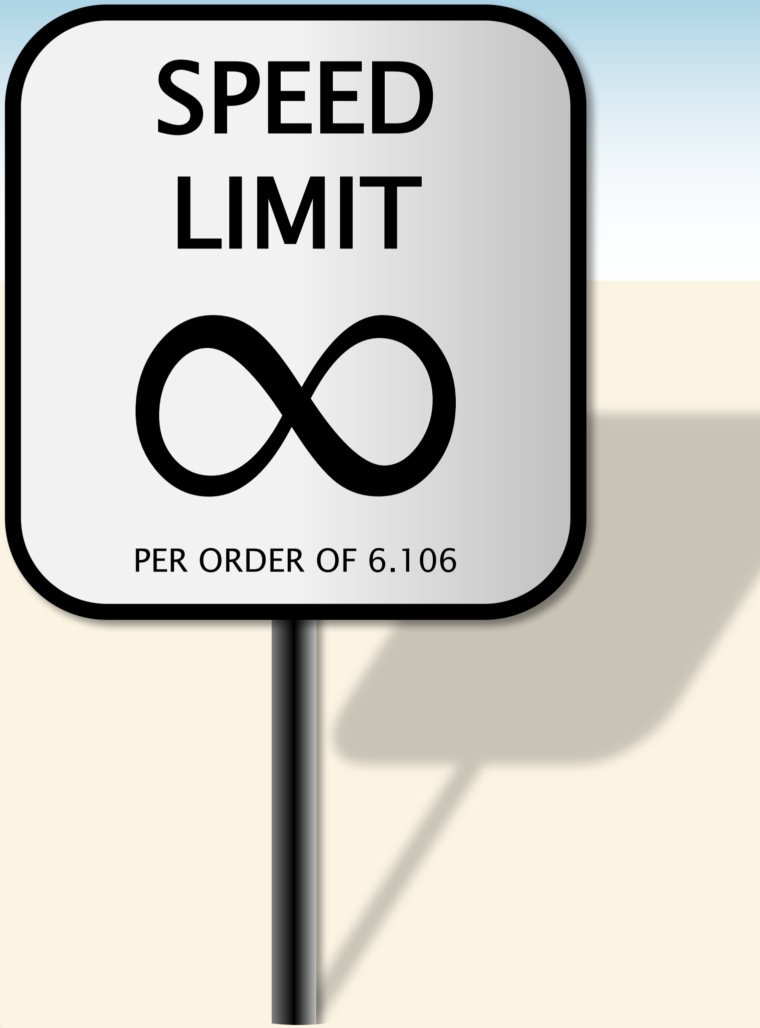
Performance
Engineering of
Software Systems

LECTURE 3

Bit Hacks

Srini Devadas

September 15, 2022



Binary Representations

Binary Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w–bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k \, .$$

Binary Representation

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w–bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k \; .$$

For example, the 8–bit word `0b10101100` represents the unsigned value $172 = 4 + 8 + 32 + 128$.

Binary Representation

Let $x = \langle x_{w-1}x_{w-2}\ldots x_0\rangle$ be a w–bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k \,.$$

The prefix 0b designates a Boolean constant.

For example, the 8–bit word 0b10101100 represents the unsigned value $172 = 4 + 8 + 32 + 128$.

Binary Representation

Let $x = \langle x_{w-1}x_{w-2}\ldots x_0\rangle$ be a w-bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

For example, the 8-bit word `0b10101100` represents the unsigned value $172 = 4 + 8 + 32 + 128$.

The signed integer (two's complement) value stored in x is

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

For example, the same 8-bit word `0b10101100` represents the signed value $-84 = 4 + 8 + 32 - 128$.

Binary Representation

Let $x = \langle x_{w-1}x_{w-2}\ldots x_0\rangle$ be a w-bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

For example, the 8-bit word 0b10101100 represents the unsigned value $172 = 4 + 8 + 32 + 128$.

The signed integer (two's complement) value stored in x is

sign bit

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

For example, the same 8-bit word 0b10101100 represents the signed value $-84 = 4 + 8 + 32 - 128$.

Two's Complement

We have $\texttt{0b00...0} = 0$.

What is the value of $x = \texttt{0b11...1}$?

$$
\begin{aligned}
x &= \left(\sum_{k=0}^{w-2} x_k 2^k\right) - x_{w-1} 2^{w-1} \\
&= \left(\sum_{k=0}^{w-2} 2^k\right) - 2^{w-1} \\
&= \left(2^{w-1} - 1\right) - 2^{w-1} \\
&= -1 \,.
\end{aligned}
$$

Complementary Relationship

Important identity

Since we have `~x + x = -1`, it follows that

$$\texttt{\~{}x + 1} = \texttt{-x} \, .$$

Complementary Relationship

Important identity

Since we have `~x + x = -1`, it follows that

$$\sim x + 1 = -x \; .$$

Example

```
 x = 0b0001100000011100
~x = 0b1110011111100011
-x = 0b1110011111100100
```

