CSCI 2330

TODO

- Assignment 0 is assigned (due September 11)
- Assignment 1 will be assigned this week

LA hours

- Xan Middents
- Searles 224
  - Monday 6-8pm
  - Thursday 7-9pm

Overview for today

- Recap of signed numbers
- Logical operators
- Sign extension

Bitwise vs. logical operators

Bitwise operator	Logical operator
&	&&
|	||
^	
~	!

Difference

- Bitwise – operate across each bit independently
- Logical – reduce each operand to a 0 (false) or 1 (true) first
  - Then the traditional bit operation is applied to that single bit

Comparison operators

- Recall the convention that:
  - 0 is false
  - 1 is true
- `x == y`: 1 if x equals y, else 0
- `x != y`: 0 if x equals y, else 1
- For two's complement integers, x == y if they have identical bits
  - Note: this is why "negative zero" is unideal

Logical operations

```
    !x → (x == 0)
x && y → (x != 0) & (y != 0)
x || y → (x != 0) | (y != 0)
```

For these…

```
x && y → (x != 0) & (y != 0)
x || y → (x != 0) | (y != 0)
```

Are these equivalent?

```
x && y → (x == 1) & (y == 1)
x || y → (x == 1) | (y == 1)
```

Are th[illegible]valent?

```
x && y → (
x || y →
```

Examples

- 0x0F & 0xF0 → ?

Examples

- 0x0F & 0xF0 → 0

Examples

- 0x0F && 0xF0 → ?

Examples

- 0x0F && 0xF0 → 1

Examples

- 0x0F | 0xF0 → ?

Examples

- 0x0F | 0xF0 → 0xFF

Examples

- 0x0F || 0xF0 → ?

Examples

- 0x0F || 0xF0 → 1

Examples

- 0x0F ^ 0xF0 → ?

Examples

- 0x0F ^ 0xF0 → 0xFF

Examples

- 0xFF & 0xF0 → ?

Examples

- 0xFF & 0xF0 → 0xF0

Examples

- 0xFF | 0xF0 → ?

Examples

- 0xFF | 0xF0 → 0xFF

Examples

- 0xFF ^ 0xF0 → ?

Examples

- 0xFF ^ 0xF0 → 0x0F

Two special operators

- Bit shifts: >>, <<

Two special operators

- Bit shifts: >>, <<
- integer >> # of bits to shift right by
- integer << # of bits to shift left by

Examples

- 0x0F << 4 → ?

Examples

- 0x0F << 4 → 0xF0

Examples

- 0x0F >> 4 → ?

Examples

- 0x0F >> 4 → 0

What do shifts do arithmetically?

- $x << n == x * 2^n$
- $x >> n == x / 2^n$ (floor division)

Examples

- 7 << 1 → ?

Examples

- 7 << 1 → 14

Examples

- 1 << 2 → ?

Examples

- 1 << 2 → 4

Examples

- 8 >> 2 → ?

Examples

- 8 >> 2 → 2

Unsigned wraparound

```
x =  11111110 (254)
x++
x == 11111111 (255)
x++
x == 00000000 (0)
```

