Bits and Bytes

Dr. Steve Goddard
goddard@cse.unl.edu

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 $1523_{10}$ as $1110110110110_2$
- Represent $1.20_{10}$ as $1.0011001100111[0011]..._2$
- 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 (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

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.
Encoding Byte Values

Byte = 8 bits
- Binary 00000000 to 11111111
- Decimal: 0 to 255
- Hexadecimal 00 to FF
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write FA1D37B in C as 0xFA1D37B
    - Or 0xFA1D37B
  - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'

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 $1.8 \times 10^{19}$ 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 “BigEndian” machines
  - Least significant byte has highest address
- Alphas, PC’s are “LittleEndian” machines
  - Least significant byte has lowest address

Big Endian Example

| Variable x has 4-byte representation 0x01234567 |
| Address given by &x is 0xa100 |
| Big Endian | Little Endian |
| 0x01 0x23 0x45 0x67 | 0x67 0x45 0x23 0x01 |

Little Endian Example

| Variable x has 4-byte representation 0x01234567 |
| Address given by &x is 0xa100 |
| Big Endian | Little Endian |
| 0x01 0x23 0x45 0x67 | 0x67 0x45 0x23 0x01 |

Page 2
**Reading Byte-Reversed Listings**

Disassembly
- Text representation of binary machine code
- Generated by program that reads the machine code

**Example Fragment**

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

**Deciphering Numbers**

- Value: 0x12ab
- Pad to 4 bytes: 0xa0000000
- Split into bytes: ab 12 00 00
- Reverse: ba 24

---

**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 \t 0 x % .2 x \ n " ,
                start + i , start [ i ] ) ;
    printf ( " \ n " ) ;
}
```

---

**Representing Integers**

- Integer A = 15213 ;
- Integer B = -15213 ;
- Long int C = 15213 ;

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>80483e5 : 5b</td>
<td>pop &amp;ebx</td>
<td></td>
</tr>
<tr>
<td>80483e6 : 81 3 b 12 00 00</td>
<td>add 0x12ab, &amp;ebx</td>
<td></td>
</tr>
<tr>
<td>80483e6c : 83 bb 28 00 00 00</td>
<td>cmp 0, 0x28 (&amp;ebx)</td>
<td></td>
</tr>
</tbody>
</table>

**Representing Pointers**

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

**Result (Linux):**

```c
int a = 15213 ;
0xf0000000 0xd6
0x00000000 0x00b
```

---

**Representing Floats**

Float F = 15213.0 ;

**IEEE Single Precision Floating Point Representation**

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

---

**Different compilers & machines assign different locations to objects**
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 0 has code 0x30
- 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)
  - 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;
}
```

- For this example, Alpha & Sun use two 4-byte instructions
  - Use different 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

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 \( B = 1 \)
- \( A & B = 0 \) when either \( A = 0 \) or \( B = 0 \)

Or
- \( A | B = 1 \) when either \( A = 1 \) or \( B = 1 \)
- \( A | B = 0 \) when both \( A = 0 \) and \( B = 0 \)

Not
- \( \sim A = 1 \) when \( A = 0 \)
- \( \sim A = 0 \) when \( A = 1 \)

Exclusive-Or (Xor)
- \( A ^ B = 1 \) when either \( A = 1 \) or \( B = 1 \)
  - but not both

Integer Algebra

Integer Arithmetic
- \( (\mathbb{Z}, +, \cdot, 0, 1) \) 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**
- \( \langle 0,1 \rangle, \lor, \land, \neg, 0, 1 \) 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*
  \[ A \lor (B \land C) = (A \lor B) \land (A \lor C) \]

- Boolean: *Identity*
  \[ A \lor A = A \]

- Boolean: *Absorption*
  \[ A \lor (A \land B) = A \]

- Boolean: *Laws of Complements*
  \[ A \land \neg A = 0 \]

- Boolean: *Ring: Every element has additive inverse*
  \[ A \land \neg A = 0 \]

**Relations Between Operations**

- **De Morgan’s Laws**
  - Express \& in terms of |, and vice-versa
    \[ A \land B = \neg (\neg A \lor \neg B) \]
    - A and B are true if and only if neither A nor B is false
  - \[ A \lor B = \neg (\neg A \land \neg B) \]
    - A or B are true if and only if A and B are not both false

- **Exclusive-Or using Inclusive Or**
  - \[ A \lor B = \neg (A \land B) \]
    - Exactly one of A and B is true
  - \[ A \lor B = (A \lor B) \land \neg (A \land B) \]
    - Either A is true, or B is true, but not both

**Shortened for Readability**

**Boolean Algebra = Integer Ring**

- **Commutativity**
  \[ A \lor B = B \lor A \]
  \[ A \land B = B \land A \]

- **Associativity**
  \[ (A \lor B) \lor C = A \lor (B \lor C) \]
  \[ (A \land B) \land C = A \land (B \land C) \]

- **Product distributes over sum**
  \[ A \land (B \lor C) = (A \land B) \lor (A \land C) \]

- **Sum and product identities**
  \[ A \lor 0 = A \]
  \[ A \land 1 = A \]

- **Zero is product annihilator**
  \[ A \land 0 = 0 \]

- **Cancellation of negation**
  \[ \neg (\neg A) = A \]

**General Boolean Algebras**

- **Operate on Bit Vectors**
  \[ 01101001 01101001 01101001 \]

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

**Representation**
- Width w bit vector represents subsets of \{0, ..., w-1\}
- \(a_j = 1 \iff j \in A\)
- \(01101001\) \(\{0, 2, 4, 6\}\)
- \(76543210\) \(\{0, 2, 4, 6\}\)

**Operations**
- & Intersection \(01000001\) \(\{0\}\)
- | Union \(01111101\) \(\{0, 2, 3, 4, 5\}\)
- ^ 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 data type)**
- \(~0x41 \rightarrow 0x5F\)
- \(~0x00 \rightarrow 0xFF\)
- \(0x69 \& 0x55 \rightarrow 0x41\)
- \(01101001, 01101010 \rightarrow 01000001\)
- \(0x69 \mid 0x55 \rightarrow 0x7D\)
- \(01101011, 01101011 \rightarrow 01111111\)

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
- Early termination

**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

Cool Stuff with Xor

**Bitwise Xor is a 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 */
}
```

**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