Complementary Relationship

Important identity

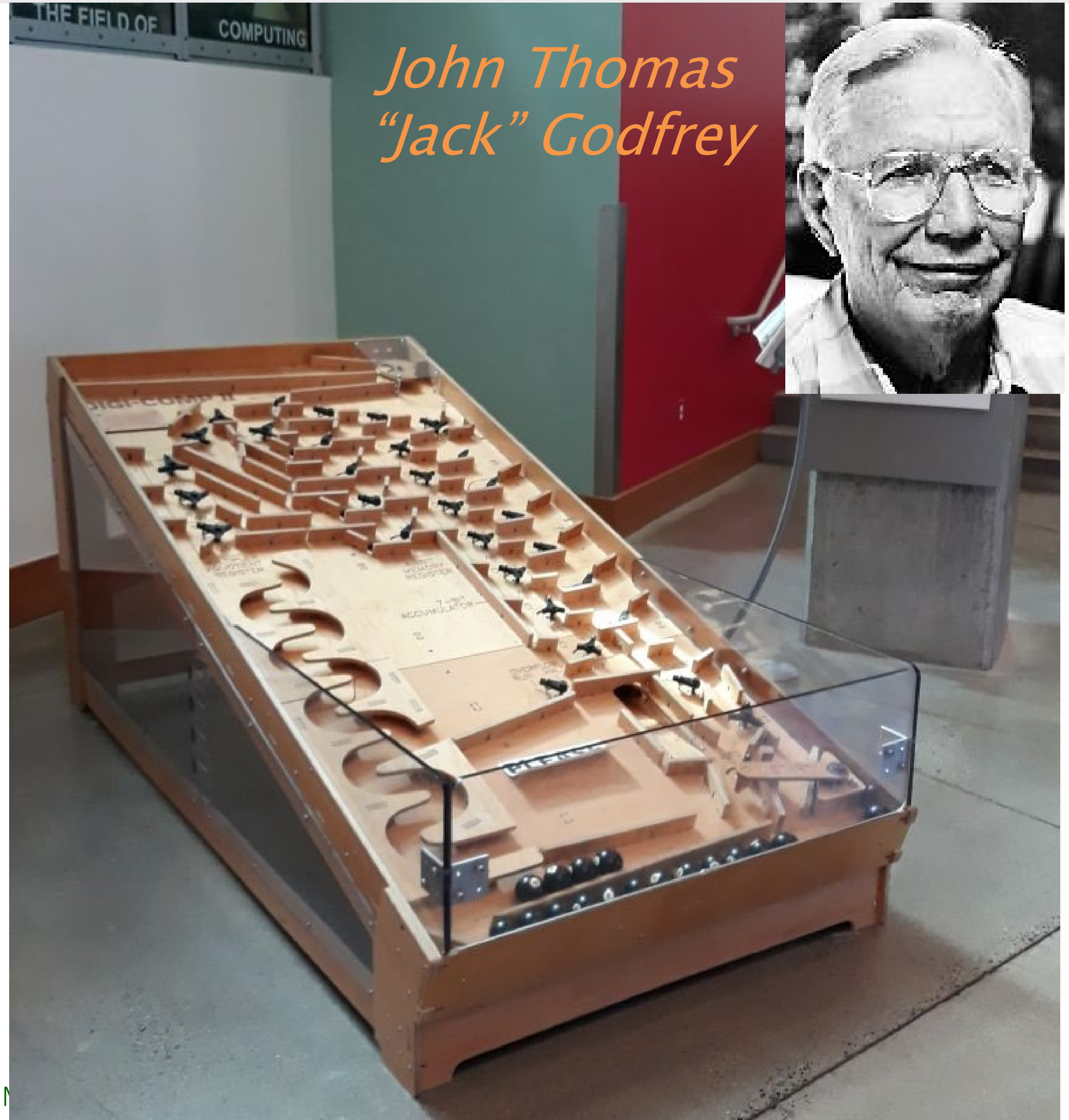
Since we have `~x + x = -1`, it follows that

$$\texttt{\~{}x + 1 = -x}\ .$$

Example

```
 x = 0b0001100000011100
~x = 0b1110011111100011
-x = 0b1110011111100100
```

DIGI-COMP II



For fun with two’s complement, check out the **DIGI-COMP II** in the Stata Center, an oversized recrea-tion of a 1960’s educational toy invented by Jack Godfrey. The DIGI-COMP algorithm for negating a number is really cool, and the machine can add, multiply, and divide numbers, as well.

Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

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

Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

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

Example: `0xDEC1DE2C0DE4F00D` is

$$\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{1100}_{C}\underbrace{0001}_{1}\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{0010}_{2}\underbrace{1100}_{C}\underbrace{0000}_{0}\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{0100}_{4}\underbrace{1111}_{F}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{1101}_{D}$$

Binary and Hexadecimal

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
[illegible]	[illegible]01	5	13	1101	D
[illegible]	[illegible]10	6	14	1110	E
[illegible]	[illegible]11	7	15	1111	F

The prefix **0x** designates a hex constant.

To translate [illegible]om hex to binary, translate each hex digit to its binary equiva[illegible]ent, and concatenate the bits.

Example: 0xDEC1DE2C0DE4F00D is

$$\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{1100}_{C}\underbrace{0001}_{1}\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{0010}_{2}\underbrace{1100}_{C}\underbrace{0000}_{0}\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{0100}_{4}\underbrace{1111}_{F}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{1101}_{D}$$

Binary and Hexadecimal

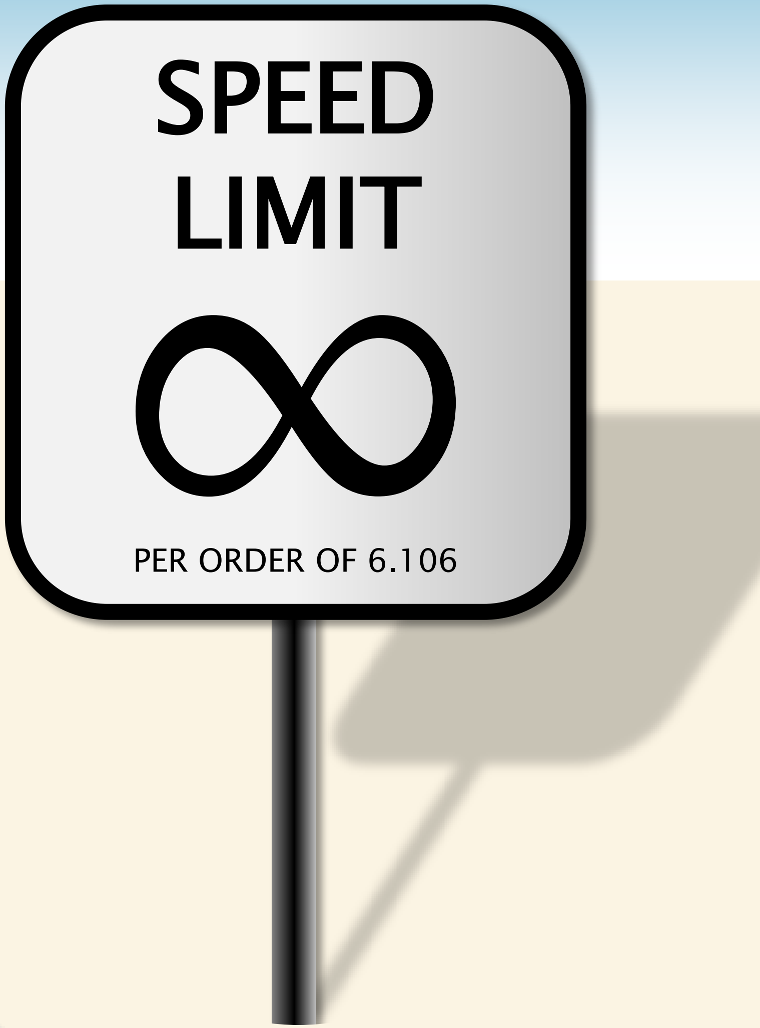
Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

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

Example: `0xDEC1DE2C0DE4F00D` is

$$\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{1100}_{C}\underbrace{0001}_{1}\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{0010}_{2}\underbrace{1100}_{C}\underbrace{0000}_{0}\underbrace{1101}_{D}\underbrace{1110}_{E}\underbrace{0100}_{4}\underbrace{1111}_{F}\underbrace{0000}_{0}\underbrace{0000}_{0}\underbrace{1101}_{D}$$

Elementary Bit Hacks



C Bitwise Operators

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

Examples (8-bit word)

```
A = 0b10110011
B = 0b01101001
```

```
A&B = 0b00100001        ~A = 0b01001100
A|B = 0b11111011    A >> 3 = 0b00010110
A^B = 0b11011010    A << 2 = 0b11001100
```

Set the kth Bit

Problem

Set kth bit in a word x to 1.

