# Introduction, Bits

September 3, 2018



## Computer Systems

Study low-level properties, but *programmer-centric*

We do not
- Design processors
- Implement operating systems
- Write compilers
- Simulate networks

Instead we
- Look at consequences *for the programmer* of existing designs
- See how processors, operating systems, compilers, and networks work together





## Prerequisites and Assumptions

- Proficiency with:
  - Representing numbers in different bases
  - Writing reasonably complex programs in Java/C/C++
  - Data structures such as: linked lists, arrays, stacks, trees
  - Debugging
- Experience with:
  - Terminal window and command line
  - Learning new languages and applications
  - Experimenting and being confused
  - Searching for and reading documentation





## The Course in a Nutshell

- Textbooks
  - Required:
    - Bryant and O'Halloran, *Computer Systems: A Programmer's Perspective*, **third edition,** Pearson, 2016 *or* electronic equivalent
    - Avoid paperback editions and pdf's on the web!
  - Optional: some reference for the C language
    - Kernighan and Ritchie, *The C Programming Language,* second edition, Prentice Hall, 1988
    - Miller and Quilici, *The Joy of C,* third edition, Wiley, 1997
    - Be cautious about web resources!
- Classes
  - Come prepared—do the reading first!





## Nutshell, continued

- Participation
  - 5% of the grade
- Labs
  - Tremendous fun, work in pairs
  - 40% of the grade
  - Start tomorrow! Be sure to have an accounts and passwords
- Midterm exams
  - October 3 and November 7
  - 15% of the grade each
- Final exam
  - Tuesday, December 17, 2:00—5:00 pm
  - 25% of the grade





## Resources

- `http://www.cs.pomona.edu/classes/cs105`
- Links from the course page:
  - Piazza, for questions and discussion
  - Lab assistants and mentors, schedule
  - Submission site
- Sakai, for recording lab grades only

## (Real) Computer Systems

- Labs use two different systems, in combination
- Two userids and two passwords
  - Pomona College Computer Science system
  - CAS ID (ITS, email)
- Be ready tomorrow with both passwords!





## Numbers Every Programmer Should Know

| | |
|---|---|
| execute typical instruction | 1/1,000,000,000 sec = 1 nanosec |
| fetch from L1 cache memory | 0.5 nanosec |
| branch misprediction | 5 nanosec |
| fetch from L2 cache memory | 7 nanosec |
| Mutex lock/unlock | 25 nanosec |
| fetch from main memory | 100 nanosec |
| send 2K bytes over 1Gbps network | 20,000 nanosec |
| read 1MB sequentially from memory | 250,000 nanosec |
| fetch from new disk location (seek) | 8,000,000 nanosec |
| read 1MB sequentially from disk | 20,000,000 nanosec |
| send packet US to Europe and back | 150 milliseconds = 150,000,000 nanosec |

These will make more sense later.

From Peter Norvig, *Teach Yourself Programming in Ten Years*. Made even more famous by Google's Jeff Dean.





## Five Great Realities

from the textbook

1. `int`s are not integers; floats are not real numbers
2. You must know assembly
3. Memory matters
4. There is more to performance than asymptotic complexity
5. Computers do more than execute programs

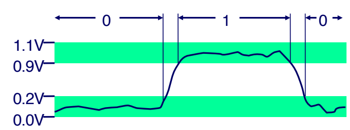
We try to balance theory and practice!





## Bits

- 0 or 1




- Sequences of bits represent
  - numbers
  - text
  - images
  - sound
  - instructions
  - .....





## Bits and Bytes

- Bits = 0 or 1
- 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 |
| `long long` | 8 | 8 | 8 |
| `float` | 4 | 4 | 4 |
| `double` | 8 | 8 | 8 |
| pointer | 4 | 8 | 8 |

Sizes in bytes

## Bits and Bytes Require Interpretation

0x00353031, or 00000000 00110101 00110000 00110001, might be interpreted as

- The integer $3{,}485{,}745_{10}$
- A floating point number close to $4.884569 \times 10^{-39}$
- The string "105"
- A portion of an image or video
- An address in memory





## Memory: An Array of Bytes

- An index into the array is an *address*, *location*, or *pointer*
  - Often expressed in hexadecimal
- We speak of the *value* in memory at an address
  - The value may be a single byte …
  - … or a multi-byte quantity starting at that address
- Assignment: **`x = y + 42;`**
  - Take the value from the location reserved for **`y`**
  - Add 42 to it
  - Place the result in the location reserved for **`x`**





## Pointers as Addresses

- A pointer is an address in memory

```
int x;     // an integer
int *p     // a pointer to an integer
```

- 4 bytes are reserved for **`x`**
- 8 bytes are reserved for **`p`**
  - Those 8 bytes are interpreted as an address of an integer (somewhere else) in memory





## Memory Access in C

```
int x;     // an integer
int *p     // a pointer to an integer

// normal initialization:
x = 0;

// silly, but illustrative:
p = &x;    // & means "address of"
*p = 0;    // * means "memory at address"
```

- & and * are inverses of one another
- prefix vs infix operators
- **`x`** occupies 4 bytes in memory; **`p`** occupies 8





## Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic---encode "True" as 1 and "False" as 0

And

| & | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Or

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Not

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

Exclusive-Or (Xor)

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





## General Boolean algebras

- Bitwise operations on words

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

- How does this map to set operations?