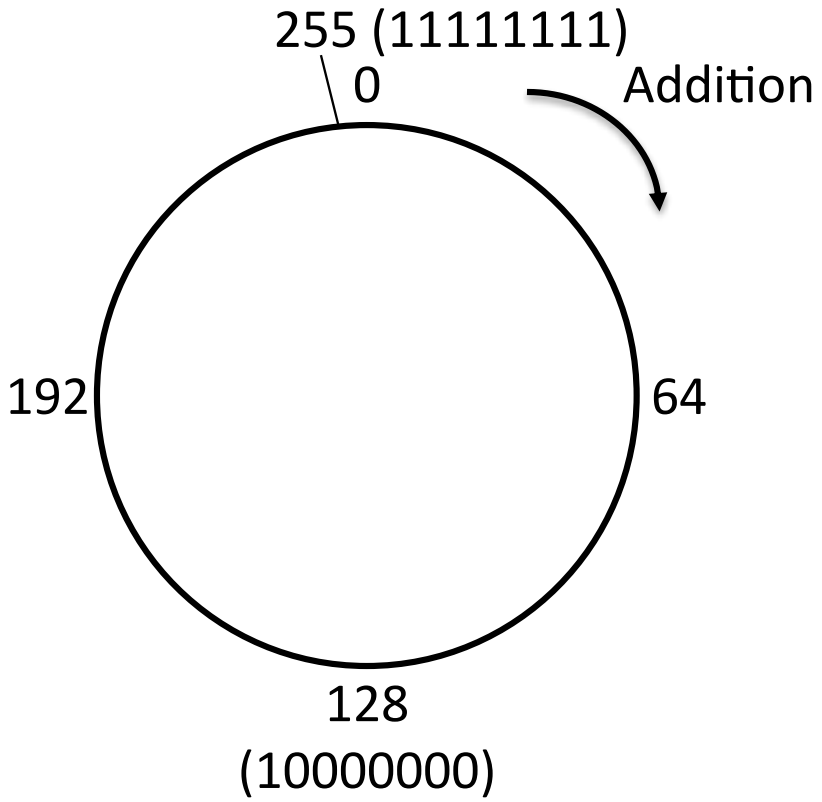
Unsigned wraparound

```
x =  11111110 (254)
x++
x == 11111111 (255)
x++
x == 00000000 (0)
```

```
x =  00000001 (1)
x--
x == 00000000 (0)
x--
x == 11111111 (255)
```

Unsigned wraparound

```
x =  11111110 (254)

x++

x == 11111111 (255)

x++

x == 00000000 (0)
```

