**CSE351**

- **Announcements:**
  - HW0, having fun?
  - Use discussion boards!
  - Sign up for cse351@cs mailing list
    - If you enrolled recently, you might not be on it
Today’s topics

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

- CPU
- Memory
- Bus
- Disks
- Net
- USB
- Etc.
Hardware: Semi-Logical View

Intel® P45 Express Chipset Block Diagram
Hardware: Physical View
Performance: It's Not Just CPU Speed

- Data and instructions reside in memory
  - To execute an instruction, it must be fetched onto the CPU
  - Then, the data the instruction operates on must be fetched onto 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” on the CPU chip
    - The next slide is just an introduction. We'll see a more full explanation later in the course.
There are a fixed number of **registers** on the CPU
- Registers hold data

There is an **I-cache** on the CPU holding 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 out of its cache
Introduction to 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**  \(00000000_2 \rightarrow 11111111_2\)
  - Byte = 8 bits (binary digits)

- **Decimal**  \(0_{10} \rightarrow 255_{10}\)

- **Hexadecimal**  \(00_{16} \rightarrow FF_{16}\)
  - Byte = 2 hexadecimal (hex) or base 16 digits
  - Base-16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
  - Write \(FA1D37B_{16}\) in C
    - as \(0xFA1D37B\) or \(0xFA1D37B\)

<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>
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
  - Most current machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - High-end systems use 64 bits (8 bytes) words
    - Potential address space $\approx 1.8 \times 10^{19}$ bytes
    - x86-64 machines support 48-bit addresses: 256 Terabytes
    - Can’t be real physical addresses -> virtual addresses
  - Machines support multiple data formats
    - Fractions or multiples of word size
    - Always integral number of bytes
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>0000</td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0008</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td>Addr = 0012</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0003</td>
<td>0003</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0004</td>
<td>0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0005</td>
<td>0005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0006</td>
<td>0006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0007</td>
<td>0007</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0008</td>
<td>0008</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0009</td>
<td>0009</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0012</td>
<td>0012</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0013</td>
<td>0013</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0015</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 $15F_{16}$)
## 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

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

<table>
<thead>
<tr>
<th>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>
## 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>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>(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?

- Conventions!
  - Big-endian, Little-endian
  - Based on Gulliver stories, tribes cut eggs on different sides (big, little)
Byte Ordering Example

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

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

**Example**
- Variable has 4-byte representation \(0x1234567\)
- Address of variable is \(0x100\)

![Diagram showing byte ordering example for Big-Endian and Little-Endian](image-url)
Byte Ordering Example

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

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

**Example**
- Variable has 4-byte representation \(0x1234567\)
- Address of variable is \(0x100\)

\[
\begin{array}{cccc}
0x100 & 0x101 & 0x102 & 0x103 \\
\hline
\text{Big Endian} & & \text{01} & 23 & 45 & 67 & \text{ } \\
\text{Little Endian} & 0x100 & 0x101 & 0x102 & 0x103 & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } & \text{ } \end{array}
\]
Byte Ordering Example

- **Big-Endian** (PPC, 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
            01     23     45     67

Little Endian:  0x100  0x101  0x102  0x103
                67     45     23     01
```
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 *
  - 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

- We can do arithmetic on pointers
  - ptr = ptr + 1; // 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 y, may not – dangerous!)

- To use the value pointed to by a pointer we use de-reference
  - y = *ptr + 1; is the same as y = x + 1;
  - But, 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

& = ‘address of value’
* = ‘value at address’ or ‘de-reference’

*(&x) is equivalent to x
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**
  - int x, y;
  - x = y; // get value at y and put it in x

```
int x, y;
x = y; // get value at y and put it in x
```
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**
  - `int x, y;`
  - `x = y; // get value at y and put it in x`

```
<table>
<thead>
<tr>
<th>00</th>
<th>27</th>
<th>D0</th>
<th>3C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0008</td>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0014</td>
<td>0018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td>0024</td>
<td></td>
<td></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
  - int x, y;
    x = y; // get value at y and put it in x
  - int * x; int y;
    x = &y + 12; // get address of y add 12

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0000</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0004</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0008</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>000C</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0014</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
<td>0018</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>001C</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0020</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>0024</td>
<td></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
  - 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 to location x points

<table>
<thead>
<tr>
<th>LHS Location</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>0010</td>
<td>0010</td>
</tr>
<tr>
<td>0014</td>
<td>0014</td>
</tr>
<tr>
<td>0018</td>
<td>0018</td>
</tr>
<tr>
<td>0020</td>
<td>0020</td>
</tr>
<tr>
<td>0024</td>
<td>0024</td>
</tr>
</tbody>
</table>
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");
}
```

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

Some printf directives:

- %p: Print pointer
- %x: Print hexadecimal
- "\n": New line
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((pointer) &a, sizeof(int));

Result (Linux):

int a = 12345;
0x11fffffcb8 0x39
0x11fffffcb9 0x30
0x11fffffcba 0x00
0x11fffffcb0 0x00
## 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 Integers

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

<table>
<thead>
<tr>
<th>Decimal: 12345</th>
<th>Binary: 0011 0000 0011 1001</th>
<th>Hex: 3 0 3 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two’s complement representation for negative integers (covered later)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Representing Integers

- \texttt{int A = 12345;}
- \texttt{int B = -12345;}
- \texttt{long int C = 12345;}

\textbf{Decimal:} 12345  
\textbf{Binary:} 0011 0000 0011 1001  
\textbf{Hex:} 3 0 3 9

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

- \( \text{int} \ A = 12345; \)
- \( \text{int} \ B = -12345; \)
- \( \text{long} \ \text{int} \ C = 12345; \)

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

Two’s complement representation for negative integers (covered later)
Representing 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
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
Harry Potter \0
```

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

- Computing string length?
Compatibility

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

- **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|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></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

  \[
  \begin{array}{ccc}
  01101001 & 01101001 & 01101001 \\
  \& 01010101 & | 01010101 & ^ 01010101 \\
  \end{array}
  \]

- All of the properties of Boolean algebra apply

  \[
  \begin{array}{c}
  01010101 \\
  ^ 01010101 \\
  \end{array}
  \]

- 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 $j \in A$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$01101001$</td>
<td></td>
<td>${0,3,5,6}$</td>
<td></td>
</tr>
<tr>
<td>$01010101$</td>
<td></td>
<td>${0,2,4,6}$</td>
<td></td>
</tr>
</tbody>
</table>

**Operations**
- $\&$ Intersection:
  - $01000001$ $\{0,6\}$
- $|$ Union:
  - $01111101$ $\{0,2,3,4,5,6\}$
- $^\wedge$ Symmetric difference:
  - $00111100$ $\{2,3,4,5\}$
- $\sim$ Complement:
  - $10101010$ $\{1,3,5,7\}$
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
    - ~00000000₀₂ --> 11111111₁₂
  - 0x69 & 0x55 --> 0x41
    - 01101001₁₂ & 01010101₁₂ --> 01000001₁₂
  - 0x69 | 0x55 --> 0x7D
    - 01101001₁₂ | 01010101₁₂ --> 01111101₁₂
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)