Computer Arithmetic

Signed and Unsigned

Topics

Overflow

Addition and carry, subtraction and borrow

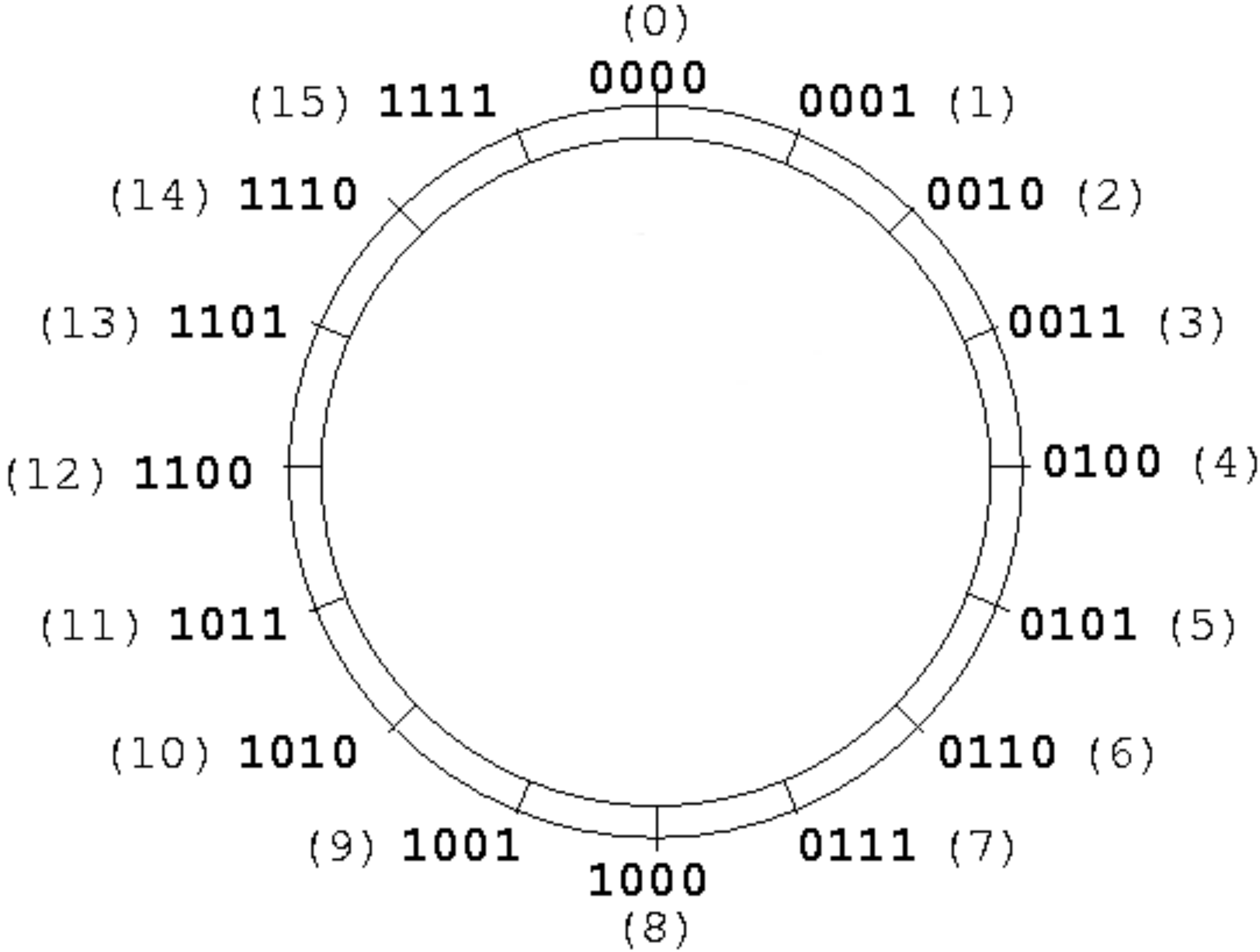
Processor flags: Z, N, C, V

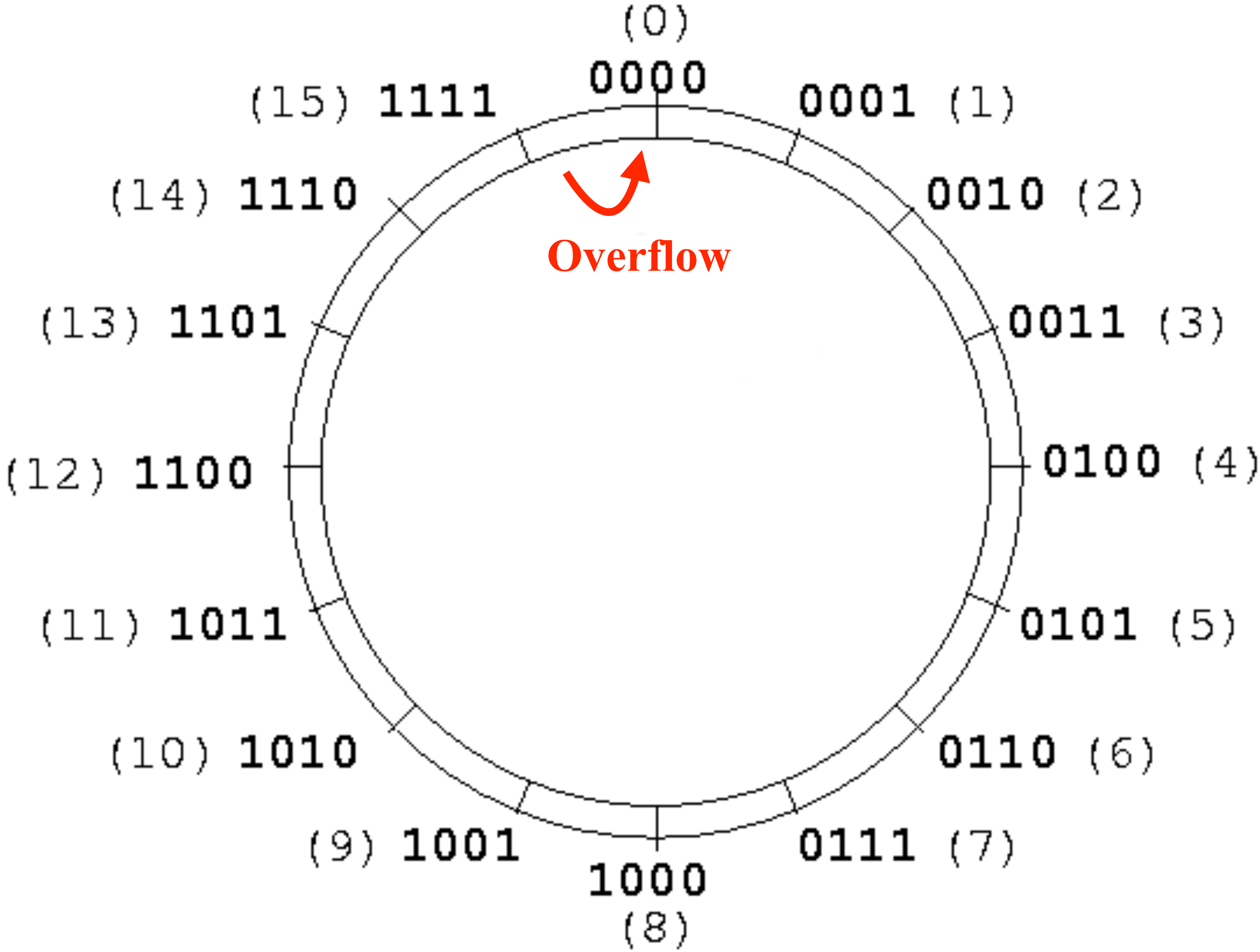
2's complement representation of negative numbers