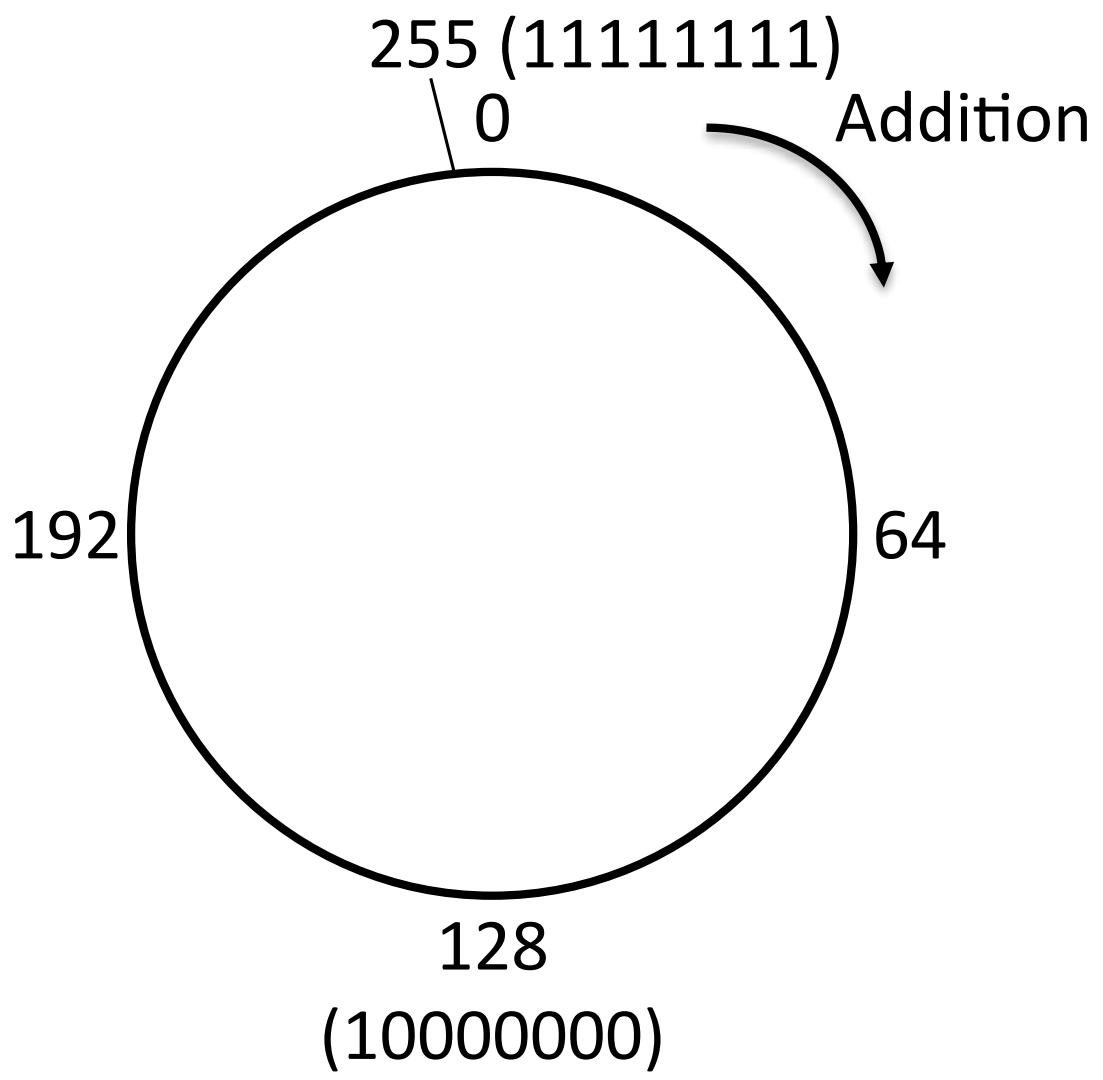


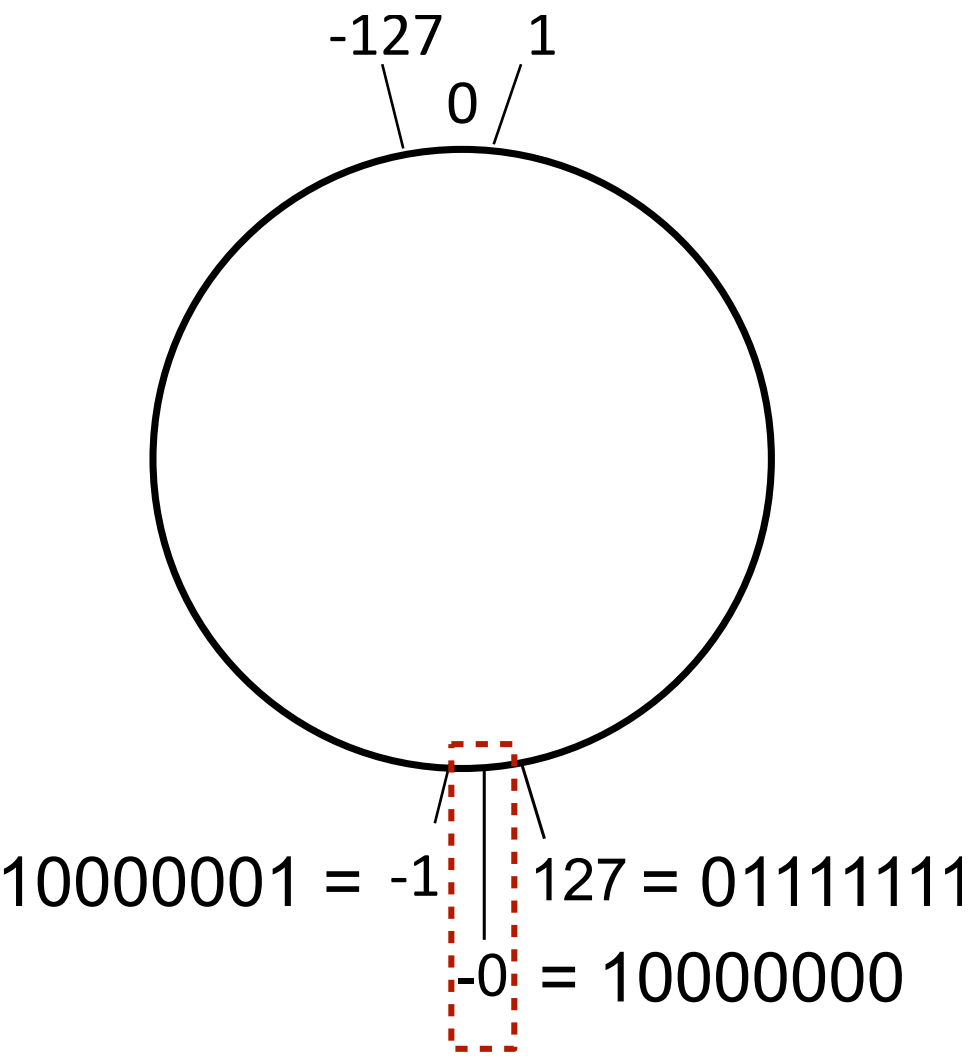
Signed magnitude approach

- Simplest way: sign bit
- Say you have 8-bit number
- Use the highest order bit to represent sign
- Traditionally, 0 positive, 1 negative
- 00000101 → 5
- 10000101 → -5

