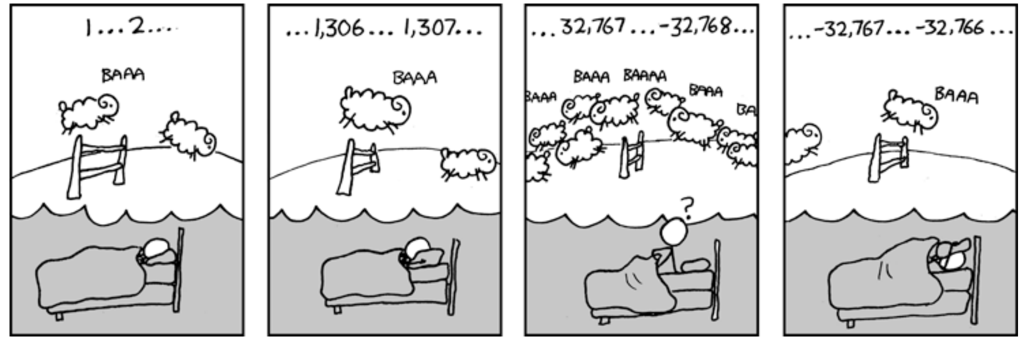
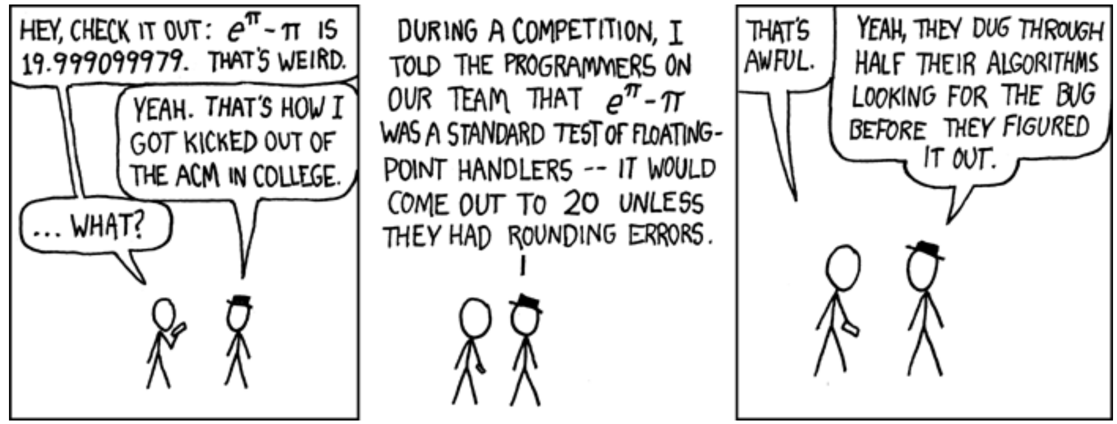
Admin

Retest cycle

Issues A2, A3



https://xkcd.com/571/



https://xkcd.com/217/

Today: Numbers & Arithmetic

Integer representation

`signed` vs `unsigned`

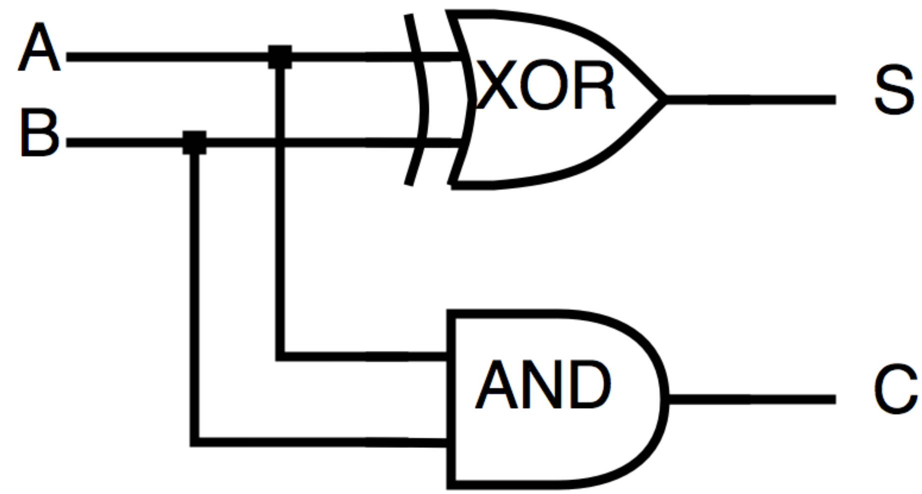
Type conversion

Floating point

Add two 1-bit nums (half adder)

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

lsb of sum `S = a^b`
msb of sum `C = a&b`



Create addition out of logical ops!

Add two 8-bit numbers

```
  0 0 0 0 1 1 1     Carry
  00000111
+ 00001011
  --------
  00010010     Sum
```

