

# COS 217: Introduction to Programming Systems

## Numbers (in C and otherwise)

Q: Why do computer programmers confuse Christmas and Halloween?

A: Because `25 Dec == 31 Oct`





# The Decimal Number System

## Name

- From Latin *decem* (“ten”)

## Characteristics

- For us, these symbols (Not universal ...)
- 0 1 2 3 4 5 6 7 8 9

European (descended from the West Arabic)	0	1	2	3	4	5	6	7	8	9	
Arabic-Indic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩	
Eastern Arabic-Indic (Persian and Urdu)	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹	
Devanagari (Hindi)	०	१	२	३	४	५	६	७	८	९	
Tamil		௦	௧	௨	௩	௪	௫	௬	௭	௮	௯

[https://commons.wikimedia.org/wiki/File:Arabic\\_numerals-en.svg](https://commons.wikimedia.org/wiki/File:Arabic_numerals-en.svg)

- Positional

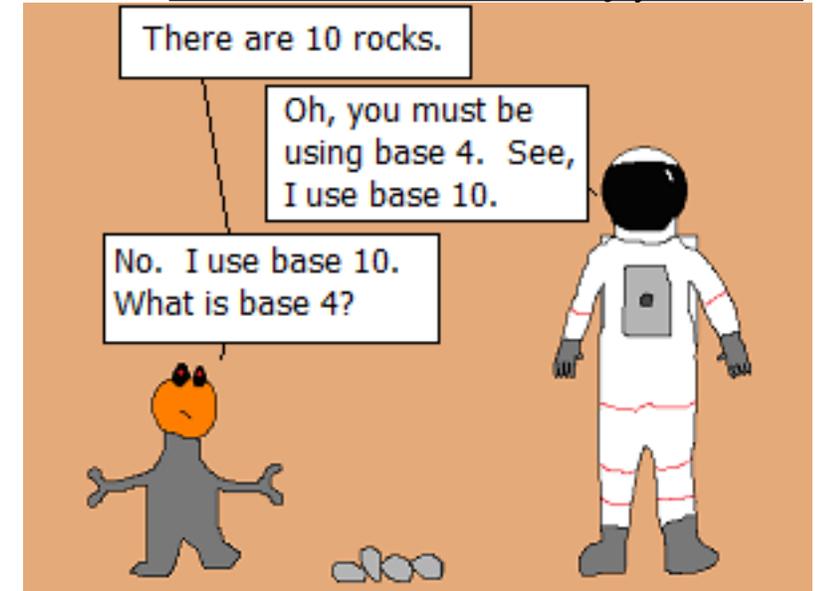
- $2945 \neq 2495$

- $2945 = (2 \cdot 10^3) + (9 \cdot 10^2) + (4 \cdot 10^1) + (5 \cdot 10^0)$

2 (Most) people use the decimal number system



Cowbirds in Love #43 – Sanjay Kulkacek



Every base is base 10.



# The Binary Number System

## binary

*adjective:* being in a state of one of two mutually exclusive conditions such as on or off, true or false, molten or frozen, presence or absence of a signal.

From late Latin *binarius* (“consisting of two”), from classical Latin *bis* (“twice”)

## Characteristics

- Two symbols: 0 1
- Positional:  $1010_B \neq 1100_B$

Most (digital) computers use the binary number system



## Terminology

- **Bit:** a single binary symbol (“binary digit”)
- **Byte:** (typically) 8 bits
- **Nibble / Nybble:** 4 bits – we'll see a more common name for 4 bits soon.



# Decimal-Binary Equivalence

<u>Decimal</u>	<u>Binary</u>
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

<u>Decimal</u>	<u>Binary</u>
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010
27	11011
28	11100
29	11101
30	11110
31	11111
...	...



# Decimal-Binary Conversion

Binary to decimal: expand using positional notation

$$\begin{aligned} 100101_B &= (1 \cdot 2^5) + (0 \cdot 2^4) + (0 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) \\ &= 32 + 0 + 0 + 4 + 0 + 1 \\ &= 37 \end{aligned}$$

Most-significant  
bit (msb)

Least-significant  
bit (lsb)



# Integer-Binary Conversion

(Decimal) Integer to binary: do the reverse

- Determine largest power of 2 that's  $\leq$  number; write template

$$37 = (? * 2^5) + (? * 2^4) + (? * 2^3) + (? * 2^2) + (? * 2^1) + (? * 2^0)$$

- Fill in template

$$\begin{array}{r} 37 = (1 * 2^5) + (0 * 2^4) + (0 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0) \\ \underline{-32} \\ 5 \\ \underline{-4} \\ 1 \\ \underline{-1} \\ 0 \end{array} \qquad 100101_B$$



