Using Different Bases in Code

How do we define numbers in our code (C, Java, Python, ...)?

	Old Languages	New Languages
binary		
octal		
decimal		
hexadecimal		

Bitwise Operations

CS 2130: Computer Systems and Organization 1

September 5, 2025

Announcements

- Quiz 1 opens this afternoon, due Sunday night
- Homework 1 due September 15

Representing Negative Integers

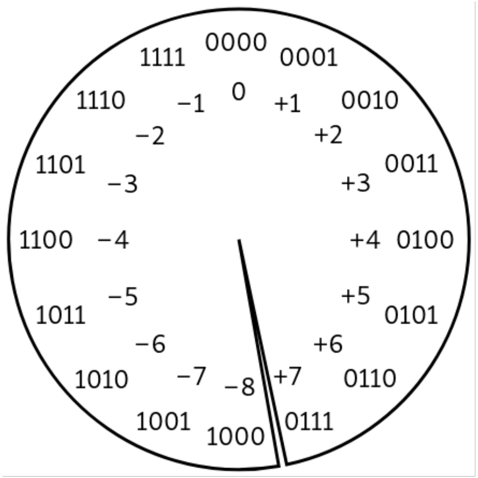
Computers store numbers in fixed number of wires

- Ex: consider 4-digit decimal numbers
- Throw away the last borrow:
  - 0000 - 0001 = 9999 == -1
  - 9999 - 0001 = 9998 == -2
  - Normal subtraction/addition still works
  - Ex: -2 + 3
- This works the same in binary

Two's Complement

This scheme is called **Two's Complement**

- More generically, a *signed* integer
- There is a break as far away from 0 as possible
- First bit acts vaguely like a minus sign
- Works as long as we do not pass number too large to represent



Two's Complement

Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

$$11010011$$

What is its value in decimal?

Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

$$11010011$$

What is its value in decimal?

1. Flip all bits
2. Add 1

Addition

```
  01001010
+ 01111100
----------
```

Subtraction

```
  01001010
- 01111100
----------
```

Operations

So far, we have discussed:

- Addition: $x + y$
  - Can get multiplication
- Subtraction: $x - y$
  - Can get division, but more difficult
- Unary minus (negative): $-x$
  - Flip the bits and add 1

Operations (on Integers)

Bit vector: fixed-length sequence of bits (ex: bits in an integer)

- Manipulated by bitwise operations

Bitwise operations: operate over the bits in a bit vector

- Bitwise not: `~x` - flips all bits (unary)
- Bitwise and: `x & y` - set bit to 1 if x, y have 1 in same bit
- Bitwise or: `x | y` - set bit to 1 if either x or y have 1
- Bitwise xor: `x ^ y` - set bit to 1 if x, y bit differs

Example: Bitwise AND

```
  11001010
& 01111100
----------
```

Example: Bitwise OR

```
  11001010
| 01111100
----------
```

Example: Bitwise XOR

```
  11001010
^ 01111100
----------
```

Your Turn!

What is: `0x1a ^ 0x72`

Operations (on Integers)

- Logical not: $!x$
  - $!0 = 1$ and $!x = 0, \forall x \neq 0$
  - Useful in C, no booleans
  - Some languages name this one differently

Operations (on Integers)

- Left shift: $x << y$ - move bits to the left
  - Effectively multiply by powers of 2
- Right shift: $x >> y$ - move bits to the right
  - Effectively divide by powers of 2
  - Signed (extend sign bit) vs unsigned (extend 0)

Left Bit-shift Example

```
01011010 << 2
```

Right Bit-shift Example

`01011010 >> 3`

Bit-shift

Computing bit-shift effectively multiplies/divides by powers of 2

Consider decimal:

$$2130 \ll_{10} 2 = 213000 = 2130 \times 100$$

$$2130 \gg_{10} 1 = 213 = 2130 \;/\; 10$$

Right Bit-shift Example 2

`11001010 >> 1`

Right Bit-shift Example 2

For **signed** integers, extend the sign bit (1)

- Keeps negative value (if applicable)
- Approximates divide by powers of 2

```
11001010 >> 1
```

Bit fiddling example