Computer Arithmetic

and

Speed

What is the difference between (signed) int and unsigned int?

and

How to make your code fast.

Addition

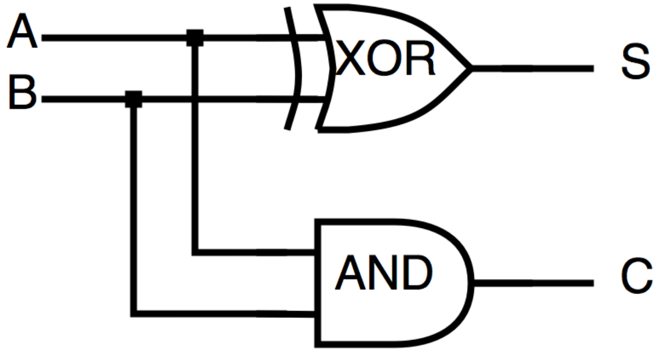
Adding 2 1-bit numbers (Half Adder)

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

Adding 2 1-bit numbers (Half Adder)

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

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



Have reduced addition to logical operations!

Adding 2 8-bit numbers

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

Adding 2 8-bit numbers

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

Adding 2 8-bit numbers

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

Adding 2 8-bit numbers

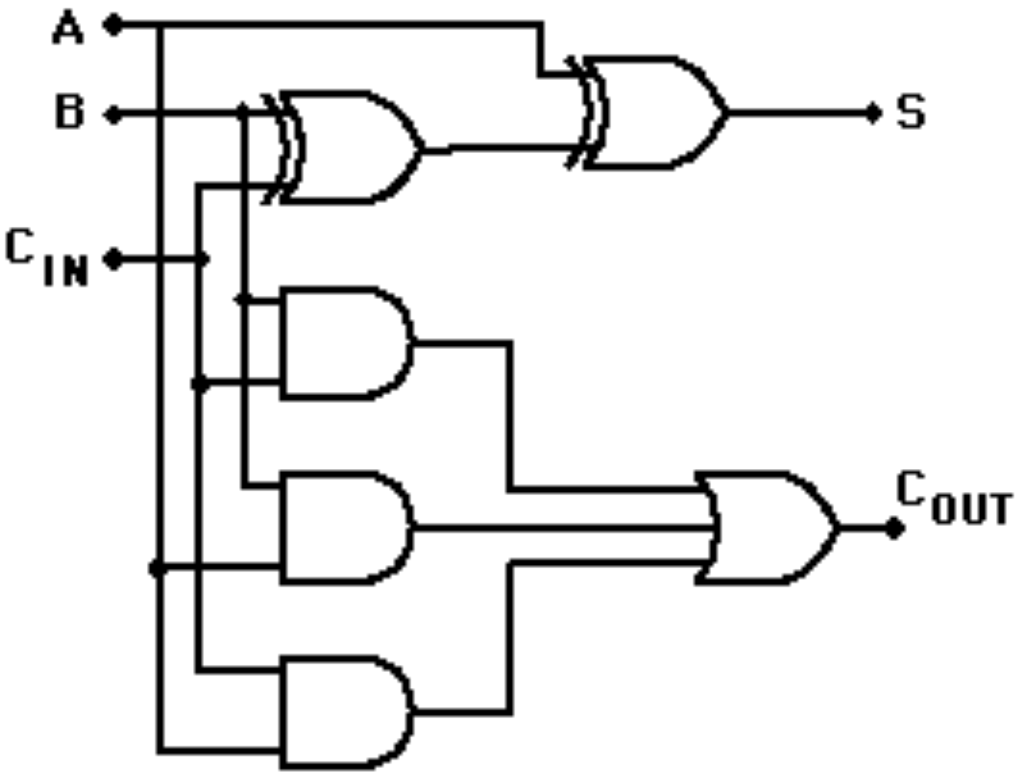
```
00001111  Carry
 00000111 A
+00001011 B
---------
 00010010 Sum
```

Adding 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

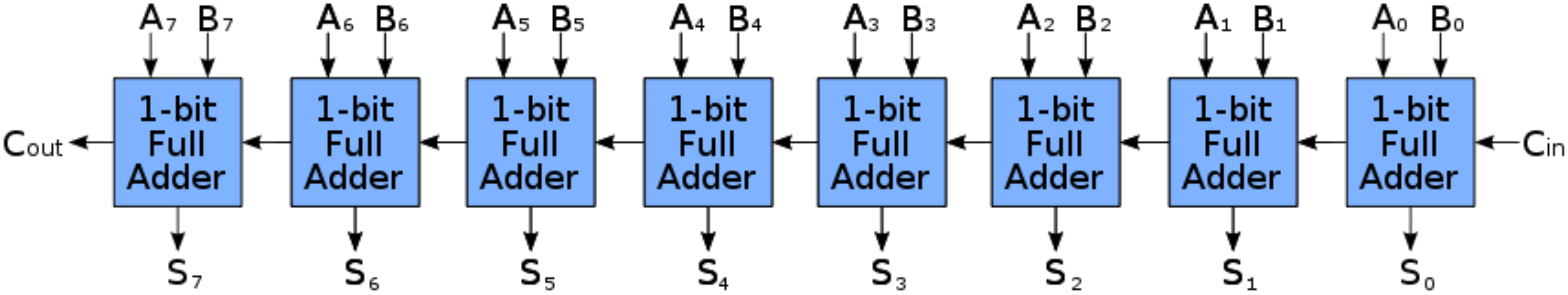
Adding 3 1-bit numbers (Full Adder)