# Integer-Binary Conversion

## Integer to binary division method

- Repeatedly divide by 2, consider remainder

37	/	2	=	18	R	1
18	/	2	=	9	R	0
9	/	2	=	4	R	1
4	/	2	=	2	R	0
2	/	2	=	1	R	0
1	/	2	=	0	R	1



Read from bottom  
to top:  $100101_B$



# The Hexadecimal Number System

## Name

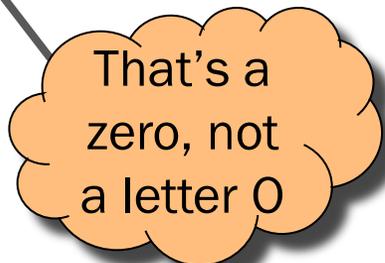
- From ancient Greek ἑξ (*hex*, “six”) + Latin-derived *decimal*

## Characteristics

- Sixteen symbols (“hexits”)
  - 0 1 2 3 4 5 6 7 8 9 A B C D E F
- Positional
  - $A13D_H \neq 3DA1_H$

Computer programmers often use hexadecimal (“hex”)

- In C: 0x prefix (0xA13D, etc.)





# Binary-Hexadecimal Conversion

Observation:

- $16^1 = 2^4$ , so every 1 hexit corresponds to a nybble (4 bits)

Binary to hexadecimal

1010000100111101 <sub>B</sub>
A 1 3 D <sub>H</sub>

Number of bits in binary number  
not a multiple of 4?  $\Rightarrow$   
pad with zeros on left

Hexadecimal to binary

A 1 3 D <sub>H</sub>
1010000100111101 <sub>B</sub>

Discard leading zeros from binary  
number if appropriate



# Integer-Hexadecimal Conversion

Hexadecimal to (decimal) integer: expand using positional notation

$$\begin{aligned} 25_{\text{H}} &= (2 \cdot 16^1) + (5 \cdot 16^0) \\ &= 32 + 5 \\ &= 37 \end{aligned}$$

Integer to hexadecimal: use the division method

$$\begin{aligned} 37 / 16 &= 2 \text{ R } 5 \\ 2 / 16 &= 0 \text{ R } 2 \end{aligned}$$



Read from bottom  
to top:  $25_{\text{H}}$



# Are you $539_H$ ?



Convert binary 101010 into decimal and hex

- A. 21 decimal, A2 hex
- B. 21 decimal, A8 hex
- C. 18 decimal, 2A hex
- D. 42 decimal, 2A hex

hint: convert to hex first

challenge: once you've voted and discussed with a neighbor, figure out why this slide's title is what it is.



# The Octal Number System

## Name

- “octo” (Latin)  $\Rightarrow$  eight

## Characteristics

- Eight symbols
  - 0 1 2 3 4 5 6 7
- Positional
  - $17430 \neq 73140$



Computer programmers sometimes use octal (so does Mickey!)

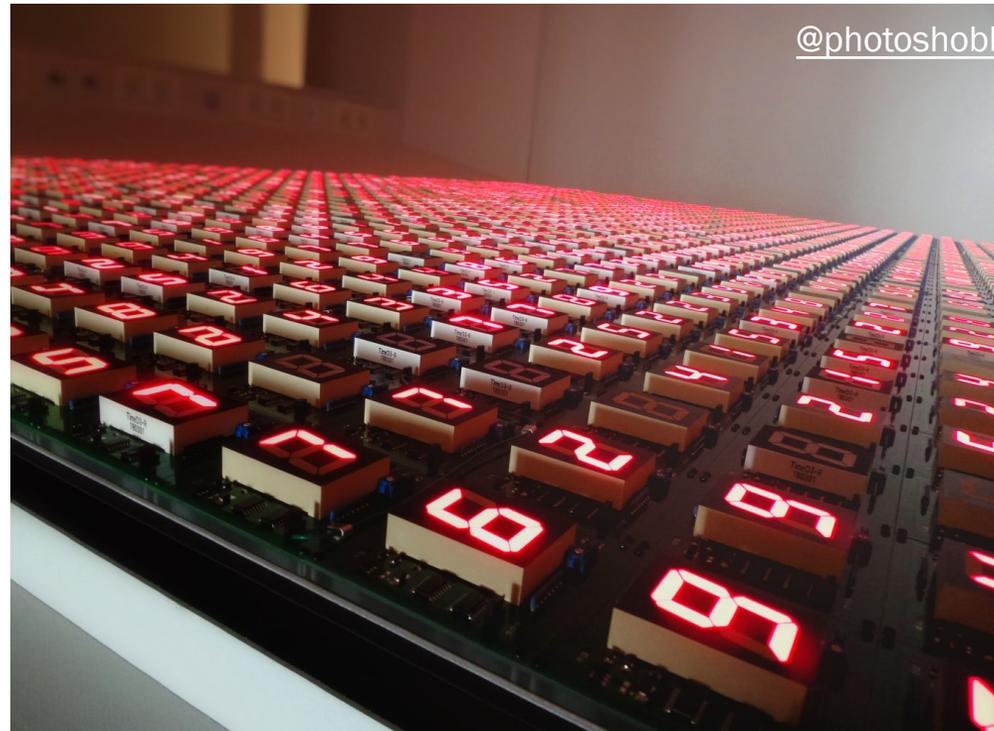
- In C: 0 prefix (01743, etc.)



```
[cmoretti@tars:tmp]$ ls -l myFile
-rw-r--r-- 1 cmoretti wheel 0 Sep  7 10:58 myFile
[cmoretti@tars:tmp]$ chmod 755 myFile
[cmoretti@tars:tmp]$ ls -l myFile
-rwxr-xr-x 1 cmoretti wheel 0 Sep  7 10:58 myFile
```



