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 $11101101101_2$
- Represent $1.20_{10}$ as $1.00100110011001100[0011]..._2$
- 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

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: \(00000000_2\) to \(11111111_2\)
  - Decimal: \(0_{10}\) to \(255_{10}\)
  - Hexadecimal: \(00_{16}\) to \(FF_{16}\)

  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B in C as `0xFA1D37B`
    - Or `0xFA1D37B` in C

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 X 10^18 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)**
  - For C Data Type Compaq Alpha
    - Typical 32-bit
    - Intel IA32
    - **int**: 4
    - **long int**: 8
    - **char**: 1
    - **short**: 2
    - **float**: 4
    - **double**: 8
    - **long double**: 8
    - **char ***: 8
      - 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

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>0x0100</td>
<td>0x100 0x101 0x102 0x103</td>
<td>01 23 45 67</td>
</tr>
<tr>
<td>0x0101</td>
<td>0x100 0x101 0x102 0x103</td>
<td>67 45 23 01</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%tp\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;
0x11ffffc8 0x6d
0x11ffffc9 0x3b
0x11ffffe 0x00
0x11fffebb 0x00
```

Representing Integers

```c
int A = 15213;
A = 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:</td>
<td></td>
</tr>
<tr>
<td>Alpha A</td>
<td>3B 6D</td>
</tr>
<tr>
<td>Sun A</td>
<td></td>
</tr>
</tbody>
</table>

Two's complement representation (Covered next lecture)

Representing Pointers

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

<table>
<thead>
<tr>
<th>Alpha Address</th>
<th>Hex: 1 F F F F F C A 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary:</td>
<td>0001 1111 1111 1111 1111 1110 1010 0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sun Address</th>
<th>Hex: E F F F F B 2 C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary:</td>
<td>1110 1111 1111 1111 1111 1011 0010 1100</td>
</tr>
</tbody>
</table>

Different compilers & machines assign different locations to objects

Representing Floats

```c
Float F = 15213.0;
```

<table>
<thead>
<tr>
<th>Linux/Alpha A</th>
<th>Sun f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex:</td>
<td></td>
</tr>
<tr>
<td>Alpha A</td>
<td>46 B4</td>
</tr>
<tr>
<td>Sun A</td>
<td>6D B4</td>
</tr>
</tbody>
</table>

IEEE Single Precision Floating Point Representation

```c
Hex: 4 6 6 D B 4 0 0 |
Binary: 0100 0110 0110 1101 1011 0100 0000 0000 |
15213: 1111 1101 1011 01 |
```

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
  - Other encodings exist, but uncommon
  - Character '0' has code 0x30
  - Digit / 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)!

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

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 NT and for Linux
  - NT / Linux not fully binary compatible

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
- A&B = 1 when both A=1 and B=1

Or
- A|B = 1 when either A=1 or B=1

Not
- ~A = 1 when A=0

Exclusive-Or (Xor)
- A^B = 1 when either A=1 or B=1, but not both
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
  
  \[ A \land \neg B \]
  \[ \neg A \lor B \]
  
  Connection when
  
  \[ A \land \neg B \lor \neg A \land B \]
  
  \[ A \land B \lor \neg A \land \neg B \]

- 21 -

Booleans Algebra

- \[ \langle 0, 1, \lor, \land, \neg, 0, 1 \rangle \] forms a “Boolean algebra”
- Or is “sum” operation
- And is “product” operation
- \( \neg \) is “complement” operation (not additive inverse)
- 0 is identity for sum
- 1 is identity for product

- 22 -

Integer Algebra

Integer Arithmetic

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

- 23 -

Boolean Algebra \( \approx \) Integer Ring

- **Commutativity**
  - \( A + B = B + A \)
  - \( A \ast B = B \ast A \)
- **Associativity**
  - \( (A + B) + C = A + (B + C) \)
  - \( (A \ast B) \ast C = A \ast (B \ast C) \)
- **Product distributes over sum**
  - \( A \ast (B + C) = (A \ast B) + (A \ast C) \)
  - \( A \ast 0 = 0 \)
- **Sum and product identities**
  - \( A + 0 = A \)
  - \( A \ast 1 = A \)
- **Zero is product annihilator**
  - \( A \ast 0 = 0 \)
- **Cancellation of negation**
  - \( \neg \neg A = A \)
  - \( \neg A = \neg A \)