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



```
s  = a^b^ci
co = (a&b)|(b&c)|(c&a)
```

8-bit Ripple Adder



Note Cin (carry in) and Cout (carry out)

```
// Multiple precision addition
// https://gcc.godbolt.org/z/6TRmY8

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

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

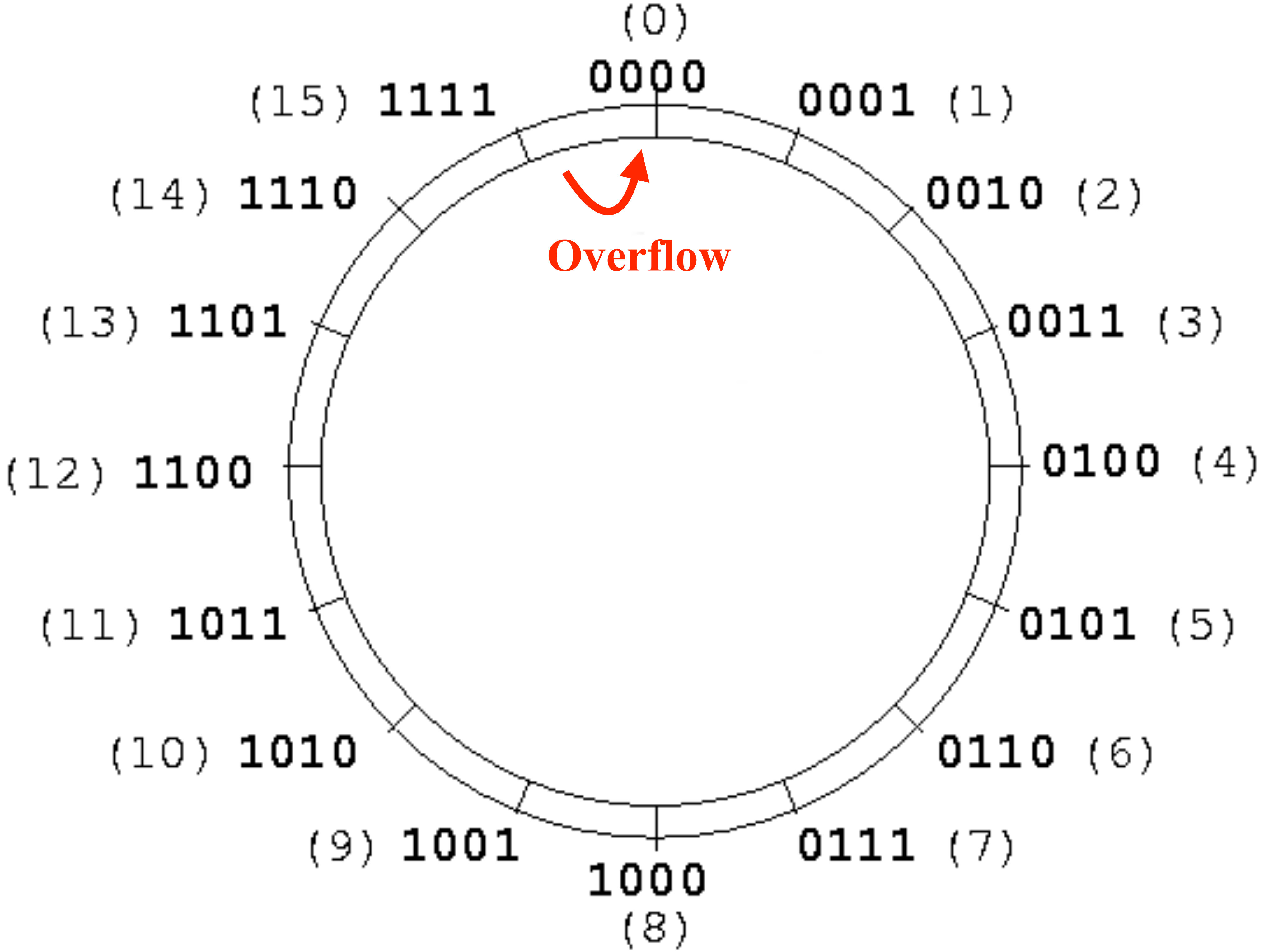
Binary Addition - Modular Arithmetic

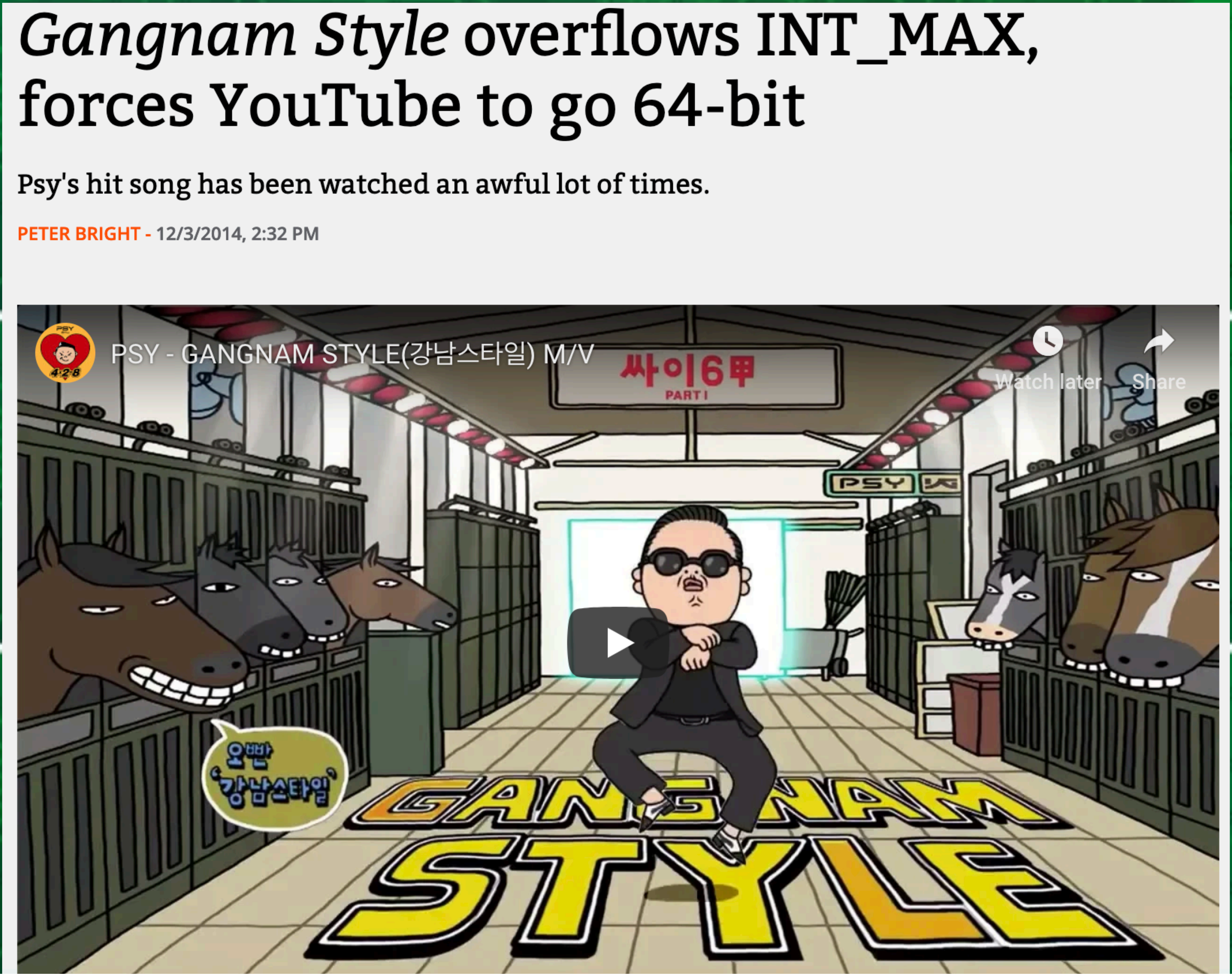
```
11111111  Carry
 11111111 A
+00000001 B
---------