# INTEGERS





# Representing Unsigned (Non-Negative) Integers

## Mathematics

- Non-negative integers' range is 0 to  $\infty$

## Computers

- Range limited by computer's **word** size
- Word size is  $n$  bits  $\Rightarrow$  range is 0 to  $2^n - 1$  representing with an  $n$  bit binary number
- Exceed range  $\Rightarrow$  **overflow**

## Typical computers today

- $n = 32$  or  $64$ , so range is 0 to  $2^{32} - 1$  (~4 billion) or  $2^{64} - 1$  (huge ... ~ $1.8e19$ )

## Pretend computer for these slides, hereafter on these slides:

- Assume  $n = 4$ , so range is 0 to  $2^4 - 1$  (15)
- All points generalize to larger word sizes like 32 and 64



# Representing Unsigned Integers

On 4-bit pretend computer

<u>Unsigned Integer</u>	<u>Rep</u>
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111



# Adding Unsigned Integers

## Addition

			<b>1</b>
	3		0011 <sub>B</sub>
+	10	+	1010 <sub>B</sub>
--		--	----
	13		1101 <sub>B</sub>

			<b>111</b>
	7		0111 <sub>B</sub>
+	10	+	1010 <sub>B</sub>
--		--	----
	1		0001 <sub>B</sub>

Start at right column  
Proceed leftward  
Carry 1 when necessary

Beware of overflow

How would you detect overflow programmatically?

Results are mod  $2^4$

$$7 + 10 = 17$$

$$17 \bmod 16 = 1$$



# Subtracting Unsigned Integers

## Subtraction

10	1010 <sub>B</sub>
- 7	- 0111 <sub>B</sub>
--	----
3	0011 <sub>B</sub>

3	0011 <sub>B</sub>
- 10	- 1010 <sub>B</sub>
--	----
9	1001 <sub>B</sub>

Start at right column  
 Proceed leftward  
 Borrow when necessary

Beware of overflow

Results are mod 2<sup>4</sup>

$$3 - 10 = -7$$

$$-7 \bmod 16 = 9$$

How would you detect overflow programmatically?

# Reminder: negative numbers exist





# Obsolete Attempt #1: Sign-Magnitude

<u>Integer</u>	<u>Rep</u>
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

## Definition

High-order bit indicates sign

0  $\Rightarrow$  positive

1  $\Rightarrow$  negative

Remaining bits indicate magnitude

$$0101_B = 101_B = 5$$

$$1101_B = -101_B = -5$$

## Pros and cons

- + easy to understand, easy to negate
- + symmetric
- two representations of zero
- need different algorithms to add signed and unsigned numbers

Not widely used for integers today



# Obsolete Attempt #2: Ones' Complement

<u>Integer</u>	<u>Rep</u>
-7	1000
-6	1001
-5	1010
-4	1011
-3	1100
-2	1101
-1	1110
-0	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit has weight  $-(2^{b-1}-1)$

$$1010_B = (1 * -7) + (0 * 4) + (1 * 2) + (0 * 1) = -5$$

$$0010_B = (0 * -7) + (0 * 4) + (1 * 2) + (0 * 1) = 2$$

Computing negative = flipping all bits

Similar pros and cons to sign-magnitude



# Two's Complement

<u>Integer</u>	<u>Rep</u>
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

## Definition

High-order bit has weight  $-(2^{b-1})$

$$1010_B = (1 * -8) + (0 * 4) + (1 * 2) + (0 * 1) \\ = -6$$

$$0010_B = (0 * -8) + (0 * 4) + (1 * 2) + (0 * 1) \\ = 2$$



# Two's Complement (cont.)

<u>Integer</u>	<u>Rep</u>
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Computing negative

$$\text{neg}(x) = \sim x + 1$$

$$\text{neg}(x) = \text{onescomp}(x) + 1$$

$$\text{neg}(0101_{\text{B}}) = 1010_{\text{B}} + 1 = 1011_{\text{B}}$$

$$\text{neg}(1011_{\text{B}}) = 0100_{\text{B}} + 1 = 0101_{\text{B}}$$

Pros and cons

- not symmetric

(“extra” negative number;  $-(-8) = -8$ )