Comparisons

Implicit type conversions (C craziness)


999999
999999
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


(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

What Happens at Overflow?

Grade School Addition

```
 247
   9
 ---
```

Grade School Addition

```
  1
 247
   9
 ---
   6
```

Grade School Addition

```
  1
 247
   9
 ---
  56
```

Grade School Addition

```
  1
 247
  09
 ---
 256
```

Addition (Hexadecimal)

```
  F7
  09
 ---
```

Addition (Hexadecimal)

```
  1
  F7
  09
 ---
   0
```

Addition (Hexadecimal)

```
 11
  F7
  09
 ---
  00
```

Addition (Hexadecimal)

```
 11
  F7
  09
 ---
 100
```

Addition (Binary)

```
  11110111
  00001001
 ---------
```

Addition (Binary)

```
        1
  11110111
  00001001
 ---------
         0
```

Addition (Binary)

```
     11
 11110111
 00001001
---------
       00
```

Addition (Binary)

```
     111
 11110111
 00001001
---------
      000
```

Addition (Binary)

```
    1111
 11110111
 00001001
---------
     0000
```

Addition (Binary)

```
    11111
  11110111
  00001001
----------
     00000
```

Addition (Binary)

```
   111111
  11110111
  00001001
 ---------
    000000
```

Addition (Binary)

```
  1111111
  11110111
  00001001
---------
   0000000
```

Addition (Binary)

```
 11111111
  11110111
  00001001
 ---------
  00000000
```

Addition (Binary)

```
 11111111
  11110111
  00001001
 ---------
  00000000
```

Result: 00000000 (only room for 8-bits)
Carry (C): 1 (extra bit)

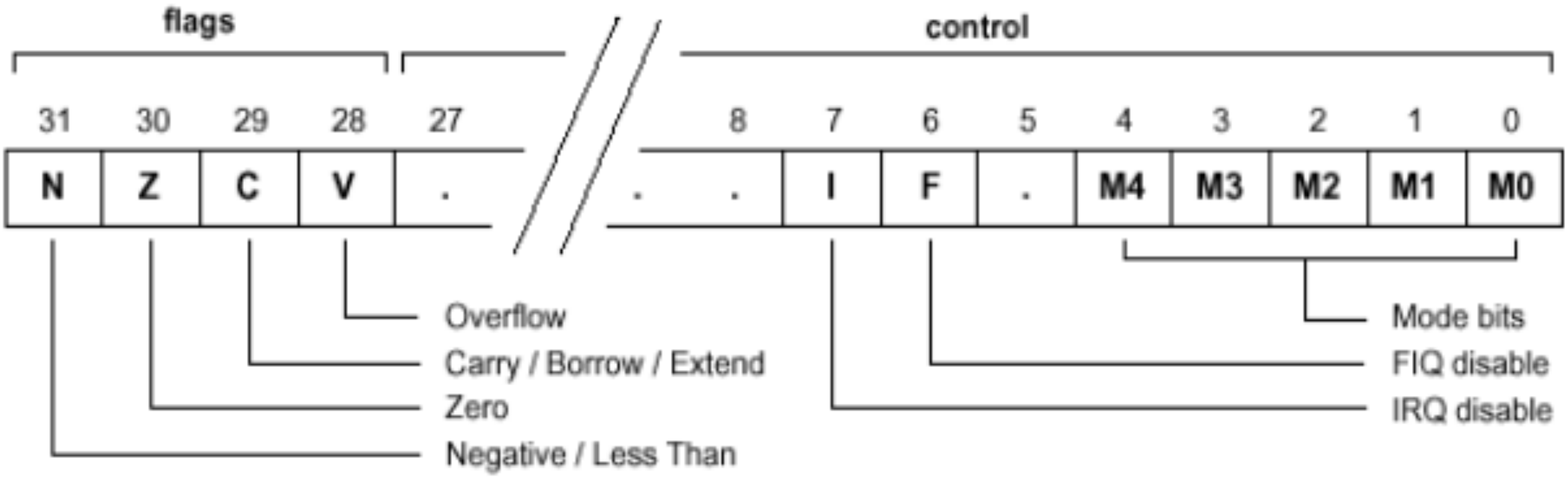
To hold the result of adding two n-bit numbers requires n+1 bits

add32/

add32

```
00000000 + 00000000 = 00000000 : Z=1, C=0
00000000 + 00000001 = 00000001 : Z=0, C=0
00000001 + 00000001 = 00000002 : Z=0, C=0
00000002 + 00000001 = 00000003 : Z=0, C=0
...
fffffffe + 00000001 = ffffffff : Z=0, C=0
ffffffff + 00000001 = 00000000 : Z=1, C=1
...
ffffffff + ffffffff = fffffffe : Z=0, C=1
```