Idea

Shift and OR.

```
x | (1 << k);
```

Set the kth Bit

Problem

Set kth bit in a word x to 1.

Idea

Shift and OR.

```
x | (1 << k);
```

truth table for OR

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

Set the kth Bit

Problem

Set kth bit in a word x to 1.

Idea

Shift and OR.

```
x | (1 << k);
```

truth table for OR

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

Set the kth Bit

Problem

Set kth bit in a word x to 1.

Idea

Shift and OR.

```
x | (1 << k);
```

truth table for OR

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

Clear the kth Bit

Problem

Clear the kth bit in a word x.

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

Clear the kth Bit

Problem

Clear the kth bit in a word x.

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

truth table for AND

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

Clear the kth Bit

Problem

Clear the kth bit in a word x.

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

truth table for AND

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

Clear the kth Bit

Problem

Clear the kth bit in a word x.

Idea

Shift, complement, and AND.

```
x & ~(1 << k);
```

truth table for AND

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

Toggle the kth Bit

Problem

Flip the kth bit in a word x.

Idea

Shift and XOR.

```
x ^ (1 << k);
```

Toggle the kth Bit

Problem

Flip the kth bit in a word x.

Idea

Shift and XOR.

```
x ^ (1 << k);
```

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Example (0 → 1)

k = 7

x	1011110101101101
1 << k	0000000010000000
x ^ (1 << k)	1011110111101101

Toggle the kth Bit

Problem

Flip the kth bit in a word x.

Idea

Shift and XOR.

```
x ^ (1 << k);
```

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Example (0 → 1)

k = 7

x	1011110101101101
1 << k	0000000010000000
x ^ (1 << k)	1011110111101101

Toggle the kth Bit

Problem

Flip the kth bit in a word x.

Idea

Shift and XOR.

```
x ^ (1 << k);
```

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Example (0 → 1)

k = 7

x	1011110101101101
1 << k	0000000010000000
x ^ (1 << k)	1011110111101101

Toggle the kth Bit

Problem

Flip the kth bit in a word x.

Idea

Shift and XOR.

```
x ^ (1 << k);
```

truth table for XOR

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Example (1 → 0)

k = 7

x	1011110111101101
1 << k	0000000010000000
x ^ (1 << k)	1011110101101101

Extract a Bit Field

Problem

Extract a bit field from a word x.

Idea

Mask and shift.

```
(x & mask) >> shift;
```

Extract a Bit Field

Problem

Extract a bit field from a word `x`.

Idea

Mask and shift.

```
(x & mask) >> shift;
```

Example

`shift = 7`

`x`	`1011110101101101`
`mask`	`0000011110000000`
`x & mask`	`0000010100000000`
`(x & mask) >> shift`	`0000000000001010`

Set a Bit Field

Problem

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

Idea

Invert mask to clear, and OR the shifted value.

```
(x & ~mask) | (y << shift);
```

Set a Bit Field

Problem

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

Idea

Invert mask to clear, and OR the shifted value.

```
(x & ~mask) | (y << shift);
```

Example

`shift = 7`

x	1011110101101101
y	0000000000000011
mask	0000011110000000
x & ~mask	1011100001101101
y << shift	0000000110000000
(x & ~mask) | (y << shift)	1011100111101101

Set a Bit Field *Dangerously*

Problem

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

Idea

Invert mask to clear, and OR the shifted value.

```
(x & ~mask) | (y << shift);
```

Dangerous example

`shift = 7`

`x`	`1000110101101101`
`y`	`0000000000100011`
`mask`	`0000011110000000`
`x & ~mask`	`1000100001101101`
`y << shift`	`0001000110000000`
`(x & ~mask) | (y << shift)`	`1001100111101101`

Set a Bit Field *Safely*

Problem

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

Idea

Invert mask to clear, and OR the masked shifted value.

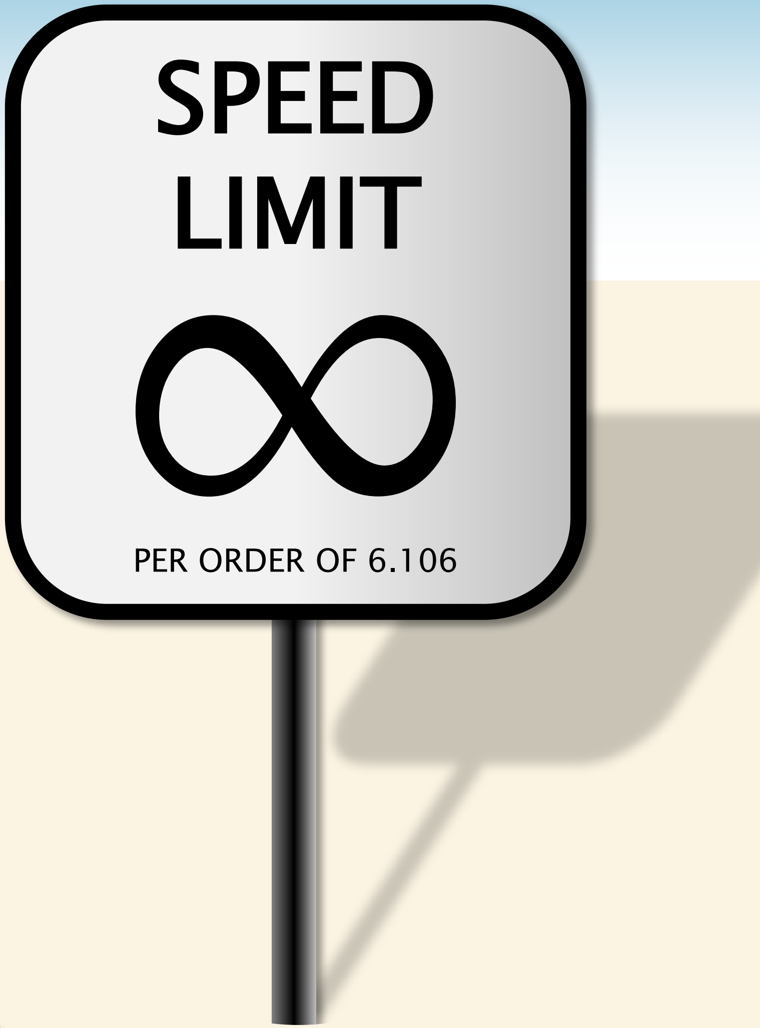
```
(x & ~mask) | ((y << shift) & mask);
```