+ one representation of zero

+ same algorithms add/subtract  
signed and unsigned integers



# Adding Signed Integers

pos + pos

		<b>11</b>
3		0011 <sub>B</sub>
+ 3	+	0011 <sub>B</sub>
--		----
6		0110 <sub>B</sub>

pos + pos (overflow)

		<b>111</b>
7		0111 <sub>B</sub>
+ 1	+	0001 <sub>B</sub>
--		----
-8		1000 <sub>B</sub>

pos + neg

		<b>1111</b>
3		0011 <sub>B</sub>
+ -1	+	1111 <sub>B</sub>
--		----
2		0010 <sub>B</sub>

How would you detect overflow programmatically?

neg + neg

		<b>11</b>
-3		1101 <sub>B</sub>
+ -2	+	1110 <sub>B</sub>
--		----
-5		1011 <sub>B</sub>

neg + neg (overflow)

		<b>1 1</b>
-6		1010 <sub>B</sub>
+ -5	+	1011 <sub>B</sub>
--		----
5		0101 <sub>B</sub>



# Subtracting Signed Integers

How would you compute  $3 - 4$ ?

3		0011 <sub>B</sub>
- 4	-	0100 <sub>B</sub>
--		----
?		???? <sub>B</sub>



# Subtracting Signed Integers

Perform subtraction  
with borrows

3	0011 <sub>B</sub>
- 4	- 0100 <sub>B</sub>
--	----
-1	1111 <sub>B</sub>

or

Compute two's comp  
and add

3	0011 <sub>B</sub>
+ -4	+ 1100 <sub>B</sub>
--	----
-1	1111 <sub>B</sub>

-5	1011 <sub>B</sub>
--2	- 1110 <sub>B</sub>
--	----
-3	1101 <sub>B</sub>



	<b>1</b>
-5	1011 <sub>B</sub>
+ 2	+ 0010 <sub>B</sub>
--	----
-3	1101 <sub>B</sub>



# Negating Signed Ints: Math

Question: Why does two's comp arithmetic work?

Answer:  $[-b] \bmod 2^4 = [\text{twoscomp}(b)] \bmod 2^4$

$$\begin{aligned} &[-b] \bmod 2^4 \\ &= [2^4 - b] \bmod 2^4 \\ &= [2^4 - 1 - b + 1] \bmod 2^4 \\ &= [(2^4 - 1 - b) + 1] \bmod 2^4 \\ &= [\text{onescomp}(b) + 1] \bmod 2^4 \\ &= [\text{twoscomp}(b)] \bmod 2^4 \end{aligned}$$

So:  $[a - b] \bmod 2^4 = [a + \text{twoscomp}(b)] \bmod 2^4$

$$\begin{aligned} &[a - b] \bmod 2^4 \\ &= [a + 2^4 - b] \bmod 2^4 \\ &= [a + 2^4 - 1 - b + 1] \bmod 2^4 \\ &= [a + (2^4 - 1 - b) + 1] \bmod 2^4 \\ &= [a + \text{onescomp}(b) + 1] \bmod 2^4 \\ &= [a + \text{twoscomp}(b)] \bmod 2^4 \end{aligned}$$



# (AT LONG<sup>o</sup> LAST) INTEGERS IN C



30

<sup>o</sup> no pun intended, I swear!

@hannahbusing



# Integer Data Types in C

Integer types of various sizes: {signed, unsigned} {char, short, int, long}

- Shortcuts: signed assumed for short/int/long; unsigned means unsigned int
- char is 1 byte
  - Number of bits per byte is unspecified (but in the 21<sup>st</sup> century, safe to assume it's 8)
  - Signedness is system dependent, so for arithmetic use “signed char” or “unsigned char”
- Sizes of other integer types not fully specified but constrained:
  - int was intended to be “natural word size” of hardware, but isn't always
  - $2 \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})$

## On arm lab:

- Natural word size: 8 bytes (“64-bit machine”)
- char: 1 byte
- short: 2 bytes
- int: 4 bytes (compatibility with widespread 32-bit code)
- long: 8 bytes

What decisions did the designers of Java make?



# Integer Types in Java vs. C

	Java	C
Unsigned types	<code>char // 16 bits</code>	<code>unsigned char</code> <code>unsigned short</code> <code>unsigned (int)</code> <code>unsigned long</code>
Signed types	<code>byte // 8 bits</code> <code>short // 16 bits</code> <code>int // 32 bits</code> <code>long // 64 bits</code>	<code>signed char</code> <code>(signed) short</code> <code>(signed) int</code> <code>(signed) long</code>

1. Not guaranteed by C, but on **armlab**, `short` = 16 bits, `int` = 32 bits, `long` = 64 bits
2. Not guaranteed by C, but on **armlab**, `char` is unsigned



# sizeof Operator

- Applied at compile-time
- Operand can be a data type
- Operand can be an expression, from which the compiler infers a data type

