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 $1.11011011011011 \times 2^{13}$

Electronic Implementation
- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires

Byte-Oriented Memory Organization

Programs Refer to Virtual Addresses
- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
  - SRAM, DRAM, disk
  - Only allocate for regions actually used by program
- In Unix and Windows NT, address space private to particular “process”
  - Program being executed
  - Program can clobber 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
Encoding Byte Values

Byte = 8 bits
- Binary: 00000000₂ to 11111111₂
- Decimal: 0₁₀ to 255₁₀
- Hexadecimal: 0₁₆ to FF₁₆
  - 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>

Literary Hex

Common 8-byte hex filler:
- 0xdeadbeef
- Can you think of other 8-byte fillers?

Hex poetry (Bruce “the Bard” Maggs, 2003):

61caca
afadacad
abaddeed
acabead
adeaddeb

Machine Words

Machine Has “Word Size”
- Nominal size of integer-valued data
  - Including addresses
- Most current machines are 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems are 64 bits (8 bytes)
  - Potentially address ≈ 1.8 × 10¹⁹ bytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

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)
Data Representations

Sizes of C Objects (in Bytes)
- C Data Type    Compaq Alpha    Typical 32-bit    Intel IA32
  - int           4              4              4
  - long int      8              4              4
  - char          1              1              1
  - short         2              2              2
  - float         4              4              4
  - double        8              8              8
  - long double   8              8              10/12
  - char *        8              4              4

» Or any other pointer

Byte Ordering

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

Conventions
- Sun’s, Mac’s are “Big Endian” machines
  - Least significant byte has highest address
- Alphas, PC’s are “Little Endian” machines
  - 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 x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

Big Endian

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>0x101</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>0x102</td>
<td>83 bb 28 00 00 00</td>
<td>cmpl $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

Little Endian

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>0x101</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>0x102</td>
<td>83 bb 28 00 00 00</td>
<td>cmpl $0x0,0x28(%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 Fragment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000012ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00 00 12 ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ab 12 00 00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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):
```
int a = 15213;
0x11fffcbb 0x6d
0x11fffcba 0x3b
0x11fffcba 0x00
```

Representing Integers

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

<table>
<thead>
<tr>
<th>Decimal: 15213</th>
<th>Binary: 0011 1011 0110 1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex: 3B 6D</td>
<td></td>
</tr>
</tbody>
</table>

**Representing Pointers**

```c
int B = -15213;
int *P = &B;
```

<table>
<thead>
<tr>
<th>Alpha P</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
</tr>
<tr>
<td>FC</td>
</tr>
<tr>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sun P</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
</tr>
<tr>
<td>FF</td>
</tr>
<tr>
<td>FB</td>
</tr>
<tr>
<td>2C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1F FF FF FF FC A0</td>
</tr>
<tr>
<td>Binary: 0001 1111 1111 1111 1111 1111 1110 1010 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linux P</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4</td>
</tr>
<tr>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
</tr>
<tr>
<td>BF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>B F F F F F 8 D 4</td>
</tr>
<tr>
<td>Binary: 1011 1111 1111 1111 1111 1111 1000 1101 0100</td>
</tr>
</tbody>
</table>

**Representing Pointers**

```c
int C = 15213;
long int D = 15213;
```

<table>
<thead>
<tr>
<th>Linux/Alpha A</th>
<th>Sun A</th>
</tr>
</thead>
<tbody>
<tr>
<td>6D</td>
<td>00</td>
</tr>
<tr>
<td>3B</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>6D</td>
</tr>
<tr>
<td>00</td>
<td>3B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linux/Alpha B</th>
<th>Sun B</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>FF</td>
</tr>
<tr>
<td>C4</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>C4</td>
</tr>
<tr>
<td>FF</td>
<td>93</td>
</tr>
</tbody>
</table>

Two’s complement representation (Covered next lecture)

Different compilers & machines assign different locations to objects
### Representing Floats

Float $F = 15213.0$;

<table>
<thead>
<tr>
<th></th>
<th>Linux/Alpha</th>
<th>Sun</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00$</td>
<td>B4</td>
<td>46</td>
</tr>
<tr>
<td>$B4$</td>
<td>6D</td>
<td>B4</td>
</tr>
<tr>
<td>$6D$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$46$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IEEE Single Precision Floating Point Representation**

Hex: $466DB400$

Binary: $010001101101011100000000$

15213: $1110110110101$

Not same as integer representation, but consistent across machines

Can see some relation to integer representation, but not obvious

### Representing Strings

**Strings in C**

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character "0" has code 0x30
    - Digit / has code 0x30+i
- String should be null-terminated
  - Final character = 0