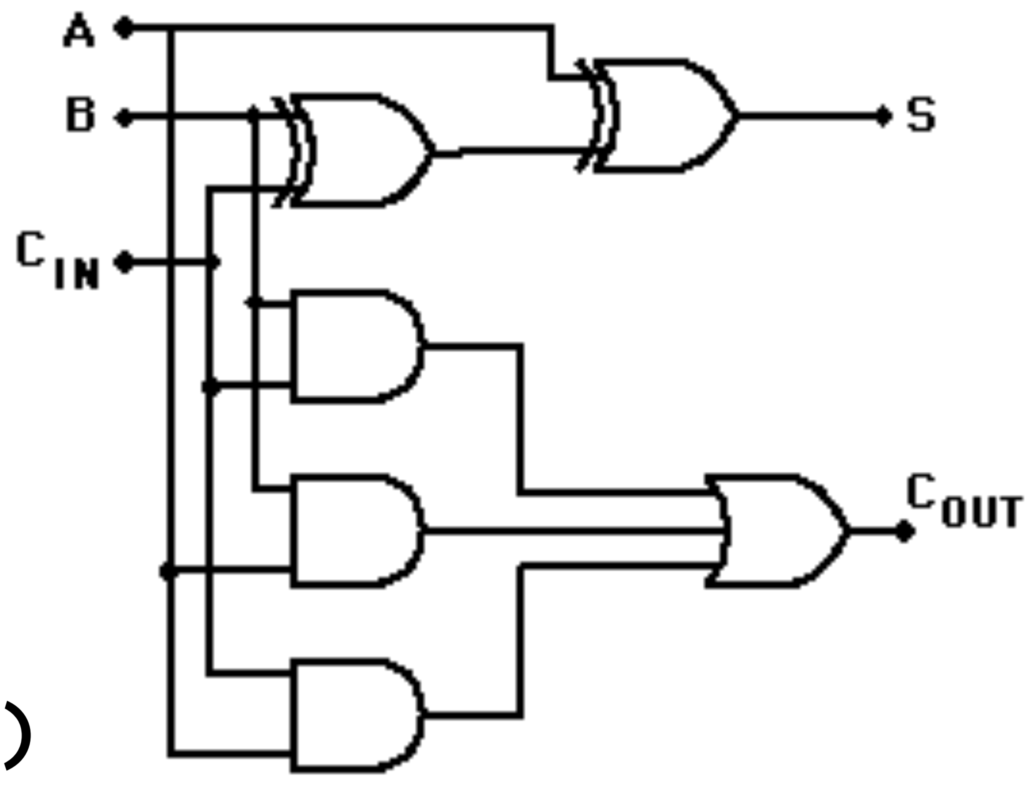
Add three 1-bit nums (full adder)

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

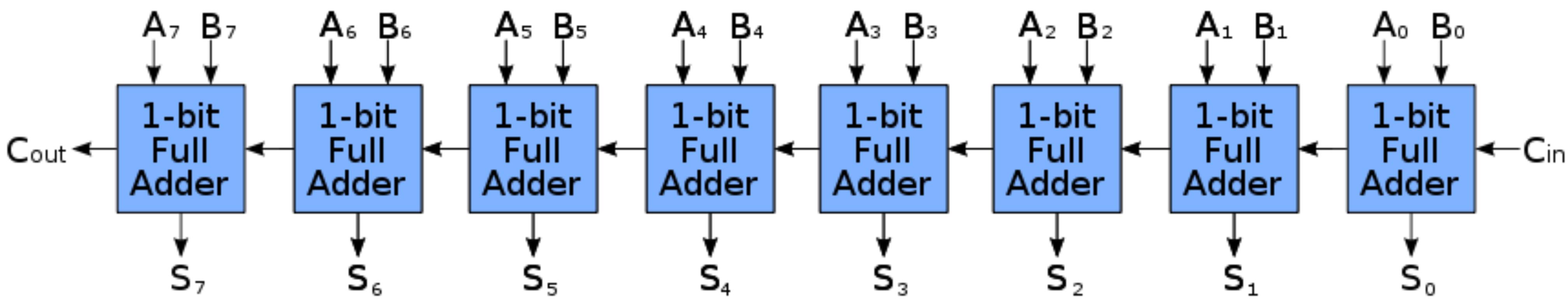
Full adder (carry in, carry out)

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 Rippler Adder



`Cin` (carry in) **`Cout`** (carry out)