## Examples, on arm1ab using gcc217

- `sizeof(int)` evaluates to 4
- `sizeof(i)` evaluates to 4 if `i` is a variable of type `int`
- `sizeof(1+2)` evaluates to 4



# Integer Literals in C

- Decimal int: 123
- Prefixes to indicate a different base
  - Octal int: 0173 = 123
  - Hexadecimal int: 0x7B = 123
  - No prefix to indicate binary int literal
- Suffixes to indicate a different type
  - Use "L" suffix to indicate long literal
  - Use "U" suffix to indicate unsigned literal
  - No suffix to indicate char or short literals; instead, cast

char:	'{' (← really int, as seen last time), (char) 123, (char) 0173, (char) 0x7B
int:	123, 0173, 0x7B
long:	123L, 0173L, 0x7BL
short:	(short)123, (short)0173, (short)0x7B
unsigned int:	123U, 0173U, 0x7BU
unsigned long:	123UL, 0173UL, 0x7BUL
unsigned short:	(unsigned short)123, (unsigned short)0173, (unsigned short)0x7B



# sizeof synthesis



Q: What is the value of the following sizeof expression on the armlab machines?

```
int i = 1;  
sizeof(i + 2L)
```

- A. 3
- B. 4
- C. 8
- D. 12
- E. error



# OPERATIONS ON NUMBERS



Shawn Rossi ©



# Reading / Writing Numbers

## Motivation

- Must convert between external form (sequence of character codes) and internal form
- Could provide `getchar()`, `putshort()`, `getint()`, `putfloat()`, etc.
- Alternative implemented in C: parameterized functions

## `scanf()` and `printf()`

- Can read/write any primitive type of data
- First parameter is a format string containing conversion specs: size, base, field width
- Can read/write multiple variables with one call

See King book for details



# Operators in C

- Typical arithmetic operators: + - \* / %
- Typical relational operators: == != < <= > >=
  - Each evaluates to FALSE  $\Rightarrow$  0, TRUE  $\Rightarrow$  1
- Typical logical operators: ! && ||
  - Each interprets 0  $\Rightarrow$  FALSE, non-0  $\Rightarrow$  TRUE
  - Each evaluates to FALSE  $\Rightarrow$  0, TRUE  $\Rightarrow$  1
- Cast operator: (type)
- Bitwise operators: ~ & | ^ >> <<

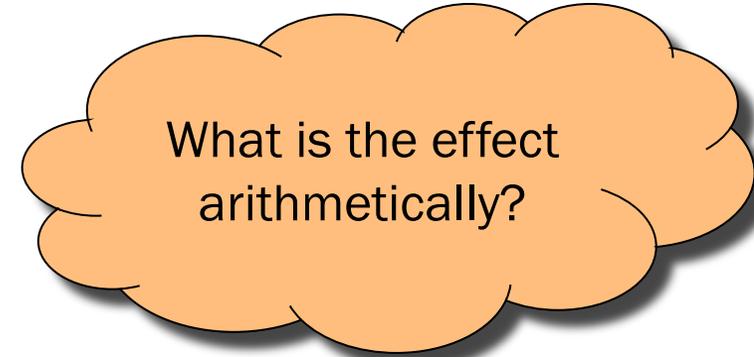


# Shifting Unsigned Integers

Bitwise right shift ( $\gg$  in C): fill on left with zeros

`10 >> 1 ⇒ 5`  
 $1010_B$       $0101_B$

`10 >> 2 ⇒ 2`  
 $1010_B$       $0010_B$

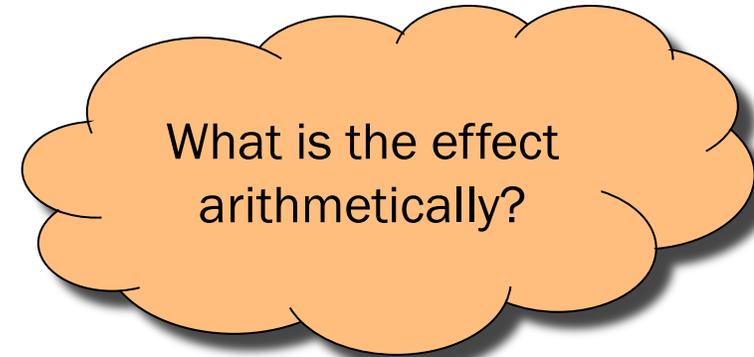


Bitwise left shift ( $\ll$  in C): fill on right with zeros

`5 << 1 ⇒ 10`  
 $0101_B$       $1010_B$

`3 << 2 ⇒ 12`  
 $0011_B$       $1100_B$

`3 << 3 ⇒ 8`  
 $0011_B$       $1000_B$



← Results are mod  $2^4$



# Other Bitwise Operations on Unsigned Integers

## Bitwise NOT (~ in C)

- Flip each bit (don't forget leading 0s!)

