Giving credit where credit is due

Most of slides for this lecture are based on slides created by Drs. Bryant and O'Hallaron, Carnegie Mellon University.
I have modified them and added new slides.

Topics

- Why bits?
  - Representing information as bits
    - Binary/Hexadecimal
    - Byte representations
      - numbers
      - characters and strings
      - Instructions
  - Bit-level manipulations
    - Boolean algebra
    - Expressing in C

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
- 3.3V
- 2.8V
- 0.5V
- 0.0V

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 (and 2000), address space is private to a 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: 0 to 255₁₀
  - Hexadecimal: 00₁₆ to FF₁₆
- Use characters '0' to '9' and 'A' to 'F'
- Example: Write FA1D37B₁₆ in C as 0xFA1D37B or 0xfa1d37b

Machine Words

- **Machine Has “Word Size”**
  - Nominal size of integer-valued data
  - Including addresses
  - Most current machines are 32 bits (4 bytes)
  - Becoming too small for memory-intensive applications
  - High-end systems are 64 bits (8 bytes)
  - Machines support multiple data formats
  - 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)

32-bit Words

<table>
<thead>
<tr>
<th>Addr</th>
<th>0000</th>
<th>0001</th>
<th>0002</th>
<th>0003</th>
<th>0004</th>
<th>0005</th>
<th>0006</th>
<th>0007</th>
<th>0008</th>
<th>0009</th>
<th>0010</th>
<th>0011</th>
<th>0012</th>
<th>0013</th>
<th>0014</th>
<th>0015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>0000</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0004</td>
<td>0005</td>
<td>0006</td>
<td>0007</td>
<td>0008</td>
<td>0009</td>
<td>0010</td>
<td>0011</td>
<td>0012</td>
<td>0013</td>
<td>0014</td>
<td>0015</td>
</tr>
</tbody>
</table>

data

32-bit Words

<table>
<thead>
<tr>
<th>Addr</th>
<th>0000</th>
<th>0001</th>
<th>0002</th>
<th>0003</th>
<th>0004</th>
<th>0005</th>
<th>0006</th>
<th>0007</th>
<th>0008</th>
<th>0009</th>
<th>0010</th>
<th>0011</th>
<th>0012</th>
<th>0013</th>
<th>0014</th>
<th>0015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr</td>
<td>0000</td>
<td>0001</td>
<td>0002</td>
<td>0003</td>
<td>0004</td>
<td>0005</td>
<td>0006</td>
<td>0007</td>
<td>0008</td>
<td>0009</td>
<td>0010</td>
<td>0011</td>
<td>0012</td>
<td>0013</td>
<td>0014</td>
<td>0015</td>
</tr>
</tbody>
</table>

Machine Has “Word Size”

- Nominal size of integer-valued data
- Including addresses
- Most current machines are 32 bits (4 bytes)
- Becoming too small for memory-intensive applications
- High-end systems are 64 bits (8 bytes)
- Machines support multiple data formats
- Always integral number of bytes

Data Representations

Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Compaq Alpha</th>
<th>Typical 32-bit</th>
<th>Intel IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<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>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>8</td>
<td>8</td>
<td>10/12</td>
</tr>
<tr>
<td>char*</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Or any other pointer

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 0xa1234567
- Address given by `&x` is 0xa100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</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 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 00 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

```
typedef unsigned char *pointer;
void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++)
        printf("0x%p	0x%.2x\n", start+i, start[i]);
    printf("\n");
}
```

Printf directives:
- %p: Print pointer
- %x: Print Hexadecimal

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

Result (Linux):
```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcbb 0x00
0x11ffffcbb 0x00
```

Representing Integers
- cast pointer to signed char *

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

Decimal: 15213
Binary:
- Linux/Alpha: 0011 1011 0110 1101
- Sun: 0100 0110 0110 1101

Hex:
- Linux/Alpha: 3  B  6  D
- Sun: 4  6  3  B

Two's complement representation (Covered next lecture)