**Compatibility**

- Byte ordering not an issue
- Text files generally platform independent
  - Except for different conventions of line termination character(s)!
    - Unix (\'\n\' = 0x0a = ^J)
    - Mac (\'\r\' = 0x0d = ^M)
    - DOS and HTTP (\'\r\n\' = 0x0d0a = ^M^J)

### 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 instruction types and encodings for different machines
  - Most code not binary compatible

**Programs are Byte Sequences Too!**

### Representing Instructions

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

<table>
<thead>
<tr>
<th></th>
<th>Alpha sum</th>
<th>Sun sum</th>
<th>PC sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00$</td>
<td>$81$</td>
<td>$C3$</td>
<td>$55$</td>
</tr>
<tr>
<td>$00$</td>
<td>$30$</td>
<td>$E0$</td>
<td>$E5$</td>
</tr>
<tr>
<td>$42$</td>
<td>$08$</td>
<td></td>
<td>$8B$</td>
</tr>
<tr>
<td>$01$</td>
<td>$90$</td>
<td></td>
<td>$4E$</td>
</tr>
<tr>
<td>$80$</td>
<td>$02$</td>
<td></td>
<td>$0C$</td>
</tr>
<tr>
<td>$FA$</td>
<td>$00$</td>
<td></td>
<td>$03$</td>
</tr>
<tr>
<td>$6B$</td>
<td>$09$</td>
<td></td>
<td>$45$</td>
</tr>
<tr>
<td></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

**And**

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

**Or**

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

**Not**

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

**Exclusive-Or (Xor)**

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

\[ A \oplus B | \sim A \& B \]

\[ = A \& \sim B \]

---

**Integer Algebra**

**Integer Arithmetic**

- \( \langle Z, +, *, -, 0, 1 \rangle \) forms a “ring”
- Addition is “sum” operation
- Multiplication is “product” operation
- \( - \) is additive inverse
- 0 is identity for sum
- 1 is identity for product

---

**Boolean Algebra**

**Boolean Algebra**

- \( \langle \{0,1\}, |, \&, \sim, 0, 1 \rangle \) forms a “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 ≠ Integer Ring**

- **Commutativity**
  \[ A \land B = B \land A \]
  \[ A \lor B = B \lor A \]
- **Associativity**
  \[ (A \land B) \land C = A \land (B \land C) \]
  \[ (A \lor B) \lor C = A \lor (B \lor C) \]
- **Product distributes over sum**
  \[ A \land (B \lor C) = (A \land B) \lor (A \land C) \]
  \[ A \lor (B \land C) = (A \lor B) \land (A \lor C) \]
- **Sum and product identities**
  \[ A \land 0 = 0 \]
  \[ A \land 1 = A \]
  \[ A \lor 0 = A \]
  \[ A \lor 1 = 1 \]
- **Zero is product annihilator**
  \[ A \land 0 = 0 \]
  \[ A \land 1 = A \]
- **Cancellation of negation**
  \[ \neg (\neg A) = A \]
  \[ (A \land B) \land C = (A \land B) \land C \]
  \[ (A \lor B) \lor C = (A \lor B) \lor C \]
  \[ A \land 0 = A \land 1 = A \]
  \[ A \lor 0 = A \lor 1 = 1 \]
  \[ A \land 0 = 0 \]
  \[ A \lor 1 = 1 \]
- **Idempotency**
  \[ A \land A = A \]
  \[ A \lor A = A \]
- **Absorption**
  \[ A \land (A \lor B) = A \]
  \[ A \lor (A \land B) = A \]
- **Laws of Complements**
  \[ A \land \neg A = 0 \]
  \[ A \lor \neg A = 1 \]
- **Ring: Every element has additive inverse**
  \[ A + (B \cdot C) = (A + B) \cdot (B + C) \]
  \[ A \cdot (B + C) = A \cdot B + A \cdot C \]
  \[ A + B = B + A \]
  \[ A \cdot B = B \cdot A \]

**Boolean Algebra = Integer Ring**

- **Commutativity**
  \[ A \land B = B \land A \]
  \[ A \lor B = B \lor A \]
- **Associativity**
  \[ (A \land B) \land C = A \land (B \land C) \]
  \[ (A \lor B) \lor C = A \lor (B \lor C) \]
- **Product distributes over sum**
  \[ A \land (B \lor C) = (A \land B) \lor (A \land C) \]
  \[ A \lor (B \land C) = (A \lor B) \land (A \lor C) \]
- **Sum and product identities**
  \[ A \land 0 = 0 \]
  \[ A \land 1 = A \]
  \[ A \lor 0 = A \]
  \[ A \lor 1 = 1 \]
- **Zero is product annihilator**
  \[ A \land 0 = 0 \]
  \[ A \land 1 = A \]
- **Cancellation of negation**
  \[ \neg (\neg A) = A \]
  \[ (A \land B) \land C = (A \land B) \land C \]
  \[ (A \lor B) \lor C = (A \lor B) \lor C \]
  \[ A \land 0 = A \land 1 = A \]
  \[ A \lor 0 = A \lor 1 = 1 \]
  \[ A \land 0 = 0 \]
  \[ A \lor 1 = 1 \]
- **Idempotency**
  \[ A \land A = A \]
  \[ A \lor A = A \]
- **Absorption**
  \[ A \land (A \lor B) = A \]
  \[ A \lor (A \land B) = A \]
- **Laws of Complements**
  \[ A \land \neg A = 0 \]
  \[ A \lor \neg A = 1 \]
- **Ring: Every element has additive inverse**
  \[ A + (B \cdot C) = (A + B) \cdot (B + C) \]
  \[ A \cdot (B + C) = A \cdot B + A \cdot C \]
  \[ A + B = B + A \]
  \[ A \cdot B = B \cdot A \]

