Unit 3 Bit Hacks

When using bit hacks, keep in mind that usually compiler can do simple bit hacks better. Apply O3 level of optimization might do the job. However, sometimes compiler will not be able to perform bit hacks and we will have to do it by hand.

Set and Clean kth Bit

```
// set kth bit in x
y = x | (1 << k);
// clean kth bit in x
y = x & ~(1 << k);
```

Toggle (flip) kth Bit

```
// flip kth bit in x
y = x ^ (1 << k);
```

Extract a Bit Field

```
// Extract a bit field from x
(x & mask) >> shift;
```

Set a Bit Field

```
// Set a bit field in x to y
x = (x & ~mask) | (y << shift);
```

No-Temp Swap

```
// Swap x and y
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Minimum of Two Integers

Branching involve branch prediction and if the prediction failed, all the prefetched instruction will have to be discarded. Therefore, branching in program is expensive.

```
    // FInd the minimum r of two integers x and y
    r = y ^ ((x ^ y) & -(x < y));
```

If $x < y$ is true, *-(x < y)* is -1 which is all 1's in twos complement representation. Therefore, we have *y ^ (x ^ y)* which is *x*. On the other hand, if $x < y$ is false, *-(x < y)* will be 0 and *(x ^ y) & 0* will be 0. Therefore, we have $y \wedge 0$ which is y.

Φ *__restrict* key word can be used in C to tell compiler that the pointer is the only pointer that will point to a certain data. Therefore, compiler will have more freedom to do optimizations.

Modular Addition

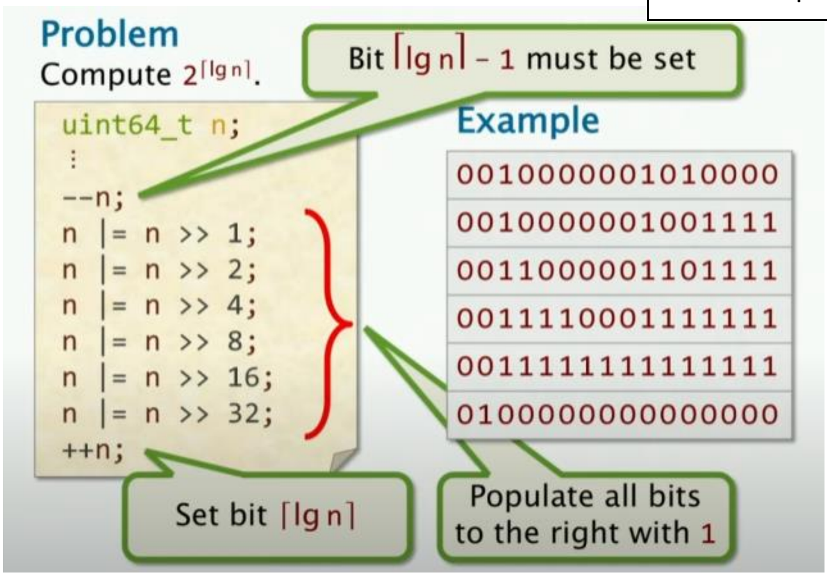
Compute $(x + y) \bmod n$, assuming that $0 \le x < n$ and $0 \le y < n$.

Φ Division is expensive, unless by a power of 2

```
z = x + y;
r = z - (n & -(z >= 0));
```

Round up to a Power of 2

If n is the power of 2, the bit will not be set



Log Base 2 of a Power of 2