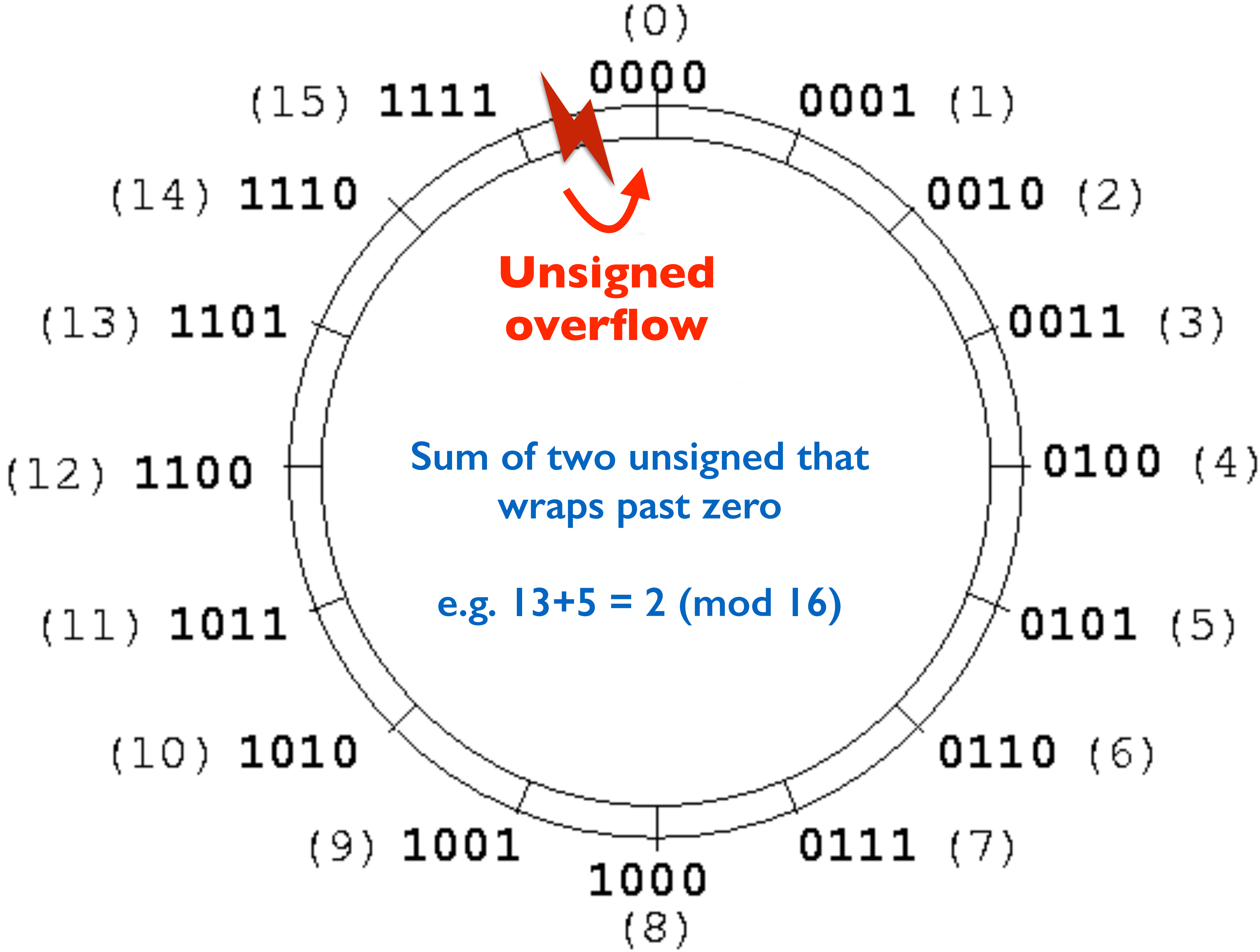
Modular arithmetic

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

Represent sum of two n-bit numbers at full precision requires n+1 bits

Store into n bits discards final carry out (overflow)

```
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
Unsigned overflow
Sum of two unsigned that wraps past zero
e.g. 13+5 = 2 (mod 16)

Unsigned overflow

```
unsigned long timer_get_ticks(void);

void timer_delay_us(unsigned long us)  {
  unsigned long elapsed = us*TICKS_PER_USEC;
  unsigned long start = timer_get_ticks();

  while (timer_get_ticks() - start < elapsed) {}
}
```

Tick count continuously increments. Above code works even when tick count overflows/wraps around — trace why.

But, does **not** work if expression is rearranged:

