Number Systems II:

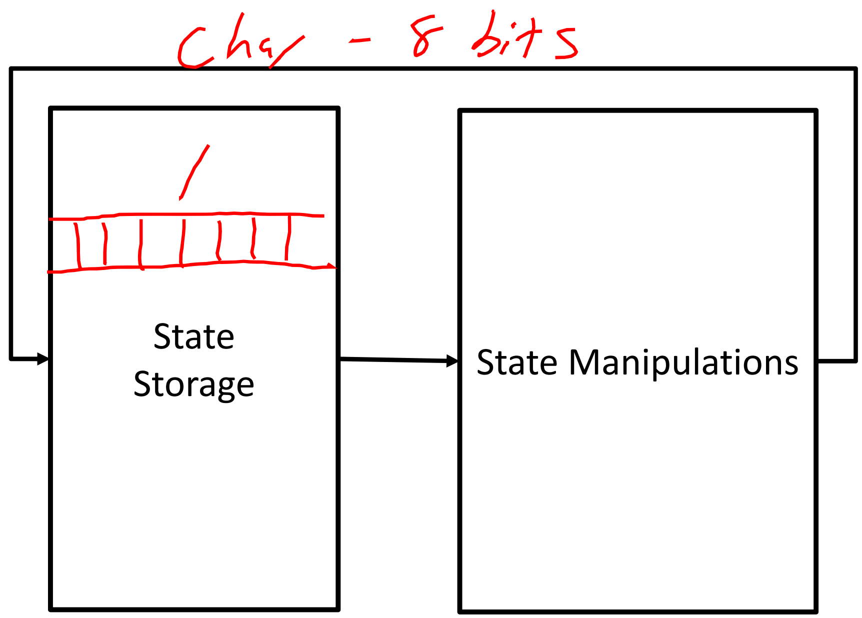
2's Complement, Arithmetic, Overflow, &
Writing Bit-wise Logical & Shifting Code

Lab 2 part 1 due Thursday

Logic and arithmetic are two primary ways of manipulating stored state

Computer can do 2 things

1) Store state (How do we interpret stored bits?)
2) Manipulate state (How do we perform meaningful mathematical operations?)



233 in one slide!

Today we continue with how to interpret state as data

- The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)
    1. You will build a simple computer processor
       Build and create **state** machines with **data**, **control**, and **indirection**
    2. You will learn how high-level language code executes on a processor
       Time limitations create **dependencies** in the **state** of the processor
    3. You will learn why computers perform the way they do
       Physical limitations require **locality** and **indirection** in how we access **state**
    4. You will learn about hardware mechanisms for parallelism
       Locality, dependencies, and **indirection** on performance enhancing drugs

- We will have a SPIMbot contest!

