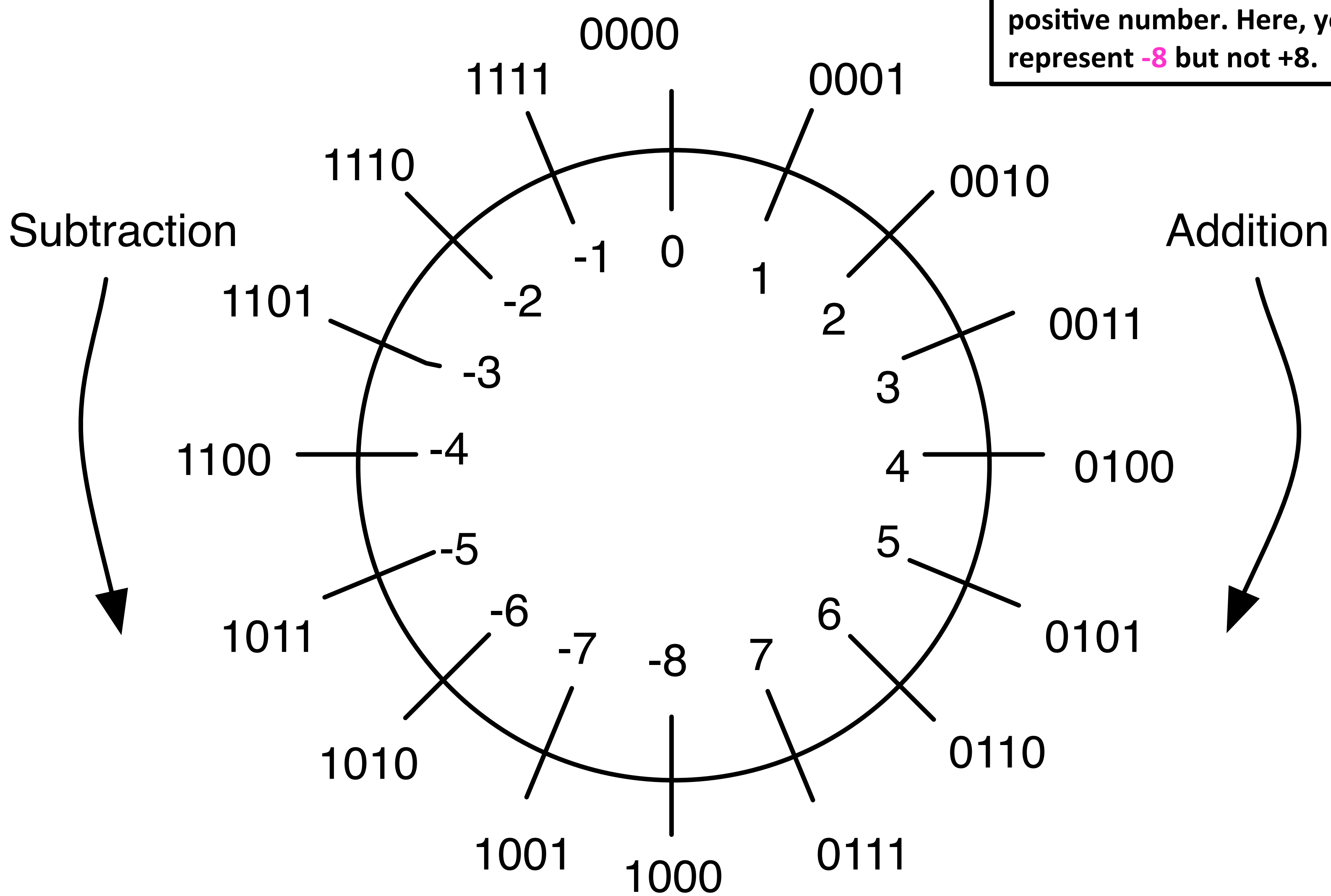
Number Systems II:
2’s Complement, Arithmetic, Overflow, & Writing Bit-wise Logical & Shifting Code

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.

