Bits and Bytes

Today
- Why bits?
- Binary/hexadecimal
- Byte representations
- Boolean algebra
- Expressing in C
Why don’t computers use Base 10?

- Base 10 number representation
  - “Digit” in many languages also refers to fingers/toes
    - Of course, decimal (from Latin *decimus*), means tenth
  - A position numeral system (unlike, say Roman numerals)
  - Natural representation for financial transactions
  - Even carries through in scientific notation

- Implementing electronically
  - Hard to store
    - ENIAC (First electronic computer) used 10 vacuum tubes / digit
  - Hard to transmit
    - Need high precision to encode 10 signal levels on single wire
  - Messy to implement digital logic functions
    - Addition, multiplication, etc.
Binary representations

- **Base 2 number representation**
  - Represent $15213_{10}$ as $11101101101101_2$
  - Represent $1.20_{10}$ as $1.0011001100110011[0011]..._2$

- **Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires
  - Straightforward implementation of arithmetic functions
Byte-oriented memory organization

- Programs refer to virtual addresses
  - Conceptually very large array of bytes
  - Each byte with its own address
  - All addresses – virtual address space
  - In Unix and Windows, address space private to particular “process”
    - Program being executed
    - Program can manipulate its own data, but not that of others

- Compiler + run-time system control allocation
  - Where different program objects should be stored
  - Multiple mechanisms: static, stack, and heap
  - In any case, all allocation within single virtual address space
How do we represent the address space?

- Hexadecimal notation
- Byte = 8 bits
  - Binary \(00000000_2\) to \(11111111_2\)
  - Decimal: \(0_{10}\) to \(255_{10}\)
  - Binary is too verbose, Decimal is hard to convert to/from bit patterns
  - Hexadecimal \(00_{16}\) to \(FF_{16}\)
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write \(\text{FA1D37B}_{16}\) in C as \(0x\text{FA1D37B}\)
      - 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>

\[1100 1001 0111 1011 \quad \rightarrow \quad 0xC97B\]
Machine words

- Machine has “word size”
  - Nominal size of integer-valued data
  - More importantly – a virtual address is encoded by such a word
    - Hence, it determines max size of virtual address space
  - Most current machines are 32 bits (4 bytes)
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Newer systems are 64 bits (8 bytes)
    - Potentially address \( \approx 1.8 \times 10^{19} \) bytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes
Data representations

- Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data type</th>
<th>32 bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long int</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>char*</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

- Portability:
  - Many programmers assume that object declared as int can be used to store a pointer
    - OK for a typical 32-bit machine
    - Problems on a 64-bit machine
Addressing and byte ordering

- For objects that span multiple bytes (e.g. integers), we need to agree on two things
  - what would be the address of the object?
  - how would we order the bytes in memory?
Word-oriented memory organization

- Addresses specify byte locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)
Byte ordering

- How to order bytes within multi-byte word in memory
- Conventions
  - (most) Sun’s, IBM’s are “Big Endian” machines
    - Least significant byte has highest address (comes last)
  - (most) Intel’s are “Little Endian” machines
    - Least significant byte has lowest address (comes first)
- Example
  - Variable \( x \) has 4-byte representation \( 0x01234567 \)
  - Address given by \&x is \( 0x100 \)

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

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>( 0x100 )</th>
<th>( 0x101 )</th>
<th>( 0x102 )</th>
<th>( 0x103 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
Reading byte-reversed Listings

- For most programmers, these issues are invisible
- Except with networking or disassembly
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- Example fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- Deciphering Numbers
  - Value: 0x12ab
  - Pad to 4 bytes: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00
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");
}
```

Printf directives:
- %p: Print pointer
- %x: Print Hexadecimal
show_bytes execution example

```c
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```c
int a = 15213;
0x11ffffffcb8 0x6d
0x11ffffffcb9 0x3b
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
```

0011 1011 0110 1101₂
3 b 6 d₁₆
Representing strings

- **Strings in C**
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Other encodings exist, but uncommon
    - Character “0” has code 0x30
      - Digit \( i \) has code 0x30+\( i \)
  - String should be null-terminated
    - Final character = 0

- **Compatibility**
  - Byte ordering not an issue
    - Data are single byte quantities
  - Text files generally platform independent
    - Except for different conventions of line termination character(s)!

