CSE351

- **Announcements:**
  - HW0, having fun?
  - Use discussion boards!
  - Check if office hours work for you, let us know if they don’t.
  - Make sure you are subscribed to the mailing lists.
    - 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

[Diagram of Intel P45 Express Chipset Block Diagram]
Hardware: Physical View

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

This slide is just an introduction. We'll see a more full 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 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

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

![Diagram of different voltage levels: 0.0V, 0.5V, 2.8V, 3.3V.]

Encoding Byte Values

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

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

- **Hexadecimal** $00_{16}$ -- FF
  - 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?
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 $15F_{16}$)

Address 0004 stores the value 351 (or $15F_{16}$)

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

```
 00 00 01 5F 0000 0004 0008 000C 0010 0014 0018 0020 0024
```

- Address 0014 stores the value 12
  - Is it a pointer?
Data Representations

- **Sizes of objects (in bytes)**
  - | Java Data Type | C Data Type | Typical 32-bit x86-64 |
  - | boolean | bool | 1 | 1 |
  - | byte | char | 1 | 1 |
  - | char | int | 2 | 2 |
  - | short | short int | 2 | 2 |
  - | int | int | 4 | 4 |
  - | float | float | 4 | 4 |
  - | long int | long int | 4 | 8 |
  - | double | double | 8 | 8 |
  - | long | long long | 8 | 8 |
  - | long double | long double | 8 | 16 |
  - | (reference) | pointer * | 4 | 8 |

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 0x01234567
  - Address of variable is 0x100

```
Big Endian
0x100 0x101 0x102 0x103

Little Endian
0x100 0x101 0x102 0x103
```
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 pointed.
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]);
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>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>24 00 00 00</td>
</tr>
<tr>
<td>0004</td>
<td>0008</td>
</tr>
<tr>
<td>0010</td>
<td>0014</td>
</tr>
<tr>
<td>0018</td>
<td>0020</td>
</tr>
<tr>
<td>0024</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0004</td>
</tr>
<tr>
<td>0008</td>
<td>0010</td>
</tr>
<tr>
<td>0014</td>
<td>0018</td>
</tr>
<tr>
<td>0020</td>
<td>0024</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

```
24 00 00 00 0000
   0004
   0008
   000C
   0010
   0014
   0018
   001C
   0020
   0024
```
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");
}
```

```c
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 Integers

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

### Decimal: 12345

### Binary: 0011 0000 0011 1001

### Hex: 3 0 3 9

#### IA32, x86-64 A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>00</td>
<td>00</td>
<td>39</td>
</tr>
<tr>
<td>30</td>
<td>00</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

#### Sun A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

#### IA32, x86-64 B

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C7</td>
<td>FF</td>
<td>FF</td>
<td>C7</td>
</tr>
<tr>
<td>CF</td>
<td>FF</td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td></td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Sun B

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

#### IA32 C

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>30</td>
<td>00</td>
<td>39</td>
</tr>
</tbody>
</table>

#### X86-64 C

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>30</td>
<td>00</td>
<td>39</td>
</tr>
</tbody>
</table>

#### Sun C

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>00</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>39</td>
</tr>
</tbody>
</table>

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

#### IA32, x86-64 A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>00</td>
<td>00</td>
<td>39</td>
</tr>
<tr>
<td>30</td>
<td>00</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

#### Sun A

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

#### IA32, x86-64 B

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C7</td>
<td>FF</td>
<td>FF</td>
<td>C7</td>
</tr>
<tr>
<td>CF</td>
<td>FF</td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td></td>
<td>CF</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Sun B

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

#### IA32 C

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>30</td>
<td>00</td>
<td>39</td>
</tr>
</tbody>
</table>

#### X86-64 C

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>30</td>
<td>00</td>
<td>39</td>
</tr>
</tbody>
</table>

#### Sun C

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>00</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>39</td>
</tr>
</tbody>
</table>

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 Integers

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

| Decimal:  | 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.

<table>
<thead>
<tr>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
<th>46</th>
<th>47</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>.</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Null-terminated Strings

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

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>97</td>
<td>114</td>
<td>114</td>
<td>121</td>
<td>32</td>
<td>80</td>
<td>111</td>
<td>116</td>
<td>116</td>
<td>101</td>
<td>114</td>
<td>0</td>
</tr>
</tbody>
</table>

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

- Computing string length?
Compatibility

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|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
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01101001 & \quad \sim 01010101 \\
\& 01010101 & \mid 01010101 & \wedge 01010101 & \sim 01010101 \\
\end{align*}
\]

- All of the properties of Boolean algebra apply

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

- How does this relate to set operations?

<table>
<thead>
<tr>
<th>Operations</th>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; Intersection</td>
<td>01000001</td>
<td>{0, 6}</td>
</tr>
<tr>
<td></td>
<td>Union</td>
<td>01111101</td>
</tr>
<tr>
<td>^ Symmetric difference</td>
<td>00111100</td>
<td>{2, 3, 4, 5}</td>
</tr>
<tr>
<td>~ Complement</td>
<td>10101010</td>
<td>{1, 3, 5, 7}</td>
</tr>
</tbody>
</table>
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
    - 01101001 & 010101012 --> 010000012
  - 0x69 | 0x55 --> 0x7D
    - 011010012 | 010101012 --> 011110112

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)