Signed magnitude approach

- With 8 bits:
  - What is the largest number we can represent?
    - 01111111 == 127
  - What is the smallest number?
    - 11111111 == -127


255 (11111111)
0
Addition
192
64
128
(10000000)

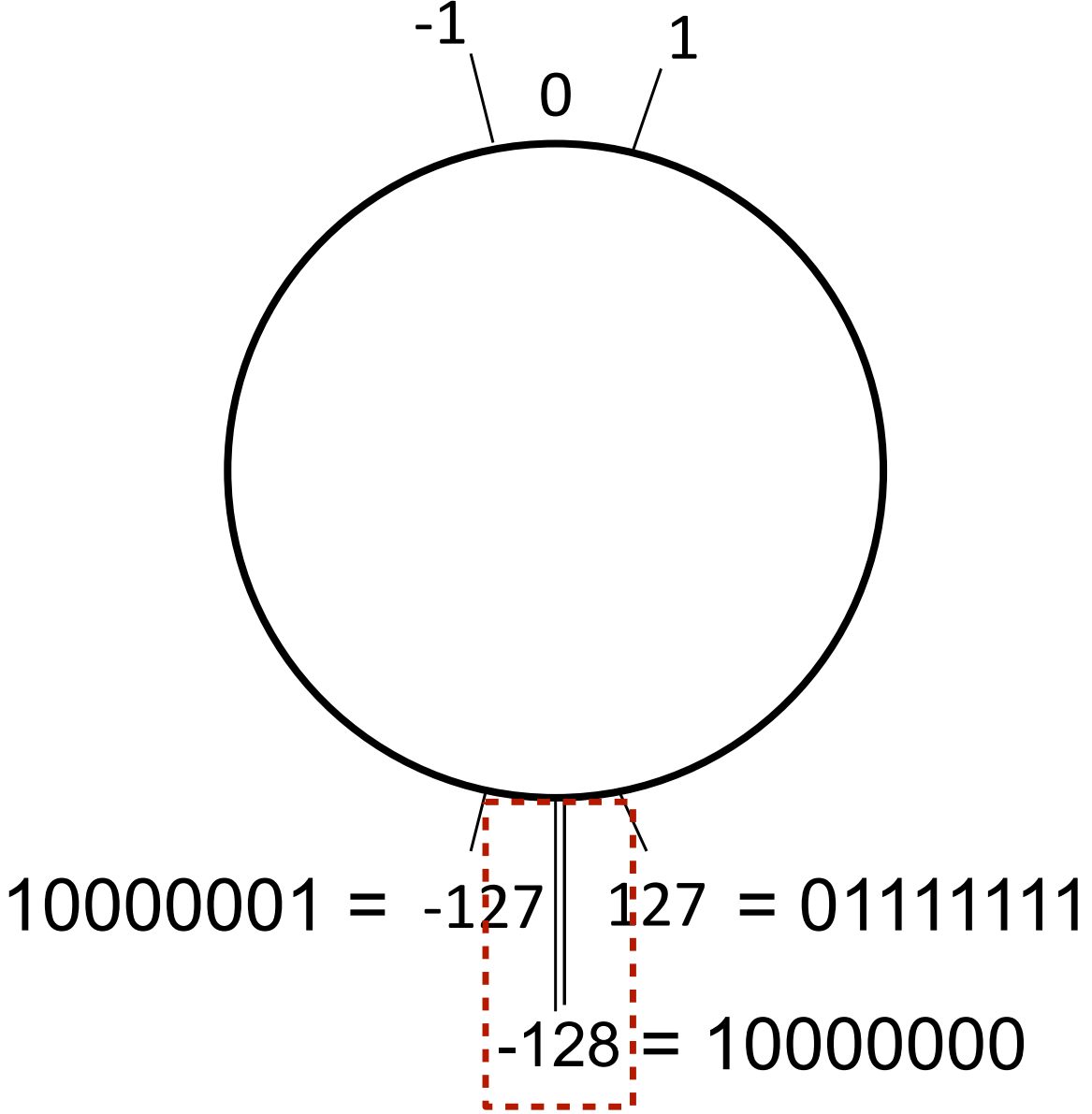

-127
1
0
10000001 = -1
127 = 01111111
-0 = 10000000

Pros and cons

- Pros
  - Simple to understand
  - Simple to flip sign
- Cons
  - Positive and negative zero!?
  - Grade school arithmetic is broken