100000000 Sum
```

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

But we only have 8-bits!

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


(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



Gangnam Style overflows INT_MAX, forces YouTube to go 64-bit

Psy's hit song has been watched an awful lot of times.

PETER BRIGHT - 12/3/2014, 2:32 PM

Subtraction

BIG IDEA: Define subtraction using addition

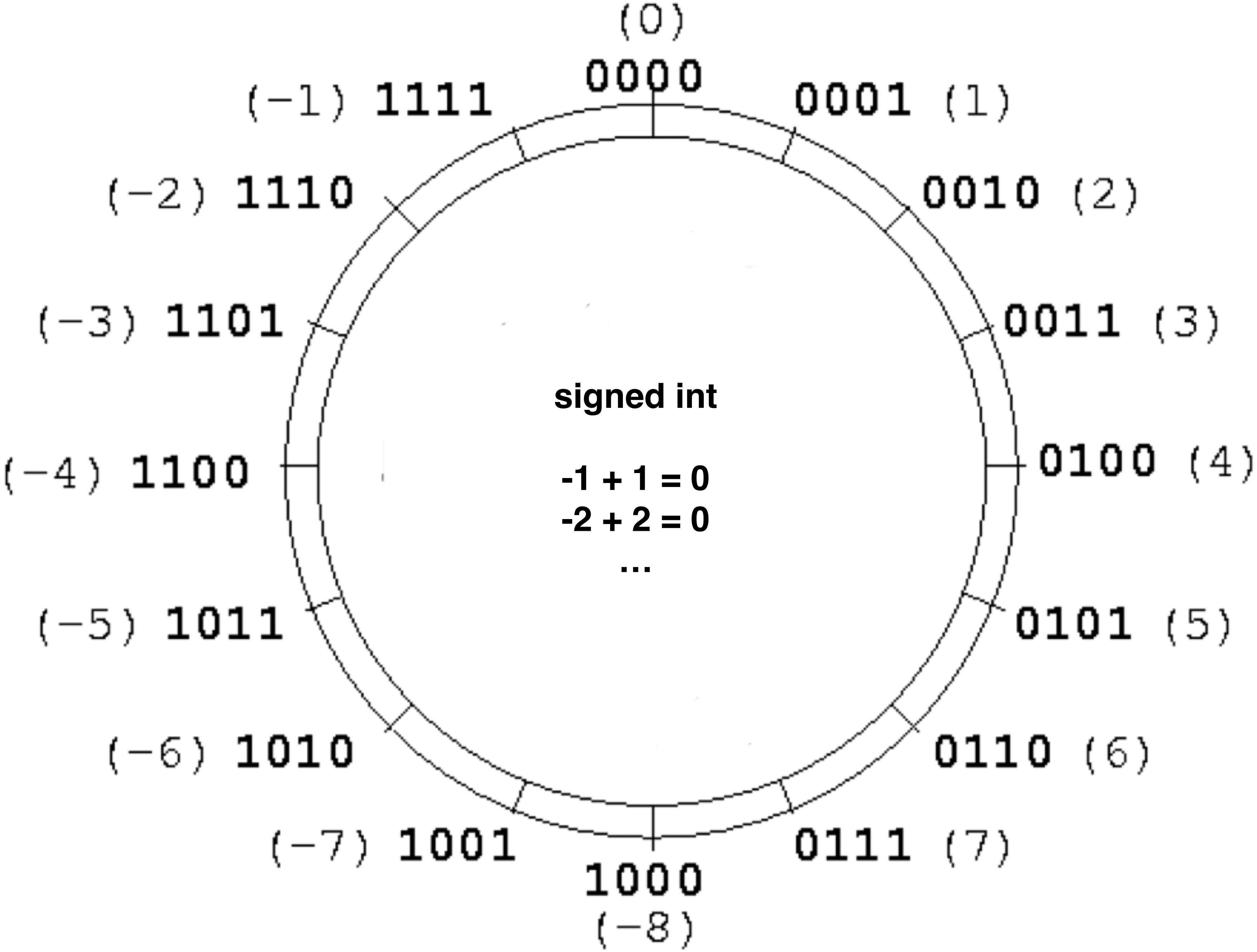
A clever way of defining subtraction by 1 is to find a number to add that yields the same result as the subtract by 1.

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



signed int

-1 + 1 = 0
-2 + 2 = 0
…

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?

Find a number -x, s.t. `(x + (-x)) % 256 = 0`

Subtract it from 256 = 2^8 = 100000000
`-x = 100000000 - x`
Since then `(x + (-x)) % 256 = 0`

E.g., for 1:

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

Thus the term *two's complement*

Another way to negate

Rewrite `100000000 = (11111111 + 1)`

