**JDEP 284H**
**Foundations of Computer Systems**

# Bits and Bytes

**Dr. Steve Goddard**
*goddard@cse.unl.edu*

*http://cse.unl.edu/~goddard/Courses/JDEP284*

## 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.

2

## Topics

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

3

## 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.

4

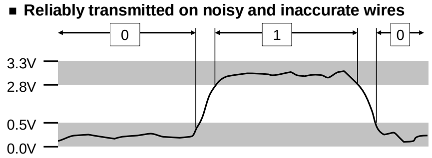
## Binary Representations

**Base 2 Number Representation**

- Represent $15213_{10}$ as $11101101101101_2$
- Represent $1.20_{10}$ as $1.0011001100110011[0011]\ldots_2$
- 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**

5

## 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

6

# 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`

| Hex | Decimal | Binary |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |



# Machine Words

**Machine Has "Word Size"**

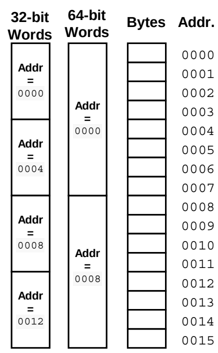
- Nominal size of integer-valued data
  - Including addresses
- Most current machines are 32 bits (4 bytes)
  - Limits addresses to 4GB
  - Becoming too small for memory-intensive applications
- High-end systems are 64 bits (8 bytes)
  - Potentially address ≈ $1.8 \times 10^{19}$ bytes
- Machines support 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 "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

| | | 0x100 | 0x101 | 0x102 | 0x103 | | |
|---|---|---|---|---|---|---|---|
| | | 01 | 23 | 45 | 67 | | |

Little Endian

| | | 0x100 | 0x101 | 0x102 | 0x103 | | |
|---|---|---|---|---|---|---|---|
| | | 67 | 45 | 23 | 01 | | |

## Reading Byte-Reversed Listings

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

### Example Fragment

| Address | Instruction Code | Assembly Rendition |
|---|---|---|
| 8048365: | 5b | pop %ebx |
| 8048366: | 81 c3 ab 12 00 00 | add $0x12ab,%ebx |
| 804836c: | 83 bb 28 00 00 00 00 | cmpl $0x0,0x28(%ebx) |

### Deciphering Numbers
- Value: 0x12ab
- Pad to 4 bytes: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

13

## 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\t0x%.2x\n",
           start+i, start[i]);
  printf("\n");
}
```

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

14

## `show_bytes` Execution Example

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

Result (Linux):

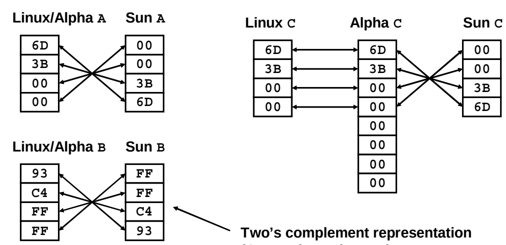
```
int a = 15213;
0x11ffffcb8   0x6d
0x11ffffcb9   0x3b
0x11ffffcba   0x00
0x11ffffcbb   0x00
```

15

## Representing Integers

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

| | |
|---|---|
| Decimal: | 15213 |
| Binary: | 0011 1011 0110 1101 |
| Hex: | 3 B 6 D |




Two's complement representation (Covered next lecture)

16

## Representing Pointers

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

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

Alpha P: A0 FC FF FF 01 00 00 00

Sun P: EF FF FB 2C

| Sun Address | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Hex: | E | F | F | F | F | B | 2 | C |
| Binary: | 1110 | 1111 | 1111 | 1111 | 1111 | 1011 | 0010 | 1100 |

Linux P: D4 F8 FF BF

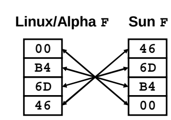
| Linux Address | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Hex: | B | F | F | F | F | 8 | D | 4 |
| Binary: | 1011 | 1111 | 1111 | 1111 | 1111 | 1000 | 1101 | 0100 |

*Different compilers & machines assign different locations to objects*

17

## Representing Floats