Two’s complement

- $-x = 2^n - x$
- $-x = \sim x + 1$


-1
0
1
10000001 = -127
127 = 01111111
-128 = 10000000

Two’s complement

- Pros
  - Simple to identify sign
  - Simple to flip sign
  - Unsigned and signed use same arithmetic
  - Only one zero
- Con
  - Not as simple for humans to understand
  - INT_MIN…

One weird thing…

- Largest number in two’s complement? (8 bits)

  ????????

One weird thing…

- Largest number in two’s complement? (8 bits)

  01111111 = 127

One weird thing…

- Largest number in two’s complement? (8 bits)

  01111111 = 127

- Smallest number?

  ????????

One weird thing…

- Largest number in two's complement? (8 bits)

  01111111 = 127

- Smallest number?

  -127? What is that in binary?

  ????????

One weird thing…

- Largest number in two's complement? (8 bits)

  01111111 = 127

- Smallest number?

  -127? What is that in binary?

  ~01111111 + 1 == 10000000 + 1 == 10000001

One weird thing…

- Largest number in two's complement? (8 bits)

  01111111 = 127

- Smallest number?

  -127? What is that in binary?

  ~01111111 + 1 == 10000000 + 1 == 10000001

One weird thing…

- Largest number in two's complement? (8 bits)

  01111111 = 127

- Smallest number?

  -127? What is that in binary?

  ~01111111 + 1 == 10000000 + 1 == 10000001

  Subtract 1 from that and get… ????????

One weird thing…

- Largest number in two's complement? (8 bits)

  01111111 = 127

- Smallest number?

  -127? What is that in binary?

  ~01111111 + 1 == 10000000 + 1 == 10000001

  Subtract 1 from that and get… 10000000 == -128!?

- This smallest number is called `INT_MIN` or T_{min}

One weird thing…

- INT_MIN (T_{min}) has some unusual properties…
- What is -(-128)?

One weird thing…

- INT_MIN (T_{min}) has some unusual properties…
- What is -(-128)?
- `  ~10000000 + 1`

One weird thing…

- INT_MIN (T_{min}) has some unusual properties…
- What is -(-128)?
- `  ~10000000 + 1`
- `= ???????? + 1`

One weird thing…

- INT_MIN (T_{min}) has some unusual properties…
- What is -(-128)?
- `  ~10000000 + 1`
- `= 01111111 + 1`
- `= ????????`

One weird thing…

- INT_MIN (T_{min}) has some unusual properties…
- What is -(-128)?
- `  ~10000000 + 1`
- `= 01111111 + 1`
- `= 10000000`

Be careful when negating

- Make sure that your number isn’t INT_MIN!

Modern computers

- Nearly all modern computers use two’s complement to represent signed integers

Signed vs. unsigned

- Remember that not all numbers are signed
- Everything we learned about unsigned numbers is still valid and used by computers too

