Computer Arithmetic

What is the difference between `int` and `unsigned int`?

Pat Hanrahan

cs107e

Addition

Binary Addition

```
          Carry
 00000111 A
+00001011 B
---------
          Sum
```

Binary Addition

```
       1  Carry
 00000111 A
+00001011 B
---------
        0 Sum
```

Binary Addition

```
      11  Carry
  00000111 A
 +00001011 B
 ---------
        10 Sum
```

Binary Addition

```
 00001111  Carry
  00000111 A
 +00001011 B
 ---------
  00010010 Sum
```

Binary Addition

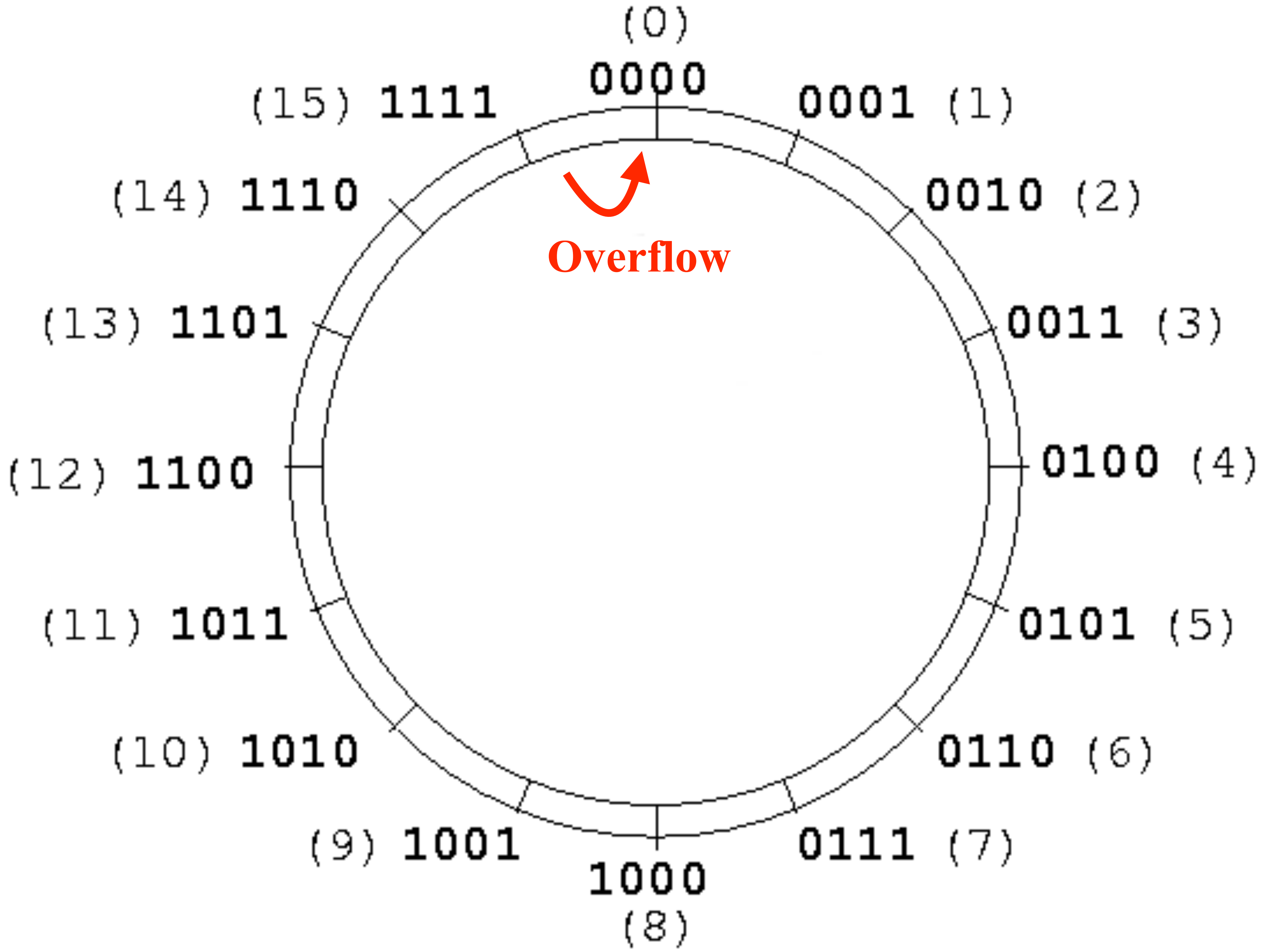
```
 11111111  Carry
  11111111 A
 +00000001 B
 ---------
 100000000 Sum
```

To represent the result of adding two n-bit numbers with full precision requires n+1 bits

But we only have 8-bits!

```
  sum = 0b00000000 = (A+B)%256
```

999999
99999
000000

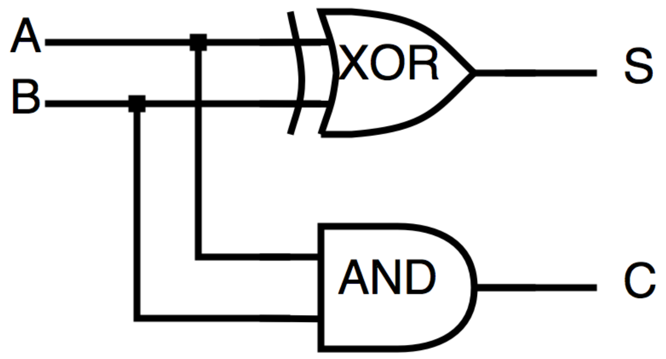

(0)
0000
0001 (1)
0010 (2)
0011 (3)
0100 (4)
0101 (5)
0110 (6)
0111 (7)
1000
(8)
(9) 1001
(10) 1010
(11) 1011
(12) 1100
(13) 1101
(14) 1110
(15) 1111
Overflow