```
-x = (11111111+1)-x
   = (11111111-x)+1
   = ~x + 1
```

Bitwise invert: `~x = 11111111-x` (one's complement)

For example, `-1`

```
~00000001 = 11111111
           -00000001
           ---------
            11111110

 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!

```
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);
}

// The timer continuously ticks.

// Does this code work if the timer
// overflows?
```

Addition and Subtraction of signed and unsigned numbers are the same!

Comparison (cmp)

Subtract and throw away result

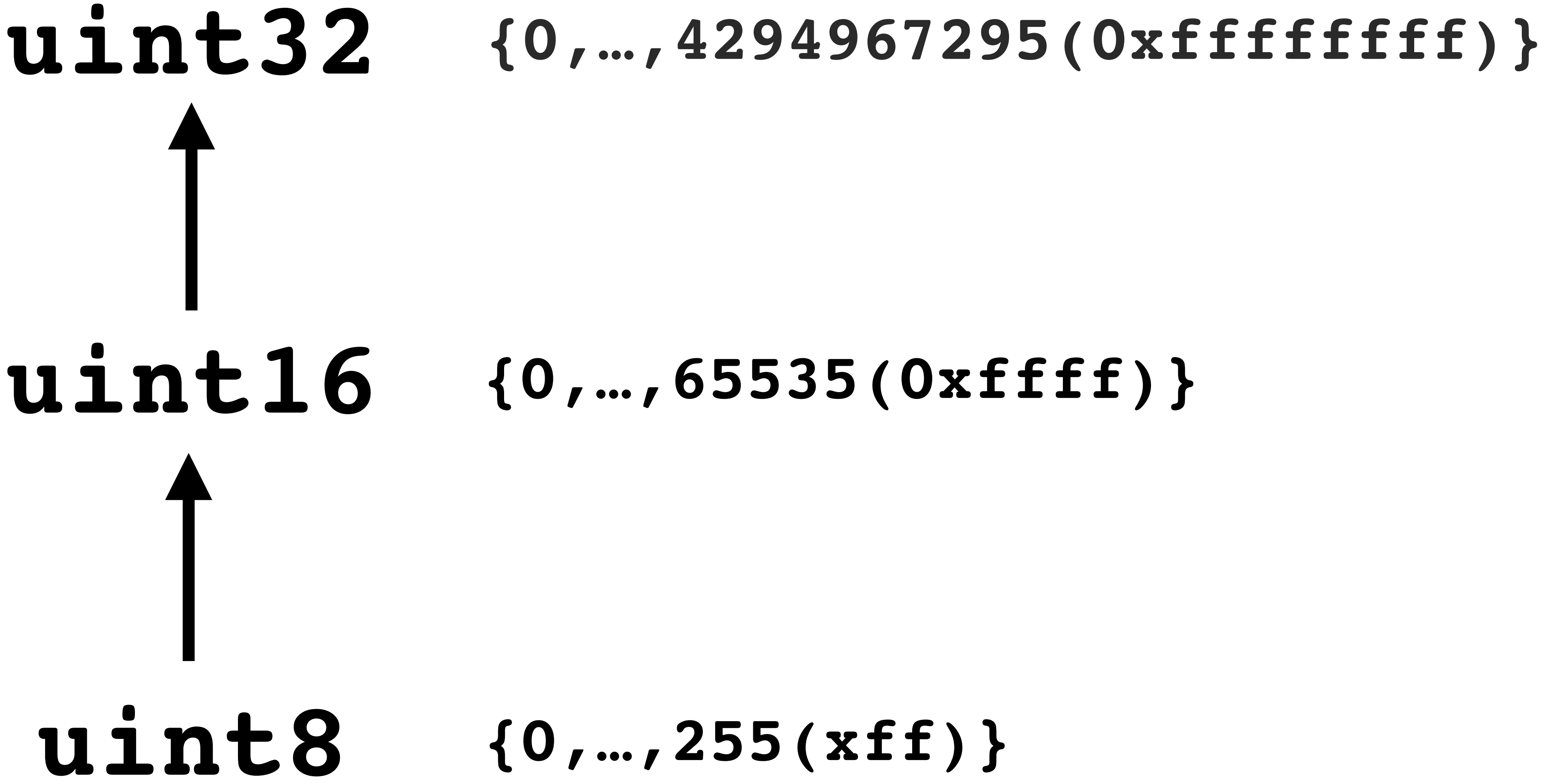
Always set the Flags

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!

Types
and
Type Conversion

Unsigned Type Hierarchy



uint32 {0,…,4294967295(0xffffffff)}

uint16 {0,…,65535(0xffff)}

uint8 {0,…,255(xff)}

Types are *sets* of allowed values

Arrow indicate *subsets*: uint16 ⊂ uint32

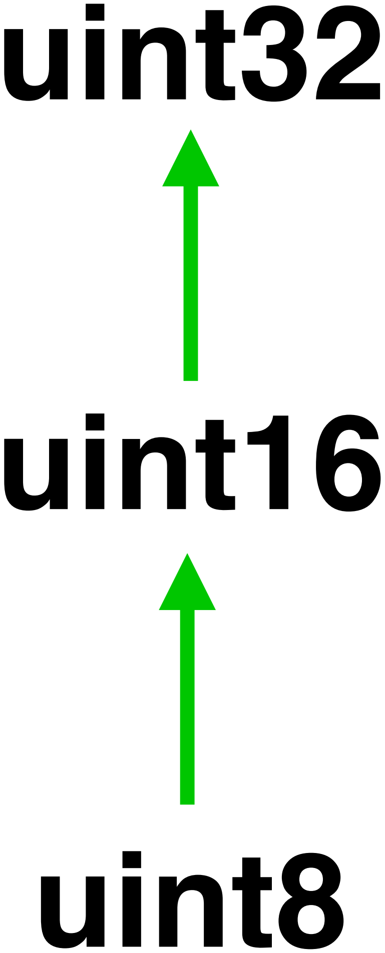
Type Conversion

Type conversion is a way of converting data from one type to another type

Explicit type conversion means that the programmer must specify type conversions. Often called *casting*.

Implicit type conversions means that the language has rules for automatically performing type conversion. Often called *coercion*

Casting usually refers to a reinterpretation of the same bits as a different type (int* a = void* b)