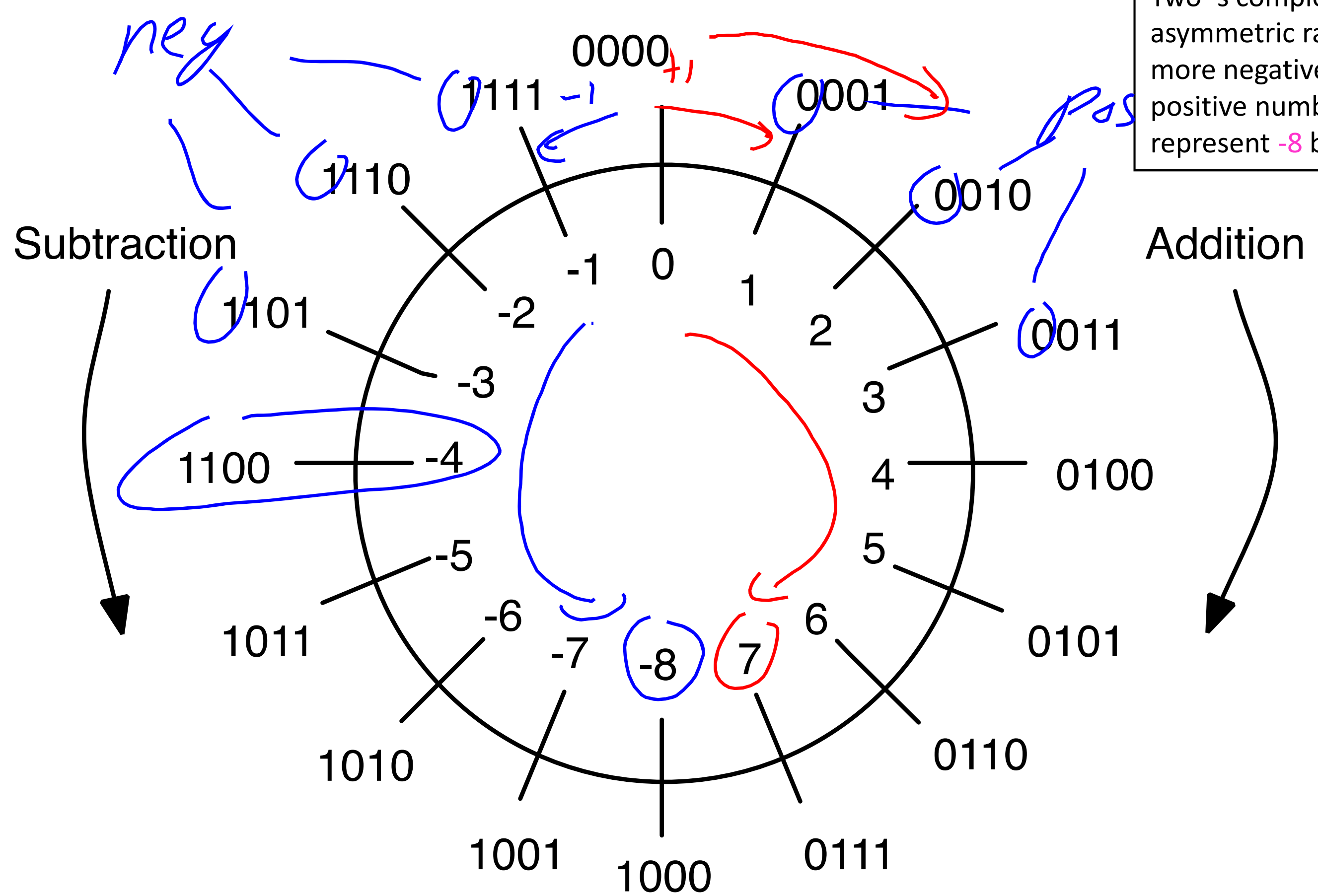
Today’s lecture

- Two’s complement signed binary representation
  - Negating numbers in Two’s complement
  - Sign extension

- Bit-wise shift operations
  - Writing bit-wise logical and shifting code

- Two’s complement arithmetic
  - Addition
  - Subtraction
  - Overflow

Review: 4-bit 2's complement



Two's complement has asymmetric ranges; there is one more negative number than positive number. Here, you can represent -8 but not +8.

Negating Numbers in 2's Complement

- To negate a number:
  - Complement each bit and then add 1.

$(\sim X) + 1$

- Example:

0100	$= +4_{10}$	(a positive number in 4-bit two's complement)
1011	=	(invert all the bits)
1100	$= -4_{10}$	(and add one)
0011	= $\sim y$	(invert all the bits)
0100	$= +4_{10}$	(and add one)

$$\begin{array}{r} 1011 \\ +\ \ \ \ 1 \\ \hline 1100 \end{array} \qquad \begin{array}{r} 0011 \\ +\ \ \ \ 1 \\ \hline 0100 \end{array}$$

Sometimes, people talk about "taking the two's complement" of a number. This is a confusing phrase, but it usually means to negate some number that's already in two's complement format.

Negating Numbers in 2's Complement

- To negate a number:
  - Complement each bit and then add 1.
- Example:

0100	$= +4_{10}$	(a positive number in 4-bit two's complement)
1011	=	(invert all the bits)
1100	$= -4_{10}$	(and add one)
0011	=	(invert all the bits)
0100	$= +4_{10}$	(and add one)

Converting 2's Complement to Decimal

4-bit unsigned

$2^3 \quad 2^2 \quad 2^1 \quad 2^0$

- Algorithm 1:
  - if negative, negate; then do unsigned binary to decimal
- Algorithm 2:
  - **Same as with n-bit unsigned binary**
    - **Except, the MSB is worth –(2^{n-1})**