$\sim 10 \Rightarrow 5$
$1010_B \quad 0101_B$

$\sim 5 \Rightarrow 10$
$0101_B \quad 1010_B$

## Bitwise AND (& in C)

- AND (1=True, 0=False) corresponding bits

10	1010 <sub>B</sub>
& 7	& 0111 <sub>B</sub>
--	----
2	0010 <sub>B</sub>

10	1010 <sub>B</sub>
& 2	& 0010 <sub>B</sub>
--	----
2	0010 <sub>B</sub>

Useful for “masking” bits to 0

$x \& 0$  is 0,  $x \& 1$  is  $x$



# Other Bitwise Operations on Unsigned Ints

## Bitwise OR: (`|` in C)

- Logical OR corresponding bits

10	1010 <sub>B</sub>
1	0001 <sub>B</sub>
--	----
11	1011 <sub>B</sub>

Useful for “masking” bits to 1

$x | 1$  is 1,  $x | 0$  is  $x$

## Bitwise exclusive OR (`^` in C)

- Logical exclusive OR corresponding bits

10	1010 <sub>B</sub>
^ 10	^ 1010 <sub>B</sub>
--	----
0	0000 <sub>B</sub>

$x \wedge x$  sets all bits to 0



# Logical vs. Bitwise Ops

## Logical AND (&&) vs. bitwise AND (&)

- `2 (TRUE) && 1 (TRUE) ==> 1 (TRUE)`

Decimal	Binary
2	00000000 00000000 00000000 00000010
<code>&amp;&amp; 1</code>	00000000 00000000 00000000 00000001
----	-----
1	00000000 00000000 00000000 00000001

- `2 (TRUE) & 1 (TRUE) ==> 0 (FALSE)`

Decimal	Binary
2	00000000 00000000 00000000 00000010
<code>&amp; 1</code>	00000000 00000000 00000000 00000001
----	-----
0	00000000 00000000 00000000 00000000

## Implication:

- Use **logical** AND to control flow of logic
- Use **bitwise** AND only when doing bit-level manipulation
- Same for OR and NOT



# A *Bit* Complicated ... challenge for the bored



How do you set bit  $k$  (where the least significant bit is bit 0)  
of unsigned variable  $u$  to zero (leaving everything else in  $u$  unchanged)?

- A.  $u \&= (0 \ll k);$
- B.  $u |= (1 \ll k);$
- C.  $u |= \sim(1 \ll k);$
- D.  $u \&= \sim(1 \ll k);$
- E.  $u = \sim u \wedge (1 \ll k);$



# Aside: Using Bitwise Ops for Arithmetic

Can use  $\ll$ ,  $\gg$ , and  $\&$  to do some arithmetic efficiently

$$x * 2^y == x \ll y$$

- $3 * 4 = 3 * 2^2 = 3 \ll 2 \Rightarrow 12$

Fast way to multiply  
by a power of 2

$$x / 2^y == x \gg y$$

- $13 / 4 = 13 / 2^2 = 13 \gg 2 \Rightarrow 3$

Fast way to divide  
unsigned by power of 2

$$x \% 2^y == x \& (2^y - 1)$$

- $13 \% 4 = 13 \% 2^2 = 13 \& (2^2 - 1)$   
 $= 13 \& 3 \Rightarrow 1$

Fast way to mod  
by a power of 2

13	1101 <sub>B</sub>
& 3	& 0011 <sub>B</sub>
--	----
1	0001 <sub>B</sub>

Many compilers will  
do these transformations  
automatically!



# Shifting Signed Integers

Bitwise left shift (<< in C): fill on right with zeros

$$\begin{array}{l} 3 \ll 1 \Rightarrow 6 \\ 0011_B \quad 0110_B \end{array}$$

$$\begin{array}{l} -3 \ll 1 \Rightarrow -6 \\ 1101_B \quad 1010_B \end{array}$$

$$\begin{array}{l} -3 \ll 2 \Rightarrow 4 \\ 1101_B \quad 0100_B \end{array}$$

What is the effect arithmetically?

Results are mod  $2^4$

Bitwise right shift: fill on left with ???



# Shifting Signed Integers (cont.)

Bitwise *arithmetic* right shift: fill on left with sign bit

6 >> 1 ⇒ 3  
0110<sub>B</sub>      0011<sub>B</sub>

-6 >> 1 ⇒ -3  
1010<sub>B</sub>      1101<sub>B</sub>

What is the effect arithmetically?

Bitwise *logical* right shift: fill on left with zeros

6 >> 1 ⇒ 3  
0110<sub>B</sub>      0011<sub>B</sub>

-6 >> 1 ⇒ 5  
1010<sub>B</sub>      0101<sub>B</sub>

What is the effect arithmetically???