Dangerous example (no longer)

`shift = 7`

x	1000110101101101
y	0000000000100011
mask	0000011110000000
x & ~mask	1000100001101101
((y << shift) & mask)	0000000110000000
(x & ~mask) | ((y << shift) & mask)	1000100111101101

SWAPPING



Ordinary Swap

Problem

Swap two integers x and y.

```
t = x;
x = y;
y = t;
```

Ordinary Swap

Problem

Swap two integers x and y.

```
t = x;
x = y;
y = t;
```

Example

x	10111101	10111101	00101110	10111101
y	00101110	00101110	00101110	00101110
t		10111101	10111101	10111101

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101			
y	00101110			

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011		
y	00101110	00101110		

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	
y	00101110	00101110	10111101	

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

No-Temp Swap

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse:

(x ^ y) ^ 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

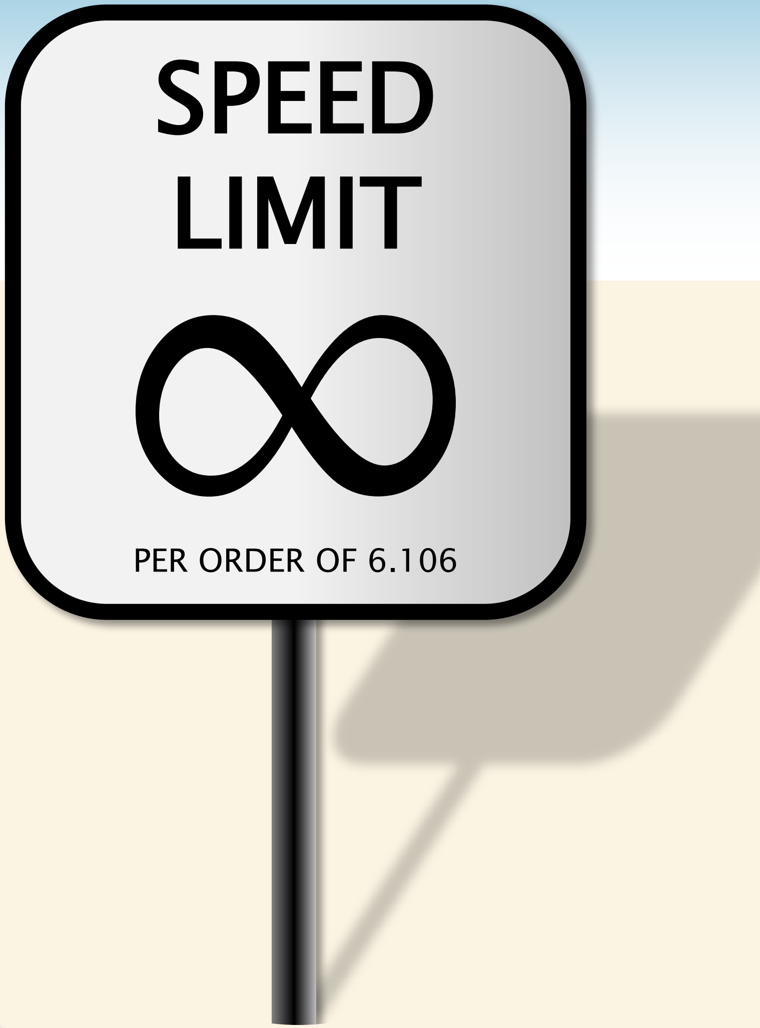
No-Temp Swap (Why it works)

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

```
x = xold ^ yold;
y = x ^ yold = (xold ^ yold) ^ yold = xold;
x = x ^ y = (xold ^ yold) ^ xold = yold;
```



AVOIDING UNPREDICTABLE CODE BRANCHES

Minimum of Two Integers

Problem

Find the minimum `r` of two integers `x` and `y`.

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

or

```
r = (x < y) ? x : y;
```

Performance

A mispredicted branch empties the processor pipeline.

Caveat

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

ars TECHNICA BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STORE

TECH —

“Meltdown” and “Spectre:” Every modern processor has unfixable security flaws

Immediate concern is for Intel chips, but everyone is at risk.

PETER BRIGHT - 1/3/2018, 7:30 PM

462

Enlarge / Spectre

Windows, Linux, and macOS have all received security patches that significantly alter how the operating systems handle virtual memory in order to protect against a hitherto undisclosed flaw. This is more than a little notable; it has been clear that Microsoft and the Linux kernel developers have been informed of some non-public security issue and have been rushing to fix it. But nobody knew quite what the problem was, leading to lots of speculation and experimentation based on pre-releases of the patches.

Now we know what the flaw is. And it's not great news, because there are in fact two related families of flaws with similar impact, and only one of them has any easy fix.

The flaws have been named Meltdown and Spectre. Meltdown was independently discovered by three groups—researchers from the Technical University of Graz in Austria, German security firm Cerberus Security, and Google's Project Zero. Spectre was discovered independently by Project Zero and independent researcher Paul Kocher.

At their heart, both attacks take advantage of the fact that processors execute instructions speculatively. All modern processors perform speculative execution to a greater or lesser extent; they'll assume that, for example, a given condition will be true and execute instructions accordingly. If it later turns out that the condition was false, the speculatively executed instructions are discarded as if they had no effect.

However, while the discarded effects of this speculative execution don't alter the outcome of a program, they do make changes to the lowest level architectural features of the processors. For example, speculative execution can load data into cache even if it turns out that the data should never have been loaded in the first place. The presence of the data in the cache can then be detected, because accessing it will be a little bit quicker than if it weren't cached. Other data

No-Branch Minimum

Problem

Find the minimum of two integers `x` and `y` without using a branch.

```
y ^ ((x ^ y) & -(x < y));
```

Why it works

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

Merging Two Sorted Arrays

```
static void merge(int64_t * __restrict C,
                  int64_t * __restrict A,
                  int64_t * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
}
```

The `__restrict` keywords say that `A`, `B`, and `C` don't *alias*, meaning that they do not overlap in memory.