- 24 -
Boolean Algebra ≠ Integer Ring

- **Boolean: Sum distributes over product**
  \[ A \mid (B \& C) = (A \mid B) \& (A \mid C) \quad A + (B \ast C) = (A + B) \ast (B + C) \]

- **Boolean: Idempotency**
  \[ A \mid A = A \quad A + A = A \]

- **Boolean: Absorption**
  \[ A \mid (A \& B) = A \quad A + (A \ast B) = A \]

- **Boolean: Laws of Complements**
  \[ A \mid \sim A = 1 \quad A + \sim A = 1 \]

- **Ring: Every element has additive inverse**
  \[ A \mid \sim A = 0 \quad A + \sim A = 0 \]

Relations Between Operations

**DeMorgan's Laws**

- **Express & in terms of |, and vice-versa**
  \[ A \& B = \sim(\sim A \mid \sim B) \]
  - A and B are true if and only if neither A nor B is false
  \[ A \mid B = \sim(\sim A \& \sim B) \]
  - A or B are true if and only if A and B are not both false

**Exclusive-Or using Inclusive Or**

- **A \& B = (\sim A \& B) \mid (A \& \sim B)**
  - Exactly one of A and B is true
  \[ A \& B = (A \mid B) \& \sim(A \& B) \]
  - Either A is true, or B is true, but not both

Boolean Ring

- **Properties of & and ^**
  \[ (0,1), \& , \ast , \{0,1\} \]
  - Identical to integers mod 2
  - \( / \) is identity operation: \( /A = A \quad A \& 0 = 0 \quad A \& 1 = A \quad A + 1 = 1 \)

**Property**

- **Commutative sum**
  \[ A \& B = B \& A \]

- **Commutative product**
  \[ A \& B = B \& A \]

- **Associative sum**
  \[ (A \& B) \& C = A \& (B \& C) \]

- **Associative product**
  \[ (A \& B) \& C = A \& (B \& C) \]

- **Prod. over sum**
  \[ A \& (B \& C) = (A \& B) \& (B \& C) \]

- **0 is sum identity**
  \[ A \& 0 = A \]

- **1 is prod. identity**
  \[ A \& 1 = A \]

- **0 is product annihilator**
  \[ A \& 0 = 0 \]

- **Additive inverse**
  \[ A \& A = 0 \]

General Boolean Algebras

Operate on Bit Vectors

- **Operations applied bitwise**
  \[
  \begin{array}{ccc}
  01 & 10 & 01 \\
  01 & 01 & 10 \\
  10 & 10 & 01 \\
  \end{array}
  \]

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

**Representation**
- Width $w$ bit vector represents subsets of $\{0, \ldots, w-1\}$
- $b_j = 1 \text{ if } j \in A$

<table>
<thead>
<tr>
<th>$b$</th>
<th>$\text{bit vector}$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>{0, 3, 5, 6}</td>
<td>6543210</td>
</tr>
<tr>
<td>01010101</td>
<td>{0, 2, 4, 6}</td>
<td>6543210</td>
</tr>
</tbody>
</table>

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

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
- ~0x00 --> 0xFF
- 0x69 & 0x55 --> 0x41
- 0x101001$\text{u}$ & 0x1010101$\text{u}$ --> 0x000001$\text{s}$
- 0x69 | 0x55 --> 0x7D
- 0x101001$\text{u}$ | 0x1010101$\text{u}$ --> 0x111111$\text{s}$

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
- 0x101001 --> 0x01
- 0x69 && 0x55 --> 0x01
- 0x69 || 0x55 --> 0x01
- 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 left
  - Useful with two's complement integer representation
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 */
}
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A^B</td>
<td>(A^B)^B = A</td>
</tr>
<tr>
<td>3</td>
<td>(A*B)^A = B</td>
</tr>
<tr>
<td></td>
<td></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