Add 2 1-bit numbers

a	b	sum
0	0	00
0	1	01
1	0	01
1	1	10

Add 2 1-bit numbers (Half Adder)

```
a b sum
0 0  00
0 1  01
1 0  01
1 1  10
```



bit 0 of sum: S = a^b
bit 1 of sum: C = a&b

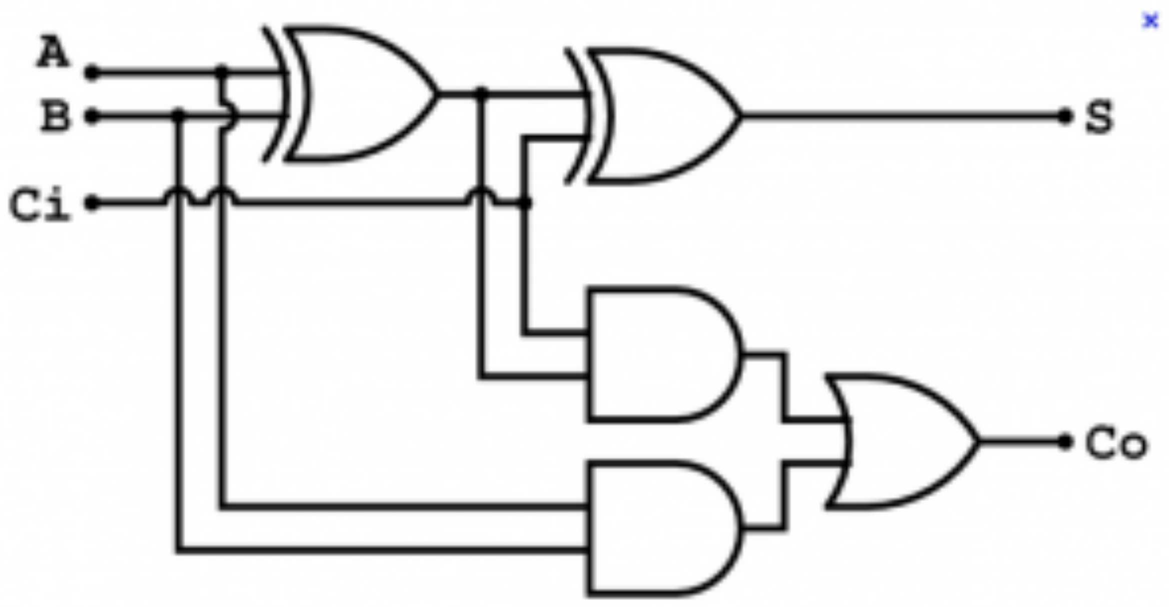
Have reduced addition to logical operations!

Add 3 1-bit numbers

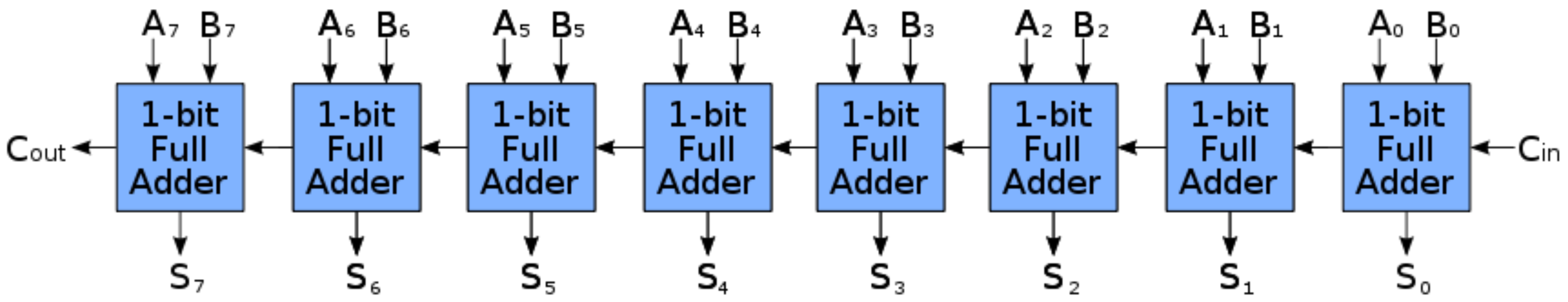
a	b	c	=	c	s
0	0	0		0	0
0	1	0		0	1
1	0	0		0	1
1	1	0		1	0
0	0	1		0	1
0	1	1		1	0
1	0	1		1	0
1	1	1		1	1

Add 3 1-bit numbers (Full Adder)

a	b	ci	=	co	s
0	0	0		0	0
0	1	0		0	1
1	0	0		0	1
1	1	0		1	0
0	0	1		0	1
0	1	1		1	0
1	0	1		1	0
1	1	1		1	1



8-bit Ripple Adder



Note `Cin` **and** `Cout`

```
// Multiple precision addition
// http://godbolt.org/g/HMYrme

uint64_t add64(uint64_t a, uint64_t b)
{
  return a + b;
}

add64:
  adds r0, r0, r2
  adc  r1, r1, r3
  bx   lr
```

Subtraction

Binary Subtraction

```
 00000001  Borrow
  00000110 A
 -00000001 B
 ---------
  00000101 Sub
```

Do we need to build subtraction hardware?

BIG IDEA: Define subtraction using addition

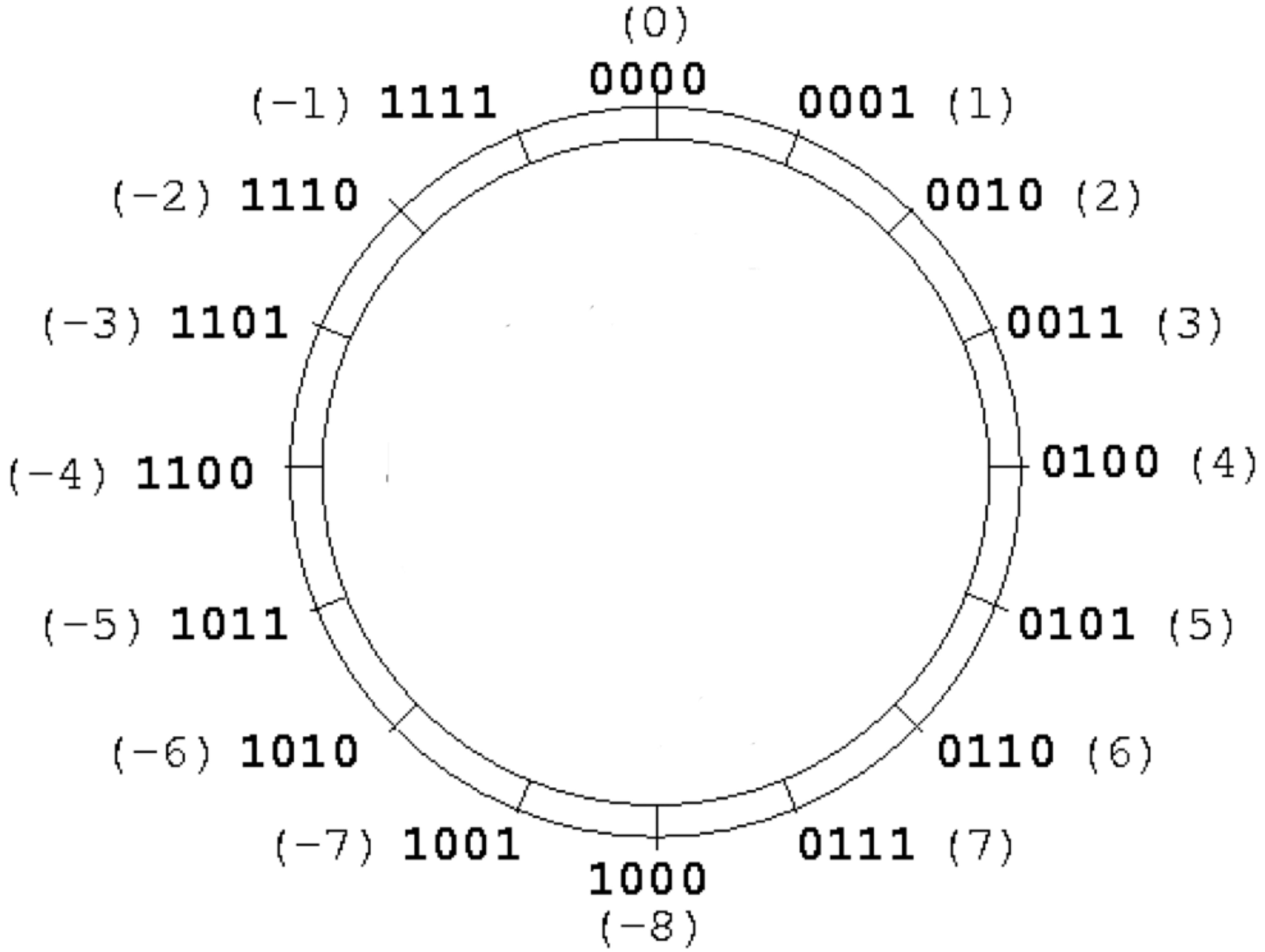
A clever way of defining subtraction by 1 is to find a number to add that yields the same result as the subtract by 1.