Merging Two Sorted Arrays

```
static void merge(int64_t * __restrict C,
                  int64_t * __restrict A,
                  int64_t * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
}
```

3	12	19	46

4	14	21	23

Branching

```
static void merge(long * __restrict C,
                  long * __restrict A,
                  long * __restrict B,
                  size_t na,
                  size_t nb) {
4   while (na > 0 && nb > 0) {
3     if (*A <= *B) {
        *C++ = *A++; na--;
      } else {
        *C++ = *B++; nb--;
      }
    }
2   while (na > 0) {
      *C++ = *A++;
      na--;
    }
1   while (nb > 0)  {
      *C++ = *B++;
      nb--;
    }
}
```

Branch	Predictable?
1	Yes
2	Yes
3	No
4	Yes

Branchless

```
static void merge(int64_t * __restrict C,
                  int64_t * __restrict A,
                  int64_t * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    long cmp = (*A <= *B);
    long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A +=  cmp; na -=  cmp;
    B += !cmp; nb -= !cmp;
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
}
```

This optimization works well on some machines, but on modern machines using `clang -O3`, the branchless version is usually slower than the branching version. 👎 Modern compilers can perform this optimization better than you can!

Why Learn Bit Hacks?

Why learn bit hacks if they don't perform?

- Because the compiler does them, and it will help to understand how the compiler is optimizing when you look at the assembly code.
- Because sometimes the compiler doesn't optimize, and you have to optimize your code by hand.
- Because many bit hacks for words extend naturally to bit, byte, and word hacks for vectors.
- Because these tricks arise in other domains, and so it pays to be educated about them.
- Because they're fun!

Modular Addition

Problem

Compute `r = (x + y) mod n`, assuming that `0 ≤ x < n` and `0 ≤ y < n`.

```
r = (x + y) % n;
```

Division is expensive.

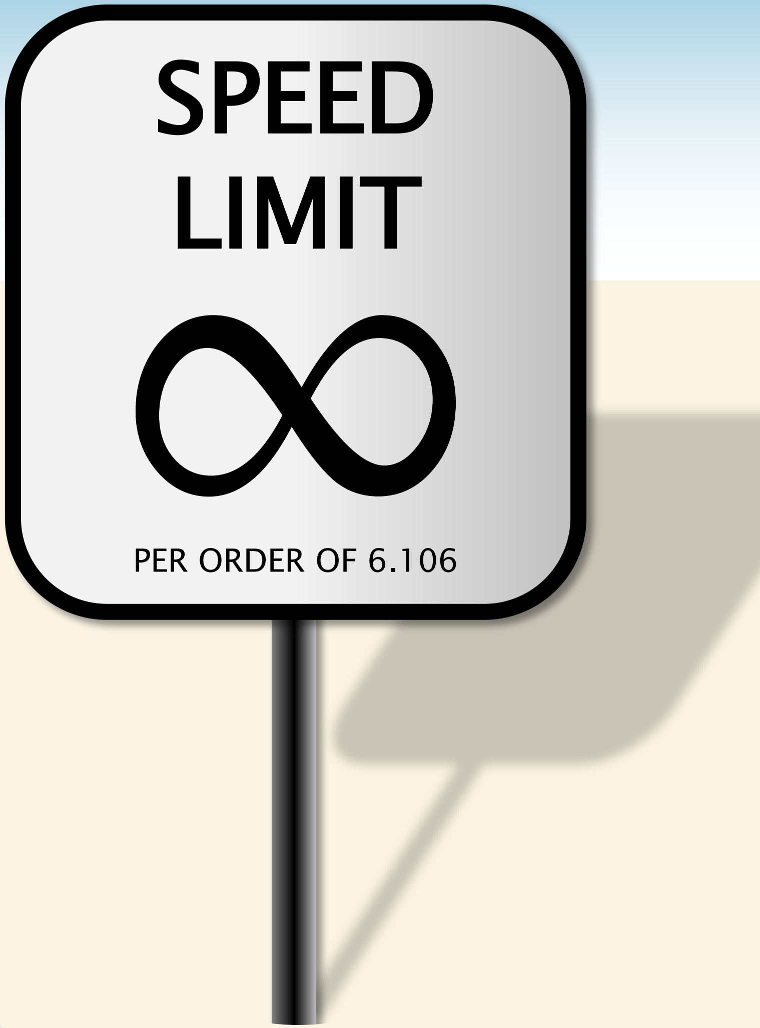
```
z = x + y;
r = (z < n) ? z : z - n;
```

Unpredictable branch is expensive.

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

Same trick as minimum.

POWERS OF 2



Is an Integer a Power of 2?

Problem

Is $x = 2^k$ for some integer k?

```
x == x & -x
```

Example

`x`	00001000	00101000
`-x`	11111000	11011000
`x & -x`	00001000	00001000
`x == x & -x`	00000001	00000000

Bug!

What if $x = 0$?

```
(x != 0) & (x == x & -x)
```

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

Notation

$\lg n = \log_2 n$

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000

Why decrement?

To handle the boundary case when n is a power of 2.

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

```
uint64_t n;
 ⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Bit $\lceil \lg n \rceil - 1$ must be set.

Populate all the bits to the right with 1.

Set bit $\lceil \lg n \rceil$.

Example

00100000001010000
00100000001001111
00110000001101111
00111100001111111
00111111111111111
1000000000000000

Least-Significant 1

Problem

Compute the mask of the least-significant 1 in word x.

```
r = x & (-x);
```

Example

x	0010000001010000
-x	1101111110110000
x & (-x)	0000000000010000

Why it works

The binary representation of -x is (~x)+1.

Question

How do you find the index of the bit, i.e., lg r?

Count Trailing Zeros

Problem

Compute lg x, where x is a power of 2.

```
const uint64_t deBruijn = 0x022fdd63cc95386d;
const int convert[64] = {
   0,  1,  2, 53,  3,  7, 54, 27,
   4, 38, 41,  8, 34, 55, 48, 28,
  62,  5, 39, 46, 44, 42, 22,  9,
  24, 35, 59, 56, 49, 18, 29, 11,
  63, 52,  6, 26, 37, 40, 33, 47,
  61, 45, 43, 21, 23, 58, 17, 10,
  51, 25, 36, 32, 60, 20, 57, 16,
  50, 31, 19, 15, 30, 14, 13, 12
};
r = convert[(x * deBruijn) >> 58];
```

Count Trailing 0's of a Power of 2

Why it works

A **deBruijn sequence** `s` of length 2^k is a cyclic 0–1 sequence such that each of the 2^k 0–1 strings of length k occurs exactly once as a substring of `s`.

Example: $k=3$

	00011101
0	00011101
1	00111010
2	01110100
3	11101000
4	11010001
5	10100011
6	01000111
7	10001110

Count Trailing 0's of a Power of 2

Why it works

A **deBruijn sequence** `s` of length 2^k is a cyclic `0–1` sequence such that each of the 2^k `0–1` strings of length `k` occurs exactly once as a substring of `s`.

Example: k=3

	00011101
0	00011101
1	00111010
2	01110100
3	11101000
4	11010001
5	10100011
6	01000111
7	10001110

```
const int convert[8]
= {0,1,6,2,7,5,4,3};
```

Count Trailing 0's of a Power of 2

Why it works

A **deBruijn sequence** `s` of length 2^k is a cyclic `0–1` sequence such that each of the 2^k `0–1` strings of length `k` occurs exactly once as a substring of `s`.