```
     while (timer_get_ticks() < start + elapsed) {}
```

Trace what happens instead

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

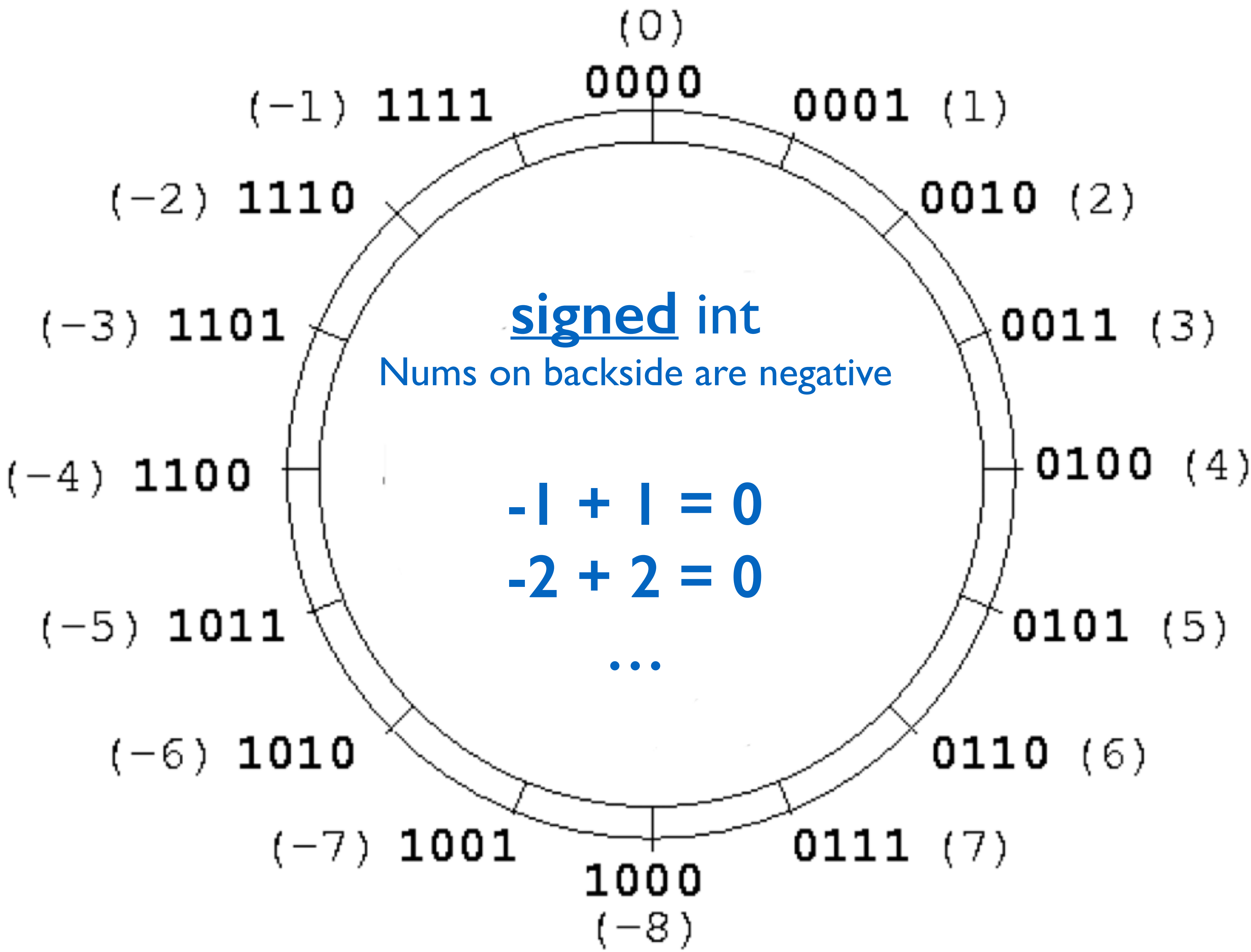
BIG IDEA: Define subtraction using addition

A clever way to define subtraction by N is to find a number that when added yields same result as subtract by N. This number is *negative* N.

Define negative N s.t **`N + negative N = 0`** (mod)

`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 int
Nums on backside are negative
-1 + 1 = 0
-2 + 2 = 0
...

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 % 256`

Thus the term *two's complement*

Another way to negate

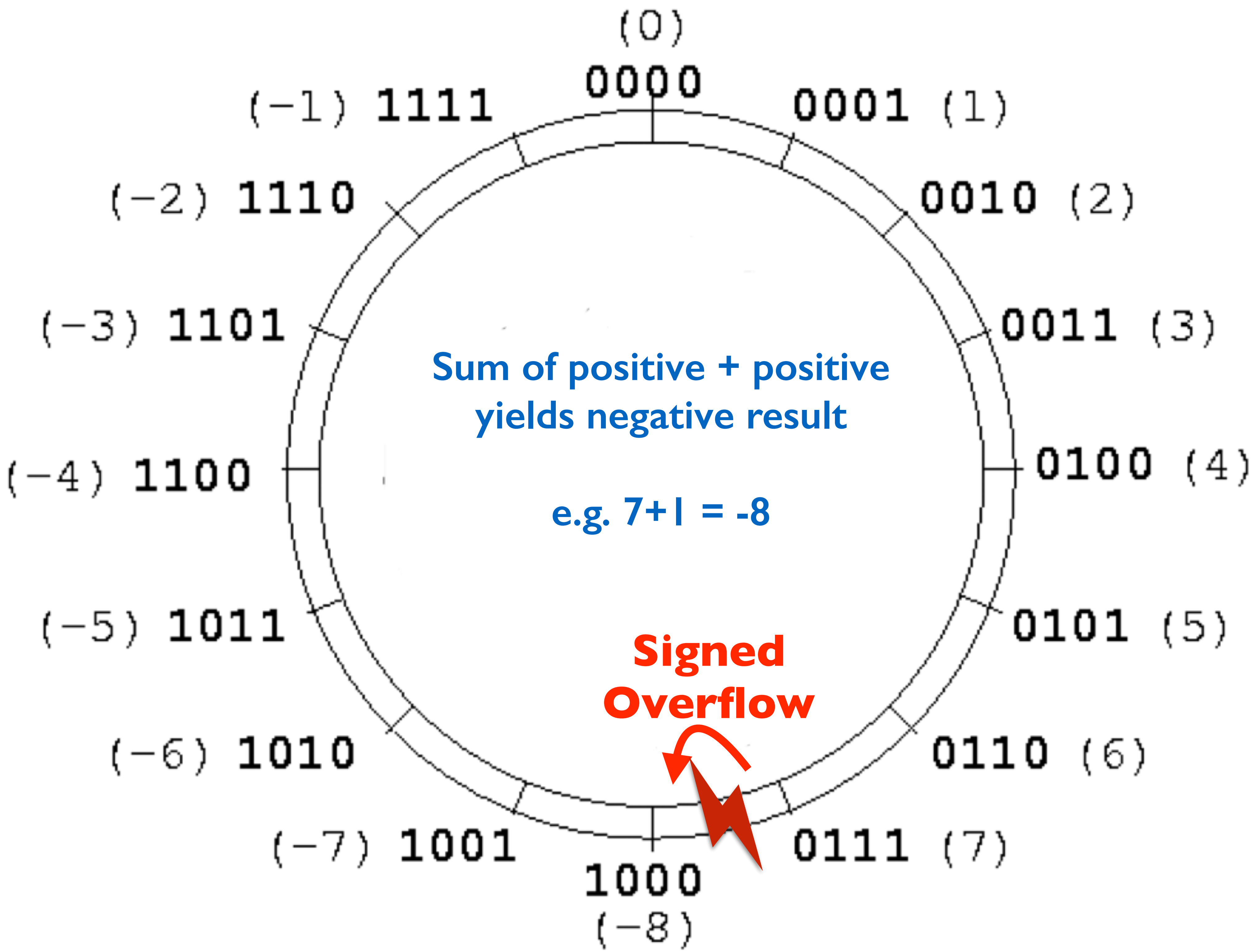
```
Rewrite 100000000 = (11111111 + 1)