This number is the *negative* of the number.

More precisely, this number is the number that when added to 1, results in 0 (mod 16)

```
0x1 - 0x1 = 0x1 + 0xf = 0x10 % 16 = 0x0
```

0xf can be *interpreted* as -1


(0)
0000
0001 (1)
0010 (2)
0011 (3)
0100 (4)
0101 (5)
0110 (6)
0111 (7)
1000
(−8)
(−7) 1001
(−6) 1010
(−5) 1011
(−4) 1100
(−3) 1101
(−2) 1110
(−1) 1111

Signed 4-bit numbers,

```
0x0 =  0
0xf = -1
0xe = -2
…
0x8 = -8 (could be interpreted as 8)
0x7 =  7
…
0x1 =  1
0x0 =  0
```

if we choose to *interpret* 0x8 as -8, then the most-significant bit of the number indicates that it is negative (n)

`signed int` **vs as** `unsigned int`

Are just different interpretations of the bits comprising the number

`0xff` **vs** `-1`

Negation

```
How do we negate an 8-bit number?

Subtract it from 2^8 (0b100000000)

-x = 0b100000000 - x (two's complement)

Since then (x + (-x)) % 256 = 0

 11111111  Borrow     100000000 Carry
 100000000             00000001
 -00000001            +11111111
 ---------            ---------
  11111111             00000000
```

Another way

```
Rewrite 100000000 = (11111111 + 1)

-x = (11111111+1)-x
   = (11111111-x)+1 (one's complement)
   = ~x+1

E.g. -1 = 0b11111111

~00000001 = 11111110 (~ is invert)

 11111110 + 00000001 = 11111111
```

Subtraction is converted negation + addition

-B is implemented using ~B+1

A - B = A + ~B + 1

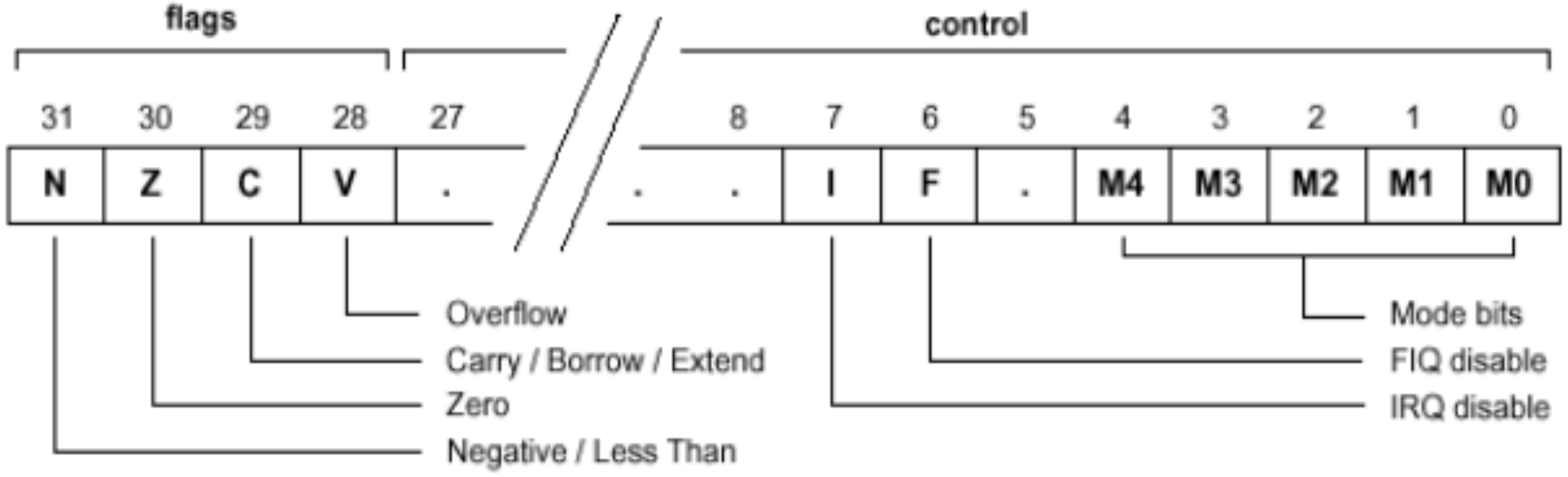
```
01 - 00 = 01 + ff + 01 = 01 + c
01 - 01 = 01 + fe + 01 = 00 + c
01 - 02 = 01 + fd + 01 = ff
```