## Bitwise vs Logical Operations in C

- Apply to any "integral" data type
  - `long, int, short, char, unsigned`
- Bitwise Operators &, |, ~, ^
  - View arguments as bit vectors
  - operations applied bit-wise in parallel
- Logical Operators &&, ||, !
  - View 0 as "False"
  - View anything nonzero as "True"
  - Always return 0 or 1
  - Early termination





## Bitwise vs Logical Operations in C

- Exercises (char data type, one byte)
  - `~0x41`
  - `~0x00`
  - `0x69 & 0x55`
  - `0x69 | 0x55`
  - `!0x41`
  - `!0x00`
  - `!!0x41`
  - `0x69 && 0x55`
  - `0x69 || 0x55`

| 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 |
| pointer | 4 | 8 | 8 |





## Powers of Two

- $2^6 = 64$
- $2^8 = 256$

Memorize these!

- $2^{10} = 1024 \approx 1000$, K or kilo
- $2^{20} \approx 10^6$, M or mega
- $2^{30} \approx 10^9$, G or giga
- $2^{40} \approx 10^{12}$, tera
- $2^{50} \approx 10^{15}$, peta





## Representations

- 15213 as a 32-bit, 4-byte number
- Decimal: $15213_{10}$
- Binary: $15213_{10}$ = 0011 1011 0110 $1101_2$
- Hexadecimal: $15213_{10}$ = **0x3B6D**





## Representing Numbers

- Practice:
  - what is $10547_{10}$ in binary?
  - what is $8.75_{10}$ in binary?

| 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 |
| pointer | 4 | 8 | 8 |





## Representing Unsigned Integers

- Think of bits as the binary representation

$$\text{UnsignedValue}(x) = \sum_{j=0}^{w-1} x_j \cdot 2^j$$

- Can only represent non-negative numbers
- If you have w bits, what is the range?

## Representing Signed Numbers

- Option 1: sign-magnitude
  - One bit for sign; interpret rest as magnitude
- Option 2: one's complement
  - Flip every bit to get the negation
- Option 3: excess-K
  - Choose a positive K in the middle of the unsigned range
  - SignedValue(w) = UnsignedValue(w) – K
  - Used in floating point representations
- Difficulties?





## Representing Signed Integers

- Option 4: two's complement
  - Most commonly used
  - Like unsigned, except the high-order contribution is *negative*

$$\text{SignedValue}(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{j=0}^{w-2} x_j \cdot 2^j$$

- Assume C short (2 bytes)
  - What is the hex/binary representation for 47?
  - What is the hex/binary representation for -47?





## Two's Complement Signed Integers

- "Signed" does not mean "negative"
- High order bit is the *sign bit*
  - To negate, complement all the bits and add 1
  - Remember the arithmetic right shift
  - Sign extension
- Arithmetic is the same as unsigned—same circuitry
- Error conditions and comparisons are different





## Fun with Integers: Using of Bitwise Operations

- x & 1 "x is odd"
- (x + 7) & 0xFFFFFFF8 "round up to a multiple of 8"
- p & ~0x3FF "start of 1K block containing p" (almost)
- ((p >> 10) << 10) same location (really)
- p & 0x3FF "offset of p within the block"





## Fun with Integers: Shift Operations in C

- Left Shift: **x << y**
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
    - Fill with 0's on right
- Multiply integer by a power of 2
- 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
- Divide by a power of 2
  - Rounds toward $-\infty$
- Choice between logical and arithmetic depends on the type of data
- Undefined Behavior
  - Shift amount < 0 or ≥ word size





## Tomorrow: Data Lab Puzzles

```
/*
 * upperBits - pads n upper bits with 1's
 * You may assume 0 <= n <= 32
 * Example: upperBits(4) = 0xF0000000
 * Legal ops: ! ~ & ^ | + << >>
 * Max ops: 10
 * Rating: 1
 */
int upperBits(int n) {

  return ??? ;

}
```

Typical example. Find an expression to replace ???.

# Things to Do Right Away

- For lab tomorrow
  - Be sure you have an accounts and passwords on both the Pomona CS system and the CAS ID from ITS
- For class on Thursday
  - Begin the reading: Chapters 1 and 2
- This week
  - Accept the invitation to our course's Piazza site
  - Enroll in CS 105 on **`submit.cs.pomona.edu`**