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

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

Example: -5 = ~5 + 1
    ~5                             -5
~00000101 + 1 = 11111010 + 1 = 11111011

     5          -5          0
 00000101 + 11111011 = (1)00000000
```

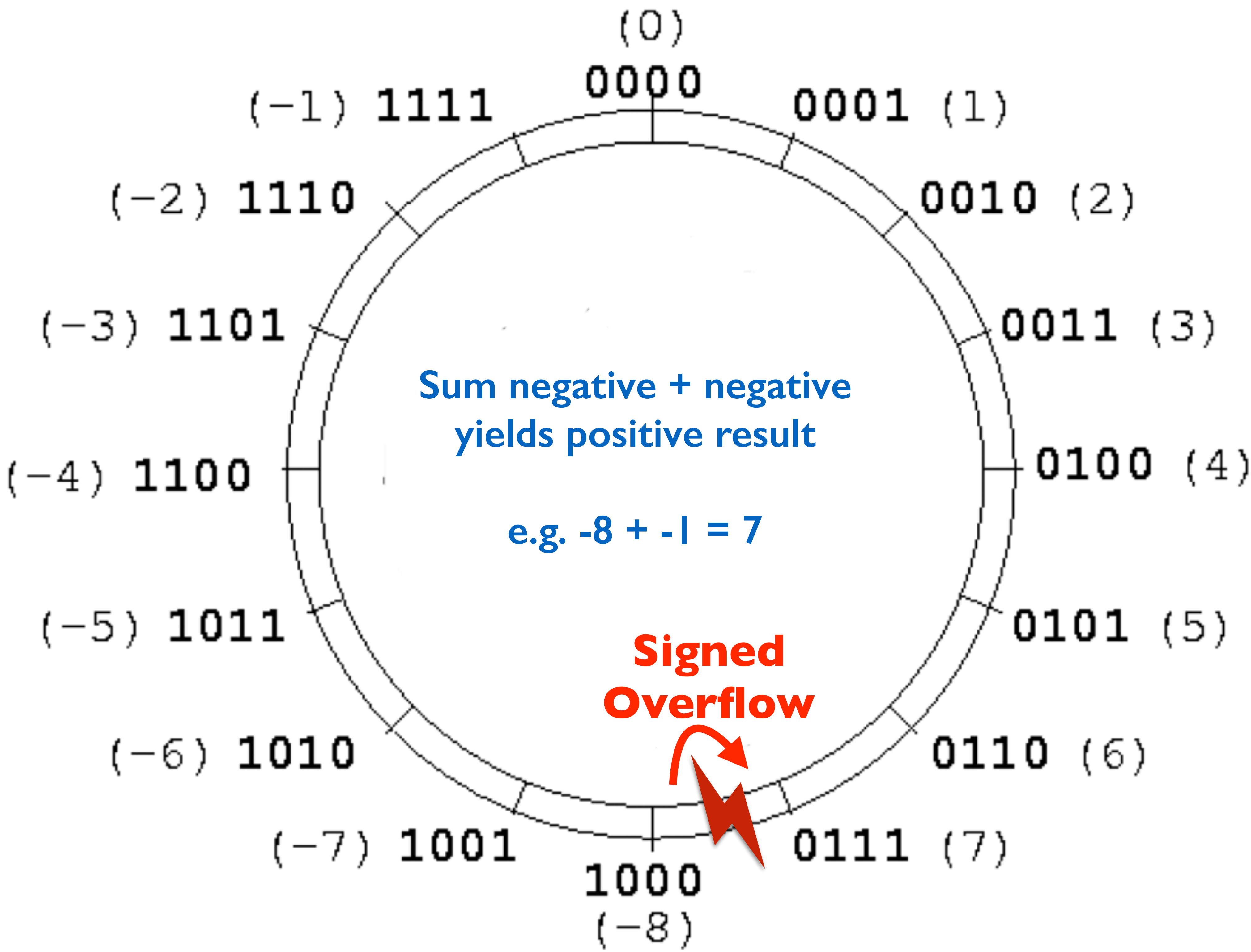
Add and subtract

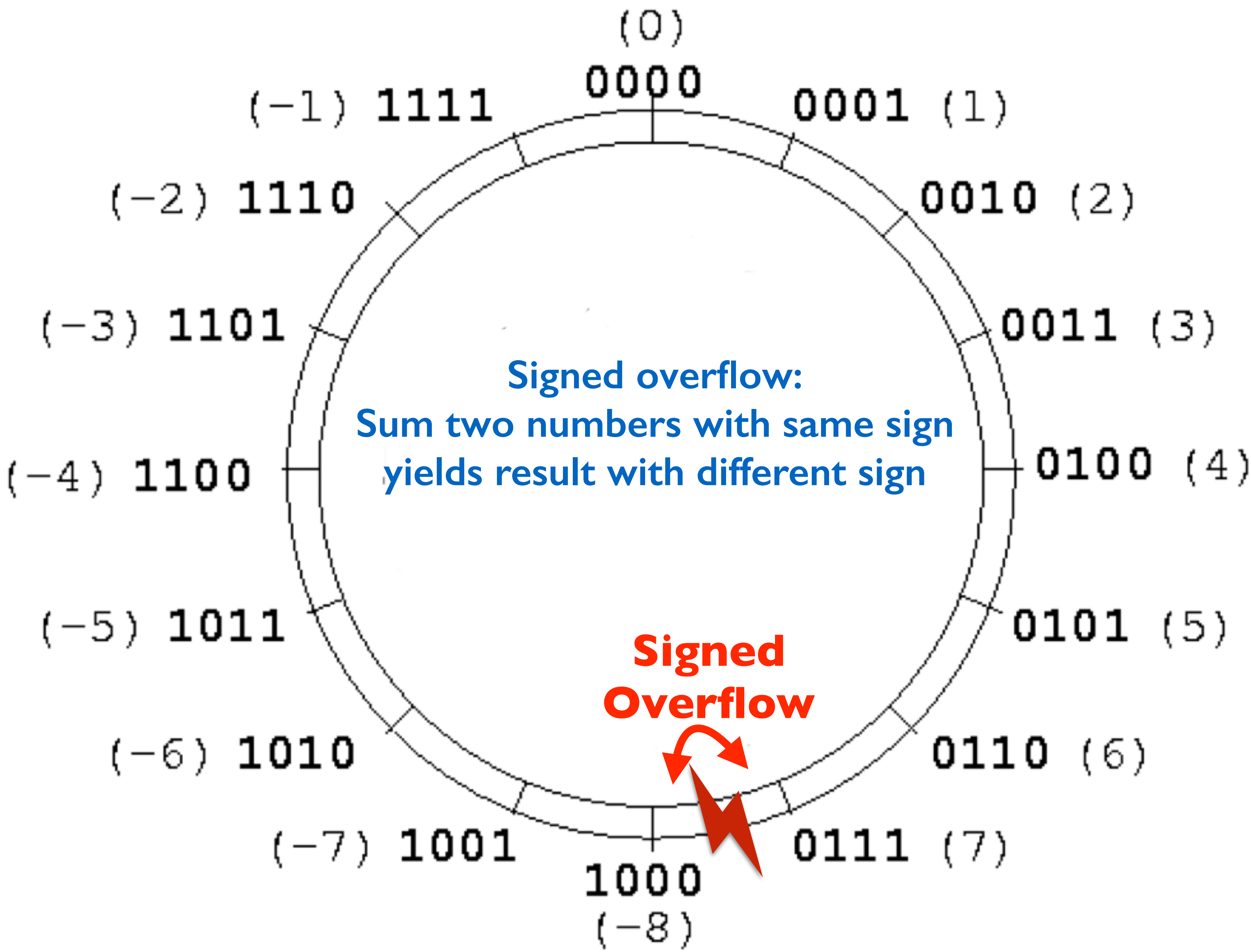
`signed` and `unsigned` numbers

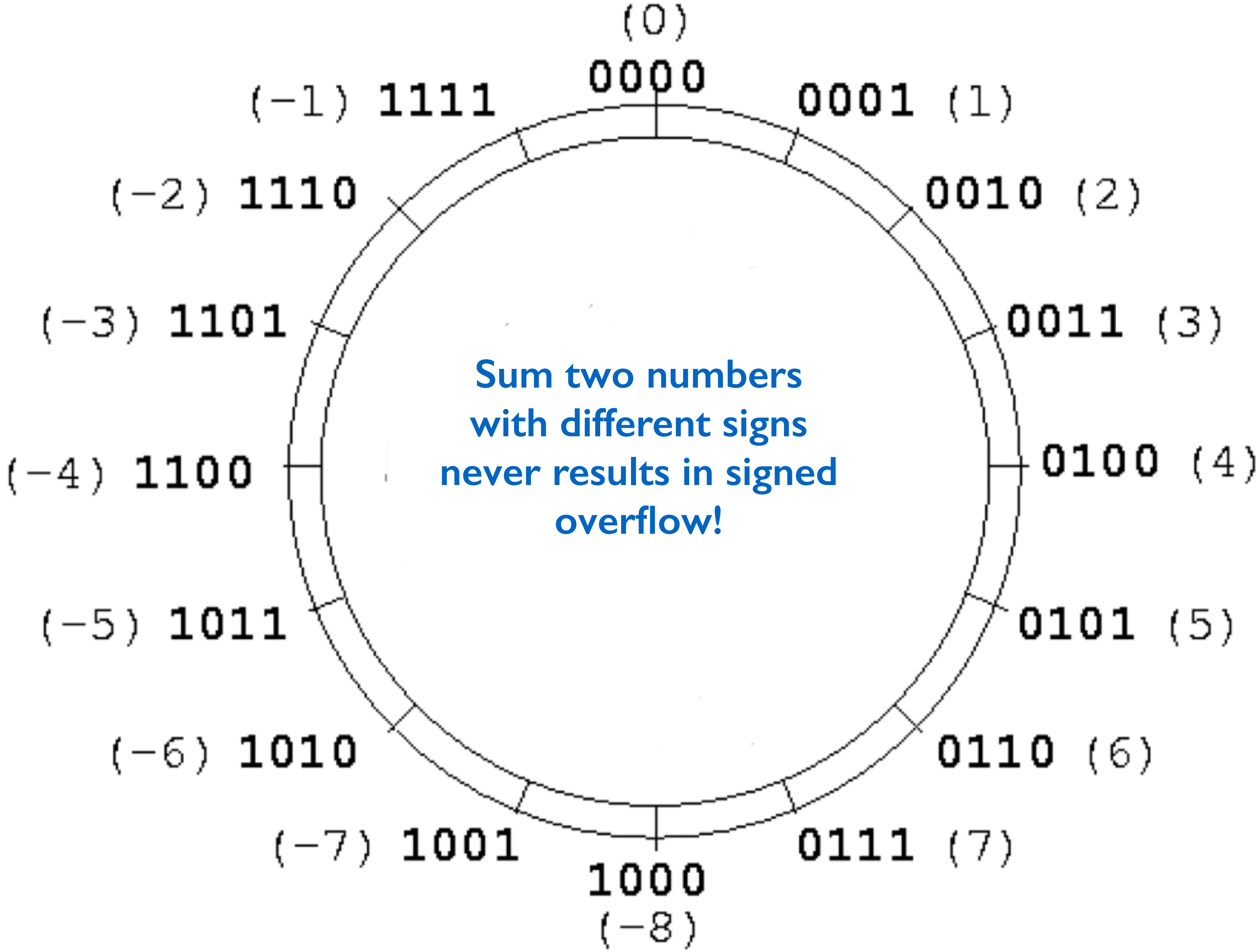
use exact same adder

Neat!


(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
Sum of positive + positive yields negative result
e.g. 7+1 = -8
Signed Overflow

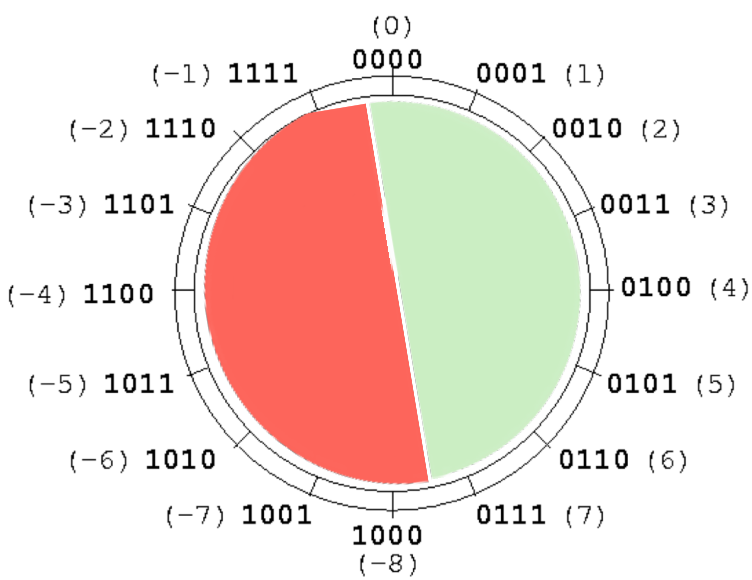

(0) 0000
0001 (1)
0010 (2)
0011 (3)
0100 (4)
0101 (5)
0110 (6)
0111 (7)
1000 (-8)
(-7) 1001
(-6) 1010
(-5) 1011
(-4) 1100
(-3) 1101
(-2) 1110
(-1) 1111
Sum negative + negative yields positive result
e.g. -8 + -1 = 7
Signed Overflow


(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 overflow:
Sum two numbers with same sign
yields result with different sign
Signed
Overflow


(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
Sum two numbers with different signs never results in signed overflow!

signed vs unsigned

different interpretations of exact same bits!



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

Signed/unsigned in C

Constants

Default to `int` type (signed)

`U` suffix means unsigned: `0U, 4294967259U`

Explicit cast

Bits unchanged, no conversion, change in interpretation

