Integers I
CSE 351 Summer 2020

Instructor:
Porter Jones

Teaching Assistants:
Amy Xu
Callum Walker
Sam Wolfson
Tim Mandzyuk

http://xkcd.com/257/
Administrivia

- No lecture on Friday 7/3 (campus holiday)
- hw3 due Wednesday 7/1 – 10:30am
- hw4 due Monday 7/6 – 10:30am
  - As a heads up, hw5 released 7/1, also due 7/6
- Lab 1a released
  - Workflow:
    1) Edit `pointer.c`
    2) Run the Makefile (`make`) and check for compiler errors & warnings
    3) Run `ptest` (`./ptest`) and check for correct behavior
    4) Run rule/syntax checker (`python dlc.py`) and check output
  - Due Monday 7/6 at 11:59pm **recommended to finish by 7/3**
    (to give time to complete lab 1b).
    - Lab 1b will be released later this week, due 7/10
Lab Reflections

- All subsequent labs (after Lab 0) have a “reflection” portion
  - The Reflection questions can be found on the lab specs and are intended to be done after you finish the lab
  - You will type up your responses in a .txt file for submission on Gradescope
  - These will be graded “by hand” (read by TAs)

- Intended to check your understand of what you should have learned from the lab
Poll Everywhere For Credit

- Poll everywhere counts for credit starting today!
  - Remember that you get credit for any answer, not just correct answers.
  - Make sure you enter a poll response before I close the poll.

- Makeup quizzes released after every lecture on Canvas.
  - Only submit this if you did not answer in lecture.
  - Must provide explanation for your answer to receive full credit.
  - Due before the next lecture at 10:30am.
Memory, Data, and Addressing

- Representing information as bits and bytes
  - Binary, hexadecimal, fixed-widths
- Organizing and addressing data in memory
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
  - Assignment
  - Pointers, pointer arithmetic, and arrays

- 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: \( \sim (A | B) = \sim A \& \sim B \), \( \sim (A \& B) = \sim A | \sim B \)

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

Two values like binary
General Boolean Algebras

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

\[
\begin{array}{c c c c}
01101001 & 01101001 & 01101001 \\
\& 01010101 & 01010101 & ^ 01010101 \\
\hline
01000000 & 01111110 & \sim 01010101 \\
\end{array}
\]

- Examples of useful operations:

\[
x \wedge x = 0
\]

"sets to 1"

\[
x \mid 1 = 1,
0 \mid 1 = 1
1 \mid 1 = 1
\]

"leaves as is"

\[
x \mid 0 = x
0 \mid 0 = 0
1 \mid 0 = 1
\]
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 1111 1110->0x3E
- 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`  
  - If `p` is the null pointer (0x0), then `p` is never dereferenced!
- `!0x00` -> `0x01`
- `!!0x41` -> `0x01`

Examples of logical operators:

- `0xCC && 0x33` -> `0x01`
- `0x00 || 0x33` -> `0x01`
- `0xCC && 0x33` -> `0x01`
Roadmap

C:

```c
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:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

- Windows 10
- OS X Yosemite

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
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

\[
52 \text{ bits} \frac{}{\text{fits in}} 7 \text{ bytes (56 bits)}
\]

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

\[
4 \text{ suits} \quad 13 \text{ numbers}
\]

\[
17 \text{ bits} \rightarrow 3 \text{ bytes}
\]
Two better representations

3) Binary encoding of all 52 cards – only 6 bits needed

- \(2^6 = 64 \geq 52\)
- \(2^5 = 32 \geq 52\)
- 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)

- \(2^4 = 16\)
- Also fits in one byte, and easy to do comparisons

<table>
<thead>
<tr>
<th>Suit</th>
<th>♣</th>
<th>♦</th>
<th>♥</th>
<th>♠</th>
</tr>
</thead>
<tbody>
<tr>
<td>♣</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>♦</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>♥</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>♠</td>
<td>11</td>
<td></td>
<td></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>
<tbody>
<tr>
<td>1101</td>
<td>1100</td>
<td>1011</td>
<td>. . .</td>
<td>0011</td>
<td>0010</td>
<td>0001</td>
</tr>
</tbody>
</table>
Compare Card Suits

```c
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)));
    // return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

**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.

SUIT_MASK = 0x30 = \[ \begin{array}{cccccccc}
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array} \]

- **suit** (keep)
- **value** (discard)
Compare Card Suits

```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);
}
```

**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.

! (x^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 = 0000011111

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

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

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, 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 $2^w$ distinct bit patterns
  - Unsigned values: $0 \ldots 2^w - 1$
  - Signed values: $-2^{w-1} \ldots 2^{w-1} - 1$

- **Example**: 8-bit integers (*e.g.* `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