Note the carry out bit c

The +1 can be done by setting Cin to 1

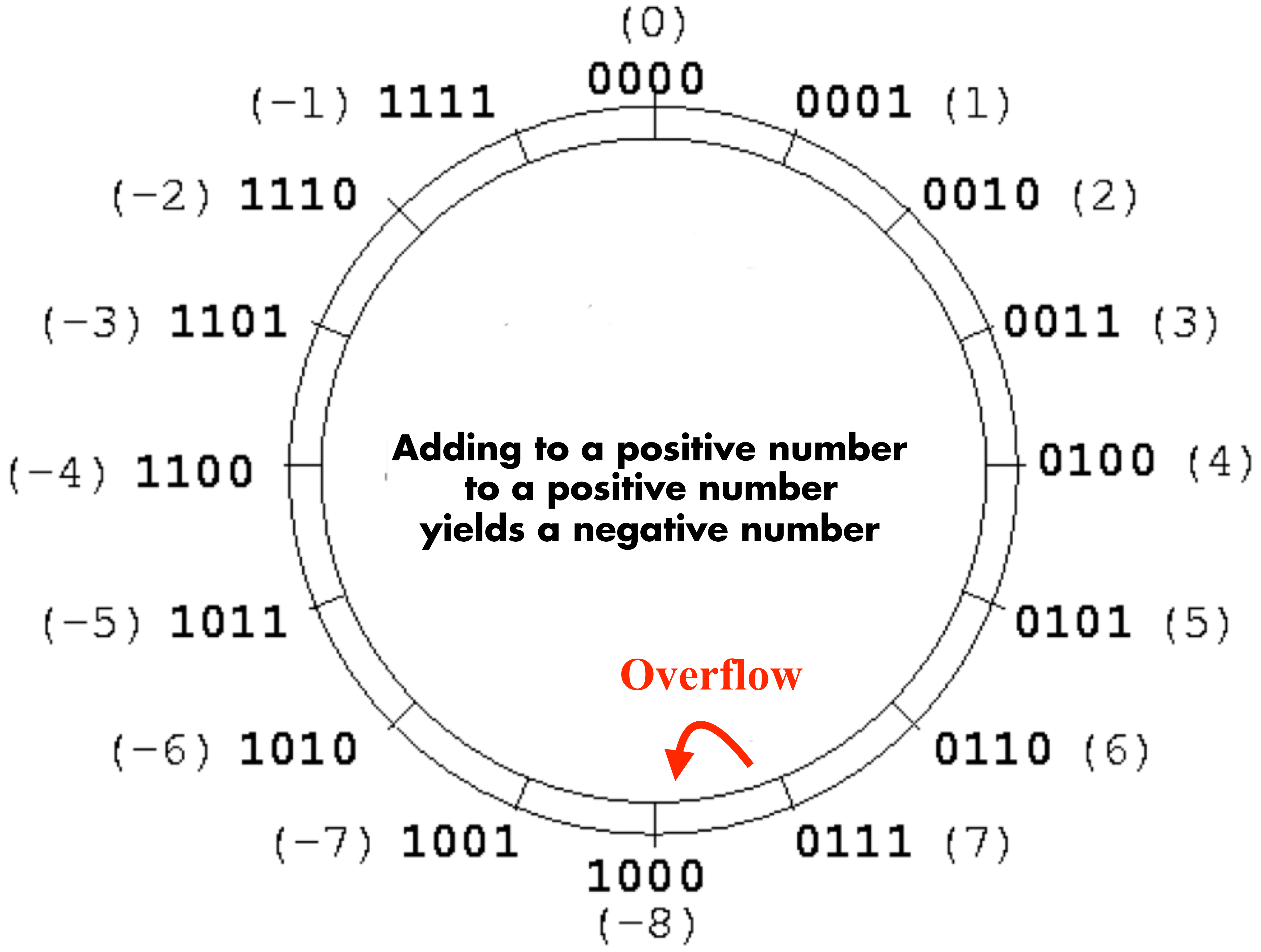
Addition, Subtraction, and Negation of `signed` and `unsigned` numbers are the same!

CPSR

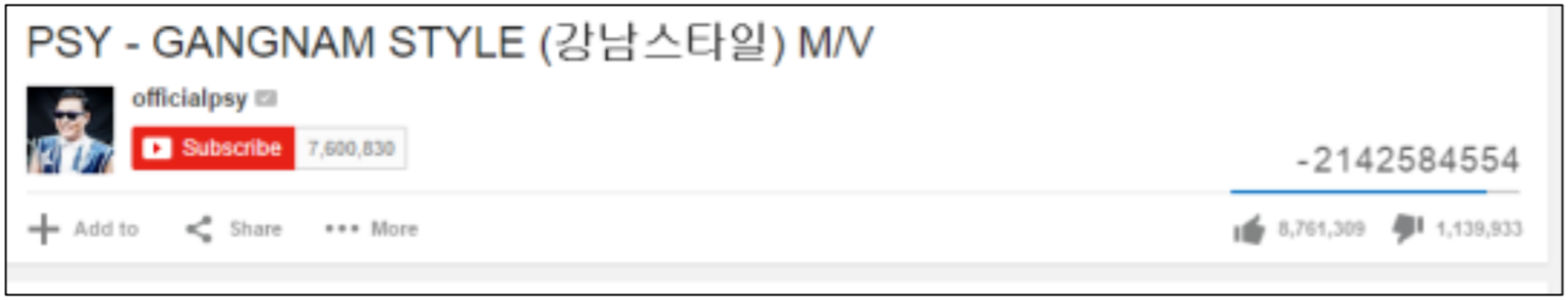


Arithmetic instructions set N, Z, C, V
Logic instructions just set N, Z

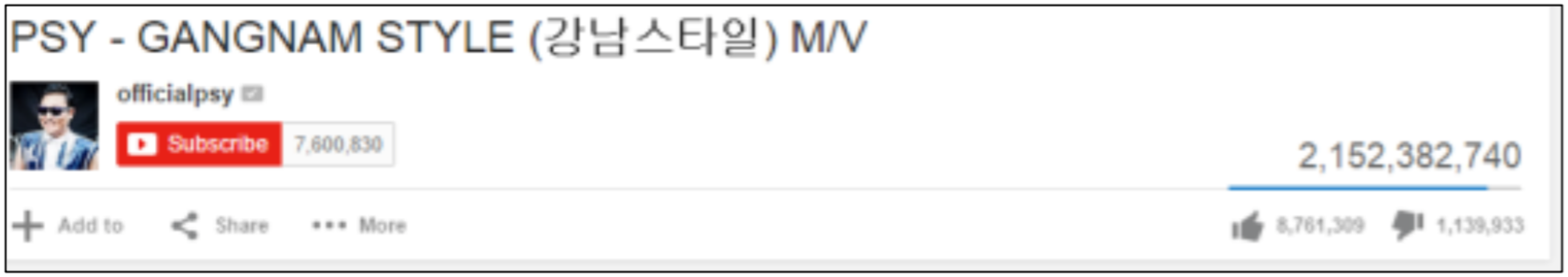
What is V?