```
int tx, ty;
unsigned int ux, uy;
tx = (int) ux;
uy = (unsigned int) ty;
```

Implicit cast

Assignment, function call, co-mingle types

```
tx = ux;
uy = ty;
```

Type Conversions

Promotion/Widening

Result is richer type (e.g. larger bit width)

Value preserved

Demotion/Narrowing

Result is lesser type (e.g. smaller bit width)

Possible loss/truncation of value

Conversion/Coercion

Convert from one type to another

Unsigned type hierarchy

`uint32_t` {0,…,0xffffffff(4294967295)}

`uint16_t` {0,…,0xffff(65535)}

`uint8_t` {0,…,0xff(255)}

Type *Promotion*
Safe: all values preserved

Type demotion/narrowing
Defined: remove most significant bits
Unsafe: truncates some values

Signed type hierarchy

`int32_t` {-2147483648 … 2147483647}

`int16_t` {-32768 … 32767}

`int8_t` {-128 … 127}

Type *Promotion*
Safe: all values preserved
(sign-extension)

Type demotion/truncation
Defined: remove most significant bits
Dangerous: not preserve all values/sign

Co-mingling signed/unsigned

`uint32` ⇄ `int32`

`uint16` ⇄ `int16`

`uint8` ⇄ `int8`

Defined: copy bits

Unsafe: reinterpret large positive/negative

⚠ What happens?

```
uint16_t before = 0xffff;
uint32_t after = before;
// after = ?

uint32_t before = 0x12340001;
uint16_t after = before;
// after = ?

int16_t before = -1; //  negative -> sign extension
int32_t after = before;
// after = ?

int32_t before = -50000; // 0xffff3cb0
int16_t after = before;
// after = ?

int32_t before = -1;
uint32_t after = before; // signed -> unsigned, ack!
// after = ?
```

Type table for binary ops

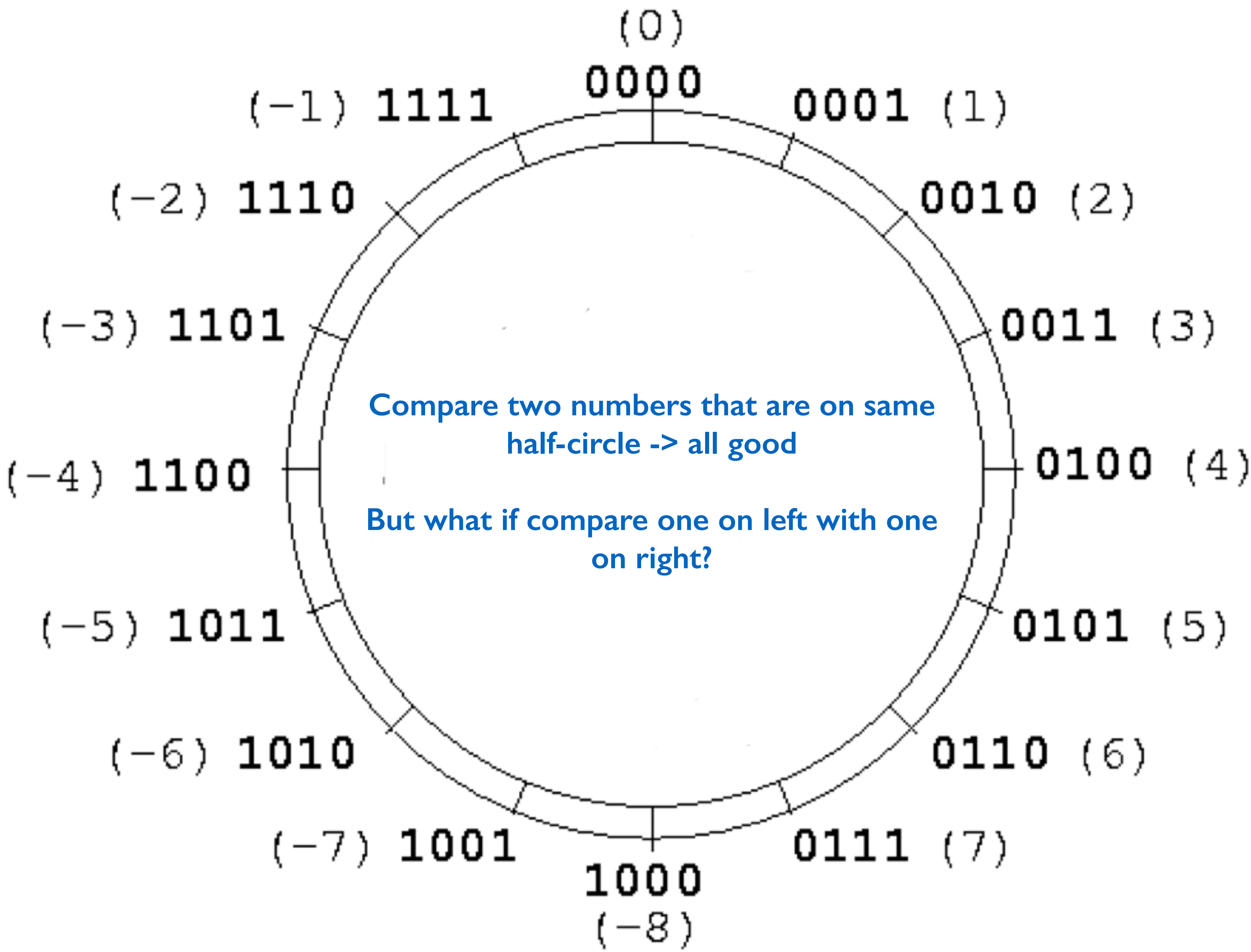
Type of result can be different than operand types!

	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

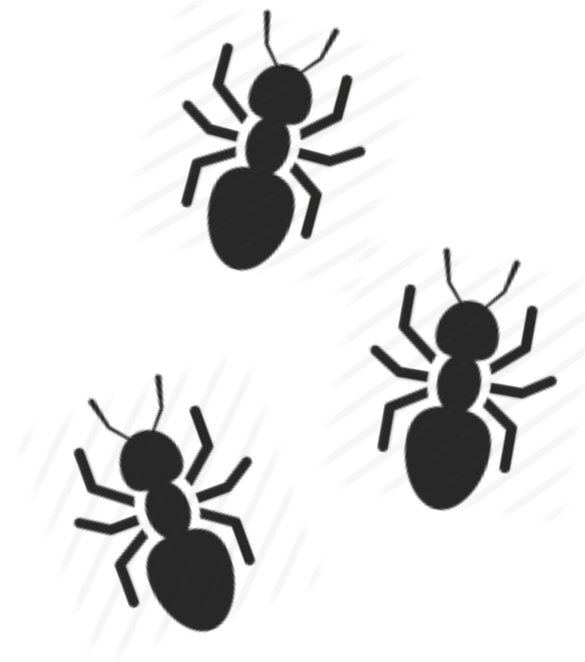
`riscv64-unknown-elf-gcc` type promotions

Operations to ***compare***
`signed` and **`unsigned`** numbers
are NOT the same!

"Whenever you mix
signed and unsigned numbers
you get in trouble."
— Bjarne Stroustrup


(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
Compare two numbers that are on same half-circle -> all good
But what if compare one on left with one on right?