<table>
<thead>
<tr>
<th>63</th>
<th>00111111</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 8</td>
<td>+00001000</td>
</tr>
<tr>
<td>71</td>
<td>01000111</td>
</tr>
</tbody>
</table>

- Useful formula: \[ 2^{N-1} + 2^{N-2} + \cdots + 2 + 1 = 2^N - 1 \]
  - i.e. N ones 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”
  - sign=0: positive numbers; sign=1: negative numbers

- Benefits:
  - Using MSB as sign bit matches positive numbers with unsigned
    - unsigned: \( 0b\ 0010 = 2^1 = 2 \)
    - sign+mag: \( 0b\ 0010 = +2^1 = 2 \)
  - 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!
  - This is why we represent one more negative than positive number \((-2^{N-1} \text{ to } 2^{N-1} - 1)\)
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!
  \((\sim x + 1 == -x)\)
Polling Question [Int I - b]

- 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/pbjones](http://pollev.com/pbjones)

A. -4

B. -5

C. 11

D. -3

E. We’re lost...
Integers

- Binary representation of integers
  - Unsigned and signed
- **Shifting and arithmetic operations** – useful for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension
Shift Operations

- Left shift \((x << n)\) bit vector \(x\) by \(n\) positions
  - Throw away (drop) extra bits on left
  - Fill with 0s on right

- Right shift \((x >> n)\) bit-vector \(x\) by \(n\) positions
  - Throw away (drop) extra bits on right
  - Logical shift (for unsigned values)
    • Fill with 0s on left
  - Arithmetic shift (for signed values)
    • Replicate most significant bit on left
    • Maintains sign of \(x\)
Shift Operations

- **Left shift** \((x<<n)\)
  - Fill with 0s on right

- **Right shift** \((x>>n)\)
  - **Logical shift** (for **unsigned** values)
    - Fill with 0s on left
  - **Arithmetic shift** (for **signed** values)
    - Replicate most significant bit on left

**Notes:**

- Shifts by \(n<0\) or \(n\geq w\) (\(w\) is bit width of \(x\)) are **undefined**
- **In C:** behavior of \(>>\) is determined by compiler
  - In gcc / C lang, depends on data type of \(x\) (signed/unsigned)
- **In Java:** logical shift is \(>>>\) and arithmetic shift is \(>>\)
Shifting Arithmetic?

- What are the following computing?
  - `x>>n` / \( \frac{4}{2^n} = 2 \)
    - `0b 0100 >> 1 = 0b 0010`
    - `0b 0100 >> 2 = 0b 0001`
    - **Divide by** \(2^n\)
  - `x<<n` \( \times \frac{1}{2^n} = 2 \)
    - `0b 0001 << 1 = 0b 0010`
    - `0b 0001 << 2 = 0b 0100`
    - **Multiply by** \(2^n\)

- Shifting is faster than general multiply and divide operations
Left Shifting Arithmetic 8-bit Example

- No difference in left shift operation for unsigned and signed numbers (just manipulates bits)
  - Difference comes during interpretation: \( x \times 2^n \)?

\[
\begin{align*}
x & = 25; \quad 00011001 = 25 \quad 25 \\
L1 = x << 2; \quad 01100100 = 100 \quad 100 \\
L2 = x << 3; \quad 11001000 = -56 \quad 200 \\
L3 = x << 4; \quad 11001000 = -112 \quad 144
\end{align*}
\]

Signed overflow

Unsigned overflow
Right Shifting Arithmetic 8-bit Examples

- **Reminder:** C operator `>>` does *logical* shift on unsigned values and *arithmetic* shift on signed values
  - **Logical Shift:** $x/2^n$?

\[
x_{u} = 240_{u};\quad \begin{array}{c}
11110000 \\
\mathbin{\div}_{3}\end{array} = 240_{u}
\]
\[
R_{1u}=x_{u}>>3;\quad \begin{array}{c}
000111110000 \\
\mathbin{\div}_{2^3}\end{array} = 30
\]
\[
R_{2u}=x_{u}>>5;\quad \begin{array}{c}
00000111100000 \\
\mathbin{\div}_{2^5}\end{array} = 7
\]

**rounding (down)**
Right Shifting Arithmetic 8-bit Examples

- **Reminder:** C operator `>>` does *logical* shift on unsigned values and *arithmetic* shift on signed values

- **Arithmetic Shift:** \( x/2^n \)?

\[
\begin{align*}
x_s &= -16; \quad 11110000 &= -16 \\
R_{1s} &= xu >> 3; \quad 11111110000 &= -2 \\
R_{2s} &= xu >> 5; \quad 1111111110000 &= -1
\end{align*}
\]

- [rounding (down)](yellow)
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
- Shifting is a useful bitwise operator
  - Right shifting can be arithmetic (sign) or logical (0)
  - Can be used in multiplication with constant or bit masking