```
char S[6] = "15213";
```
Machine-level code representation

- Encode program as sequence of instructions
  - Each simple operation
    - Arithmetic operation
    - Read or write memory
    - Conditional branch
  - Instructions encoded as bytes
    - Alpha’s, Sun’s, Mac’s use 4 byte instructions
      - Reduced Instruction Set Computer (RISC)
    - PC’s use variable length instructions
      - Complex Instruction Set Computer (CISC)
  - Different machines $\rightarrow$ different ISA & encodings
    - Most code not binary compatible

- A fundamental concept:
  Programs are byte sequences too!
Representing instructions

```c
int sum(int x, int y)
{
    return x+y;
}
```

- **Sun** use 2 4-byte instructions
  - Differing numbers in other cases
- **PC** uses instructions with lengths 1, 2, and 3 bytes
  - Mostly the same for NT and for Linux
  - NT / Linux not fully binary compatible

<table>
<thead>
<tr>
<th>Linux 32</th>
<th>55</th>
<th>89</th>
<th>E5</th>
<th>8B</th>
<th>45</th>
<th>0C</th>
<th>03</th>
<th>45</th>
<th>08</th>
<th>C9</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>55</td>
<td>89</td>
<td>E5</td>
<td>8B</td>
<td>45</td>
<td>0C</td>
<td>03</td>
<td>45</td>
<td>08</td>
<td>5D</td>
<td>C3</td>
</tr>
<tr>
<td>Sun</td>
<td>81</td>
<td>C3</td>
<td>E0</td>
<td>08</td>
<td>90</td>
<td>02</td>
<td>00</td>
<td>09</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Different machines use totally different instructions and encodings*
Boolean algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - $\langle\{0,1\},|,\&,\neg,0,1\rangle$
  - | is “sum” operation, & is “product” operation
  - $\neg$ is “complement” operation (not additive inverse)
  - 0 is identity for sum, 1 is identity for product

<table>
<thead>
<tr>
<th>Not $\neg A$</th>
<th>And $A &amp; B$</th>
<th>Or $A \mid B$</th>
<th>Xor $A ^ B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg$</td>
<td>$&amp;$ 0 1</td>
<td>1 0 1</td>
<td>$\neg$ 0 1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0 1</td>
<td>1 1 1</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
Application of Boolean Algebra

- Applied to Digital Systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

Connection when

\[ A \& \neg B \mid \neg A \& B \]

\[ = A^\wedge B \]
Relations between operations

- **DeMorgan’s Laws**
  - Express & in terms of |, and vice-versa
    - A & B = ~(~A | ~B)
      - A and B are true if and only if neither A nor B is false
    - A | B = ~(~A & ~B)
      - A or B are true if and only if A and B are not both false

- **Exclusive-Or using Inclusive Or**
  - A ^ B = (~A & B) | (A & ~B)
    - Exactly one of A and B is true
  - A ^ B = (A | B) & ~(A & B)
    - Either A is true, or B is true, but not both
General Boolean algebras

- Boolean operations can be extended to work on bit vectors
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 & 01101001 & 01101001 \\
& 01010101 & 01010101 & 01010101 \\
& 01000001 & 01111101 & 00111100 & 10101010
\end{array}
\]

- All of the properties of Boolean algebra apply
- Now, Boolean |, &, and \(\sim\) correspond to set union, intersection and complement
Representing & manipulating sets

- Useful application of bit vectors – represent finite sets
- Representation
  - Width $w$ bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j = 1$ if $j \in A$
    - 01101001 represents $\{0, 3, 5, 6\}$
    - 01010101 represents $\{0, 2, 4, 6\}$

- 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 &, |, ~, ^ available in C
  
  - Apply to any “integral” data type
    
    - long, int, short, char
  
  - 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
Logic operations in C – not quite the same

- Logical operations ||, && and ! (Logic OR, AND & Not)
  - Contrast to logical operators
    - View 0 as “False”
    - *But anything nonzero as “True”*
    - Always return 0 or 1
    - Early termination (*if you can answer by just looking at first argument, you are done*)

- Examples (char data type)
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
Shift operations

- **Left shift: \( x << y \)**
  - Shift bit-vector \( x \) left \( y \) positions