```
void 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");
}
```



Takeways integer type

Signed numbers are represented in two's complement

Negation: `-x = (2`^{N}` - x) = (~x + 1)`

Arithmetic **same** signed and unsigned

Comparison **not same** signed and unsigned!

Know rules for type conversion, watch out for implicit type conversions and promotions!

Floating point

Numbers represented in scientific notation

Use cases:

Real numbers: `2.5` `3.14159`

Tiny/huge numbers: $0.5*10^{-8}$ $3.5*10^{19}$

Pre-floating point world

Early computers built for scientific calculations, but no floating point data types nor special hardware

Apply large scale factor to all fp values, multiply inputs before start, operate on as int, divide outputs by scale factor at end, ugh!

Fixed width dilemma

Infinite set of integers, infinitely more reals

Hardware register/memory has finite width N bits

- 2^N representable values

For integers, range to choose is "obvious", centered around 0, next neighbor at `val +/- 1`

For reals, which values should be in representable set?

- 2^N values from 0 to 1
- 2^N powers of 2
- Something else, but what?

Range and precision

Range

Distance between smallest and largest representable value

Precision

Distance between two consecutive representable values

Desire large range and small precision

Fixed bit width means have to compromise

Scientific notation to the rescue

Divvy up bits among sign, significand, and exponent

Change in exponent "floats" point

Sliding window of precision

Relative precision is same across entire range, absolute precision runs from teeny-tiny to quite large

Small numbers very precise, close together

Big numbers have larger and larger gaps between neighbors, skips whole integers

Most values approximate, not exact

IEEE 754

Major standards success story

Established in 1985 as uniform standard for floating point arithmetic

Main idea: make numerically sensitive programs portable

Specifies two things: representation and result of floating operations

Supported by most all hardware since 80s, portability, collaborative improvement, Kahan Turing Award 1989

Focus on numerical concerns, not performance

Numerical analysts drove standard, not hardware designers

Precise standards for rounding, overflow, underflow

But... hard to make fast in hardware

Float operations often several times slower than integer

https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html

IEEE float & double

Single precision `float`

32-bit: sign bit, 8-bit exponent, 23-bit significand

Range: $2 * 10^{-38}$ to $2 * 10^{38}$

Precision: ~7 significant (decimal) digits

Double precision `double`

64-bit: sign bit, 11-bit exponent, 52-bit significand

Range: $2 * 10^{-308}$ to $2 * 10^{308}$

Precision: ~15 significant digits

How to represent float as bits?

Sign: 1 bit, easy

Exponent: 8-bit signed integer, cool

Significant: 23-bits of … what?

Bit representation is sum of **negative** powers of two

```
.0101 = 0*1/2 + 1*1/4 + 0*1/8 + 1*1/16
```

(Pardon my shameless ghosting of details of normalization, exponent bias, subnormal, exceptional, rounding)

Floating point arithmetic

Addition, subtraction

First align points, then add/subtract fractions, round/normalize

Multiplication

Multiply fractions, add exponents, round/normalize

Division

True division on fraction costly/slow

Instead Newton's method to find reciprocal and use existing multiply

Rounding/exceptions

IEEE strong specification of expected result (including rounding mode), also handling of underflow, overflow, NaN

Type conversions to/from fp

Cast/convert between `int` and `float`/`double` **changes bits**.

`int → float`

May be rounded; (32-bit int into 23-bit frac)

`int → double` or `float → double`

Exact conversion (32-bit int into 52-bit frac)

`double or float → int`

Truncates fraction (round toward zero)

`1.999 -> 1, -1.99 -> -1`

Out of range, NaN is "undefined": typical result Tmin (largest negative value, ugh)

Mango Pi float support

C language spec no dictate fp support IEEE 754 compliant (but likely true in practice)

If no hardware support or unavail, floating point operations emulated in software (libgcc) "soft-float"

RISC-V ISA extensions f and d https://www.sifive.com/blog/all-aboard-part-1-compiler-args

`march=rv64imfd  mabi=lp64d`

Compiles for use of single-precision (f) and double-precision (d) registers and fp instructions, pass fp parameters in fp registers

At runtime, must turn on float unit by setting bit in `mstatus` CSR

Hardware fp is order of magnitude faster than soft

Takeaways of float type

Float representation compromise of fixed bitwidths

Floats suffer from overflow/underflow, just like ints

Many "simple fractions" have no exact representation (e.g., 0.2)

Can also lose precision, unlike ints

Every operation gets a "slightly wrong" result

Mathematically equivalent ways of computing same expression may get different results

Violates associativity/distributivity

Never test floating point values for equality!

- Instead subtract and test if delta close enough to "epsilon"

Careful when converting between ints and fp

"95% of the folks out there are completely clueless about floating-point."

James Gosling
(Sun Fellow, inventor of Java)