uint32

uint16

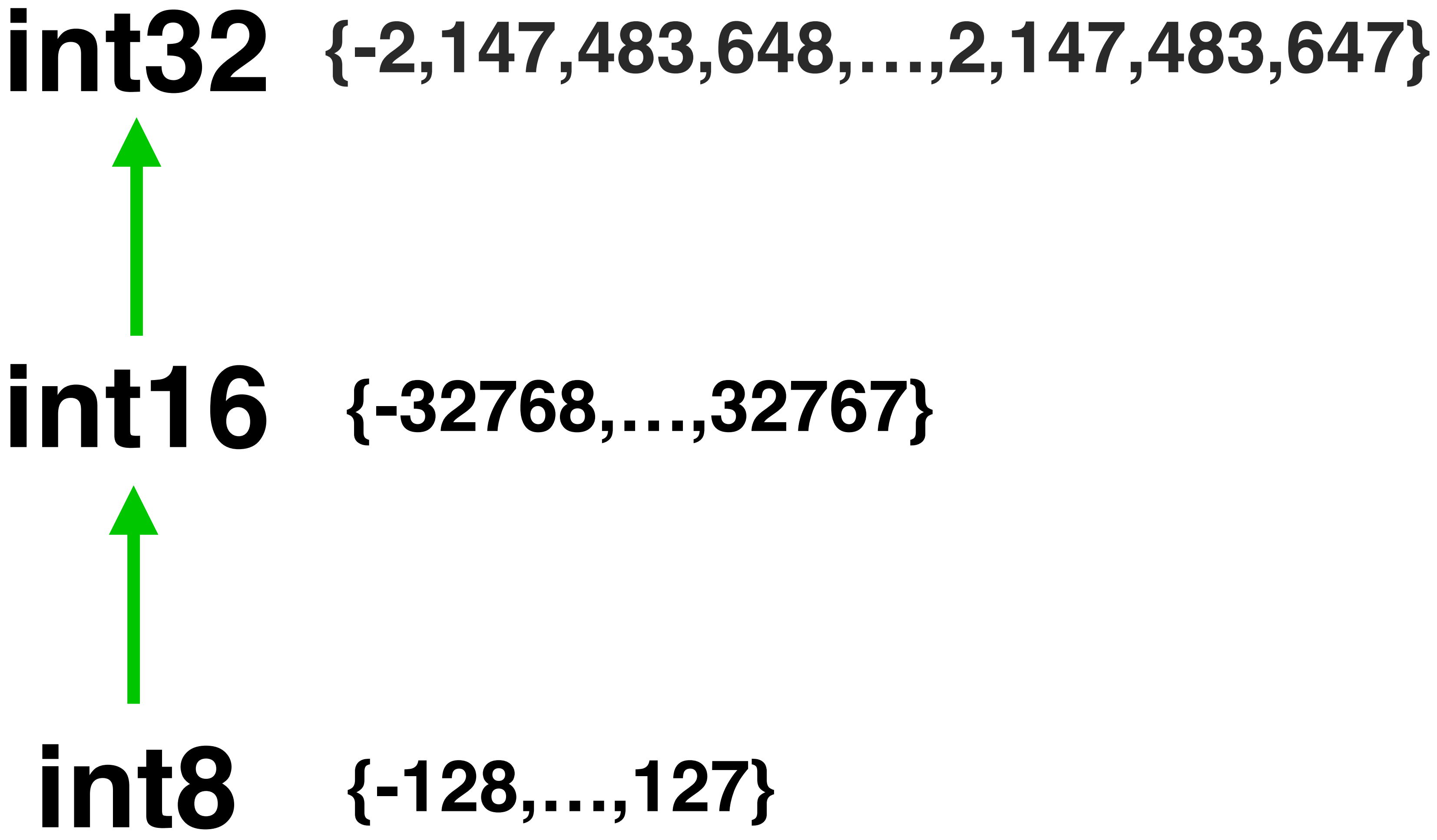
uint8

Type *Promotion* is Safe (values preserved)

```c
#include <stdint.h>

uint16_t x = 0xffff;
uint32_t y = x;

// x = 0xffff
// y = ?
```

```
#include <stdint.h>

uint16_t x = 0xffff;
uint32_t y = x;

// x = 0xffff
// y = 0x0000ffff
```

Signed Type Hierarchy



int32 {-2,147,483,648,…,2,147,483,647}

int16 {-32768,…,32767}

int8 {-128,…,127}

Arrow indicate *subsets*: int16 ⊂ int32

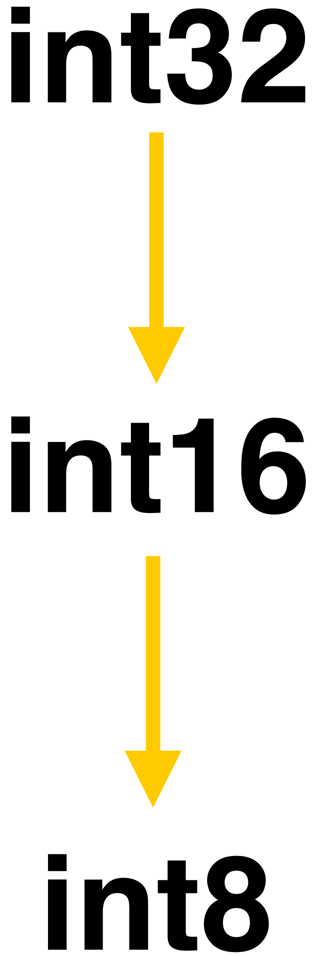
```
int16_t x = -1;
int32_t y =  x;

// x = -1
// y = ?
```

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

// x = -1
// y = -1
// positive
int16_t x =  1;
int32_t y =  x;

// x = 1 = 0x0001
// y = 1 = 0x00000001

// negative
int16_t x = -1;
int32_t y =  x;

// x = -1 = 0xffff
// y = -1 = 0xffffffff
// To preserve signed values need sign extension

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


// Sign extend instructions:
//
//  sxtb - sign extend byte to word
//  sxth - sign extend half word to word
//
```

```
int32_t x = 0x80000;
int16_t y = x;

// x = 0x80000
// y = ?
```

```
int32_t x = 0x80000;
int16_t y = x;

// x = 0x80000
// y =  0x0000
```

⚠ value has changed



int32

int16

int8

Defined (remove most significant bits)

Dangerous (doesn't preserve all values)

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

// x = -1
// y =  ?
```

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

// x = -1
// y = 0xffffffff = 4294967295
```

⚠ value has changed

x is negative, but y is positive!

