Bitwise tips :

Left shift (<<):

- shifting left is equivalent to multiplication by power of 2, for example:
  - $6 << 1 = 6 \times 2$
  - $6 << 3 = 6 \times (2^3) = 6 \times 8 = 48$

Right shift (>>):

- shifting right is equivalent to division by power of 2, for example:
  - $12 >> 1 = 12 : (2^1) = 6$
  - $12 >> 2 = 12 : (2^2) = 3$

$$X = \sum_{k=0}^{w-1} x_k 2^k \quad \text{(unsigned integer)}$$

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k\right) - \boxed{x_{w-1}} 2^{w-1} \quad \text{(sign integer)}$$

sign bit

- Unsigned integer, example:

8 bit word 0b10010110 represents the unsigned value, $150 = 2+4+16+128$

- Signed integer (two's complement), example:
8 bit word 0b10010110 represents the signed value $-106 = 2+4+16-128$

- We have $0b00\ldots0 = 0$
- What is the value of $x = 0b11\ldots1$?

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1}$$

$$= \left(\sum_{k=0}^{w-2} 2^k \right) - 2^{w-1}$$

$$= \left(2^{w-1} - 1\right) - 2^{w-1}$$

$$= -1$$

Complementary Relationship

$$x + \sim x = -1 \Rightarrow -x = \sim x + 1$$

Exp:

$x = 0b01101 1000$

$\sim x = 0b10010 0111$

$-x = 0b10010 1000$

* The prefix "0b" designates a Boolean constant.

To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

Example: $\boxed{0x}$ DEC1DE2C0DE4F00D is
→ the prefix 0x designates a hex constant

$$\underbrace{1101}_{D}\ \underbrace{1110}_{E}\ \underbrace{1100}_{C}\ \underbrace{0001}_{1}\ \underbrace{1101}_{D}\ \underbrace{1110}_{E}\ \underbrace{0010}_{2}\ \underbrace{1100}_{C}\ \underbrace{0000}_{0}\ \underbrace{1101}_{D}\ \underbrace{1110}_{E}\ \underbrace{0100}_{4}$$

$$\underbrace{1111}_{F}\ \underbrace{0000}_{0}\ \underbrace{0000}_{0}\ \underbrace{1101}_{D}$$

Bitwise operators :

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	Shift left
>>	Shift right

- Examples (8-bit word)

A = 0b 1011 0011
B = 0b 0110 1001
A&B = 0b 0010 0001
A | B = 0b 1111 1011
A ^ B = 0b 1101 1010
~A = 0b 0100 1100
A>>3 = 0b 0001 0110
A<<2 = 0b 1100 1100

PROBLEMS:

1. Set kth bit in a word x to 1.
   - IDEA:
     * SHIFT and OR.

$$y = x \mid (1 << k)$$

   - EXAMPLE:

$$k = 7$$

	7th
x	1 0 1 1 1 1 0 1 $\boxed{0}$ 1 1 0 1 1 0 1
$1 << k$	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
$x \mid (1 << k)$	1 0 1 1 1 1 0 1 1 1 1 0 1 1 0 1

2. Clear the kth bit in a word x.
   - IDEA:
     * SHIFT, complement, and AND

$$y = x \,\&\, \sim(1 << k)$$

   - EXAMPLE:

$$k = 7$$

→ 7th bit

x	1 0 1 1 1 1 0 1 $\boxed{1}$ 1 1 0 1 1 0 1
$1 << k$	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
$\sim(1 << k)$	1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1
$x \,\&\, \sim(1 << k)$	1 0 1 1 1 1 0 1 0 1 1 0 1 1 0 1

3. Toggle / Flip the kth bit in a word x

- IDEA:
  * SHIFT and XOR

$$y = x \,^\wedge\, (1 << k)$$

- EXAMPLE ($0 \rightarrow 1$):

