Today’s topics

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

Binary Representations

- Base 2 number representation
  - Represent $351_{10}$ as $0000000101011111_2$ or $101011111_2$
  - Represent $3.6_{10}$ as $11.1001100110011001[1001]..._2$
  - Represent $3.51 \times 10^2$ as $1.01011111_2 \times 2^8$

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

![Binary representation diagram]
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
      - 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>

Byte-Oriented Memory Organization

- **Programs refer to addresses**
  - Conceptually, a very large array of bytes
  - Actually implemented with hierarchy of different memory types (later...)
  - 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)

<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></td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td>Addr = 0004</td>
<td></td>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>Addr = 0008</td>
<td></td>
<td></td>
<td>0002</td>
</tr>
<tr>
<td>Addr = 000C</td>
<td></td>
<td></td>
<td>0003</td>
</tr>
<tr>
<td>Addr = 0010</td>
<td></td>
<td></td>
<td>0004</td>
</tr>
<tr>
<td>Addr = 0014</td>
<td></td>
<td></td>
<td>0005</td>
</tr>
<tr>
<td>Addr = 0018</td>
<td></td>
<td></td>
<td>0006</td>
</tr>
<tr>
<td>Addr = 001C</td>
<td></td>
<td></td>
<td>0007</td>
</tr>
<tr>
<td>Addr = 0020</td>
<td></td>
<td></td>
<td>0008</td>
</tr>
<tr>
<td>Addr = 0024</td>
<td></td>
<td></td>
<td>0009</td>
</tr>
<tr>
<td>Addr = 0028</td>
<td></td>
<td></td>
<td>0010</td>
</tr>
<tr>
<td>Addr = 002C</td>
<td></td>
<td></td>
<td>0011</td>
</tr>
<tr>
<td>Addr = 0030</td>
<td></td>
<td></td>
<td>0012</td>
</tr>
<tr>
<td>Addr = 0034</td>
<td></td>
<td></td>
<td>0013</td>
</tr>
<tr>
<td>Addr = 0038</td>
<td></td>
<td></td>
<td>0014</td>
</tr>
<tr>
<td>Addr = 003C</td>
<td></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 $15F_{16}$)
- 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)
  - Java Data Type  | C Data Type  | Typical 32-bit | x86-64 |
  - boolean         | bool        | 1             | 1      |
  - byte            | char        | 1             | 1      |
  - char            |             | 2             | 2      |
  - short           | short int   | 2             | 2      |
  - int             | int         | 4             | 4      |
  - float           | float       | 4             | 4      |
  - long            | long int    | 4             | 8      |
  - double          | double      | 8             | 8      |
  - long long       |             | 8             | 8      |
  - long double     |             | 8             | 16     |
  - (reference)     | pointer *   | 4             | 8      |
Byte Ordering

- How should bytes within multi-byte word be ordered in memory?

- Conventions
  - Big-endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little-endian: x86
    - Least significant byte has lowest address

Byte Ordering Example

- Big-Endian
  - Least significant byte has highest address

- Little-Endian
  - Least significant byte has lowest address

- Example
  - Variable has 4-byte representation \(0x01234567\)
  - Address of variable is \(0x100\)

<table>
<thead>
<tr>
<th></th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</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
Arrays

- Arrays represent adjacent locations in memory storing the same type of data object
  - E.g., int big_array[128]; allocated 400 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 + 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

- 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

  | 00 2D 00 3C | 0000 | 0004 | 0008 | 000C | 0010 | 0014 | 0018 | 001C | 0020 | 0024 |
  | 00 27 D0 3C |
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

### 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

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
      - Fits into 8 bits with a leading 0
    - Character “0” has code 0x30
      - Digit i has code 0x30 + i
  - String should be null-terminated
    - Final character = 0x00

- **Compatibility**
  - 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>
<th></th>
<th>0</th>
<th>1</th>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>^</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>~</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Application of Boolean Algebra

- Applied to digital systems by Claude Shannon
  - 1937 MIT Masters Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

\[
\text{Connection when: } \quad A \& \sim B \lor \sim A \& B = A \lor B
\]

General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{align*}
01101001 & \quad 01101001 & \quad 01110001 \\
\& 01010101 & \mid 01010101 & \quad ^01010101 & \quad ^01010101 \\
01000001 & \quad 01111110 & \quad 00111100 & \quad 10101010
\end{align*}
\]

- All of the properties of Boolean algebra apply

\[
\begin{align*}
01010101 & \quad ^01010101 \\
\& 01010101 & \quad ^01010101 \\
00000000
\end{align*}
\]
Representing & Manipulating Sets

Representation
- Width $w$ bit vector represents subsets of \{0, ..., $w$−1\}
- $a_j = 1$ if $j \in A$
  
  01101001 \quad \{0, 3, 5, 6\} \quad 76543210

  01010101 \quad \{0, 2, 4, 6\} \quad 76543210

Operations
- & Intersection 01000001 \quad \{0, 6\}
- | Union 01111101 \quad \{0, 2, 3, 4, 5, 6\}
- ^ Symmetric difference 00111100 \quad \{2, 3, 4, 5\}
- ~ Complement 10101010 \quad \{1, 3, 5, 7\}

Bit-Level Operations in C

Operations & $\mid$, $^$, $\sim$ 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)
- $\sim$0x41 $\rightarrow$ 0xBE
  - $\sim$01000001$_2$ $\rightarrow$ 10111110$_2$
- $\sim$0x00 $\rightarrow$ 0xFF
  - $\sim$00000000$_2$ $\rightarrow$ 11111111$_2$
- 0x69 $\&$ 0x55 $\rightarrow$ 0x41
  - 0101001$_2$ $\&$ 01010101$_2$ $\rightarrow$ 01000001$_2$
- 0x69 $\mid$ 0x55 $\rightarrow$ 0x7D
  - 01101001$_2$ $\mid$ 01010101$_2$ $\rightarrow$ 01111101$_2$
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)