    - Throw away extra bits on left
    - Fill with 0’s on right

- **Right shift: \( x >> y \)**
  - Shift bit-vector \( x \) right \( y \) positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0’s on left
  - Arithmetic shift
    - Replicate most significant bit on right
    - Useful with two’s complement integer representation
  - For unsigned data, \( >> \) must be logical; for signed data either could be used but mostly arithmetic
    - Which one? Most follow this but not all
Main points

- It’s all about bits & bytes
  - Numbers
  - Programs
  - Text

- Different machines follow different conventions
  - Word size
  - Byte ordering
  - Representations

- Boolean algebra is mathematical basis
  - Basic form encodes “false” as 0, “true” as 1
  - General form like bit-level operations in C
    - Good for representing & manipulating sets
Integer & Boolean algebra

- **Integer Arithmetic**
  - $\langle \mathbb{Z}, +, *, -, 0, 1 \rangle$ forms a mathematical structure called “ring”
  - Addition is “sum” operation
  - Multiplication is “product” operation
  - $-$ is additive inverse
  - $0$ is identity for sum
  - $1$ is identity for product

- **Boolean Algebra**
  - $\langle \{0, 1\}, |, \&, \sim, 0, 1 \rangle$ forms a mathematical structure called “Boolean algebra”
  - Or is “sum” operation
  - And is “product” operation
  - $\sim$ is “complement” operation (not additive inverse)
  - $0$ is identity for sum
  - $1$ is identity for product
### Boolean Algebra \(\approx\) Integer Ring

<table>
<thead>
<tr>
<th>Property</th>
<th>Boolean Expression</th>
<th>Integer Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Commutative</strong></td>
<td>(A \lor B \Rightarrow B \lor A) (A \land B \Rightarrow B \land A)</td>
<td>(A + B = B + A) (A \cdot B = B \cdot A)</td>
</tr>
<tr>
<td><strong>Associativity</strong></td>
<td>((A \lor B) \lor C \Rightarrow A \lor (B \lor C)) ((A \land B) \land C \Rightarrow A \land (B \land C))</td>
<td>((A + B) + C = A + (B + C)) ((A \cdot B) \cdot C = A \cdot (B \cdot C))</td>
</tr>
<tr>
<td><strong>Product distributes over sum</strong></td>
<td>(A \land (B \lor C) = (A \land B) \lor (A \land C))</td>
<td>(A \cdot (B + C) = A \cdot B + B \cdot C)</td>
</tr>
<tr>
<td><strong>Sum and product identities</strong></td>
<td>(A \lor 0 = A) (A \land 1 = A)</td>
<td>(A + 0 = A) (A \cdot 1 = A)</td>
</tr>
<tr>
<td><strong>Zero is product annihilator</strong></td>
<td>(A \land 0 = 0)</td>
<td>(A \cdot 0 = 0)</td>
</tr>
<tr>
<td><strong>Cancellation of negation</strong></td>
<td>(~ (\sim A) = A)</td>
<td>(\sim (\sim A) = A)</td>
</tr>
</tbody>
</table>
# Boolean Algebra ≠ Integer Ring

<table>
<thead>
<tr>
<th>Boolean: Sum distributes over product</th>
<th>$A \mid (B &amp; C) = (A \mid B) &amp; (A \mid C)$</th>
<th>$A + (B \ast C) \neq (A + B) \ast (B + C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean: Idempotency</td>
<td>$A \mid A = A$</td>
<td>$A + A \neq A$</td>
</tr>
<tr>
<td></td>
<td>$A &amp; A = A$</td>
<td>$A \ast A \neq A$</td>
</tr>
<tr>
<td>Boolean: Absorption</td>
<td>$A \mid (A &amp; B) = A$</td>
<td>$A + (A \ast B) \neq A$</td>
</tr>
<tr>
<td></td>
<td>$A &amp; (A \mid B) = A$</td>
<td>$A \ast (A + B) \neq A$</td>
</tr>
<tr>
<td>Boolean: Laws of Complements</td>
<td>$A \mid \neg A = 1$</td>
<td>$A + \neg A \neq 1$</td>
</tr>
<tr>
<td>Ring: Every element has additive inverse</td>
<td>$A \mid \neg A \neq 0$</td>
<td>$A + \neg A = 0$</td>
</tr>
</tbody>
</table>
Properties of $\&$ and $^\wedge$

- **Boolean ring**
  - $\langle \{0,1\}, ^\wedge, \&\!, I, 0, 1 \rangle$
  - Identical to integers mod 2
  - $I$ is identity operation: $I(A) = A$
    - $A \wedge A = 0$

- **Property: Boolean ring**
  - Commutative sum $A \wedge B = B \wedge A$
  - Commutative product $A \& B = B \& A$
  - Associative sum $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
  - Associative product $(A \& B) \& C = A \& (B \& C)$
  - Prod. over sum $A \& (B \wedge C) = (A \& B) \wedge (B \& C)$
  - 0 is sum identity $A \wedge 0 = A$
  - 1 is prod. identity $A \& 1 = A$
  - 0 is product annihilator $A \& 0 = 0$
  - Additive inverse $A \wedge A = 0$