**Properties of \( \land \) and \( \lor \)**

- \( \{0,1\}, \lor, \land, I, 0, 1 \)
- Idempotent to integers mod 2
- \( I \) is identity operation: \( I(A) = A \)
  \[ A \land A = 0 \]

**Relations Between Operations**

- **DeMorgan’s Laws**
  - Express \( \land \) in terms of |, and vice-versa
    - \( A \land B = \neg (\neg A \land \neg B) \)
      - A and B are true if and only if neither A nor B is false
    - \( A \lor B = \neg (\neg A \lor \neg B) \)
      - A or B are true if and only if neither A nor B is false
  - **Exclusive-Or using Inclusive Or**
    - \( A \land B = (\neg A \land B) \lor (A \land \neg B) \)
      - Exactly one of A and B is true
    - \( A \lor B = (A \land B) \lor (\neg A \land \neg B) \)
      - Either A is true, or B is true, but not both
**General Boolean Algebras**

**Operate on Bit Vectors**

- Operations applied bitwise
  
  \[
  \begin{array}{cccc}
  01101001 & 01101001 & 01101001 \\
  \& & \& & \& \\
  01000001 & 01111101 & 00111100 & 10101010 \\
  \end{array}
  \]

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 \(j \in A\)

<table>
<thead>
<tr>
<th>01101001</th>
<th>011010001</th>
<th>011010001</th>
<th>01010101</th>
<th>01010101</th>
<th>01010101</th>
</tr>
</thead>
</table>

\[
\begin{array}{cccc}
  01101001 & 01101001 & 01101001 \\
  \& & \& & \& \\
  01000001 & 01111101 & 00111100 & 10101010 \\
  \end{array}
  
  76543210
\]

**Operations**

- & Intersection
- | Union
- ^ Symmetric difference
- ~ Complement

<table>
<thead>
<tr>
<th>01000001</th>
<th>01111101</th>
<th>00111100</th>
<th>10101010</th>
</tr>
</thead>
</table>

\[
\begin{array}{cccc}
  01101001 & 01101001 & 01101001 \\
  \& & \& & \& \\
  01000001 & 01111101 & 00111100 & 10101010 \\
  \end{array}
  
  76543210
\]

---

**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 \longrightarrow 0xBE
  
  \~01000001 \longrightarrow 10111110
- \~0x00 \longrightarrow 0xFF
  
  \~00000000 \longrightarrow 11111111
- 0x69 & 0x55 \longrightarrow 0x41
  
  01101001 & 01010101 \longrightarrow 01000001
- 0x69 | 0x55 \longrightarrow 0x7D
  
  01101001 | 01010101 \longrightarrow 01111101
- 0x69 & & 0x55 \longrightarrow 0x01
- 0x69 | | 0x55 \longrightarrow 0x01
- p & & *p (avoids null pointer access)

---

**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 \longrightarrow 0x00
- !0x00 \longrightarrow 0x01
- !!0x41 \longrightarrow 0x01
- 0x69 && 0x55 \longrightarrow 0x01
- 0x69 || 0x55 \longrightarrow 0x01
- p & & *p (avoids null pointer access)
### Shift Operations

**Left Shift:** $x \ll y$
- Shift bit-vector $x$ left $y$ positions
  - Throw away extra bits on left
  - Fill with 0’s on right

**Right Shift:** $x \gg y$
- 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

<table>
<thead>
<tr>
<th>Argument $x$</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ll 3$</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. $\gg 2$</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. $\gg 2$</td>
<td>00011000</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
  $$A \land A = 0$$

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

<table>
<thead>
<tr>
<th>$*x$</th>
<th>$*y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$B$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical $\gg$ 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \land B$</td>
</tr>
<tr>
<td>$(A \land B) \land B = A$</td>
</tr>
<tr>
<td>$(A \land B) \land A = B$</td>
</tr>
<tr>
<td>$B$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
</tbody>
</table>

### 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