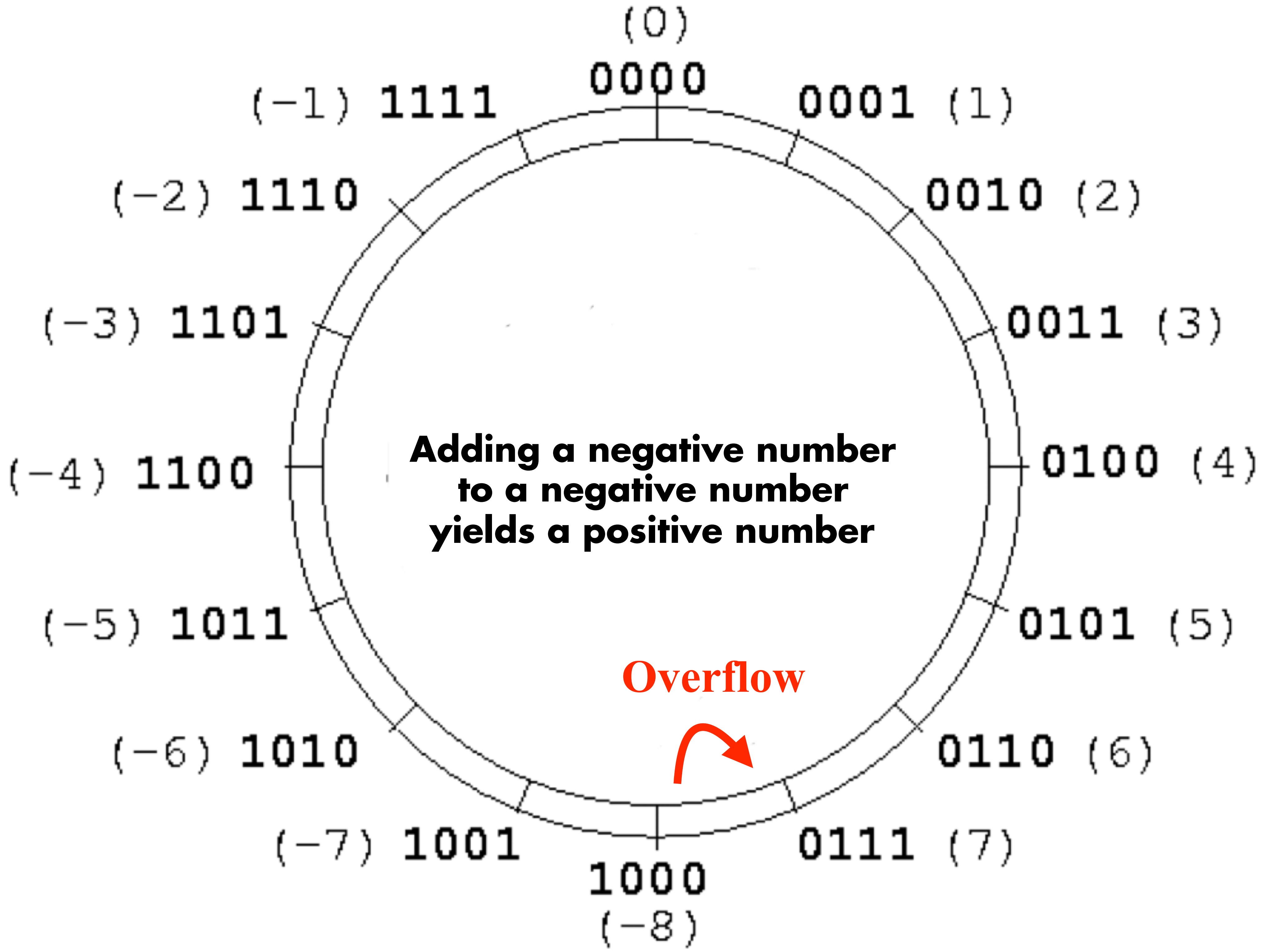

(0)
0000
0001 (1)
0010 (2)
0011 (3)
0100 (4)
0101 (5)
0110 (6)
0111 (7)
1000
(-8)
(-7) 1001
(-6) 1010
(-5) 1011
(-4) 1100
(-3) 1101
(-2) 1110
(-1) 1111
Adding to a positive number to a positive number yields a negative number
Overflow