In C, right shift (>>) could be logical (>>> in Java) or arithmetic (>> in Java)

- Not specified by standard (happens to be arithmetic on armlab)
- Best to avoid shifting signed integers



# Other Operations on Signed Ints

## Bitwise NOT (~ in C)

- Same as with unsigned ints

## Bitwise AND (& in C)

- Same as with unsigned ints

## Bitwise OR: (| in C)

- Same as with unsigned ints

## Bitwise exclusive OR (^ in C)

- Same as with unsigned ints

Best to avoid using signed ints for bit-twiddling.



# Assignment Operator

Many high-level languages provide an assignment *statement*

C provides an assignment **operator**

- Performs assignment, and then **evaluates to the assigned value**
- Allows assignment to appear within larger expressions
- But be careful about precedence! Extra parentheses often needed!



# Assignment Operator Examples

## Examples

```
i = 0;
    /* Side effect: assign 0 to i.
       Evaluate to 0. */

j = i = 0; /* Assignment op has R to L associativity */
    /* Side effect: assign 0 to i.
       Evaluate to 0.
       Side effect: assign emitted 0 to j.
       Evaluate to 0. */

while ((i = getchar()) != EOF) ...
    /* Read a character or EOF value.
       Side effect: assign that value to i.
       Evaluate to that value.
       Compare that emitted value to EOF.
       Evaluate to 0 (FALSE) or 1 (TRUE). */
```



# Special-Purpose Assignment in C

## Motivation

- The construct  $a = b + c$  is flexible
- The construct  $d = d + e$  is somewhat common
- The construct  $d = d + 1$  is very common

## Assignment in C

- Introduce  $+=$  operator to do things like  $d += e$
- Extend to  $-= *= /= \sim= \&= |= \wedge= \ll= \gg=$
- All evaluate to whatever was assigned
- Pre-increment and pre-decrement:  $++d$   $--d$
- Post-increment and post-decrement (evaluate to *old* value):  $d++$   $d--$



# Plusplus Playfulness / Confusion Plusplus



Q: What are i and j set to in the following code?

```
i = 5;  
j = i++;  
j += ++i;
```

- A. 5, 7
- B. 7, 5
- C. 7, 11
- D. 7, 12
- E. 7, 13



# Incremental Iffiness



Q: What does the following code print?

```
int i = 1;
switch (i++) {
    case 1: printf("%d", ++i);
    case 2: printf("%d", i++);
}
```

- A. 1
- B. 2
- C. 3
- D. 22
- E. 33

# Sample Exam Question (Spring 2017, Exam 1)



1(b) (12 points/100) Suppose we have a 7-bit computer. Answer the following questions.

(i) (4 points) What is the largest unsigned number that can be represented in 7 bits?

In binary:

In decimal:

(ii) (4 points) What is the smallest (i.e., most negative) signed number represented in 2's complement in 7 bits?

In binary:

In decimal:

(iii) (2 points) Is there a number  $n$ , other than 0, for which  $n$  is equal to  $-n$ , when represented in 2's complement in 7 bits? If yes, show the number (in decimal). If no, briefly explain why not.

(iv) (2 points) When doing arithmetic addition using 2's complement representation in 7 bits, is it possible to have an overflow when the first number is positive and the second is negative? (Yes/No answer is sufficient, no need to explain.)

# Sample Exam Question (Fall 2024, Exam 1)



1 (d) (1 point /32) If `acc` is of type `int`, write a statement that has the same effect as the line `acc *= 10`, without using multiplication and using no more than one addition operation.

# (Hard!) Sample Exam Question (Fall 2020, Exam 1)



- a. In the two ranges below, replace the "\_\_\_\_" with the inclusive upper and lower bounds of decimal numbers that do not change value when moving from  $i$ -bit two's complement to  $(i+1)$ -bit two's complement (for example, when moving from four bits to represent integers to using five bits to do so). The two ranges consider two different possibilities for changing an  $i$ -bit value into an  $(i+1)$ -bit value:

If we make the change by prepending a 0 onto the front of the  $i$ -bit representation (e.g., 1001  $\rightarrow$  01001):

$$\text{____} \leq x \leq \text{____}$$

If we make the change by prepending a 1 onto the front of the  $i$ -bit representation (e.g., 1001  $\rightarrow$  11001):

$$\text{____} \leq x \leq \text{____}$$

- b. In the range below, replace the "\_\_\_\_" with the inclusive upper and lower bounds of armlab C int literals for which the expression still compiles and does not change value when adding a 0 before the first character of the literal (for example, 217  $\rightarrow$  0217):

$$\text{____} \leq x \leq \text{____}$$

Hint 1: does a literal 09 compile?

Hint 2: the word "expression" is intentional; note that the first character of a signed int is not necessarily a digit.