C

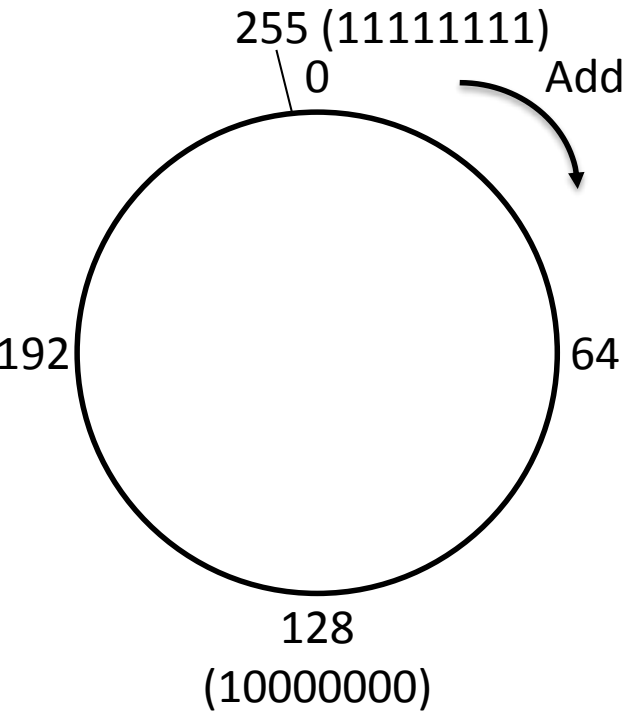
```c
#include <stdio.h>

int main(void) {
    unsigned int x = 0xdeadbeef;
    int y = 0xdeadbeef; // equivalent to "signed int"
    printf("%u %d\n", x, y);
    return 0;
}
```

C

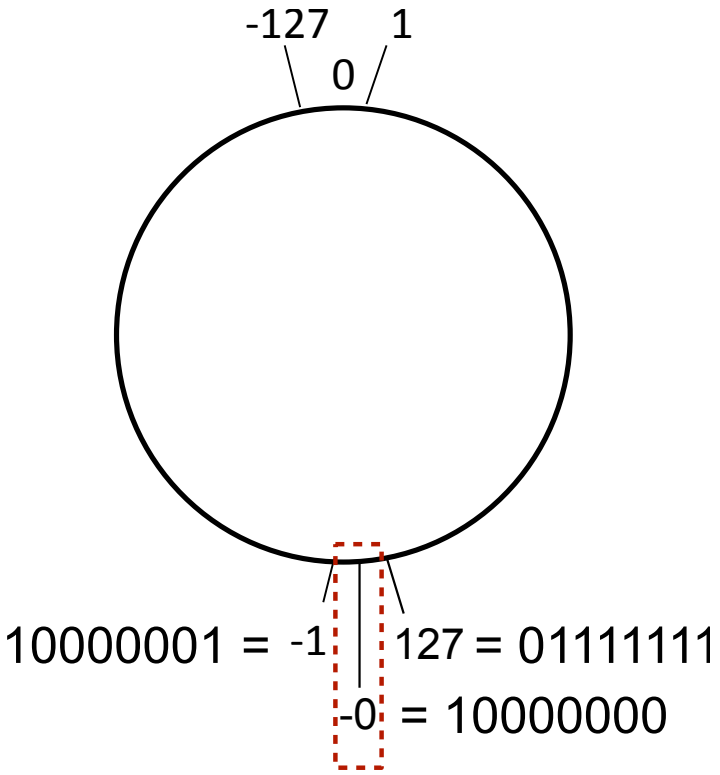
```c
#include <stdio.h>

int main(void) {
    unsigned int x = 0xdeadbeef;
    int y = 0xdeadbeef; // equivalent to "signed int"
    printf("%u %d\n", x, y);
    return 0;
}
```

```
jeffrey@polaris$ gcc main.c -o main
jeffrey@polaris$ ./main
3735928559 -559038737
```