4-bit 2's comp

MSB

$$-b_{n-1}2^{n-1} + \sum_{k=0}^{n-2} b_k 2^k$$

$1 \quad 1 \quad 0 \quad 0$

$-2^3 \quad 2^2 \quad 2^1 \quad 2^0$

- Example:

sign

$0100 = 4$

1100 = -4_{10} (a negative number in 4-bit two's complement)

0011
+0001
0100 – mag

-4_{10}

$1(-2^3) + 1\cdot(2^2)$

$= -8 + 4 = -4$

2’s Complement Negation

iclicker

- If 01011 is the 5-bit 2’s complement representation for 11, what is the 2’s complement representation for -11?
  - A: 11011
  - B: 10011
  - C: 10101
  - D: 01011
  - E: 10100

2's Complement Representation iclicker

- If 01001 is the 5-bit unsigned binary representation for 9, what is the 2's complement representation for 9?
  - A: 10110
  - B: 10111
  - C: 10101
  - D: 01001
  - E: 01010

-16 1

8 + 1 = 9

2’s Complement Negation

- If 01010 is the 5-bit representation for 10, what is the 2’s complement representation for -10?
  - A: 01011
  - B: 10101
  - C: 10110
  - D: 10111
  - E: 11010

Sign Extension

- In everyday life, decimal numbers are assumed to have an infinite number of 0's in front of them. This helps in "lining up" numbers.
- To subtract 231 and 3, for instance, you can imagine:

003 == 3

zero extension

```
   2 11
   231
 - 003
 -----
   228
```

4-bit 2's comp

$(-4)_{10}$

- This works for *positive* 2's complement numbers, but not *negative* ones.
- To preserve sign and value for negative numbers, we add more 1's.
- For example, going from 4-bit to 8-bit numbers:
  - 0101 (+5) should become 0000 0101 (+5). = 5
  - But 1100 (-4) should become 1111 1100 (-4).

1100

8-bit 2's comp

-128 32 8 4

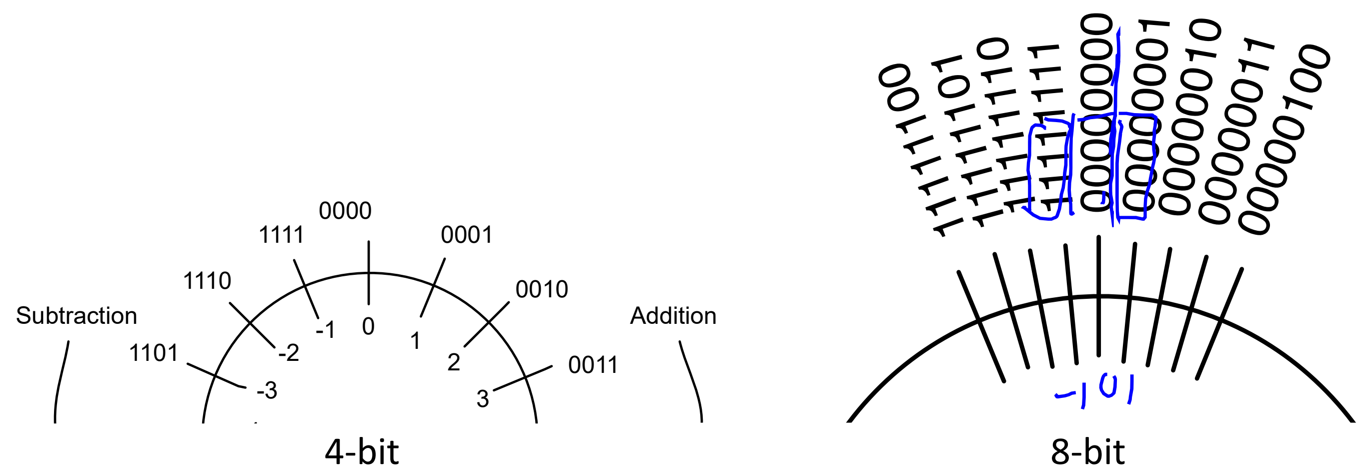
- The proper way to extend any signed binary number is to replicate the sign bit.

