Announcement

- Sorry about the delay on releasing exercise --- git repo set up issues; stay tuned.

Bits, Bytes, and Integers

B&O Readings: 2.1-2.2

CSE 361: Introduction to Systems Software

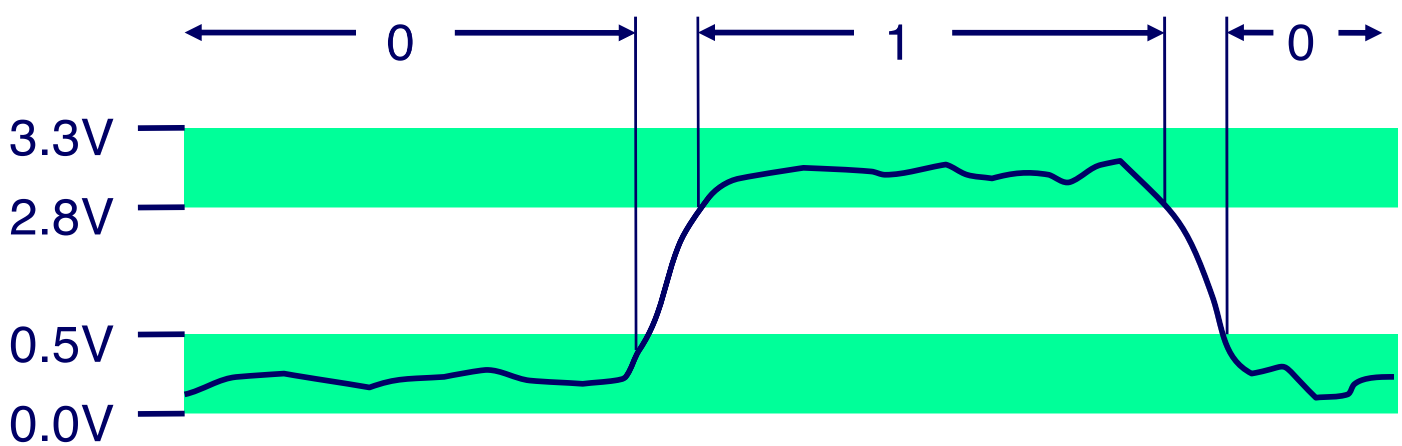
Instructor:
I-Ting Angelina Lee

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- **Bit-level manipulation**
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting

Binary Representations

- **Everything is a collection of bits (a bit: 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**
  - **Processor is made of billions of transistors**
  - **transistor: a tiny switch activated by the electronic signal it receives**
  - **Can reliably store and transmit bi-stable elements**



Binary Representations of Integral Values

- **Base 2 Number Representation**
  - Represent 161_{10} as 10100001_2
  - 161_{10}

$1 \cdot 10^0$

$6 \cdot 10^1$

$1 \cdot 10^2$

  - $10100001_2 \quad = \quad 1 \cdot 2^0 + 1 \cdot 2^5 + 1 \cdot 2^7$

most-significant bit (MSB)

least-significant bit (LSB)

Byte: Smallest Addressable Unit of Memory

- **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

Converting Between Different Bases

- **Find the hexadecimal (base 16) representation for the following number:**
  - 51966

- **How do you convert from decimal to hex?**
  - Take the value, mod it by 16 to find the quotient and remainder
  - Take the reminder as the next digit (from least-significant to most)
  - Repeat with the quotient as the new value it reaches 0
- **What about its binary representation?**
  - Each hex digit can be presented by 4 binary digits
- **What about its octal representation?**

Recap: What We Learned Thus Far

- **How values are represented in bits**
- **How to convert between:**
  - decimal (base 10)
  - hexidecimal (base 16)
  - binary (base 2)

Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- Integers
  - Representation: unsigned and signed
  - Conversion, casting

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

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**

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**
  - `~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`
    - $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

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 }

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)

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

Using Bit Masks to For Modular Arithmetic for Power of Two

```
unsigned int val = … // some value to take mod
unsigned int x = … // some power of two
unsigned int mask = x – 1;
unsigned int val_mod_x = val & mask;
```

Recab: What We Learned Thurs Far

- Bit vectors can be used to represent a set of resources (often done in system software)
- How to apply Boolean algebra to manipulate bits in C
- The distinction between Boolean algebra and logic operations in C
- Knowing bit representation of values allows one to perform perform certain tasks more efficiently

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting

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

Typically treated as a signed value, but no guarantee!

Same size if declared as unsigned

Watch out for portability issues!

Unsigned & Signed Numeric Values

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **For 1xxx values, the signed and unsigned values are +/-16 apart.**

Encoding Integers

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

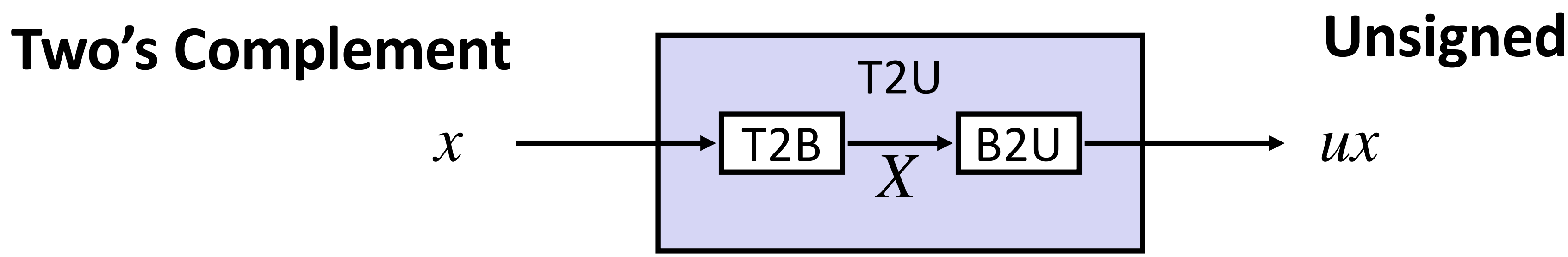
```
short int x =  361;
short int y = -361;
```