```
Float F = 15213.0;
```




| IEEE Single Precision Floating Point Representation | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Hex: | 4 | 6 | 6 | D | B | 4 | 0 | 0 |
| Binary: | 0100 | 0110 | 0110 | 1101 | 1011 | 0100 | 0000 | 0000 |
| 15213: | | | | 1110 | 1101 | 1011 | 01 | |

*Not same as integer representation, but consistent across machines*

*Can see some relation to integer representation, but not obvious*

18

# Representing Strings

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

## 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

| Linux/Alpha `S` | Sun `S` |
|---|---|
| 31 | 31 |
| 35 | 35 |
| 32 | 32 |
| 31 | 31 |
| 33 | 33 |
| 00 | 00 |

## 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

| Alpha `sum` | Sun `sum` | PC `sum` |
|---|---|---|
| 00 | 81 | 55 |
| 00 | C3 | 89 |
| 30 | E0 | E5 |
| 42 | 08 | 8B |
| 01 | 90 | 45 |
| 80 | 02 | 0C |
| FA | 00 | 03 |
| 6B | 09 | 45 |
| | | 08 |
| | | 89 |
| | | EC |
| | | 5D |
| | | C3 |

*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

| & | 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

- ~A = 1 when A=0

| ~ | |
|---|---|
| 0 | 1 |
| 1 | 0 |

## Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

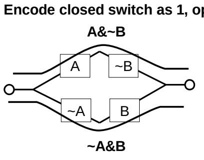
| ^ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |



# 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&~B | ~A&B

= A^B



# 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

- ⟨{0,1}, |, &, ~, 0, 1⟩ **forms a "Boolean algebra"**
- **Or is the "sum" operation**
- **And is the "product" operation**
- **~ is the "complement" operation (not additive inverse)**
- **0 is the identity for sum**
- **1 is the identity for product**



## Boolean Algebra ≈ Integer Ring

- *Commutativity*

| | |
|---|---|
| A \| B = B \| A | A + B = B + A |
| A & B = B & A | A * B = B * A |

- *Associativity*

| | |
|---|---|
| (A \| B) \| C = A \| (B \| C) | (A + B) + C = A + (B + C) |
| (A & B) & C = A & (B & C) | (A * B) * C = A * (B * C) |

- *Product distributes over sum*

| | |
|---|---|
| A & (B \| C) = (A & B) \| (A & C) | A * (B + C) = A * B + B * C |

- *Sum and product identities*

| | |
|---|---|
| A \| 0 = A | A + 0 = A |
| A & 1 = A | A * 1 = A |

- *Zero is product annihilator*

| | |
|---|---|
| A & 0 = 0 | A * 0 = 0 |

- *Cancellation of negation*

| | |
|---|---|
| ~ (~ A) = A | – (– A) = A |



## Boolean Algebra ≠ Integer Ring

- **Boolean:** *Sum distributes over product*

| | |
|---|---|
| A \| (B & C) = (A \| B) & (A \| C) | A + (B * C) ≠ (A + B) * (B + C) |

- **Boolean:** *Idempotency*

| | |
|---|---|
| A \| A = A | A + A ≠ A |

  - "A is true" or "A is true" = "A is true"

| | |
|---|---|
| A & A = A | A * A ≠ A |

- **Boolean:** *Absorption*

| | |
|---|---|
| A \| (A & B) = A | A + (A * B) ≠ A |

  - "A is true" or "A is true and B is true" = "A is true"

| | |
|---|---|
| A & (A \| B) = A | A * (A + B) ≠ A |

- **Boolean:** *Laws of Complements*

| | |
|---|---|
| A \| ~A = 1 | A + –A ≠ 1 |

  - "A is true" or "A is false"

- **Ring:** *Every element has additive inverse*

| | |
|---|---|
| A \| ~A ≠ 0 | A + –A = 0 |



## Properties of & and ^

### Boolean Ring

- ⟨{0,1}, ^, &, *I*, 0, 1⟩
- Identical to integers mod 2
- *I* is identity operation: *I*(A) = A
  - A ^ A = 0

| Property | Boolean Ring |
|---|---|
| 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 |



## Relations Between Operations

### DeMorgan's Laws

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

### Exclusive-Or using Inclusive Or

- **A ^ B = (~A & B) | (A & ~B)**
  - » Exactly one of A and B is true
- **A ^ B = (A | B) & ~(A & B)**
  - » Either A is true, or B is true, but not both



## General Boolean Algebras

### Operate on Bit Vectors

- **Operations applied bitwise**

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

### All of the Properties of Boolean Algebra Apply

## Representing & Manipulating Sets

### Representation

- Width *w* bit vector represents subsets of {0, …, *w*–1}
- $a_j = 1$ **if** $j \in A$

```
01101001        { 0, 3, 5, 6 }
76543210

01010101        { 0, 2, 4, 6 }
76543210
```

### 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 }

31

## 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`
  - $\sim01000001_2$ --> $10111110_2$
- `~0x00 --> 0xFF`
  - $\sim00000000_2$ --> $11111111_2$
- `0x69 & 0x55 --> 0x41`
  - $01101001_2$ & $01010101_2$ --> $01000001_2$
- `0x69 | 0x55 --> 0x7D`
  - $01101001_2$ | $01010101_2$ --> $01111101_2$

32

## 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`
- `!!0x41 --> 0x01`
- `0x69 && 0x55 --> 0x01`
- `0x69 || 0x55 --> 0x01`
- `p && *p` (avoids null pointer access)

33

## 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

| | |
|---|---|
| Argument `x` | `01100010` |
| `<< 3` | `00010000` |
| Log. `>> 2` | `00011000` |
| Arith. `>> 2` | `00011000` |

| | |
|---|---|
| Argument `x` | `10100010` |
| `<< 3` | `00010000` |
| Log. `>> 2` | `00101000` |
| Arith. `>> 2` | `11101000` |

34

## 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

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

| | `*x` | `*y` |
|---|---|---|
| Begin | `A` | `B` |
| 1 | `A^B` | `B` |
| 2 | `A^B` | `(A^B)^B = A` |
| 3 | `(A^B)^A = B` | `A` |
| End | `B` | `A` |

35

## 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

36