$-128+64+32+16+8+4=-4$

~~00001100 = 8+4 = 12~~

-128 84

Sign Extension, cont.



What you need to know for Lab 2.

Review: Bitwise Logical operations

```
                          5         5
unsigned char a = 0x55;   0 1 0 1   0 1 0 1

unsigned char b = 0x0f;   0 0 0 0 8 1 1 1 1
```

- Last time we introduced bit-wise logical operations:

unsigned char c = a | b; *(bit-wise OR)*

```
     0 1 0 1   0 1 0 1
OR   0 0 0 0   1 1 1 1
     ---------------------
     0 1 0 1   1 1 1 1
```

unsigned char d = a & b; *(bit-wise AND)*

```
     0 1 0 1   0 1 0 1
AND  0 0 0 0   1 1 1 1
     ---------------------
     0 0 0 0   0 1 0 1
```

unsigned char e = a ^ b; *(bit-wise XOR)*

```
     0 1 0 1   0 1 0 1
XOR  0 0 0 0   1 1 1 1
     ---------------------
     0 1 0 1   1 0 1 0
```

unsigned char n = ~a; *(bit-wise NOT)*

```
NOT  0 1 0 1   0 1 0 1
     ---------------------
     1 0 1 0   1 0 1 0
```

Bit-wise shifting

- When doing bit-wise logical operations, it can be useful to "shift" bits to the left or right within a word.

unsigned

0 0 1 0 (2)

0 1 0 0 (4)

1 0 0 0 (8)

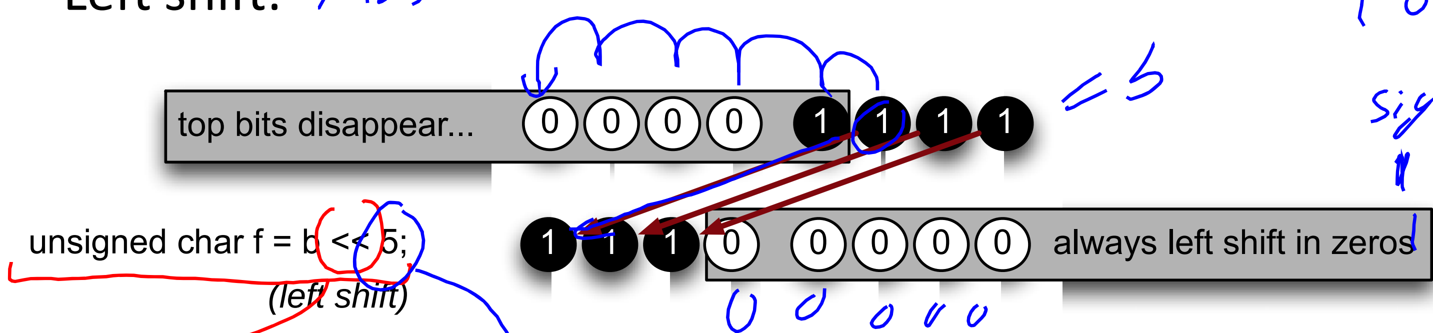
signed

1 1 1 0 (-2)

1 1 0 0 (-4)

- Left shift: toward MSb

$15 \times 2^5 = 15 \cdot 32 \rightarrow$ overflow



shift left

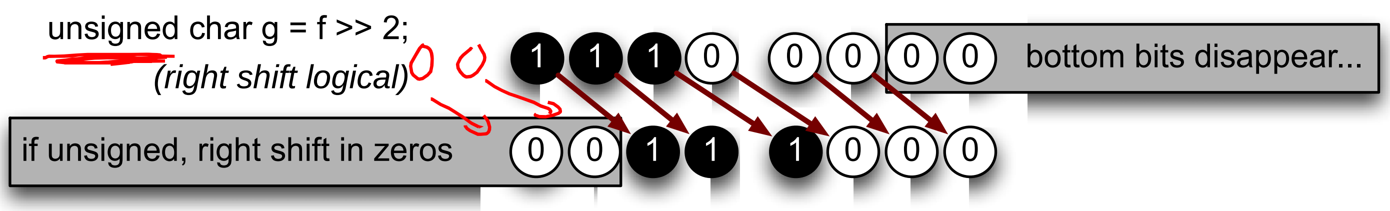
bit positions

We are shifting bits toward the most significant bit (MSB); we call this a left shift because we think of the MSB being on the left.