CPSR



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

(We will cover N soon, and V later)

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

int64_t add64(int64_t a, int64_t b)
{
  return a + b;
}

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

Negative (`signed`) Numbers

Up to now, all binary numbers have been positive (unsigned)

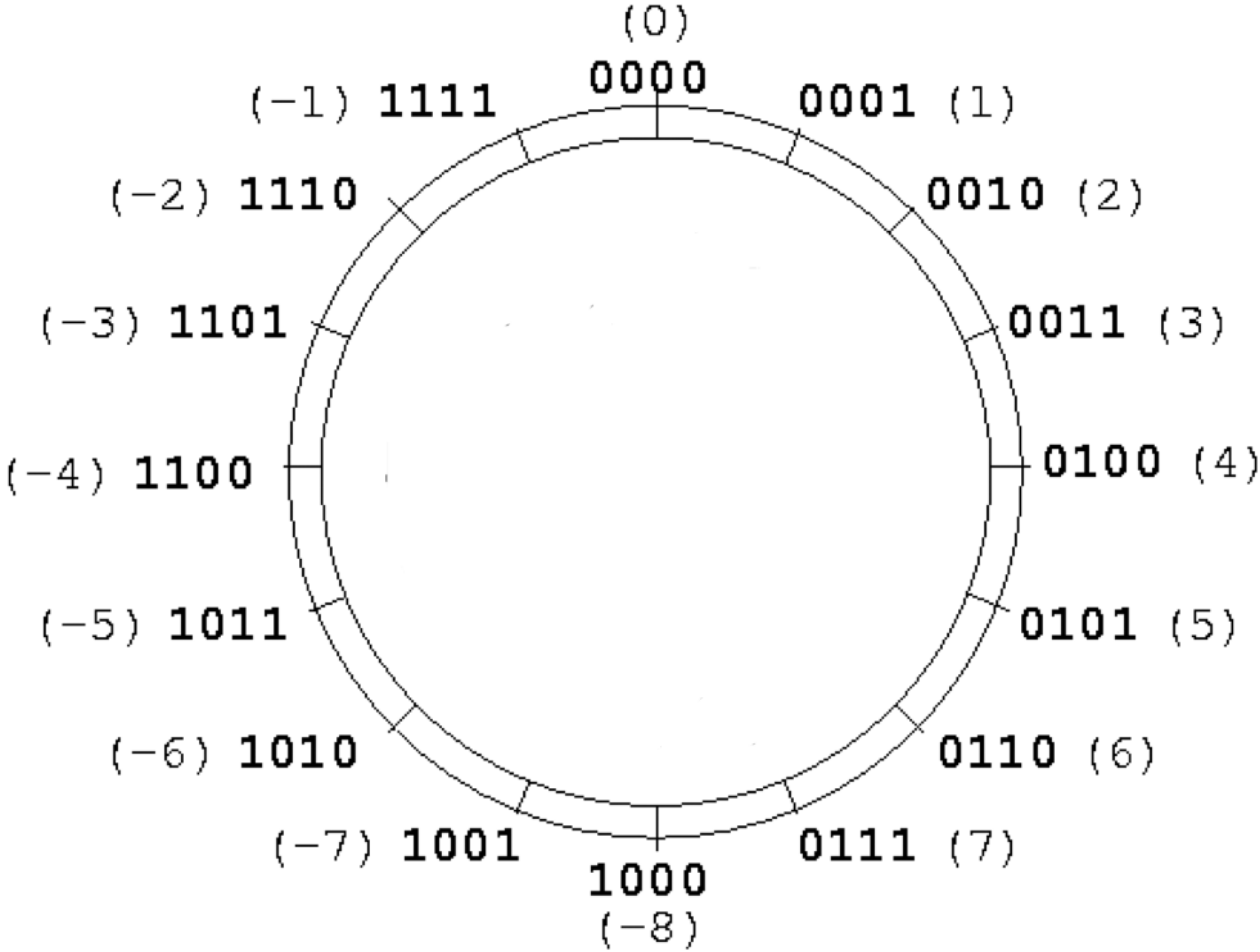
How would you define negative (signed) numbers?

A clever way of defining -1 is to say that -1 is the number that when added to 1, results in 0 (mod 16)

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

0xf can be *interpreted* as -1

Note that modulo addition of signed numbers is exactly the same as addition of unsigned numbers!!


(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

```
For 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)

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

How do we negate an 8-bit number?

Subtract the number from 100000000!

