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 \( A=1 \) and \( B=1 \)
    
    | & | 0 | 1 |
    |---|---|---|
    | 0 | 0 | 0 |
    | 1 | 0 | 1 |
  - **Or**
    - \( A | B = 1 \) when either \( A=1 \) or \( B=1 \)
    
    | | 0 | 1 |
    |---|---|---|
    | 0 | 0 | 1 |
    | 1 | 1 | 1 |
  - **Not**
    - \( \sim A = 1 \) when \( A=0 \)
    
    | ~ | 0 | 1 |
    |---|---|---|
    | 0 | 1 | 0 |
    | 1 | 0 | 1 |
  - **Exclusive-Or (Xor)**
    - \( A \wedge B = 1 \) when either \( A=1 \) or \( B=1 \), but not both
    
    | ^ | 0 | 1 |
    |---|---|---|
    | 0 | 0 | 1 |
    | 1 | 1 | 0 |
General Boolean Algebras

❖ Operate on Bit Vectors

- Operations applied bitwise

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

❖ All of the Properties of Boolean Algebra Apply
**Representing & Manipulating Sets**

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

**Operations**
- & Intersection 01000001 \ {0, 6}
- | Union 01111101 \ {0, 2, 3, 4, 5, 6}
- ^ Symmetric difference 00111100 \ {2, 3, 4, 5}
- ~ 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 type => 1 byte, 8 bits)**
  - ~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 (short-circuit evaluation, minimal evaluation)

❖ Examples (char type)
  • !0x41  -->  0x00
  • !0x00  -->  0x01
  • !!0x41  -->  0x01
  • 0x69 && 0x55  -->  0x01
  • 0x69 || 0x55  -->  0x01
  • if (p && *p) (avoids null pointer access)
# 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

### Undefined behavior
- Shift amount <0 or \( \geq \) word size

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 01100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td>( 00011000 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( 10100010 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;&lt; 3 )</td>
<td>( 00010000 )</td>
</tr>
<tr>
<td>Log. ( &gt;&gt; 2 )</td>
<td>( 00101000 )</td>
</tr>
<tr>
<td>Arith. ( &gt;&gt; 2 )</td>
<td>( 11101000 )</td>
</tr>
</tbody>
</table>
## Cool Stuff with XOR

- Bitwise XOR is form of addition
- With extra property that every value is its own additive inverse
  \[ A \oplus A = 0 \]

```c
void funny(int x, int y) {
    x = x ^ y;    /* 1 */
    y = x ^ y;    /* 2 */
    x = x ^ y;    /* 3 */
}
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A^B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A^B</td>
<td>(A^B)^B = A</td>
</tr>
<tr>
<td>3</td>
<td>(A^B)^A = B</td>
<td>A</td>
</tr>
<tr>
<td>End</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>
Why Don’t Computers Use Base 10?

- **Base 10 Number Representation**
  - That’s why fingers are known as “digits”
  - Natural representation for financial transactions
    - Floating point number cannot exactly represent $1.20$
  - Even carries through in scientific notation
    - $1.5213 \times 10^4$

- **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 $1.0011001100110011[0011]..._2$
  - Represent $1.5213 \times 10^4$ as $1.1101101101101_2 \times 2^{13}$

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

- **Byte = 8 bits**
  - Binary: $00000000_2$ to $11111111_2$
  - Decimal: $0_{10}$ to $255_{10}$
  - Hexadecimal: $00_{16}$ to $FF_{16}$
    - 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>
Machine Has “Word Size”

- Nominal size of integer-valued data
  - Including addresses
- Until recently, most machines used 32 bits (4 bytes)
  - Limits addresses to $4\text{GB} \approx 4.3 \times 10^9$
  - Becoming too small for memory-intensive applications
- Increasingly, machines use 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
## Example Data Representations

### Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>–</td>
<td>–</td>
<td>10/16</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
**Accessing Words in Memory**

- **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 are the bytes within a multi-byte word ordered in memory?

❖ Conventions
  ▪ “Big Endian” machines: Sun, PowerPC Mac, Internet
    ▪ Least significant byte has highest address
  ▪ “Little Endian” machines: x86, ARM running Android, iOS, Windows
    ▪ Least significant byte has lowest address

❖ Note:
  ▪ Alpha and PowerPC can run in either mode.
    Byte ordering is determined when the chip is powered up.
  ▪ Problematic when transferring binary data over the network between machines with different byte ordering.
**Byte Ordering Example**

- **Big Endian**
  - Least significant byte has highest address

- **Little Endian**
  - Least significant byte has lowest address

- **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></td>
<td>01</td>
<td>23</td>
<td>45</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>
### Byte Ordering Examples

- `int A = 15213;`
- `int B = -15213;`
- `long int C = 15213;`

#### Decimal: 15213
- **Binary:** 0011 1011 0110 1101
- **Hex:** 3B6D

#### Two’s complement representation

- **Intel/Alpha A**
  - 00: 6D
  - 01: 3B
  - 02: 00
  - 03: 00

- **Sun A**
  - 00: 00
  - 01: 00
  - 02: 3B
  - 03: 6D

- **Intel/Alpha B**
  - 00: 93
  - 01: C4
  - 02: FF
  - 03: FF

- **Sun B**
  - 00: FF
  - 01: FF
  - 02: C4
  - 03: 93

- **Intel C**
  - 00: 6D
  - 01: 3B
  - 02: 00
  - 03: 00

- **Alpha C**
  - 00: 6D
  - 01: 3B
  - 02: 00
  - 03: 00

- **Sun C**
  - 00: 00
  - 01: 3B
  - 02: 6D

---

Two’s complement representation
Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example Fragment** (objdump –d a.out)

<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 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
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 (null)

❖ **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)!

```c
char S[6] = "15213";
```

<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>35</td>
<td>35</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
Representing Instructions

- **Encode Program as Sequence of Instructions**
  - Each simple operation
    - Arithmetic operation
    - Read or write memory
    - Conditional branch
  - Instructions encoded as bytes
    - Alpha, Sun Sparc, IBM PowerPC use 4 byte instructions
      - Reduced Instruction Set Computer (RISC)
    - Intel/AMD x86, x86-64 use variable length instructions
      - Complex Instruction Set Computer (CISC)
  - Different instruction types and encodings for different machines
    - Most code not binary compatible

- **Programs are Byte Sequences Too!**
  - Still follow byte ordering for multi-byte instructions
Representing Instructions (Cont’d)

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

 firma For this example, Alpha & Sun use two 4-byte instructions
- Use differing numbers of instructions in other cases

❖ PC uses 7 instructions with lengths 1, 2, and 3 bytes
- Same for Windows and for Linux
- Windows / Linux not fully binary compatible

Different machines use totally different instructions and encodings
What is the output of this program?

- Solaris/SPARC (big endian): ?
- Linux/x86 (little endian): ?

```c
#include <stdio.h>

union {
    int i;
    unsigned char c[4];
} u;

int main () {
    u.i = 0x12345678;
    printf("%x %x %x %x\n",
            u.c[0], u.c[1], u.c[2], u.c[3]);
}
```
It’s All About Bits & Bytes
- Numbers
- Programs
- Text

Different Machines Follow Different Conventions
- Word size
- Byte ordering
- Representations (Integer, Floating-Point)

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