Bit-wise shifting, cont.

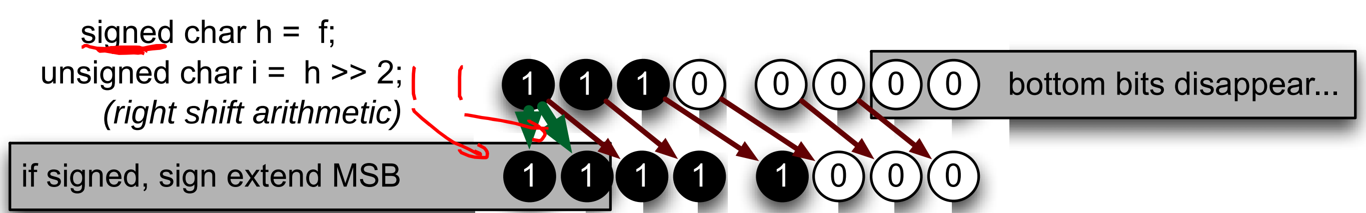
- Two kinds of right shift, depends on type of variable:
  - **Unsigned numbers**

zero-extend



  - **Signed numbers**

sign-extend



Note: x >> 1 not the same as x/2 for negative numbers; compare (-3)>>1 with (-3)/2

Bit-shifting has lower precedence than arithmetic but higher than bitwise operators

Precedence	Operator	Description
Higher	* / %	Multiplication, division, and modulus
	+ -	Addition and subtraction
	<< >>	Bitwise shifting
	& ^ |	Bitwise logical operators
Lower	&& ||	Logical operators

Useful for extracting bits

message

- We have the unsigned 8-bit word: $b_7b_6b_5b_4b_3b_2b_1b_0$
  0 1 1
- And we want the 8-bit word: 0 0 0 0 0 $b_5b_4b_3$
  0 1 1
  - i.e., we want to extract bits 3-5.

- We can do this with bit-wise logical & shifting operations
  - y = (x >> 3) & 0x7;

x	$b_7b_6b_5b_4b_3b_2b_1b_0$
x >> 3	$000b_7b_6b_5b_4b_3$
(x >> 3) & 0x7	& 00000111
	$00000b_5b_4b_3$

bit mask

Useful for merging two bit patterns

- We have 2 unsigned 8-bit words: $\underset{1\ 0\ 1\ 0\ 1\ 0\ 1\ 0}{}$ $a_7a_6a_5a_4a_3a_2a_1a_0 = A$

  0 1 0 1 0 1 0 1

  $b_7b_6b_5b_4b_3b_2b_1b_0 = B$

- And we want the 8-bit word: $\underbrace{a_7b_6a_5b_4a_3b_2a_1b_0} = C$

bit mask

$$C = (A \,\&\, 0xAA) \mid (B \,\&\, 0x55)$$

Bit-wise Logical & Shifting

iclicker

- We have 2 unsigned 8-bit words: x= $a_7a_6a_5a_4a_3a_2a_1a_0$
  y= $b_7b_6b_5b_4b_3b_2b_1b_0$
- And we want the 8-bit word: z= $a_3a_2a_1a_0b_3b_2b_1b_0$

  - A: z = (x >> 4) | (y << 4)
  - B: z = (x & 0x0f << 4) | (y & 0xf)
  - C: z = (x >> 4) | (y & 0xf)
  - D: z = (x & 0xf0) | (y & 0xf)
  - E: z = (x << 4) | (y & 0x0f)

Bit-wise Logical & Shifting

- We have 2 unsigned 8-bit words: x= $a_7a_6a_5a_4a_3a_2a_1a_0$

  y= $b_7b_6b_5b_4b_3b_2b_1b_0$
- And we want the 8-bit word: z= $b_3b_2b_1b_0a_3a_2a_1a_0$

  - A: z = (x & 0xf) | (y & 0x0f << 4)
  - B: z = (x & 0xf) | (y & 0xf0)
  - C: z = (x >> 4) | (y << 4)
  - D: z = (x & 0x0f) | (y << 4)
  - E: z = (x & 0xf) | (y >> 4)