```
0b00011101*2^4 ⇒ 0b11010000
0b11010000 >> 5 ⇒ 6
convert[6] ⇒ 4
```

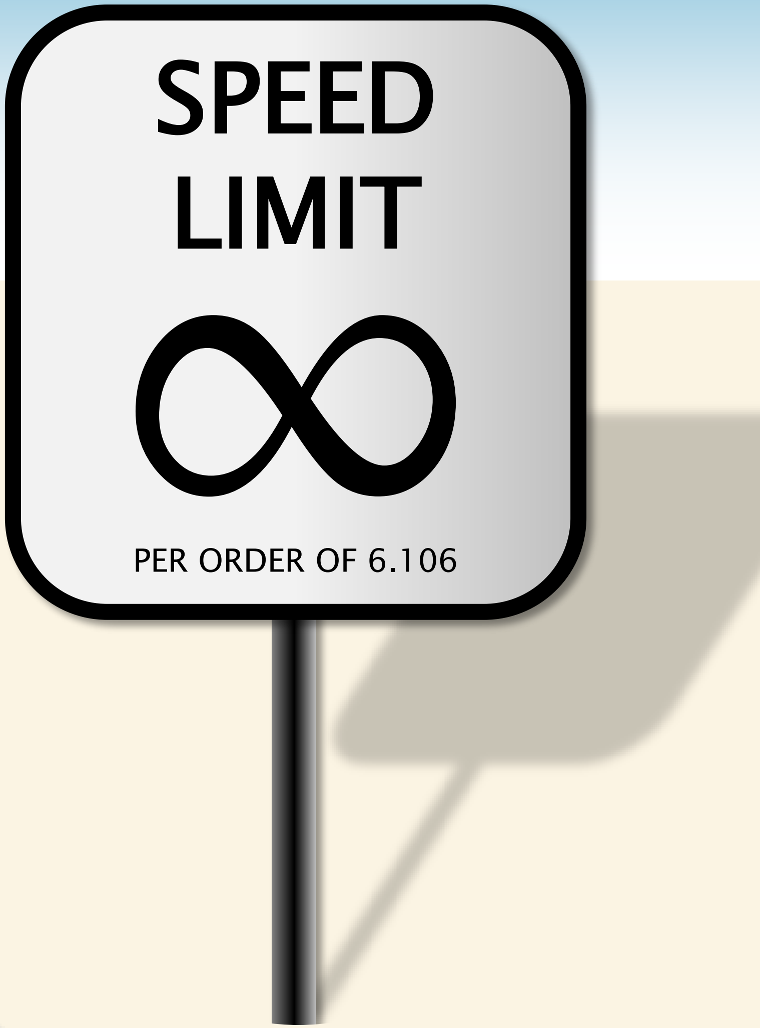
Example: k=3

00011101
0 00011101
1 00111010
2 01110100
3 11101000
4 11010000
5 10100000
6 01000000
7 10000000

Hardware instruction

```
int __builtin_ctz(int x)
```

```
const int convert[8]
= {0,1,6,2,7,5,4,3};
```



POPCOUNT

Population Count I

Problem

Count the number of 1 bits in a word x.

```
for (r=0; x!=0; ++r)
  x &= x - 1;
```

Repeatedly eliminate the least-significant 1.

Example

x	0010110111010000
x - 1	0010110111001111
x & (x - 1);	0010110111000000

Issue

Fast if the popcount is small, but in the worst case, the running time is proportional to the number of bits in the word.

Population Count II

Table lookup

```
static const int count[256] =
{ 0, 1, 1, 2, 1, 2, 2, 3, 1, …, 8 };

for (int r = 0; x != 0; x >>= 8)
  r += count[x & 0xFF];
```

Performance depends on the word size. The cost of memory operations is a major bottleneck. Typical memory latencies:

- register: 1 cycle,
- L1-cache: 4 cycles,
- L2-cache: 10 cycles,
- L3-cache: 40 cycles,
- DRAM: 200 cycles.

$\left.\begin{matrix}\\ \\ \\ \\ \end{matrix}\right\}$ per 64-byte cache line

Population Count III

Parallel divide-and-conquer

```
// Create masks
M5 = ~((-1) << 32);   // 0³²1³²
M4 = M5 ^ (M5 << 16); // (0¹⁶1¹⁶)²
M3 = M4 ^ (M4 << 8);  // (0⁸1⁸)⁴
M2 = M3 ^ (M3 << 4);  // (0⁴1⁴)⁸
M1 = M2 ^ (M2 << 2);  // (0²1²)¹⁶
M0 = M1 ^ (M1 << 1);  // (01)³²
// Compute popcount
x = ((x >> 1) & M0) + (x & M0);
x = ((x >> 2) & M1) + (x & M1);
x = ((x >> 4) + x) & M2;
x = ((x >> 8) + x) & M3;
x = ((x >> 16) + x) & M4;
x = ((x >> 32) + x) & M5;
```

Notation:

$$x^k = \underbrace{x x \cdots x}_{k \text{ times}}$$

Population Count III

1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0   x

Population Count III

1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0 `x`

`x&MO`
`(x>>1)&MO`

Population Count III

	1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0																x
	1	0	0	0	1	1	0	1	1	1	1	0	1	1	0	0	x&MO
+	1	0	0	1	0	0	1	1	1	1	0	0	0	1	1	0	(x>>1)&MO
	1 0	0 0	0 0	0 1	0 1	0 1	0 1	1 0	1 0	1 0	0 1	0 0	0 1	1 0	0 1	0 0	

Population Count III

	1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0	x
	1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 0	x&M0
+	1 0 0 1 0 0 1 1 1 1 0 0 0 1 1 0	(x>>1)&M0
	1 0 0 0 0 0 0 1 0 1 0 1 0 1 1 0 1 0 1 0 0 1 0 0 0 1 1 0 0 1 0 0	x&M1
		(x>>2)&M1

Population Count III