```
 11111111
 100000000
  00000001
 ---------
  11111111
```

11111111 (0xff) is -1

The 1's on the top are borrows

Another way

```
Rewrite 100000000 as 11111111 + 1
-x =(11111111+1)-x =(11111111-x)+1 = ~x+1

  11111111
 -00000001
 ---------
  11111110 = ~00000001 (~ means flip bits)

  11111110
 +00000001
 ---------
  11111111
```

```
Subtraction is converted to

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

Why?
- Only one arithmetic op: add
- Need to think like this
  to understand when a carry is generated
```

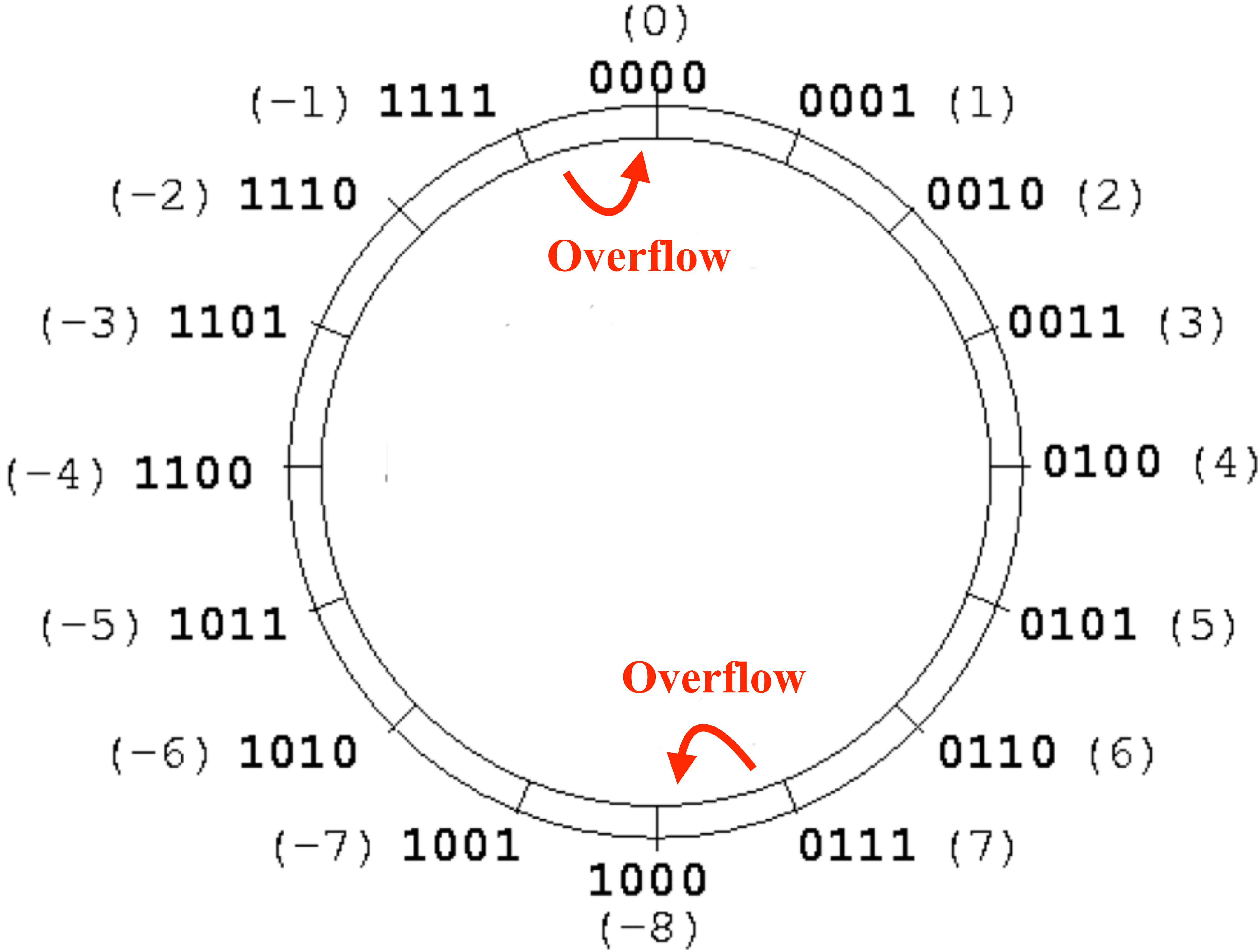
```
sub32/

00000000-00000000 = 00000000+00000000 = 00000000
00000000-00000001 = 00000000+ffffffff = ffffffff
ffffffff-00000001 = ffffffff+ffffffff = fffffffe
fffffffe-00000001 = fffffffe+ffffffff = fffffffd
...
00000000-ffffffff = 00000000+00000001 = 00000001
00000001-ffffffff = 00000001+00000001 = 00000002
00000001-fffffffd = 00000001+00000003 = 00000004
...
ffffffff-ffffffff = ffffffff+00000001 = 00000000
ffffffff-fffffffd = ffffffff+00000003 = 00000002
```

```c
// What happens if the timer overflows?
unsigned timer_get_time(void) {
    return GET32(SYSTIMERCLO);
}