- **C `short` 2 bytes long**

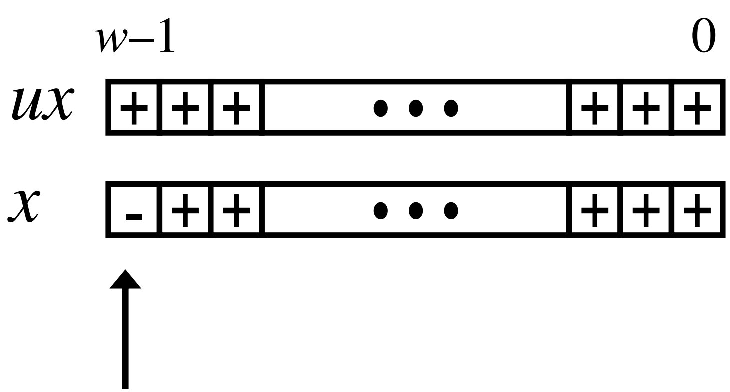
	Decimal	Hex	Binary
`x`	361	`01 69`	`00000001 01101001`
`y`	-361	`FE 97`	`11111110 10010111`

- **Sign Bit**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Relation between Signed & Unsigned



Maintain Same Bit Pattern



Large negative weight

becomes

Large positive weight

Numeric Ranges

- **Assume you have a integer type that's 5 bits long …**
  - What's the maximum value it can take if it's an unsigned int?
  - What's the minimum value it can take if it's an unsigned int?
  - What's the maxmum value it can take if it's a signed int?
  - What's the minimum value it can take if it's a signed int?
- **How do we generalize this to w-bit integer?**

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    0x000…0
  - *UMax* = $2^w - 1$
    0xFFF…1
- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    0x800…0
  - *TMax* = $2^{w-1} - 1$
    0x7FF…F
- **Other Interesting Values**
  - Minus 1
    0xFF…F

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
  - $|TMin| = TMax + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$
- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Negation: Complement & Increment

- **Claim: Following Holds for 2's Complement**

  ```
  -x = ~x + 1
  ```

- **Why is this the case?**

Recap: What We Learned Thus Far

- **Bit representation of signed and unsigned integer values**
- **How to compute the range of values a particular integral data type can take when it's**
  - an unsigned int
  - a signed int (two's complement)
- **How to convert a positive value into a negative value in two's complement**
  (Also why two's complement works so well)

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

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

Positive `x`	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

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

Positive `x`	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Negative `x`	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Shift Operations

- **Left Shift:** **`x << y`**
  - Shift bit-vector **`x`** left **`y`** positions
    - – Throw away extra bits on left

IMPORTANT:
For unsigned data, >> performs logical shift.
For signed data, the standard does not define it, but likely arithmetic.

  - Arithmetic shift
    - Replicate most significant bit on left

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

Positive x	01100010
<< 3	00010*000*
Arith. >> 2	*11*101000

Implement a pop_count function

- How do you implement pop_count, that counts the number of bits set in a 4 byte memory?
  ex: pop_count(0x000000FF) = 8

Recap: What We Learned Thus Far

- The right shift behaves differently depending on whether an expression is signed versus unsigned.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting

Casting Between Signed vs. Unsigned in C

- **Constants**
  - ***By default are considered to be signed integers***
  - Unsigned if have "U" as suffix: `0U, 4294967259U`

- **Casting**
  - **Explicit casting between signed & unsigned same as U2T and T2U**

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

Rule of Thumb: Keep bit representations and reinterpret!

  - **Implicit casting also occurs via assignments and procedure calls**

```
tx = ux;
uy = ty;
```

Use -Werror and -Wall compiler flag to catch this!

Casting Surprises

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**

Code Puzzle

- **What's the bug in this code? How do you fix it?**

```
float sum_elements(float a[], unsigned length) {
    int i;
    float result = 0;

    for (i=0; i <= length-1; i++)
        result += a[i];
    return result;
}
```

Recap: What We Learned Thus Far

- **C allows one to cast from signed to unsigned and vise versa:**
  - Bit pattern is maintained
  - But reinterpreted
  - Can have unexpected effects: adding or subtracting 2^w
- **When an expression contains both signed and unsigned, it's implicitly treated as unsigned.**

- **Why should you care? Understanding these quirks in C allows you to write correct and secure code!**

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in FreeBSD's implementation of getpeername
- There are legions of smart people trying to find vulnerabilities in programs

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious Usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

Recap: What We Learned About Casting

- **C allows one to cast from signed to unsigned and vise versa:**
  - Bit pattern is maintained
  - But reinterpreted
  - Can have unexpected effects: adding or subtracting 2^w
- **When an expression contains both signed and unsigned, it's implicitly treated as unsigned.**

- **Understanding these quirks in C allows you to write correct and secure code.**

When Should I Use Unsigned?

- ***Don't*** **Use Just Because the Number are Nonnegative**
  - Easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - Can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

- ***Do*** **Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension

Expression Evaluation Puzzles

- **Assuming int type (32 bits)**

Constant$_1$	**Constant$_2$**	**Relation**	**Evaluation**
0	0U		
-1	0		
-1	0U		
2147483647	-2147483647-1		
2147483647U	-2147483647-1		
-1	-2		
(unsigned)-1	-2		
2147483647	2147483648U		
2147483647	(int) 2147483648U		

TMIN = -2147483647-1 (0x80000000)

TMAX = 2147483647 (0x7FFFFFFF)