- **Example:**

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

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.

Converting 2's Complement to Decimal

- **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 $-(2^{n-1})$**

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

- **Example:**

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

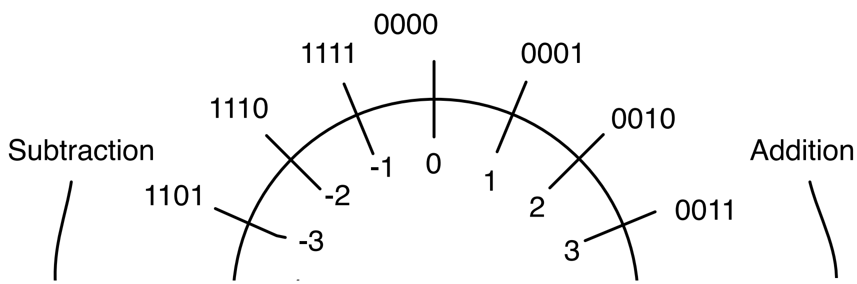
Sign Extension

- **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:**

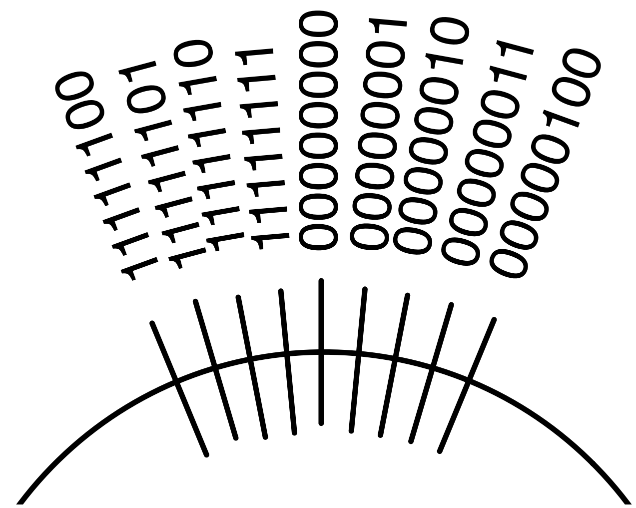
$$\begin{array}{r} 231 \\ -\ 003 \\ \hline 228 \end{array}$$

- **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).
  - But 1100 (-4) should become 1111 1100 (-4).
- **The proper way to extend any signed binary number is to replicate the sign bit.**

Sign Extension, cont.



4-bit



8-bit

What you need to know for Lab 2.