void delay_us(unsigned us) {
    unsigned rb = timer_get_time();
    while (1) {
        unsigned ra = timer_get_time();
        // subtraction results in a positive
        // value even if the timer overflows
        if ((ra - rb) >= us) {
            break;
        }
    }
}
```

Addition / subtraction of `signed` and `unsigned` numbers is the same!

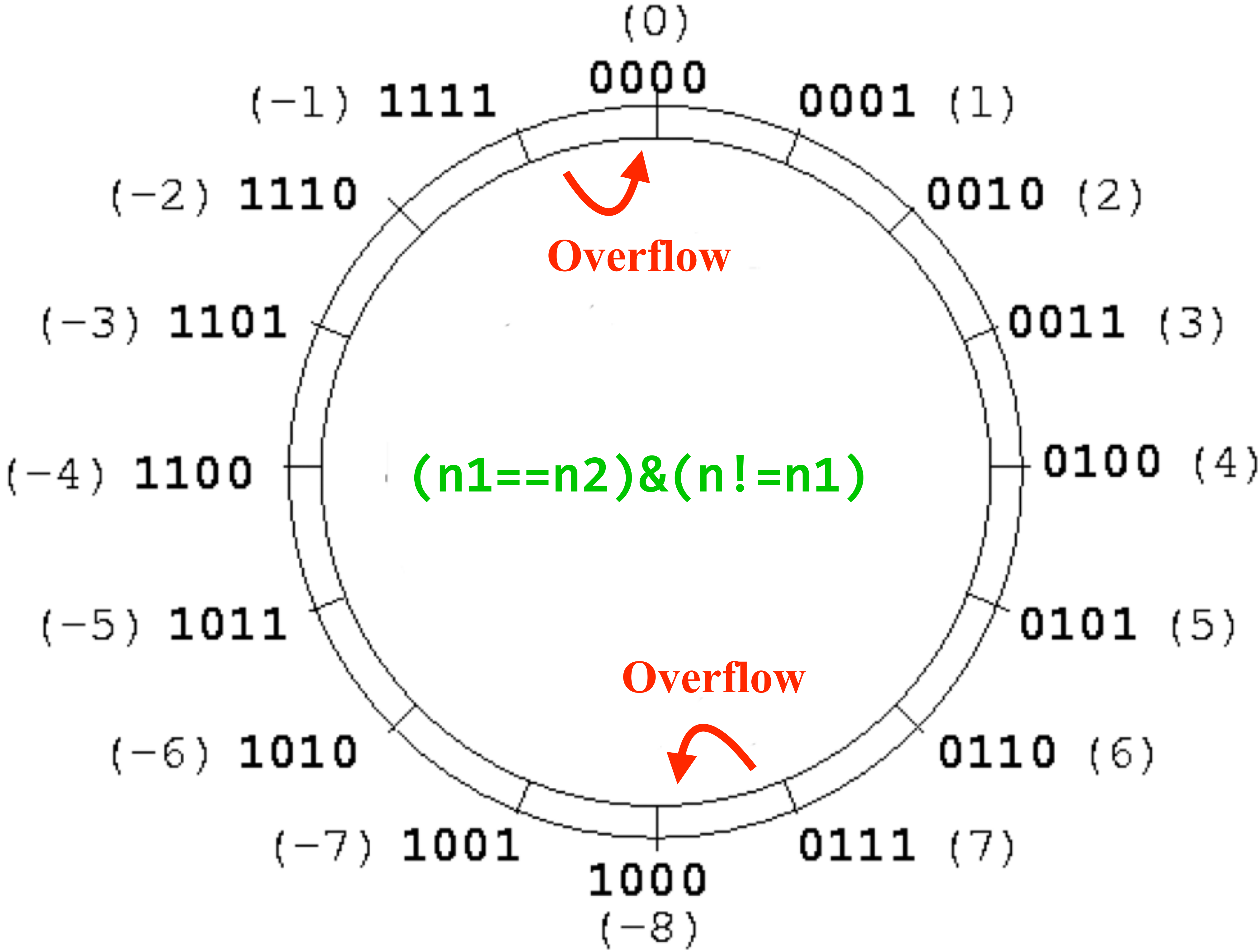
Comparison

```
unsigned comparison: ucmp32/
>
00000001-00000000=00000001: C=1
ffffffff-00000001=fffffffe: C=1
ffffffff-00000000=ffffffff: C=1
==
00000000-00000000=00000000: C=1
00000001-00000001=00000000: C=1
ffffffff-ffffffff=00000000: C=1
<
00000000-00000001=ffffffff: C=0
00000000-ffffffff=00000001: C=0
00000001-ffffffff=00000002: C=0

uge = c
```


(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
Overflow
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
Overflow
Overflow
(n1==n2)&(n!=n1)

```
signed comparison cmp32/
>
7fffffff - 00000000 = 7fffffff: N=0, V=0
00000000 - 80000000 = 80000000: N=1, V=1
7fffffff - 80000000 = ffffffff: N=1, V=1
==
00000000 - 00000000 = 00000000: N=0, V=0
80000000 - 80000000 = 00000000: N=0, V=0
7fffffff - 7fffffff = 00000000: N=0, V=0
<
00000000 - 7fffffff = 80000001: N=1, V=0
80000000 - 00000000 = 80000000: N=1, V=0
80000000 - 7fffffff = 00000001: N=0, V=1
```

overflow occurs if the sign of the two numbers being compared are the same, and the sign of the result is different (n1==n2) & (n!=n1)

```
signed comparison cmp32/
>
7fffffff - 00000000 = 7fffffff: N=0, V=0
00000000 - 80000000 = 80000000: N=1, V=1
7fffffff - 80000000 = ffffffff: N=1, V=1
==
00000000 - 00000000 = 00000000: N=0, V=0
80000000 - 80000000 = 00000000: N=0, V=0
7fffffff - 7fffffff = 00000000: N=0, V=0
<
00000000 - 7fffffff = 80000001: N=1, V=0
80000000 - 00000000 = 80000000: N=1, V=0
80000000 - 7fffffff = 00000001: N=0, V=1

bge: signed greater than or equal (n == v)
blt: signed less than (n != v)

int ge() { return !v ? !n : n }
```

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

Comparison of `signed` and `unsigned` numbers is NOT the same!

Type Conversion

Jedi Job Interview Questions

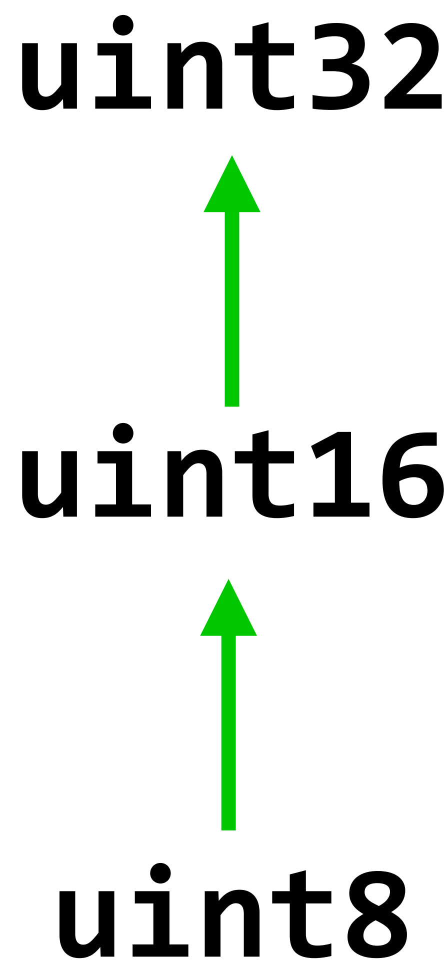
```
uint16_t x = 0xffff;
uint32_t y = x;