```
// draw_pixel(-1, -1, color);
// !!
```

uint32 ← int32

uint16 ← int16

uint8 ← int8

Defined (copies bits)

uint32 ← int32

uint16 ← int16

uint8 ← int8

Dangerous! (neg maps to pos)

uint32 → int32

uint16 → int16

uint8 → int8

Technically Not Defined
(arm: copies bits)

uint32 → int32

uint16 → int16

uint8 → int8

Dangerous!

(large positive numbers change)

**"Whenever you mix
signed and unsigned numbers
you get in trouble."**

Bjarne Stroustrup

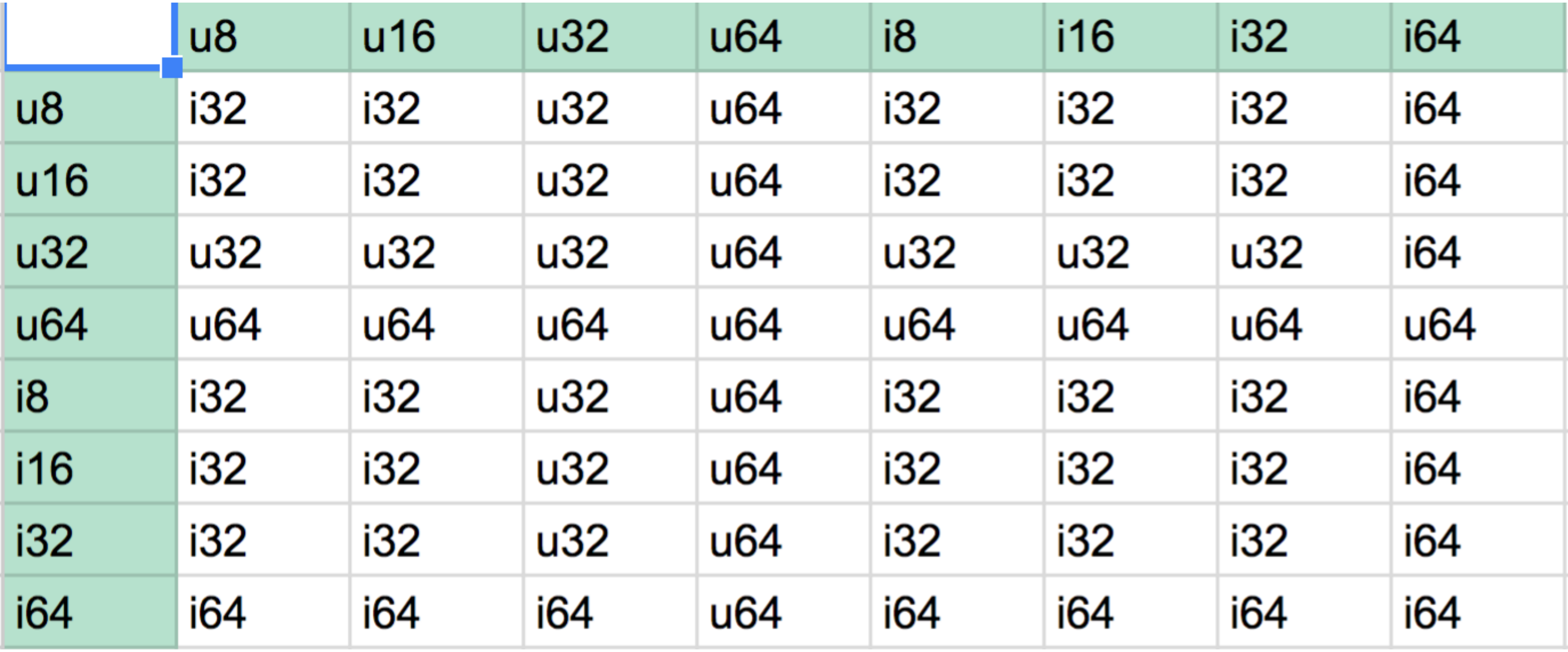
Implicit Type Promotion

in

Binary Operators

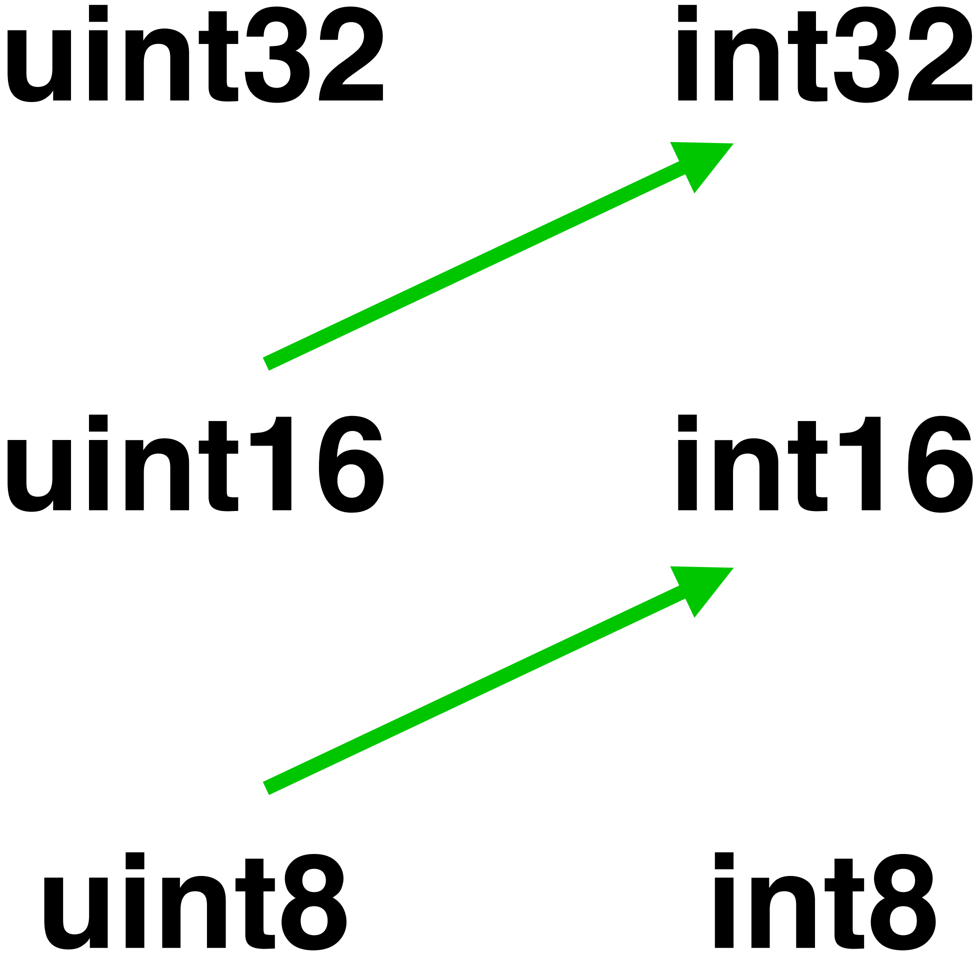