Review: Bitwise Logical operations

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

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

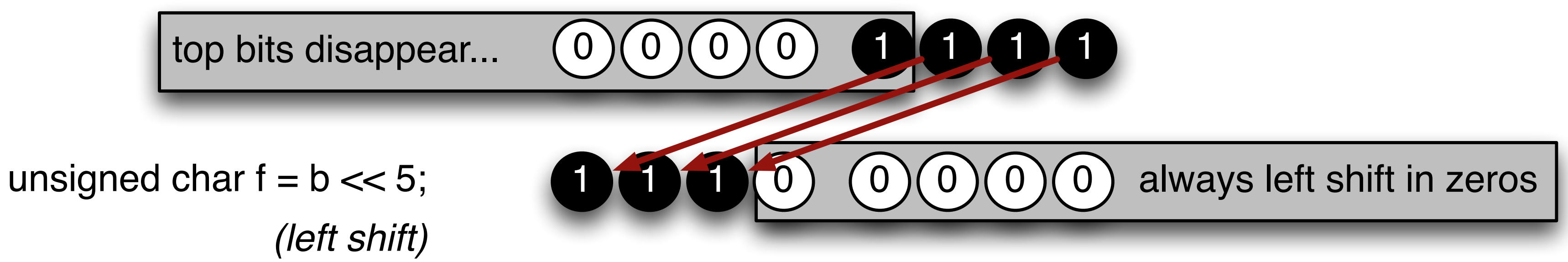
unsigned char n = ~a; *(bit-wise NOT)*

```
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.**

- **Left shift:**

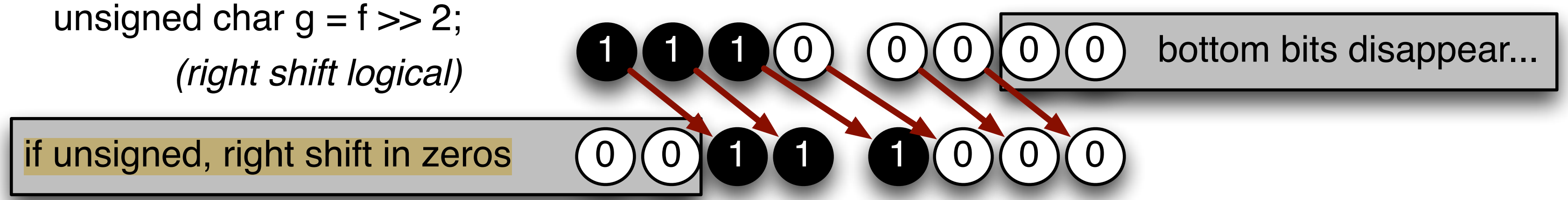


We are shifting bits toward the most significant bit (MSB); we call this a left shift because we think of the MSB being on the left.

Bit-wise shifting, cont.

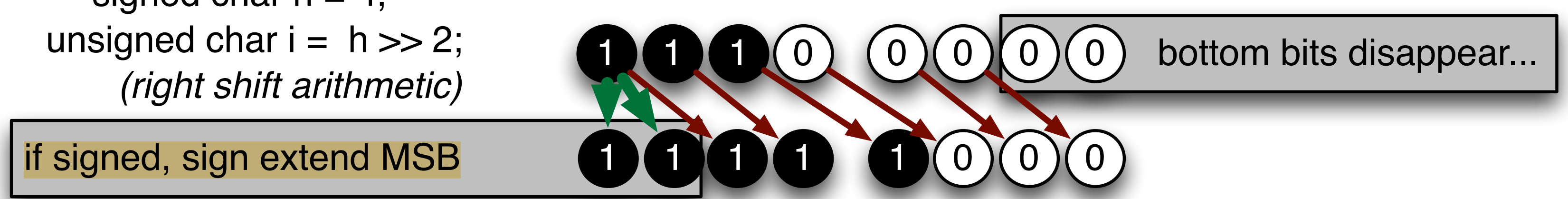
- **Two kinds of right shift, depends on type of variable:**
  - **Unsigned numbers**

unsigned char g = f >> 2;
(right shift logical)



  - **Signed numbers**

signed char h = f;
unsigned char i = h >> 2;
(right shift arithmetic)



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

Useful for extracting bits

- **We have the unsigned 8-bit word:** $b_7b_6b_5b_4b_3b_2b_1b_0$
- **And we want the 8-bit word:** $0\ 0\ 0\ 0\ 0\ b_5b_4b_3$
  - 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	
(x >> 3) & 0x7	

Useful for merging two bit patterns

- We have 2 unsigned 8-bit words: $a_7a_6a_5a_4a_3a_2a_1a_0$
  $b_7b_6b_5b_4b_3b_2b_1b_0$
- And we want the 8-bit word: $a_7b_6a_5b_4a_3b_2a_1b_0$

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

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

Subtraction

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

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

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 and we have $2^n - (A - B)$. This corresponds to the desired result, $-(A - B)$, in two's complement form.

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)

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

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

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

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)
+	0101	(+5)
	1001	(-7)

	1100	(-4)
+	1011	(-5)
	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 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