Data III & Integers I
CSE 351 Winter 2017

http://xkcd.com/257/
Administrivia

- Thanks for all feedback and bug reports 😊
  - Keep using the anonymous feedback
- My office hours now Mon 11-noon.
- Lab1 --- fun!
Examining Data Representations

- Code to print byte representation of data

```c
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

`printf` directives:
- `%p` Print pointer
- `\t` Tab
- `%x` Print value as hex
- `\n` New line
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to `char`
  - C has unchecked casts  !! DANGER !!

```c
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}

void show_int(int x) {
    show_bytes((char*) &x, sizeof(int));
}
```
show_bytes Execution Example

```c
int a = 12345; // 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((char *) &a, sizeof(int));
```

- **Result (Linux x86-64):**
  - **Note:** The addresses will change on each run (try it!), but fall in same general range

```c
int a = 12345;
0x7fffffff71dbc 0x39
0x7fffffff71dbd 0x30
0x7fffffff71dbe 0x00
0x7fffffff71dbf 0x00
```
Memory, Data, and Addressing

- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic (True → 1, False → 0)
  - AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1
  - OR: \( A | B = 1 \) when either \( A \) is 1 or \( B \) is 1
  - XOR: \( A \oplus B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both
  - NOT: \( \sim A = 1 \) when \( A \) is 0 and vice-versa
  - DeMorgan’s Law: \( \sim (A | B) = \sim A \& \sim B \)
    \( \sim (A \& B) = \sim A | \sim B \)

<table>
<thead>
<tr>
<th></th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise
  - All of the properties of Boolean algebra apply

\[\begin{align*}
01101001 \& 01010101 &= 01010101 \\
01101001 \mid 01010101 &= ^01010101 \\
01101001 \^ 01010101 &= ~01010101
\end{align*}\]

- Examples of useful operations:

\[\begin{align*}
x \^ x &= 0 & 01010101 \^ 01010101 &= 01010101 \\
x \mid 1 &= 1 & ^01010101 \mid 11110000 &= 11110000
\end{align*}\]

- How does this relate to set operations?
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of $\{0, \ldots, w-1\}$
  - $a_j = 1$ iff $j \in A$
  - 01101001 $\{0, 3, 5, 6\}$
  - 01010101 $\{0, 2, 4, 6\}$

- **Operations**
  - $\&$ Intersection $\{0, 6\}$
  - $\mid$ Union $\{0, 2, 3, 4, 5, 6\}$
  - $^\wedge$ Symmetric difference $\{2, 3, 4, 5\}$
  - $\sim$ Complement $\{1, 3, 5, 7\}$
Bit-Level Operations in C

- & (AND), | (OR), ^ (XOR), ~ (NOT)
  - View arguments as bit vectors, apply operations bitwise
  - Apply to any “integral” data type
    - long, int, short, char, unsigned

Examples with char a, b, c;

- a = (char) 0x41;  // 0x41->0b 0100 0001
  b = ~a;           // 0b 1011 1110->0xBE
- a = (char) 0x69;  // 0x69->0b 0110 1001
  b = (char) 0x55;  // 0x55->0b 0101 0101
  c = a & b;        // 0b 0100 0001->0x41
- a = (char) 0x41;  // 0x41->0b 0100 0001
  b = a;           // 0b 0100 0001
  c = a ^ b;       // 0b 0000 0000->0x00
Contrast: Logic Operations

- Logical operators in C: `&&` (AND), `||` (OR), `!` (NOT)
  - 0 is False, anything nonzero is True
  - Always return 0 or 1
  - Early termination (a.k.a. short-circuit evaluation) of `&&`, `||`

- Examples (char data type)
  - `!0x41` -> `0x00`
  - `!0x00` -> `0x01`
  - `!!0x41` -> `0x01`
  - `p && *p++`
    - Avoids null pointer (0x0) access via early termination
    - Short for: `if (p) { *p++; }`
  - `0xCC && 0x33` -> `0x01`
  - `0x00 || 0x33` -> `0x01`
Roadmap

C:

car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:

Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:

get_mpg:
    pushq %rbp
    movq %rsp, %rbp
    ...
    popq %rbp
    ret

Machine code:

0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
 Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Memory & data
Integers & floats
Machine code & C
x86 assembly
 Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
But before we get to integers....

- Encode a standard deck of playing cards
- 52 cards in 4 suits
  - How do we encode suits, face cards?
- What operations do we want to make easy to implement?
  - Which is the higher value card?
  - Are they the same suit?
Two possible representations

1) 1 bit per card (52): bit corresponding to card set to 1

- “One-hot” encoding (similar to set notation)
- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required

2) 1 bit per suit (4), 1 bit per number (13): 2 bits set

- Pair of one-hot encoded values
- Easier to compare suits and values, but still lots of bits used

✓ Can we do better?
Two better representations

3) Binary encoding of all 52 cards – only 6 bits needed
   - $2^6 = 64 \geq 52$
   - Fits in one byte (smaller than one-hot encodings)
   - How can we make value and suit comparisons easier?

4) Separate binary encodings of suit (2 bits) and value (4 bits)
   - Also fits in one byte, and easy to do comparisons

<table>
<thead>
<tr>
<th>suit</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>♣ 00</td>
<td></td>
</tr>
<tr>
<td>♦ 01</td>
<td></td>
</tr>
<tr>
<td>♥ 10</td>
<td></td>
</tr>
<tr>
<td>♠ 11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>K</th>
<th>Q</th>
<th>J</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>A</th>
</tr>
</thead>
</table>
| 1101| 1100| 1011| ... | 0011| 0010| 0001
Compare Card **Suits**

char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare

```c
    card1 = hand[0];
    card2 = hand[1];
    ...
    if ( sameSuitP(card1, card2) ) { ... }
```

**#define SUIT_MASK 0x30**

```c
    int sameSuitP(char card1, char card2) {
        return (!((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK)));
        //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
    }
```

**returns int**

SUIT_MASK = 0x30 = \[00110000]

**mask:** a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector \(v\).

Here we turns all *but* the bits of interest in \(v\) to 0.
**Compare Card Suits**

A mask is a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector \( v \). Here we turn all but the bits of interest in \( v \) to 0.

```c
#define SUIT_MASK 0x30

int sameSuitP(char card1, char card2) {
    return !((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    //return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

![Binary representation of card suits comparison](image)

\( (x \land y) \) equivalent to \( x == y \)
**Compare Card Values**

```c
#define VALUE_MASK 0x0F

int greaterValue(char card1, char card2) {
    return ((unsigned int)(card1 & VALUE_MASK) >
             (unsigned int)(card2 & VALUE_MASK));
}
```

`VALUE_MASK = 0x0F = \text{00001111}_{2}`

**mask**: A bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector \(v\).
#define VALUE_MASK 0x0F

int greaterValue(char card1, char card2) {
    return ((unsigned int)(card1 & VALUE_MASK) > (unsigned int)(card2 & VALUE_MASK));
}

Compare Card **Values**

**mask:** a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector \( v \).
Integers

- Representation of integers: unsigned and signed
- Integers in C and casting
- Sign extension
- Shifting and arithmetic
Encoding Integers

- The hardware (and C) supports two flavors of integers
  - *unsigned* – only the non-negatives
  - *signed* – both negatives and non-negatives

- Cannot represent all integers with \( w \) bits
  - Only \( 2^w \) distinct bit patterns
  - Unsigned values: \( 0 \ldots 2^w - 1 \)
  - Signed values: \( -2^{w-1} \ldots 2^{w-1} - 1 \)

- **Example:** 8-bit integers (i.e., *char*)
Unsigned Integers

- Unsigned values follow the standard base 2 system
  \[ b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + \cdots + b_12^1 + b_02^0 \]

- Add and subtract using the normal “carry” and “borrow” rules, just in binary

\[
\begin{array}{c}
63 \\
+ 8 \\
\hline
00111111 \\
+00001000 \\
\hline
000010001
\end{array}
\]

- Useful formula: \( 2^{N-1} + 2^{N-2} + 4 + 2 + 1 = 2^N - 1 \)
  - i.e., N 1’s in a row = \( 2^N - 1 \)

- How would you make *signed* integers?
Sign and Magnitude

- Designate the high-order bit (MSB) as the “sign bit”
  - \( \text{sign} = 0 \): positive numbers; \( \text{sign} = 1 \): negative numbers

- Positives:
  - Using MSB as sign bit matches positive numbers with unsigned
  - All zeros encoding is still \( = 0 \)

- Examples (8 bits):
  - \( 0x00 = 00000000_2 \) is non-negative, because the sign bit is 0
  - \( 0x7F = 01111111_2 \) is non-negative \( (+127_{10}) \)
  - \( 0x85 = 10000101_2 \) is negative \( (-5_{10}) \)
  - \( 0x80 = 10000000_2 \) is negative... zero???
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- **Drawbacks:**
  - Two representations of 0 (bad for checking equality)
Sign and Magnitude

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks:
  - Two representations of 0 (bad for checking equality)
  - Arithmetic is cumbersome
    - Example: $4 - 3 \neq 4 + (-3)$
    - Negatives “increment” in wrong direction!
Two’s Complement

- Let’s fix these problems:
  1) “Flip” negative encodings so incrementing works
Two’s Complement

- Let’s fix these problems:
  1) “Flip” negative encodings so incrementing works
  2) “Shift” negative numbers to eliminate −0

- MSB still indicates sign!
Two’s Complement Negatives

- Accomplished with one neat mathematical trick!

- $b_{w-1}$ has weight $-2^{w-1}$, other bits have usual weights $+2^i$

- 4-bit Examples:
  - $1010_2$ unsigned:
    \[1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10\]
  - $1010_2$ two’s complement:
    \[-1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -6\]

- -1 represented as:
  \[1111_2 = -2^3 + (2^3 - 1)\]
  - MSB makes it super negative, add up all the other bits to get back up to -1
Why Two’s Complement is So Great

- Roughly same number of (+) and (–) numbers
- Positive number encodings match unsigned
- Single zero
- All zeros encoding = 0

Simple negation procedure:
- Get negative representation of any integer by taking bitwise complement and then adding one!
  \(( \sim x + 1 == -x \) )
Summary

- Bit-level operators allow for fine-grained manipulations of data
  - Bitwise AND (\&), OR (\|), and NOT (~) different than logical AND (&&), OR (||), and NOT (!)
  - Especially useful with bit masks

- Choice of *encoding scheme* is important
  - Tradeoffs based on size requirements and desired operations

- Integers represented using unsigned and two’s complement representations
  - Limited by fixed bit width
  - We’ll examine arithmetic operations next lecture