Binary addition with 2's Complement

- You can add two's complement numbers just as if they are unsigned numbers.
  - Recall, this was the whole reason for this representation

carry-out

1	1	0	0	0		
	0	1	0	1	1	11
+	1	1	1	0	0	+ (-4)
	0	0	1	1	1	+7

no overflow

Subtraction

- We can implement subtraction by negating the 2nd input and then adding:

	0	1	1	0	1	13
-	0	1	0	1	0	- 10

0 0 0 1

1 0 1 0 1

\+ 1

1 0 1 1 0 (-10)

carry-out

1	1	1	0	0		
	0	1	1	0	1	13
+	1	0	1	1	0	+ (-10)
	0	0	0	1	1	+3

no overflow

Why does this work?

- For n-bit numbers, the negation of B in two's complement is 2^n - B (this is alternative way of negating a 2's-complement number).

$$\begin{aligned} A - B &= A + (-B) \\ &= A + (2^n - B) \\ &= (A - B) + 2^n \end{aligned}$$

- If $A \geq B$, then (A - B) is a positive number, and 2^n represents a carry out of 1. Discarding this carry out is equivalent to subtracting 2^n, which leaves us with the desired result (A - B).
- If $A < B$, then (A - B) is a negative number -(B – A) and we have 2^n – (B - A). This corresponds to the desired result, (A - B), in two's complement form.

2's Complement Subtraction

$$\begin{array}{rcccc} & 1 & 1 & 0 & 1 \\ - & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

- A: 0111
- B: 0011
- C: 1000
- D: 0101
- E: 1001

2's Complement Subtraction

	1	1	1	0	(-2)
-	0	0	1	1	- 3

- A: 1011
- B: 1010
- C: 0001
- D: 0101
- E: 1111

$-8+2+1=-5$

no overflow

Overflow Review

- Recall that when we add two numbers the result may be larger than we can represent.

(in 5b 2's complement we can represent -16 to +15)

overflow

	0	1	0	1	1	Augend	(11)
+	0	1	1	1	0	Addend	(14)
	1	1	0	0	1	Sum	(-7)

= 25

- The same thing can happen when we add negative numbers.

overflow

	1	1	0	0	1	Augend	(-7)
+	1	0	1	0	0	Addend	(-12)
	0	1	1	0	1	Sum	(13)

= -19

“Carry-out” is a procedure, “Overflow” is an interpretation

Carry-out

- Occurs at every bit-position
- The process of moving larger numbers to higher bit positions
- Focuses on bit-wise operations

Overflow

- Can only be seen after completing an entire mathematical operation
- When the interpretation of a set of bits does not match the expected value after a mathematical operation
- Focuses on representational range (i.e., 4 bits represent 0-15)

How can we know if overflow has occurred?

- The easiest way to detect signed overflow is to look at all of the sign bits.

```
    0100   (+4)        1100   (-4)
  + 0101   (+5)      + 1011   (-5)
  ------------       ------------
    1001   (-7)        0111   (+7)
```

- Overflow occurs only in the two situations above:
  - If you add two *positive* numbers and get a *negative* result.
  - If you add two *negative* numbers and get a *positive* result.
- Overflow cannot occur if you add a positive number to a negative number. Do you see why?

Overflow clicker

4-bit unsigned integers

```
  1 1 1 0
+ 0 0 1 1
---------
```

4-bit 2's comp integers

```
  1 1 1 0
+ 0 0 1 1
---------
```

a) Neither overflows
b) Only unsigned addition overflows
c) Only 2's comp addition overflows
d) Both overflow

Overflow

In which circumstance can overflow not occur?

- A: subtracting a positive number from a negative number
- B: subtracting a negative number from zero
- C: adding two negative numbers
- D: subtracting a negative number from a positive number
- E: subtracting a negative number from a negative number

Overflow in software (e.g., Java programs)

```
public class overflow {
 public static void main(String[] args) {
      int i = 0;
      while (i >= 0) {
         i++;
      }
      System.out.println("i = " + i);
      i--;
      System.out.println("i = " + i);
      i++;
      System.out.println("i = " + i);
 }}
```

Output:
i = -2147483648 2^{31}
i = 2147483647 $2^{31}-1$
i = -2147483648