// what is the value of y?
```

Type Hierarchy

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

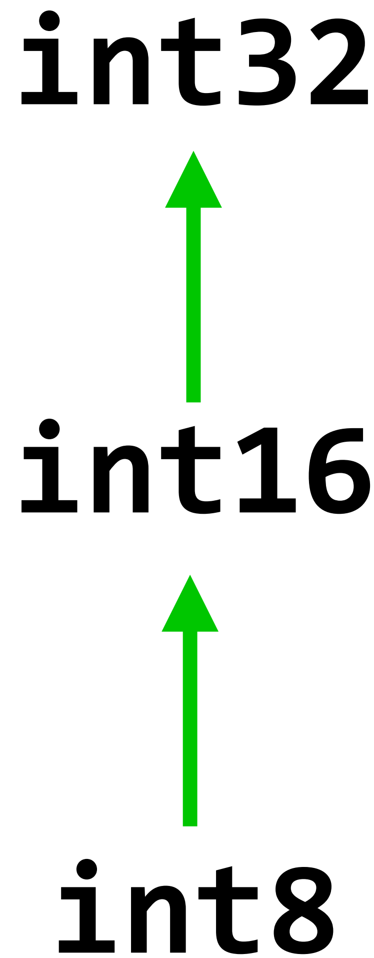
Types are *sets* of allowed values

Arrow indicate *subsets*: uint8 ⊂ uint16



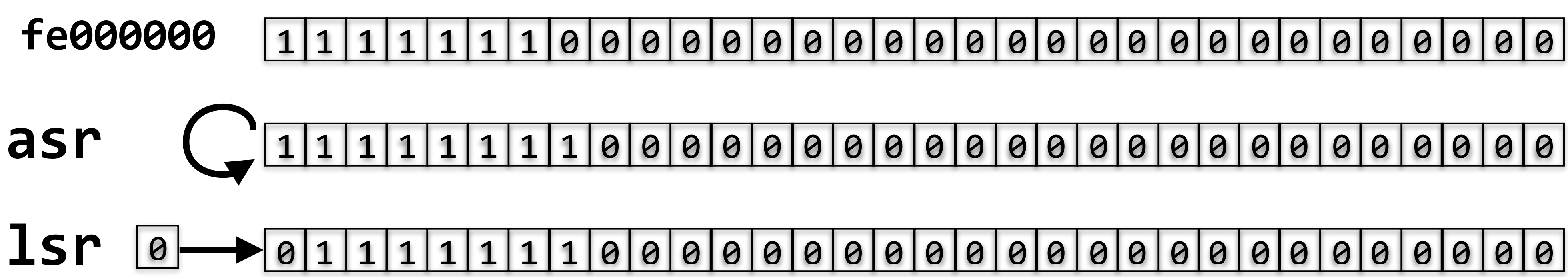
Safe (values preserved)

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

// what is the value of y?
```



Safe (values preserved)

```
// Sign extend

int8 0xFE -> int32 0xFFFFFFFE
int8 0x7E -> int32 0x0000007E


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



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

// what is the value of y?
```

int32

int16

int8

Defined (throw away msb)
Dangerous (doesn't preserve all values)

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

// what is the value of y?
```

uint32	←	int32
uint16	←	int16
uint8	←	int8

Defined
(copies bits*)

uint32 ← int32

uint16 ← int16

uint8 ← int8

Dangerous! (neg become large)

```
uint32_t x = 0xffffffff;
int32_t  y = x;