# Rational Numbers

## Mathematics

- A rational number is one that can be expressed as the ratio of two integers
- Unbounded range and precision

## Computer science

- Finite range and precision
- Approximate using floating point number



# Floating Point Numbers

Like scientific notation: e.g.,  $c$  is

$$2.99792458 \times 10^8 \text{ m/s}$$

This has the form

$$(\text{multiplier}) \times (\text{base})^{(\text{power})}$$

In the computer,

- **Multiplier** is called **mantissa**
- Base is almost always 2
- **Power** is called **exponent**



# Floating-Point Data Types

## C specifies:

- Three floating-point data types:  
float, double, and long double
- Sizes unspecified, but constrained:
- $\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$

## On ArmLab (and on pretty much any 21st-century computer using the IEEE standard)

- float: 4 bytes
- double: 8 bytes

## On ArmLab (but varying across architectures)

- long double: 16 bytes



# Floating-Point Literals

## How to write a floating-point number?

- Either fixed-point or “scientific” notation
- Any literal that contains decimal point or "E" is floating-point
- The default floating-point type is double
- Append "F" to indicate float
- Append "L" to indicate long double

## Examples

- double:                   123.456, 1E-2, -1.23456E4
- float:                     123.456F, 1E-2F, -1.23456E4F
- long double:             123.456L, 1E-2L, -1.23456E4L



# IEEE Floating Point Representation

Common finite representation: IEEE floating point

- More precisely: ISO/IEEE 754 standard

Using 32 bits (type **float** in C):

- 1 bit: sign (0⇒positive, 1⇒negative)
- 8 bits: exponent + 127
- 23 bits: binary fraction of the form 1.bbbbbbbbbbbbbbbbbbbbbbb

Using 64 bits (type **double** in C):

- 1 bit: sign (0⇒positive, 1⇒negative)
- 11 bits: exponent + 1023
- 52 bits: binary fraction of the form  
1.bbb



# When was floating-point invented?

mantissa (noun): decimal part of a logarithm, 1865, ← Answer: long before computers!  
from Latin mantisa “a worthless addition, makeweight”

COMMON LOGARITHMS  $\log_{10} x$

x	0	1	2	3	4	5	6	7	8	9.	$\Delta_m$	1	2	3
											+			
50	.6990	6998	7007	7016	7024	7033	7042	7050	7059	7067	9	1	2	3
51	.7076	7084	7093	7101	7110	7118	7126	7135	7143	7152	8	1	2	2
52	.7160	7168	7177	7185	7193	7202	7210	7218	7226	7235	8	1	2	2
53	.7243	7251	<u>7259</u>	7267	7275	7284	7292	7300	7308	7316	8	1	2	2
54	.7324	7332	7340	7348	7356	7364	7372	7380	7388	7396	8	1	2	2
55	.7404	7412	7419	7427	7435	7443	7451	7459	7466	7474	8	1	2	2



# Floating Point Example

Sign (1 bit):

- 1 ⇒ negative

11000001110110110000000000000000

32-bit representation

Exponent (8 bits):

- $10000011_B = 131$
- $131 - 127 = 4$

Mantissa (23 bits):

- $1.101101100000000000000000_B$
- $1 + (1*2^{-1})+(0*2^{-2})+(1*2^{-3})+(1*2^{-4})+(0*2^{-5})+(1*2^{-6})+(1*2^{-7})+(0*2^{-\dots})= 1.7109375$

Number:

- $-1.7109375 * 2^4 = -27.375$



# Floating Point Consequences

“Machine epsilon”: smallest positive number you can add to 1.0 and get something other than 1.0

For float:  $\varepsilon \approx 10^{-7}$

- No such number as 1.0000000001
- Rule of thumb: “almost 7 digits of precision”

For double:  $\varepsilon \approx 2 \times 10^{-16}$

- Rule of thumb: “not quite 16 digits of precision”

These are all relative numbers



# Floating Point Consequences, cont

Just as decimal number system can represent only some rational numbers with finite digit count...

- Example:  $1/3$  cannot be represented

Binary number system can represent only some rational numbers with finite digit count

- Example:  $1/5$  cannot be represented

Beware of round-off error

- Error resulting from inexact representation
- Can accumulate
- Be careful when comparing two floating-point numbers for equality

<u>Decimal</u> <u>Approx</u>	<u>Rational</u> <u>Value</u>
.3	3/10
.33	33/100
.333	333/1000
...	

<u>Binary</u> <u>Approx</u>	<u>Rational</u> <u>Value</u>
0.0	0/2
0.01	1/4
0.010	2/8
0.0011	3/16
0.00110	6/32
0.001101	13/64
0.0011010	26/128
0.00110011	51/256
...	



# Floating away ...



What does the following code print?

```
double sum = 0.0;
double i;
for (i = 0.0; i != 10.0; i++)
    sum += 0.1;
if (sum == 1.0)
    printf("All good!\n");
else
    printf("Yikes!\n");
```

- A. All good!
- B. Yikes!
- C. (Infinite loop)
- D. (Compilation error)

B: Yikes!

... loop terminates, because we can represent 10.0 exactly by adding 1.0 at a time.

... but sum isn't 1.0 because we can't represent 1.0 exactly by adding 0.1 at a time.