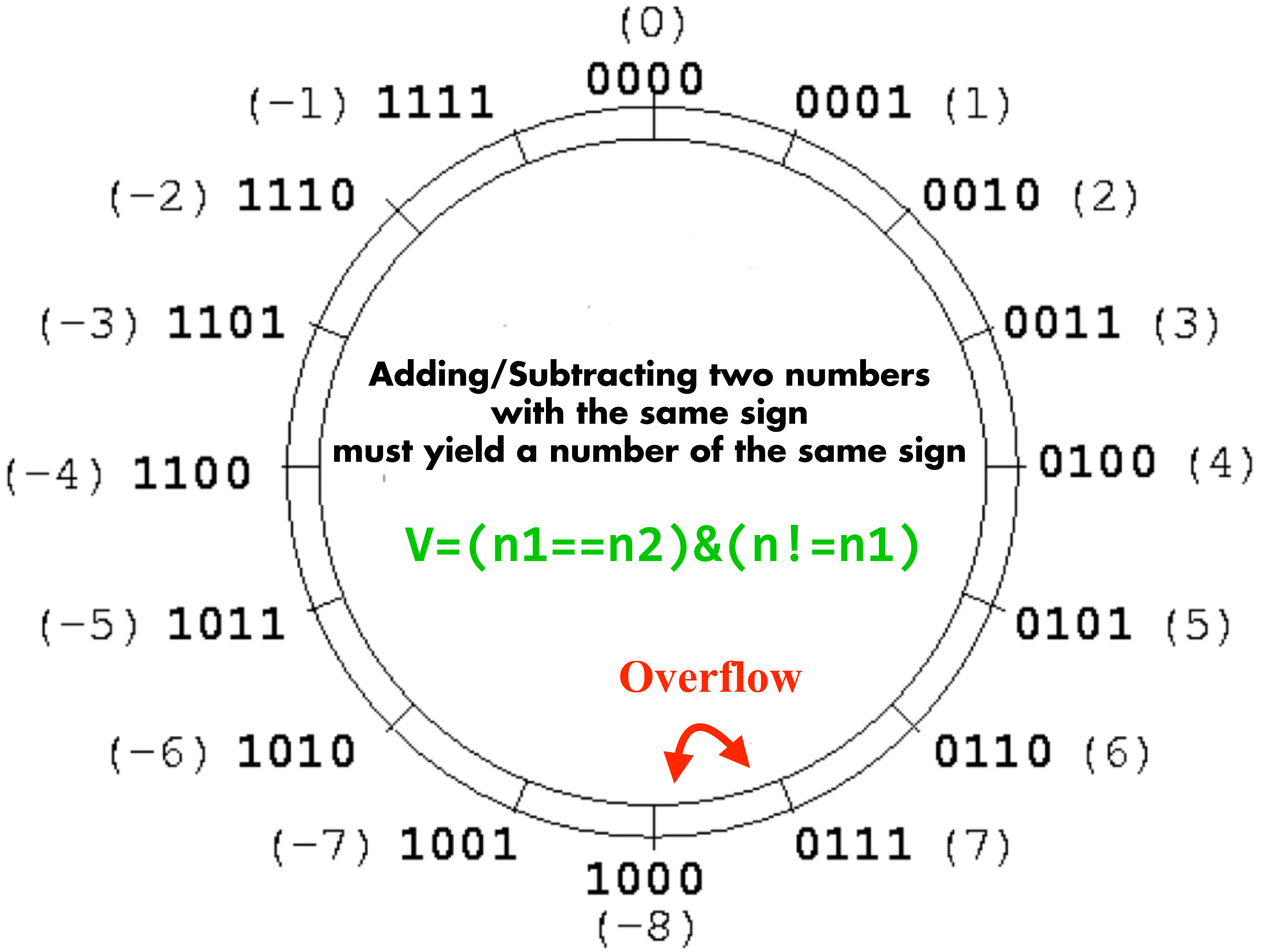
- In two's complement, when you exceed the maximum value of int (2,147,483,647), you "wrap around" to negative numbers:



- Here is the link after Google upgraded to 64-bit integers:




(0) 0000
0001 (1)
0010 (2)
0011 (3)
0100 (4)
0101 (5)
0110 (6)
0111 (7)
1000 (-8)
(-7) 1001
(-6) 1010
(-5) 1011
(-4) 1100
(-3) 1101
(-2) 1110
(-1) 1111
Adding a negative number to a negative number yields a positive number
Overflow



Adding/Subtracting two numbers with the same sign must yield a number of the same sign

`V=(n1==n2)&(n!=n1)`

Comparison
=
Subtract and Look at Flags

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Methods used to compare `signed` and `unsigned` numbers are NOT the same!

Type Conversion

Jedi Job Interview Questions

```c
#include <stdint.h>

uint16_t x = 0xffff;
uint32_t y = x;

// y?
```

Type Hierarchy

$$\begin{tikzcd}
\texttt{uint32} & \texttt{0 … 4294967295 (0xffffffff)} \\
\texttt{uint16} \arrow[u] & \texttt{0 … 65535 (0xffff)} \\
\texttt{uint8} \arrow[u] & \texttt{0 … 255 (xff)}
\end{tikzcd}$$

Types are *sets* of allowed values

Arrow indicate *subsets*: uint16 ⊂ uint32

$$\begin{tikzcd}
\texttt{uint32} \\
\texttt{uint16} \arrow[u] \\
\texttt{uint8} \arrow[u]
\end{tikzcd}$$

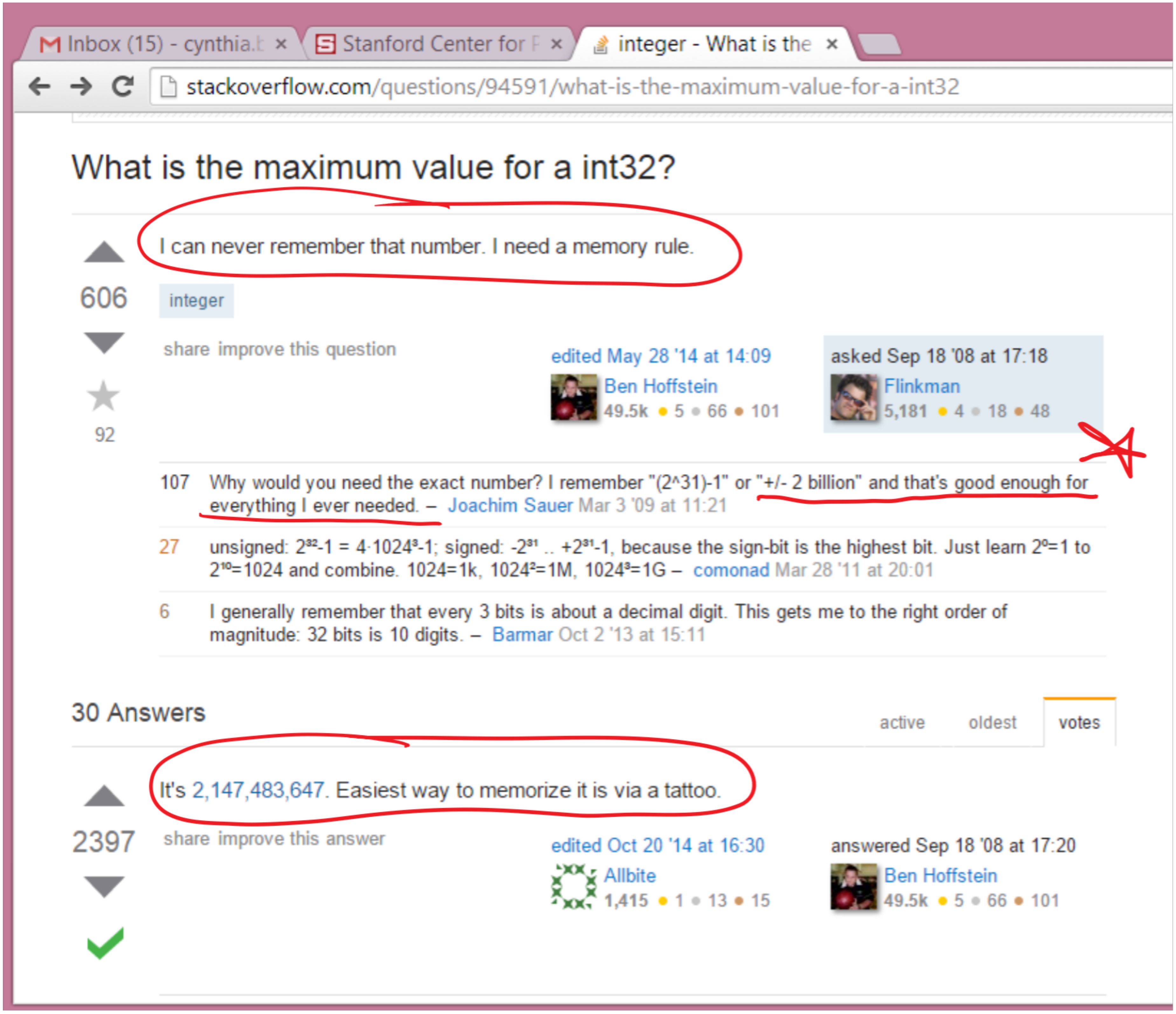
Type Conversion is Safe
(values preserved)

```c
#include <stdint.h>

uint16_t x = 0xffff;
uint32_t y = x;

// y = 0x0000ffff
```

```
int16_t x = -1;
int32_t y =  x;

// y?
```