// what is the value of y?
```

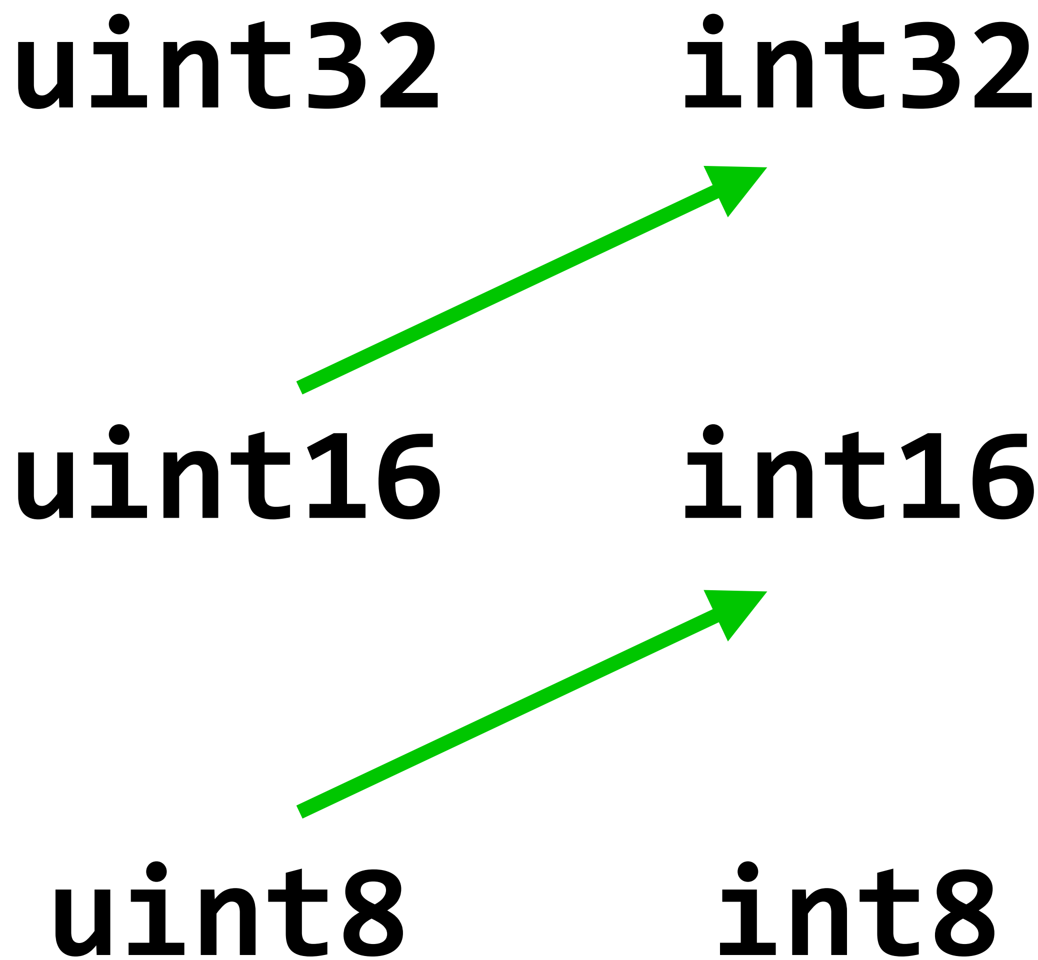
uint32 → int32

uint16 → int16

uint8 → int8

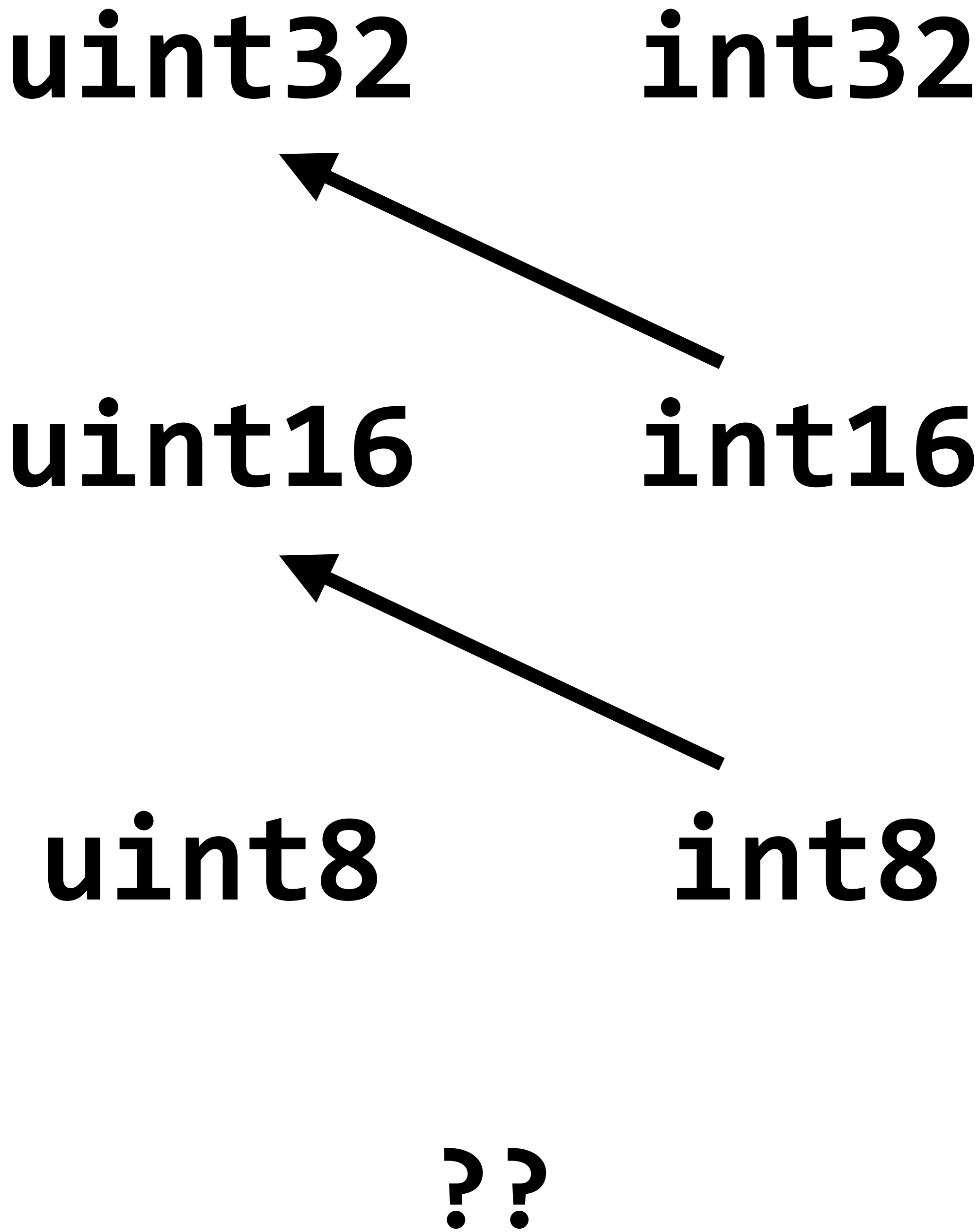
Technically Not Defined
(copies bits)

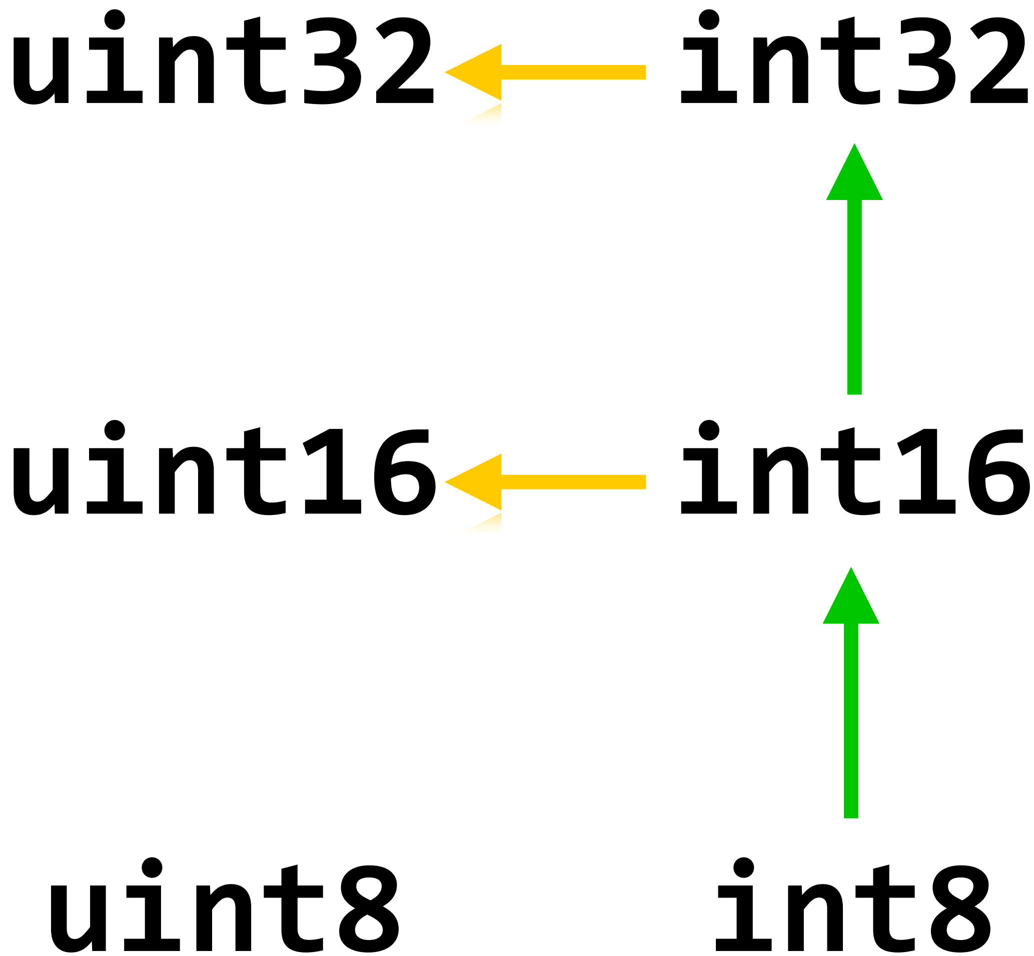
```
uint16_t x = 0xffff;
int32_t  y = x;