Representing Pointers
```
int B = -15213;
int *P = &B;
```

Alpha Address
- Hex: 1 F F F F C A 0
- Binary: 0001 1111 1111 1111 1111 1100 1010 0000

Sun Address
- Hex: 1 F F F F F B 2 C
- Binary: 1110 1111 1111 1111 1111 1010 0010 1100

Linux Address
- Hex: B F F F F F 8 D 4
- Binary: 1011 1111 1111 1111 1111 1011 1011 0100

Different compilers & machines assign different locations to objects

Representing Floats
```
float F = 15213.0;
```

```
IEEE Single Precision Floating Point Representation
```
- Hex: 4 e 6 d 4 f 0 0
- Binary: 0100 0110 0110 1111 1011 1010 0000 0000

15213:
- 1110 1111 1111 1011

Not same as integer representation, but consistent across machines
Can see some relation to integer representation, but not obvious
Representing Strings

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

- 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
  - Character "1" has code 0x31
  - Other characters unique
- 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 instruction is a 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

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

- 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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>A&amp;B</td>
</tr>
<tr>
<td>Or</td>
<td>A</td>
</tr>
<tr>
<td>Not</td>
<td>~A</td>
</tr>
<tr>
<td>Exclusive-Or (Xor)</td>
<td>A^B = 1 when either A=1 or B=1, but not both</td>
</tr>
</tbody>
</table>

Integer Algebra

Integer Arithmetic
- (Z, +, *, -, 0, 1) forms a “ring”
- Addition is the “sum” operation
- Multiplication is the “product” operation
- – is the additive inverse
- 0 is the identity for sum
- 1 is the identity for product
### Boolean Algebra

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

### Boolean Algebra ≠ Integer Ring

- **Boolean: Sum distributes over product**
  \[ A \land (B \lor C) = (A \land B) \lor (A \land C) \]
- **Boolean: Idempotency**
  - \( A \land A = A \)
  - \( A \lor A = A \)
- **Boolean: Absorption**
  - \( A \land (A \lor B) = A \land B \)
  - \( A \lor (A \land B) = A \lor B \)
- **Boolean: Laws of Complements**
  - \( A \land \neg A = 0 \land 1 \)
  - \( A \lor \neg A = 1 \lor 0 \)
- **Ring: Every element has additive inverse**
  - \( A \land \neg A = 0 \land 1 \)
  - \( A \lor \neg A = 1 \lor 0 \)

### Relations Between Operations

#### DeMorgan’s Laws
- Express \( \land \) in terms of \( \lor \), and vice-versa
  - \( A \land B = \neg \neg (A \land B) \)
  - \( A \land B = \neg \neg (A \land B) \)
- **Exclusive-Or using Inclusive Or**
  - \( A \lor B = \neg \neg (A \land B) \land \neg \neg (A \land B) \)
  - \( A \lor B = \neg \neg (A \land B) \land \neg \neg (A \land B) \)

### Boolean Algebra = Integer Ring

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

### General Boolean Algebras

**Operate on Bit Vectors**

- **Operations applied bitwise**
  - 01101001 01101001 01101001
  - 01101001 01101001 01101001
  - 01101001 01101001 01101001

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 \( j \in A \)
- Example:
  - \( 01101001 \) \( \{0, 3, 5, 6\} \)
- \( 01010101 \) \( \{0, 2, 4, 6\} \)

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

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)

- \( \sim 0x41 \rightarrow 0xBE \)
- \( \sim 0x00 \rightarrow 0xFF \)
- \( 0x69 \& 0x55 \rightarrow 0x41 \)
- \( 0x69 | 0x55 \rightarrow 0x7D \)

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

Cool Stuff with Xor

- Bitwise Xor is a form of addition
- With extra property that every value is its own additive inverse
- \( A^A = 0 \)
- Examples:
  - \( 0x69 \& 0x55 \rightarrow 0x01 \)
  - \( p \& *p \) (avoids null pointer access)

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