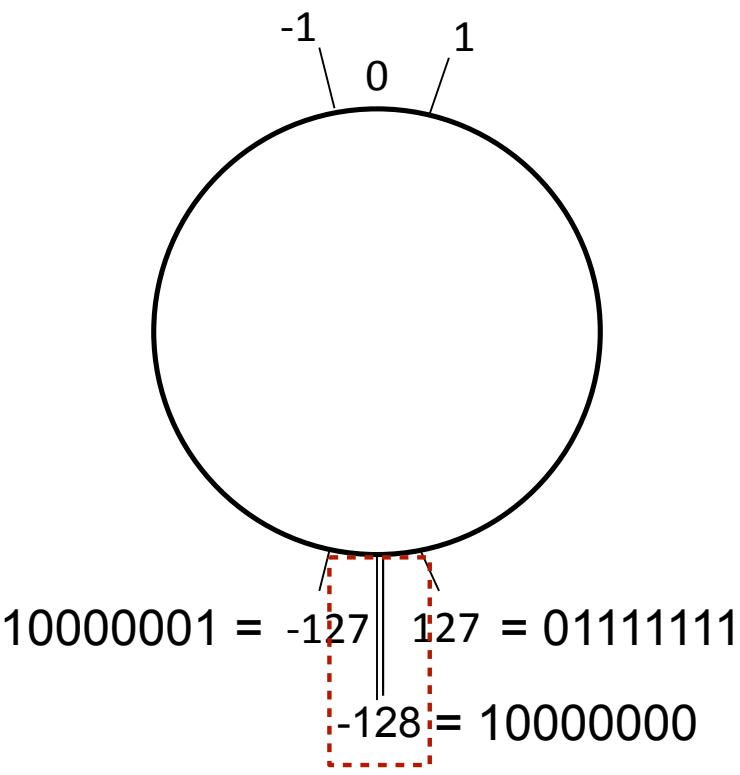
Data representation exercise



Unsigned



Signed Magnitude

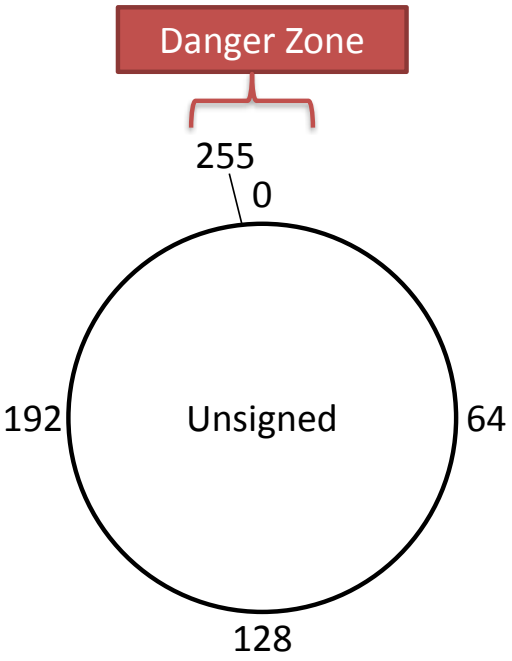


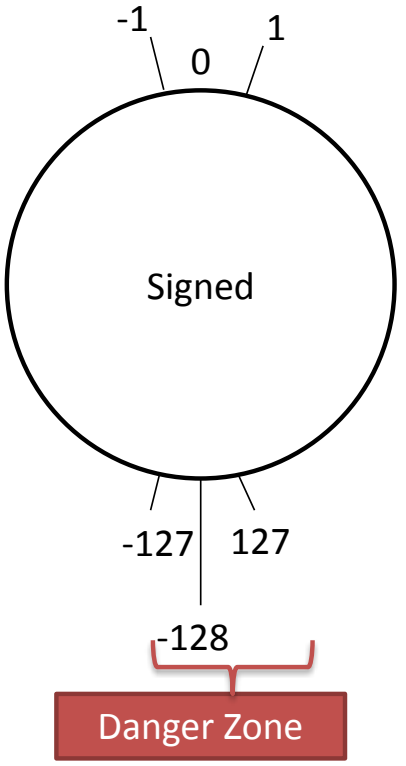
Two’s Complement

Two’s complement overflow

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Two’s complement overflow





Two’s complement overflow

```
x =  01111110 (126)
x++
x == 01111111 (127)
x++
x == 10000000 (-128)
```

```
x =  10000001 (-127)
x--
x == 10000000 (-128)
x--
x == 01111111 (127)
```

Other triggers of overflow/underflow

- Adding of two large numbers
- Adding of two very low (negative) numbers

Can hardware detect overflow?

- Unsigned overflow: carry out bit

```
  1111 (15)
+ 0001 (1)
 10000 (0)
```

Can hardware detect overflow?

- Signed overflow: high bit consistency
- Observation: overflow occurs when adding two positive numbers and the result is negative or two negative numbers and the result is positive

Can hardware detect overflow?

- Signed overflow: high bit consistency

```
 0111 (7)
+ 0001 (1)
 1000 (-8)
```

- A, B, S: high bits of addends and sum
- Overflow occurs when ~(A ^ B) & (A ^ S)
- “A is same as B but different than S.”

So hardware can detect it!