// what is the value of y?
```



Safe (values preserved)

```c
int16_t x = -1;
uint32_t y = x;

// what is the value of y?
```


uint32
int32
uint16
int16
uint8
int8
??



Defined, Dangerous

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

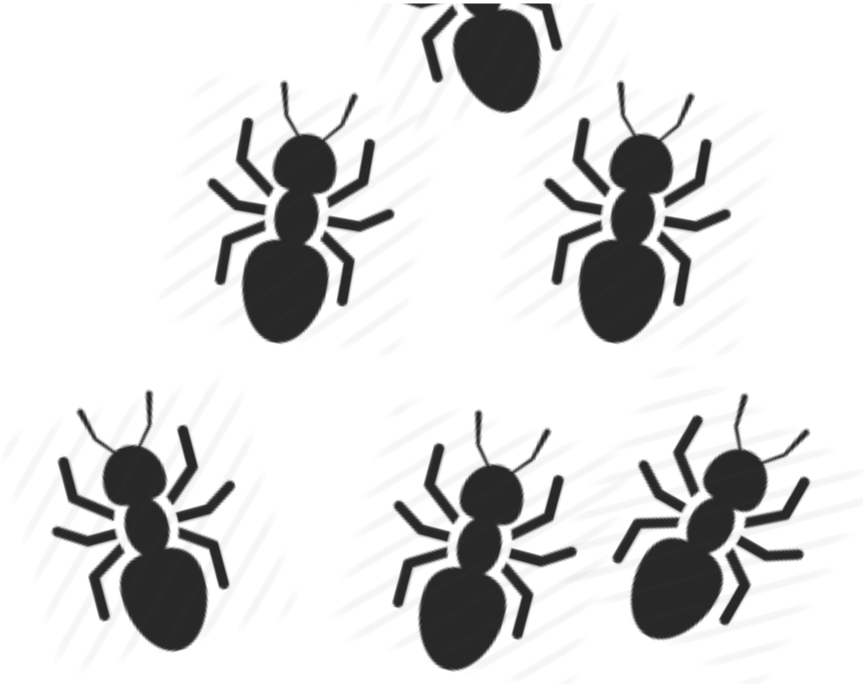
Casting between two integers of the same size (e.g. i32 -> u32) is a no-op

Binary Operations

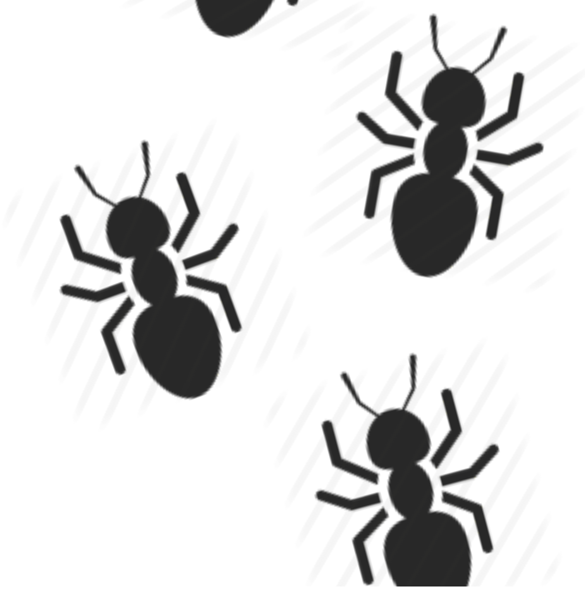
Implicit Type Conversion (Coercion)

Resulting type from a binary operator (ARM32)

	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



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

C Promotion Rules

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.