ when $A$ is 0 and vice-versa
  - DeMorgan’s Law: $\sim (A \mid B) = \sim A \& \sim B$

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

<table>
<thead>
<tr>
<th>$\mid$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\wedge$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sim$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{align*}
01101001 \& 01010101 & = 01000001 \\
01101001 \mid 01010101 & = 01111101 \\
01101001 \wedge 01010101 & = 00111100 \\
\sim 01010101 & = 10101010
\end{align*}
\]

- All of the properties of Boolean algebra apply

\[
\begin{align*}
01010101 \wedge 01010101 & = 01010101
\end{align*}
\]

- How does this relate to set operations?
Representing & Manipulating Sets

- **Representation**
  - Width $w$ bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j = 1$ if $j \in A$
    
    | Bit Vector       | Subset         |
    |------------------|----------------|
    | 01101001         | $\{0, 3, 5, 6\}$ |
    | 01010101         | $\{0, 2, 4, 6\}$ |

- **Operations**
  - $\&$ Intersection
    
    | Bit Vector       | Subset         |
    |------------------|----------------|
    | 01000001         | $\{0, 6\}$    |
  - $|$ Union
    
    | Bit Vector       | Subset         |
    |------------------|----------------|
    | 01111101         | $\{0, 2, 3, 4, 5, 6\}$ |
  - $^\wedge$ Symmetric difference
    
    | Bit Vector       | Subset         |
    |------------------|----------------|
    | 00111100         | $\{2, 3, 4, 5\}$ |
  - $\sim$ Complement
    
    | Bit Vector       | Subset         |
    |------------------|----------------|
    | 10101010         | $\{1, 3, 5, 7\}$ |
Bit-Level Operations in C

- Operations &, |, ^, ~ are available in C
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise

- Examples (char data type)
  - ~0x41 --> 0xBE
    - ~01000001₂ --> 10111110₂
  - ~0x00 --> 0xFF
    - ~00000000₂ --> 11111111₂
  - 0x69 & 0x55 --> 0x41
    - 01101001₂ & 01010101₂ --> 01000001₂
  - 0x69 | 0x55 --> 0x7D
    - 01101001₂ | 01010101₂ --> 01111101₂
Contrast: Logic Operations in C

- **Contrast to logical operators**
  - &&, ||, !
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination

- **Examples (char data type)**
  - !0x41 --> 0x00
  - !0x00 --> 0x01
  - !!0x41 --> 0x01
  - 0x69 && 0x55 --> 0x01
  - 0x69 || 0x55 --> 0x01
  - p && *p++ (avoids null pointer access, null pointer = 0x00000000)
  - if (p) *p++;