Type promotions for binary operations

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

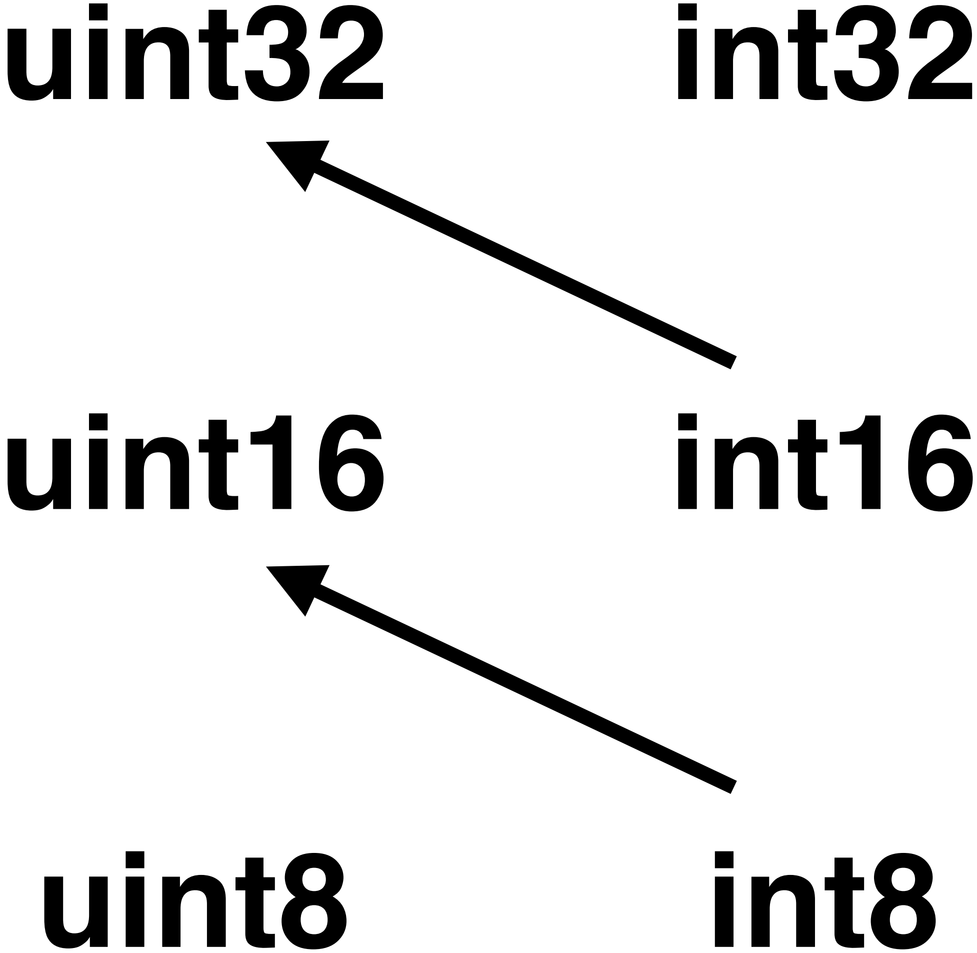


uint32 int32

uint16 int16

uint8 int8

Safe?



uint32 int32

uint16 int16

uint8 int8

Safe?

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");
}
```

Be Wary of Implicit Type Conversion

Modern languages like rust and go do not perform implicit type conversion

Summary

Signed numbers are represented in two's complement

- Negation: -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 and promotions!!

Speed and Optimization