Data III & Integers I
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/257/
Administrivia

- Lab 1 has been released
  - Based on material from today’s and Friday’s lecture
- Section 2 tomorrow
  - New rooms for AC/AG (EEB 003) and BC (GUG 218)
  - Good preparation for Lab 1
- Make good use of office hours!
  - Can go to *any* office hours for *any* TA
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: \( A ^ B = 1 \) 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:
    \[
    \sim (A \lor B) = \sim A \land \sim B \\
    \sim (A \land B) = \sim A \lor \sim B
    \]

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

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

$$\begin{array}{c}
01101001 \\
& 01010101 \\
\hline
01010000
\end{array}$$

$$\begin{array}{c}
01101001 \\
| 01010101 \\
\hline
01111101
\end{array}$$

$$\begin{array}{c}
01101001 \\
^ 01010101 \\
\hline
00111100
\end{array}$$

$$\begin{array}{c}
\sim 01010101 \\
| 11110100 \\
\hline
11111010
\end{array}$$

- Examples of useful operations:
  $$x \ ^x = 0$$
  $$x \ | 1 = 1$$

- 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 \( j \in A \)
  
  \[
  \begin{align*}
  01101001 & \quad \{0, 3, 5, 6\} \\
  76543210 & \\
  01010101 & \quad \{0, 2, 4, 6\} \\
  76543210
  \end{align*}
  \]

❖ Operations
  ▪ & Interesction  \quad 01000001 \quad \{0, 6\}
  ▪ | Union  \quad 01111101 \quad \{0, 2, 3, 4, 5, 6\}
  ▪ ^ Symmetric difference  \quad 00111100 \quad \{2, 3, 4, 5\}
  ▪ ~ Complement  \quad 10101010 \quad \{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:
```c
#define car

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

Java:
```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:
```
011101010000011000
100011010000010000000010
1000100111000010
1100000111111010000011111
```

OS:
- Windows 8
- MacOS

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>♠</td>
<td>00</td>
</tr>
<tr>
<td>♦</td>
<td>01</td>
</tr>
<tr>
<td>♥</td>
<td>10</td>
</tr>
<tr>
<td>♣</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>K</th>
<th>Q</th>
<th>J</th>
<th>. . .</th>
<th>3</th>
<th>2</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>♠</td>
<td>1101</td>
<td>1100</td>
<td>1011</td>
<td>. . .</td>
<td>0011</td>
<td>0010</td>
<td>0001</td>
</tr>
<tr>
<td>♦</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>♥</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>♣</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compare Card Suits

char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( sameSuitP(card1, card2) ) { ... }

#define SUIT_MASK  0x30

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

returns int SUIT_MASK = 0x30 = 0011100000

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**

- **mask**: 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);
}
```

### Example
- Card1: 00010010
- Card2: 00011101
- SUIT_MASK: 00110000

```
\begin{align*}
00010010 & \& 00110000 = 00010000 \\
00011101 & \& 00110000 = 00010000 \\
\end{align*}
```

- Result: 00010000

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

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

card1 = hand[0];
card2 = hand[1];
...

if ( greaterValue(card1, card2) ) { ... }

#define VALUE_MASK 0x0F

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

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

VALUE_MASK = 0x0F = 0000011111

---

**suit** **value**
#define VALUE_MASK  0x0F

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

mask: a bit vector designed to achieve a desired behavior when used with a bitwise operator on another bit vector $v$. 

Compare Card Values

```
0 0 1 0 0 0 1 0
&
0 0 0 0 1 1 1 1
=
0 0 0 0 0 0 1 0

0 0 1 0 1 1 0 1
&
0 0 0 0 1 1 1 1
=
0 0 0 0 0 1 1 0 1
```

$2_{10} > 13_{10}$

0 (false)
Integers

- Representation of integers: unsigned and signed
- Casting
- Arithmetic and shifting
- Sign extension
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 $2^w$)
  - 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

$$
\begin{array}{c}
63 \\
+ 8 \\
\hline
71
\end{array} \quad
\begin{array}{c}
00111111 \\
+00001000 \\
\hline
01000111
\end{array}
$$

- 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: $4 - 3 \neq 4 + (-3)$

<table>
<thead>
<tr>
<th>4</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>-0011</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
</tbody>
</table>

\[ \begin{array}{c|c|c}
4 & 0 & 0100 \\
-3 & 1 & 0011 \\
\hline
+3 & 4 & 1011 \\
\hline
-7 & 1111 \\
\end{array} \]

- 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!
  - This is why we represent one more negative than positive number (-2^{N-1} to 2^{N-1}+1)
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:
    \[1*2^3+0*2^2+1*2^1+0*2^0 = 10\]
  - $1010_2$ two’s complement:
    \[-1*2^3+0*2^2+1*2^1+0*2^0 = -6\]

- -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!
\[ \sim x + 1 = -x \]
Peer Instruction Question

- Take the 4-bit number encoding \( x = 0b1011 \)
- Which of the following numbers is NOT a valid interpretation of \( x \) using any of the number representation schemes discussed today?
  - Unsigned, Sign and Magnitude, Two’s Complement
  - Vote at http://PollEv.com/justinh

(A) -4
(B) -5
(C) 11
(D) -3
Summary

- Bit-level operators allow for fine-grained manipulations of data
  - Bitwise AND (\&), OR (\mid), and NOT (\sim) different than logical AND (\&\&), OR (\mid\mid), 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