- Can’t really get at this from even C though
- Later we will learn how to do it via machine code

Let’s look more at C

```c
#include <stdio.h>

int main(void) {
    unsigned int x = 0xdeadbeef;
    int y = 0xdeadbeef; // equivalent to "signed int"
    printf("%u %d\n", x, y);
    return 0;
}
```

```
jeffrey@polaris$ gcc main.c -o main
jeffrey@polaris$ ./main
3735928559 -559038737
```

Which C types are signed?

char 🥲	signed char ✅	unsigned char ❌
short ✅	signed short ✅	unsigned short ❌
int ✅	signed int ✅	unsigned int ❌
long ✅	signed long ✅	unsigned long ❌

```
signed x;   // alias for 'signed int x' aka 'int x'
unsigned x; // alias for 'unsigned x'
```

Integer literals

- Signed:
  - 12345 (int)
  - 12345L (long)
- Unsigned:
  - 12345u (unsigned int)
  - 12345uL (unsigned long)
- 0xDEADBEEFuL

Type conversions

- How do we convert types when we are programming?

Type conversions

- Explicit type conversion: casting

```
unsigned int x = 0xdeadbeefu;
unsigned long y = (unsigned long) x;
```

- Implicit type conversion: integer promotion

```
unsigned int x = 4294967295u;
unsigned long y = 109951162775uL;
(x + y) // what type is this???
```

Type conversions

- Explicit type conversion: casting

```
unsigned int x = 0xdeadbeefu;
unsigned long y = (unsigned long) x;
```

- Implicit type conversion: integer promotion

```
unsigned int x = 4294967295u;
unsigned long y = 109951162775uL;
(x + y) // x gets promoted to unsigned long
```

Promotion hierarchy

1. unsigned long
2. long
3. unsigned int
4. int
5. (Everything narrower undergoes promotion to at least int)

Promotion steps

- Promote both operands to at least int
- See which is “higher ranked”
- Promote other to that rank if necessary

Type conversions

```
unsigned int x = 2023406814;
unsigned long y = 32374509039;

(x + y) // what is the type?
```

Type conversions

```
unsigned int x = 2023406814;
unsigned long y = 32374509039;

(x + y) // unsigned long
```

int promotion

```
unsigned char a = 200;
unsigned char b = 100;

(a + b) // expression is of type int
```

Widening a type

- How do we determine the new bits?
- We determine them to preserve its value
- If the original type was unsigned, *zero extend*
- If the original type was signed, *sign extend*

unsigned char → short

- `00000000 → 0000000000000000`
- `11111111 → 0000000011111111`
- `10000000 → 0000000010000000`
- `01111111 → 0000000001111111`

signed char → short

- 00000000 → 0000000000000000
- 11111111 → 1111111111111111
- 10000000 → 1111111110000000
- 01111111 → 0000000001111111

Also applies to right bit shifting

- Unsigned:
- `00000000 >> 4 → 00000000`
- `11111111 >> 4 → 00001111`
- `10000000 >> 4 → 00001000`
- `01111111 >> 4 → 00000111`

Also applies to right bit shifting

- Signed:
- `00000000 >> 4 → 00000000`
- `11111111 >> 4 → 11111111`
- `10000000 >> 4 → 11111000`
- `01111111 >> 4 → 00000111`

Why sign-extend signed right shifts?

- To preserve this arithmetic property for right bitshifts of signed numbers:
- $x >> n == x / 2^n$ (floor division)
- We call a sign-extended right bitshift an *arithmetic shift*.

When types are equally wide

- E.g., signed int → unsigned int
- Or unsigned int → signed int
- Do not change bits, *just interpretation*

Surprising type conversions

```
unsigned int x = 500;
int y = -1000;

(x >= y) // true or false?
```

Promotion hierarchy

1. unsigned long
2. long
3. unsigned int
4. int
5. (Everything narrower undergoes promotion to at least int)

Surprising type conversions

```
unsigned int x = 500;
int y = -1000;

(x >= y) // true or false?
```

Surprising type conversions

```
unsigned int x = 500;
int y = -1000;

(x >= y) // false 🥲
```

Integer Logic Exercises

Promotion hierarchy:

1) unsigned long
2) long
3) unsigned int
4) int
5) Everything narrower undergoes int promotion

TODO

- Assignment 0 is assigned (due September 11)
- Assignment 1 will be assigned this week