$$\begin{tikzcd}
\texttt{int32} & \texttt{-2,147,483,648 … 2,147,483,647} \\
\texttt{int16} \arrow[u] & \texttt{-32768 … 32767} \\
\texttt{int8} \arrow[u] & \texttt{-128 … 127}
\end{tikzcd}$$

Type Conversion is Safe
(values preserved)



Inbox (15) - cynthia.
Stanford Center for
integer - What is the

stackoverflow.com/questions/94591/what-is-the-maximum-value-for-a-int32

What is the maximum value for a int32?

606

I can never remember that number. I need a memory rule.

integer

share improve this question

edited May 28 '14 at 14:09
Ben Hoffstein
49.5k ● 5 ● 66 ● 101

asked Sep 18 '08 at 17:18
Flinkman
5,181 ● 4 ● 18 ● 48

92

107 Why would you need the exact number? I remember "(2^31)-1" or "+/- 2 billion" and that's good enough for everything I ever needed. – Joachim Sauer Mar 3 '09 at 11:21

27 unsigned: 2^{32}-1 = 4·1024^3-1; signed: -2^{31} .. +2^{31}-1, because the sign-bit is the highest bit. Just learn 2^0=1 to 2^{10}=1024 and combine. 1024=1k, 1024^2=1M, 1024^3=1G – comonad Mar 28 '11 at 20:01

6 I generally remember that every 3 bits is about a decimal digit. This gets me to the right order of magnitude: 32 bits is 10 digits. – Barmar Oct 2 '13 at 15:11

30 Answers

active oldest votes

2397

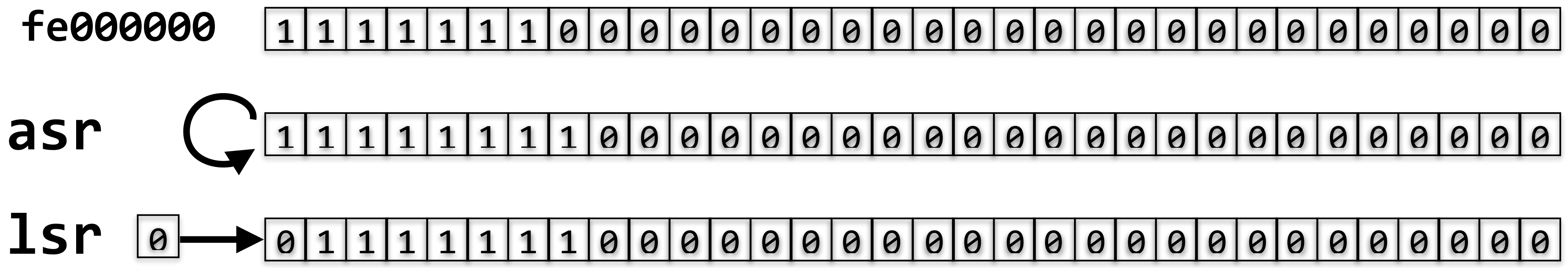
It's 2,147,483,647. Easiest way to memorize it is via a tattoo.

share improve this answer

edited Oct 20 '14 at 16:30
Allbite
1,415 ● 1 ● 13 ● 15

answered Sep 18 '08 at 17:20
Ben Hoffstein
49.5k ● 5 ● 66 ● 101

```
int16_t x = -1;
int32_t y =  x;

// what is the value of y?

// x = -1 = 0xffff
// y = -1 = 0xffffffff
// Sign extension

int8_t 0xfe -> int32_t 0xfffffffe
int8_t 0x7e -> int32_t 0x0000007e

// Assembly language
LSL r0,r0,#24
ASR r0,r0,#24
```



```
// Sign extend instructions: sxth and sxth
```

```
int32_t x = -1;
int16_t y =  x;

// y?
```

```
int32_t x = -1;
int16_t y =  x;

// y = -1
```

```
int32_t x = 4*INT16_MIN;
int16_t y = x;

// y?
```

```
int32_t x = 4*INT16_MIN;
int16_t y = x;

// y = 0
```

⚠ value has changed

$$\begin{tikzcd}
\texttt{int32} \arrow[d] \\
\texttt{int16} \arrow[d] \\
\texttt{int8}
\end{tikzcd}$$

Defined (remove most significant bits)

Dangerous (doesn't preserve all values)

```
int32_t  x = -1;
uint32_t y =  x;

// y?
```

```
int32_t  x = -1;
uint32_t y =  x;

// y = 0xffffffff
```

⚠ value has changed

(y is a large positive number!)

$$\begin{tikzcd} \texttt{uint32} & \texttt{int32} \arrow[l] \\ \texttt{uint16} & \texttt{int16} \arrow[l] \\ \texttt{uint8} & \texttt{int8} \arrow[l] \end{tikzcd}$$

`Defined (copies bits)`

$$\begin{tikzcd}
\texttt{uint32} & \texttt{int32} \arrow[l] \\
\texttt{uint16} & \texttt{int16} \arrow[l] \\
\texttt{uint8} & \texttt{int8} \arrow[l]
\end{tikzcd}$$

Dangerous! (neg become large)

$$\begin{tikzcd}
\texttt{uint32} \arrow[r] & \texttt{int32} \\
\texttt{uint16} \arrow[r] & \texttt{int16} \\
\texttt{uint8} \arrow[r] & \texttt{int8}
\end{tikzcd}$$

Technically Not Defined
(arm: copies bits)

$$\begin{tikzcd}
\texttt{uint32} & \texttt{int32} \\
\texttt{uint16} \arrow[ur] & \texttt{int16} \\
\texttt{uint8} \arrow[ur] & \texttt{int8}
\end{tikzcd}$$

Safe?

$$\begin{tikzcd}
\mathtt{uint32} & \mathtt{int32} \\
\mathtt{uint16} & \mathtt{int16} \arrow[ul] \\
\mathtt{uint8} & \mathtt{int8} \arrow[ul]
\end{tikzcd}$$

Safe?

