**CS 2630**
**Computer Organization**

# Bits and Bytes

Steve Goddard
*steve-goddard@uiowa.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.

# Today: Bits and Bytes

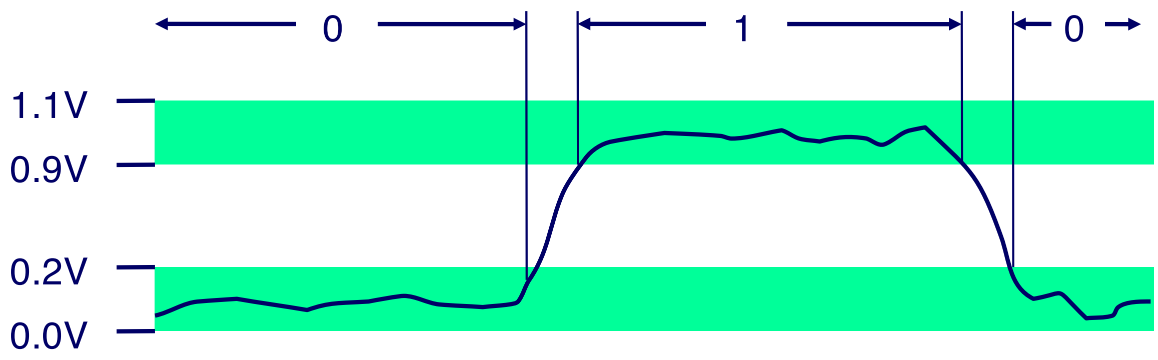
- **Representing information as bits**
- Bit-level manipulations

# 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.

# Everything is bits

- **Each bit is 0 or 1**
- **By encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Why bits?  Electronic Implementation**
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



# For example, can count in binary

- **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 $1.1101101101101_2 \times 2^{13}$

# 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
      - 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 |

# Example Data Representations

| C Data Type | Typical 32-bit | Typical 64-bit | x86-64 |
|---|---|---|---|
| **`char`** | 1 | 1 | 1 |
| **`short`** | 2 | 2 | 2 |
| **`int`** | 4 | 4 | 4 |
| **`long`** | 4 | 8 | 8 |
| **`float`** | 4 | 4 | 4 |
| **`double`** | 8 | 8 | 8 |
| **`long double`** | − | − | 10/16 |
| pointer | 4 | 8 | 8 |

# Today: Bits and Bytes

- Representing information as bits
- **Bit-level manipulations**

# 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 |

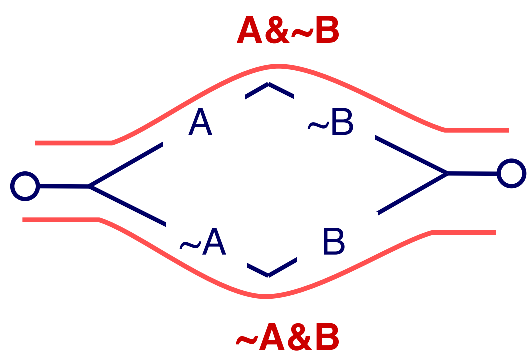
Exclusive-Or (Xor)

- 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**

# 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**

# Example: 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 } |

# Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - `long, int, short, char, unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - `~0x41 → 0xBE`
    - $\sim 01000001_2 \rightarrow 10111110_2$
  - `~0x00 → 0xFF`
    - $\sim 00000000_2 \rightarrow 11111111_2$
  - `0x69 & 0x55 → 0x41`
    - $01101001_2 \ \& \ 01010101_2 \rightarrow 01000001_2$
  - `0x69 | 0x55 → 0x7D`
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

# 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)

# Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or
    - Early termination
- **Examples (char data**
  - `!0x41 → 0x00`
  - `!0x00 → 0x01`
  - `!!0x41 → 0x01`
  - `0x69 && 0x55 → 0x01`
  - `0x69 || 0x55 → 0x01`
  - `p && *p` (avoids null pointer access)

**Watch out for && vs. & (and || vs. |)… one of the more common oopsies in C programming**

# 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
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

| Argument **x** | 01100010 |
|---|---|
| << 3 | 00010*000* |
| Log. >> 2 | *00*011000 |
| Arith. >> 2 | *00*011000 |

| Argument **x** | 10100010 |
|---|---|
| << 3 | 00010*000* |
| Log. >> 2 | *00*101000 |
| Arith. >> 2 | *11*101000 |

# 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 |