$k = 7$

kth bit

x	1 0 1 1 1 1 0 1 $\boxed{0}$ 1 1 0 1 1 0 1
$1 << k$	0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
$x \,^\wedge\, (1 << k)$	1 0 1 1 1 1 0 1 1 1 1 0 1 1 0 1

4. Extract a bit field from a word x

- IDEA:
  * mask and shift

$$(x \,\&\, mask) >> shift$$

- EXAMPLE:

shift = 7

x	10111 $\boxed{1010}$ 1101101
mask	00000 $\boxed{1111}$ 0000000
x & mask	00000 1010 0000000
x & mask >> shift	000000000000 1010

- this is a good trick to know if you're working with compressed or encoded data.

5. Set a bit field in a word x to a value y.

- IDEA:
  - invert mask to clear, and OR the shifted value

$$x = (x \mathbin{\&} \sim mask) \mid \boxed{(y \ll shift)}$$

- EXAMPLE:

shift = 7 For safety's sake. ((y<<shift) & mask)

x	10111 $\boxed{1010}$ 1101101
y	000000000000 $\boxed{0011}$
mask	00000 1111 0000000
x & ~mask	10111 0000 1101101
(x & ~mask) | (y << shift)	10111 0011 1101101

6. Ordinary swap: swap two integers x and y.

- the standard way of doing this is to use a temporary variable:

$$t = x$$
$$x = y$$
$$y = t$$

* we can do this by using bit tricks, swapping x and y without using a temporary.

$x = x \wedge y$ → Mask with 1's where bits differ.

$y = x \wedge y$ → Flip the bits in y that differ from x.

$x = x \wedge y$ → Flip the bits in x that differ from y.

For example:

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

- Why it works:

XOR is its own inverse:

$(x \wedge y) \wedge y => x$

x	y	x^y	(x^y)^y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

when we XOR something twice, it just cancel out and we get back the original thing.

– Performance:
Poor at exploiting "instruction-level parallelism (ILP)". ("the naive approach is faster).

7. Find the minimum 'r' of two integers x and y.
The standard approach will be:

```
if (x < y):
    r = x
else:
    r = y
```

- Performance:

 • one performance problem with this code is that there is a branch in this code (modern machines will do branch prediction, and for whatever branch predicts the code to take it's going to do prefetching and execute some of the instructions in advance). But the problem is, if it mispredicts the branch, it does a lot of wasted work, and the processor has to empty the pipeline and undo all of the work that it did.

- Caveat:

 • the compiler is usually smart enough to optimize away the unpredictable branch, but maybe not.

? Is there a way to do a minimum without using a branch

$$r = y \,\hat{}\, ((x \,\hat{}\, y) \,\&\, -(x < y))$$

Why it works:

- The C language represents the Booleans TRUE and FALSE with the integers 1 and 0.
- If $x < y$, then $-(x<y) \Rightarrow -1$, which is all 1's in two's complement representation. Therefore, we have $y \,\hat{}\, (x \,\hat{}\, y) \Rightarrow x$
- If $x \geq y$, then $-(x<y) \Rightarrow 0$. Therefore, we have $y \,\hat{}\, 0 \Rightarrow y$.

Info! Modern compilers can perform this optimization better than you can. (the branchless version is usually slower than the branching).

*** Why learn bit hacks if they don't even work?**

- Because the compiler does them, and it will help to understand what the compiler is doing when you look at the assembly code.
- Because sometimes the compiler doesn't optimize, and you have to do it yourself by hand.
- Because many bit hacks for words extend naturally to bit and word hacks for vectors.
- Because these tricks arise in other domains.
- Because they're fun! YESSS :)

8. Modular Addition: compute $(x+y) \bmod n$, assuming that $0 \le x < n$ and $0 \le y < n$.

```
r = (x+y) % n
```
(division is expensive, unless by a power of 2)

```
z = x + y
r = z if z < n else z - n
```
(unpredictable branch is expensive)

```
z = x + y
r = z - (n & -(z >= n))
```
→ same trick as minimum