```
    1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0    x
      1   0   0   0   1   1   0   1   1   1   1   0   1   1   0   0    x&M0
+     1   0   0   1   0   0   1   1   1   1   0   0   0   1   1   0    (x>>1)&M0
  -------------------------------------------------------------------
  |     0 0|     0 1|     0 1|     1 0|     1 0|     0 0|     1 0|     0 0|    x&M1
+ |     1 0|     0 0|     0 1|     0 1|     1 0|     0 1|     0 1|     0 1|    (x>>2)&M1
  -------------------------------------------------------------------
  |0 0 1 0|0 0 0 1|0 0 1 0|0 0 1 1|0 1 0 0|0 0 0 1|0 0 1 1|0 0 0 1|
```

Population Count III

	1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0	x
	1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 0	x&M0
+	1 0 0 1 0 0 1 1 1 1 0 0 0 1 1 0	(x>>1)&M0
	0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 0	x&M1
+	1 0 0 0 0 1 0 1 1 0 0 1 0 1 0 1	(x>>2)&M1
	0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 1	x&M2
		(x>>4)&M2

Population Count III

```
    1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0    x
      1   0   0   0   1   1   0   1   1   1   1   0   1   1   0   0    x&M0
+     1   0   0   1   0   0   1   1   1   1   0   0   0   1   1   0    (x>>1)&M0
---------------------------------------------------------------------
        0 0     0 1     0 1     1 0     1 0     0 0     1 0     0 0    x&M1
+       1 0     0 0     0 1     0 1     1 0     0 1     0 1     0 1    (x>>2)&M1
---------------------------------------------------------------------
          0 0 0 1         0 0 1 1         0 0 0 1         0 0 0 1    x&M2
+         0 0 1 0         0 0 1 0         0 1 0 0         0 0 1 1    (x>>4)&M2
---------------------------------------------------------------------
  | 0 0 0 0 0 0 1 1 | 0 0 0 0 0 0 1 1 | 0 0 0 0 0 1 0 1 | 0 0 0 0 0 1 0 0 |
```

Population Count III

```
     1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0    x
       1   0   0   0   1   1   0   1   1   1   1   0   1   1   0   0    x&M0
+      1   0   0   1   0   0   1   1   1   1   0   0   0   1   1   0    (x>>1)&M0
---------------------------------------------------------------------
         0 0     0 1     0 1     1 0     1 0     0 0     1 0     0 0    x&M1
+        1 0     0 0     0 1     0 1     1 0     0 1     0 1     0 1    (x>>2)&M1
---------------------------------------------------------------------
             0 0 0 1         0 0 1 1         0 0 0 1         0 0 0 1    x&M2
+            0 0 1 0         0 0 1 0         0 1 0 0         0 0 1 1    (x>>4)&M2
---------------------------------------------------------------------
     0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0    x&M3
                                                                        (x>>8)&M3
```

Population Count III

```
    1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0      x
      1   0   0   0   1   1   0   1   1   1   1   0   1   1   0   0      x&M0
+     1   0   0   1   0   0   1   1   1   1   0   0   0   1   1   0      (x>>1)&M0
---------------------------------------------------------------------
        0 0     0 1     0 1     1 0     1 0     0 0     1 0     0 0      x&M1
+       1 0     0 0     0 1     0 1     1 0     0 1     0 1     0 1      (x>>2)&M1
---------------------------------------------------------------------
          0 0 0 1         0 0 1 1         0 0 0 1         0 0 0 1      x&M2
+         0 0 1 0         0 0 1 0         0 1 0 0         0 0 1 1      (x>>4)&M2
---------------------------------------------------------------------
                  0 0 0 0 0 1 0 1                 0 0 0 0 0 1 0 0      x&M3
+                 0 0 0 0 0 0 1 1                 0 0 0 0 0 1 0 1      (x>>8)&M3
---------------------------------------------------------------------
    0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0|0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1
```

Population Count III

```
      1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0    x
        1   0   0   0   1   1   0   1   1   1   1   0   1   1   0   0    x&M0
+       1   0   0   1   0   0   1   1   1   1   0   0   0   1   1   0    (x>>1)&M0
-------------------------------------------------------------------
          0 0     0 1     0 1     1 0     1 0     0 0     1 0     0 0    x&M1
+         1 0     0 0     0 1     0 1     1 0     0 1     0 1     0 1    (x>>2)&M1
-------------------------------------------------------------------
              0 0 0 1         0 0 1 1         0 0 0 1         0 0 0 1    x&M2
+             0 0 1 0         0 0 1 0         0 1 0 0         0 0 1 1    (x>>4)&M2
-------------------------------------------------------------------
                      0 0 0 0 0 1 0 1                 0 0 0 0 0 1 0 0    x&M3
+                     0 0 0 0 0 0 1 1                 0 0 0 0 0 1 0 1    (x>>8)&M3
-------------------------------------------------------------------
      0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1    x&M4
                                                                         (x>>16)&M4
```

Population Count III

	1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 0 0 0 1 1 1 1 0 0 0	x
	1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 0	x&M0
+	1 0 0 1 0 0 1 1 1 1 0 0 0 1 1 0	(x>>1)&M0
	0 0 0 1 0 1 1 0 1 0 0 0 1 0 0 0	x&M1
+	1 0 0 0 0 1 0 1 1 0 0 1 0 1 0 1	(x>>2)&M1
	0 0 0 1 0 0 1 1 0 0 0 1 0 0 0 1	x&M2
+	0 0 1 0 0 0 1 0 0 1 0 0 0 0 1 1	(x>>4)&M2
	0 1 0 1 1 0 1 1 0 0 0 0 0 1 0 0	x&M3
+	1 1 0 0 0 0 1 0 0 0 0 0 0 1 0 1	(x>>8)&M3
	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1	x&M4
+	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0	(x>>16)&M4
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 $\underbrace{1\ 0\ 0\ 0\ 1}_{17}$	

Population Count III

Parallel divide-and-conquer

