Bitwise Operations

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcement

- Homework 1 due February 2, 2026

Two's Complement

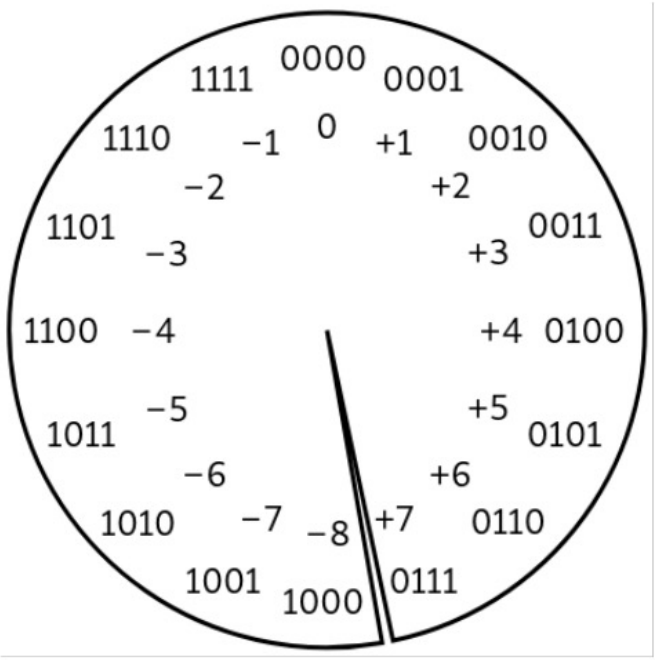
The scheme is called Two's Complement

Why do we need Two's Complement?

- We want the computer to represent both positive and negative numbers.
- And we want addition and subtraction to use the *same* hardware (just one adder), instead of building a separate "subtractor."

How does it work?

- The **leftmost bit (MSB)** is treated as negative.
  - In normal binary: the leftmost bit is +128 (for 8-bit).
  - In two's complement: the leftmost bit is −128.
- That's why $10000000_2 = -128$ instead of +128.



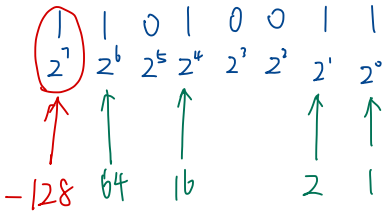
Values of Two's Complement Numbers

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

11010011

What is its value in decimal?

Method 1:



$-128+64+16+2+1=-45$

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 1 0 1 0 0 1 1
128 64 32 16 8 4 2 1

1. Flip all bits
2. Add 1

00101100
32 16 8 4 2

\+ 1

00101101

32+8+4+1 = 45 ⇒ −45

−128 +64+16+2+1 = −45

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

① Flip all bits:

11010011 → 00101100

② Add 1: 00101100 + 1 = 00101101

③ what is 00101101 in decimal?

128 64 32 16 8 4 2 1

00101101 = 32+8+4+1 = 45

④. So the value of this negtive number is −45.

Values of Two's Complement Numbers

Why "invert the bits and add 1"?

- Because in 8 bits, we have 256 total values (0–255).
- A negative number is stored as 256 − (its absolute value). $2^n - |x|$
- The "invert + 1" trick is just a fast way to compute that.

invert bits: 1 0 0 0 0 0 0 0 0

0 1 1 1 1 1 1 1 1 $\Rightarrow 255$ $(2^n - 1)$

$255 + 1 = 256$

(This is the definition of negative numbers in 2's complement).

$-2^{n-1} \leq x \leq 2^{n-1} - 1$

Values of Two's Complement Numbers

Why "invert the bits and add 1"?

- Because in 8 bits, we have 256 total values (0–255).
- A negative number is stored as 256 − (its absolute value).
- The "invert + 1" trick is just a fast way to compute that.

$-a = \sim a + 1$

$0 = \sim a + 1 + a$

$-1 = \sim a + a$

Values of Two's Complement Numbers

Consider the following decimal number:

-117

What is its value in 8-bit binary binary?

method 1:

+117: 0 1 1 1 0 1 0 1

convert: 1 0 0 0 1 0 1 0

"+1": 1 0 0 0 1 0 1 1

method 2:

256 − 117 = 139

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	0	0	1	0	1	1

139 − 128 = 11 11 − 8 = 3

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
----------
  01001000
```

Example: Bitwise OR

```
  11001010
| 01111100
----------
  11111110
```

Example: Bitwise XOR

```
  11001010
^ 01111100
----------
  10110110
```

Your Turn!

What is:

0x1a ^ 0x72

```
  00011010
^ 01110010
----------
  01101000
```

= 0x68

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

0 means false
Any non-zero value means true.

C does not have a built-in Boolean type in older standards.
Logical expressions return integers (0 or 1)

Example:
```
if(!ptr) {
    // ptr is NULL
}
```

Python: not
Java/C/C++: !
Bash: !

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

$$01011010 << 2$$

01011010 00

(01011010×2^2)

Right Bit-shift Example

$$01011010 >> 3$$

000 01011

(01011010/2^3)

Bit-shift

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

Consider decimal:

$$2130 <<_{10} 2 = 213000 = 2130 \times 100$$

$$2130 >>_{10} 1 = 213 = 2130 / 10$$