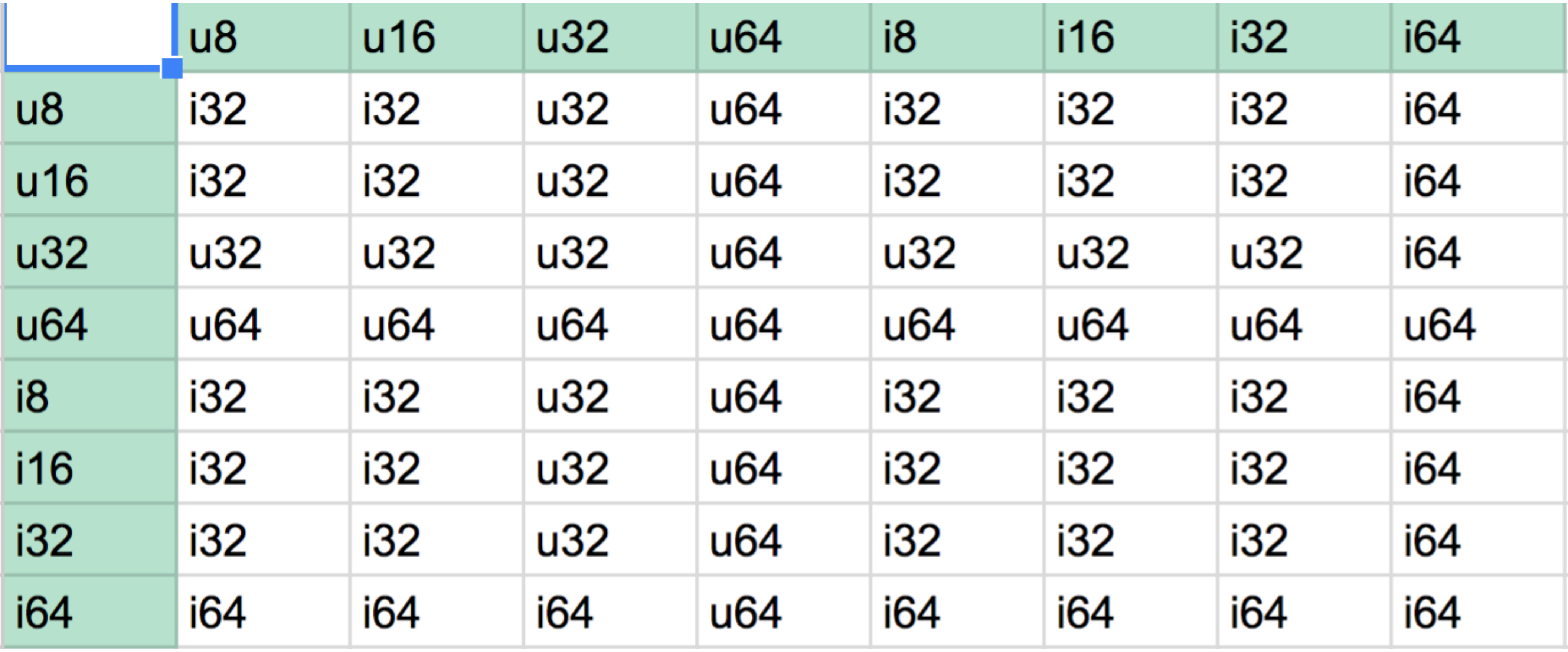
$$\begin{tikzcd}
\mathtt{uint32} & \mathtt{int32} \arrow[l] \\
\mathtt{uint16} & \mathtt{int16} \arrow[l] \arrow[u] \\
\mathtt{uint8} & \mathtt{int8} \arrow[u]
\end{tikzcd}$$

`Defined, Dangerous`

Binary Operations

Type promotions for binary operations

Note that the type of the result can be different than the type of the operands!



	u8	u16	u32	u64	i8	i16	i32	i64
u8	i32	i32	u32	u64	i32	i32	i32	i64
u16	i32	i32	u32	u64	i32	i32	i32	i64
u32	u32	u32	u32	u64	u32	u32	u32	i64
u64	u64	u64	u64	u64	u64	u64	u64	u64
i8	i32	i32	u32	u64	i32	i32	i32	i64
i16	i32	i32	u32	u64	i32	i32	i32	i64
i32	i32	i32	u32	u64	i32	i32	i32	i64
i64	i64	i64	i64	u64	i64	i64	i64	i64

arm-none-eabi-gcc type promotions

Bugs, Bugs, Bugs

```c
#include <stdio.h>

int main(void)
{
    int a = -20;
    unsigned int b = 6;

    if( a < b )
        printf("-20<6 - all is well\n");
    else
        printf("-20>=6 - omg \n");
}
```

Whenever you mix
signed and unsigned numbers
you get in trouble

Bjarne Stroustrup

Summary

Negation is performed by forming the two's complement

- Signed numbers are represented in two's complement (-x = 2^n-x = ~x+1)

In 2's complement,

- Arithmetic between signed and unsigned numbers is identical
- Comparison between signed and unsigned numbers is different

Know the rules for type conversion, watch out for implicit type conversions

C Type Conversion and Promotion Rules

The semantics of numeric casts are:

Casting from a larger integer to a smaller integer (e.g. u32 -> u8) will truncate

Casting from a smaller integer to a larger integer (e.g. u8 -> u32) will zero-extend if the source is unsigned sign-extend if the source is signed

Casting between two integers of the same size (e.g. i32 -> u32) is a no-op

6.3.1.3 Signed and unsigned integers conversions

1 When a value with integer type is converted to another integer type, if the value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

3 Otherwise, if the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

6.3.1.8 Usual arithmetic conversions

1 If both operands have the same type, then no further conversion is needed.

2 Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

3 Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

4 Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

5 Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

```c
unsigned int timer_get_ticks(void)
{
  return *SYSTIMERCLO;
}

void timer_delay_us(unsigned int usecs)
{
  unsigned int start=timer_get_ticks();
  while (timer_get_ticks()-start < usecs)
     { /* spin */ }
}

// The timer continuously ticks.
// Does this code work if the timer overflows?
```