```
// Create masks
M5 = ~((-1) << 32);    // 0³²1³²
M4 = M5 ^ (M5 << 16); // (0¹⁶1¹⁶)²
M3 = M4 ^ (M4 << 8);  // (0⁸1⁸)⁴
M2 = M3 ^ (M3 << 4);  // (0⁴1⁴)⁸
M1 = M2 ^ (M2 << 2);  // (0²1²)¹⁶
M0 = M1 ^ (M1 << 1);  // (01)³²
// Compute popcount
x = ((x >> 1) & M0) + (x & M0);
x = ((x >> 2) & M1) + (x & M1);
x = ((x >> 4) + x) & M2;
x = ((x >> 8) + x) & M3;
x = ((x >> 16) + x) & M4;
x = ((x >> 32) + x) & M5;
```

Avoid overflow

No worry about overflow.

Performance

$\Theta(\lg w)$ time, where w = word length.

Popcount Instructions

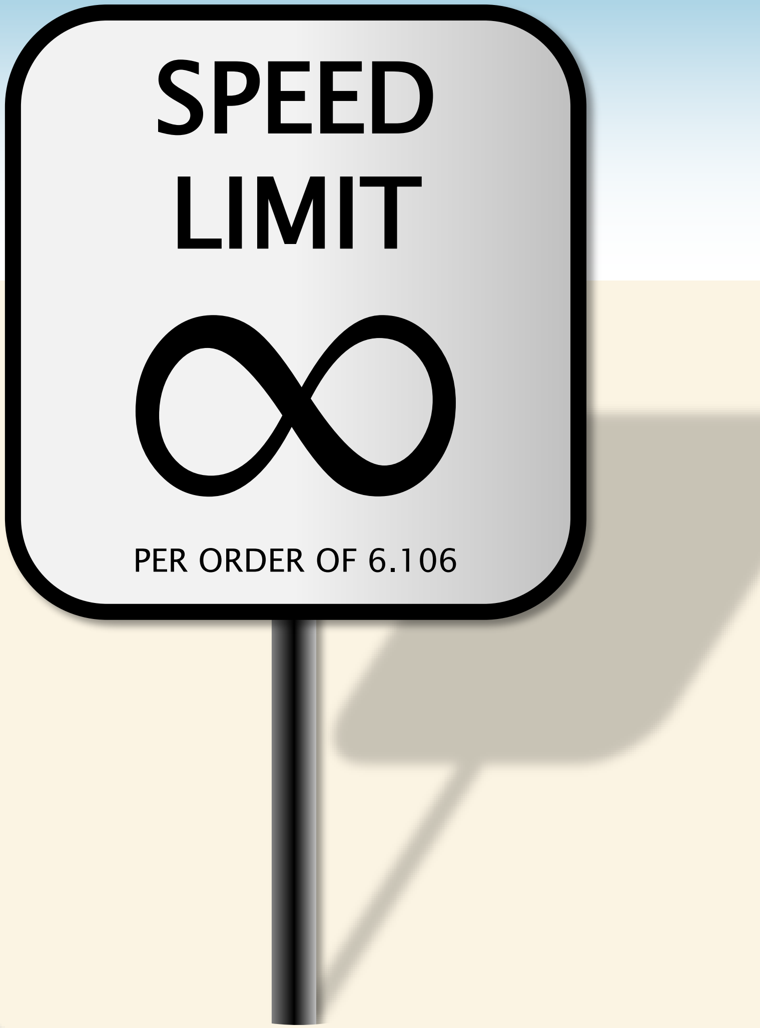
Most modern machines provide popcount instructions, which operate much faster than anything you can code yourself. You can access them via compiler intrinsics, e.g., in `clang`:

```
int __builtin_popcount (unsigned int x);
```

Warning: With some compilers, you may need to enable certain switches to access built-in functions, and your code may be less portable.

Exercise

Compute the log base 2 of a power of 2 quickly using a `popcount` instruction.

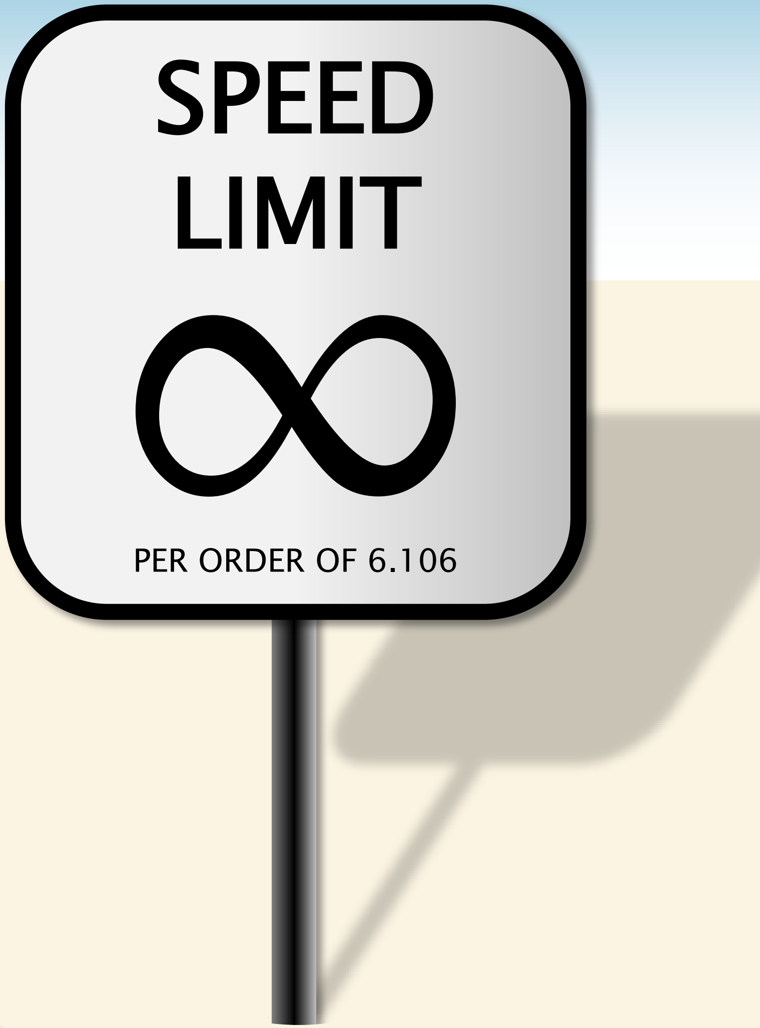


FINAL REMARKS

Further Reading

- Sean Eron Anderson, “Bit twiddling hacks,” `http://graphics.stanford.edu/~seander/bithacks.html`, 2009.
- Donald E. Knuth, *The Art of Computer Programming*, Volume 4A, *Combinatorial Algorithms, Part 1*, Addison-Wesley, 2011, Section 7.1.3.
- `http://chessprogramming.wikispaces.com/`
- Henry S. Warren, *Hacker’s Delight*, Addison-Wesley, 2003.

And remember to⋯



Support Computer Science: Every Little Bit Counts!