Integer Representation

Representation of integers: unsigned and signed

Modular arithmetic and overflow

Sign extension

Shifting and arithmetic

Multiplication

Casting

Fixed-width integer encodings

Unsigned $\subset \mathbb{N}$ **non-negative integers** only

Signed $\subset \mathbb{Z}$ both **negative** and **non-negative integers**

***n* bits** offer only **2^n distinct values.**

Terminology:

"Most-significant" bit(s)
or "high-order" bit(s)

"Least-significant" bit(s)
or "low-order" bit(s)

MSB 0110010110101001 **LSB**

Unsigned integer representation

Example in 4-bit unsigned representation.

1	0	1	1	
8	4	2	1	*weight*
2^3	2^2	2^1	2^0	*weight*
3	2	1	0	*position*

$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

n-bit unsigned numbers:

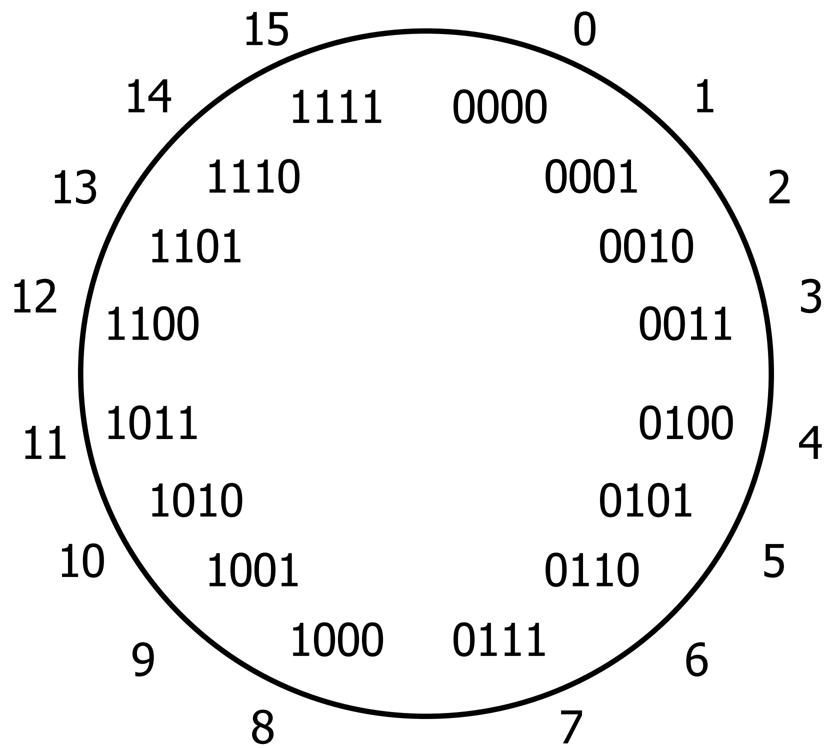
minimum =

maximum =

Unsigned **modular arithmetic, overflow**

Examples in 4-bit unsigned representation.

11 + 2 =



13 + 5 =

`x + y` in N-bit unsigned arithmetic is ***(x + y) mod 2^N*** in math

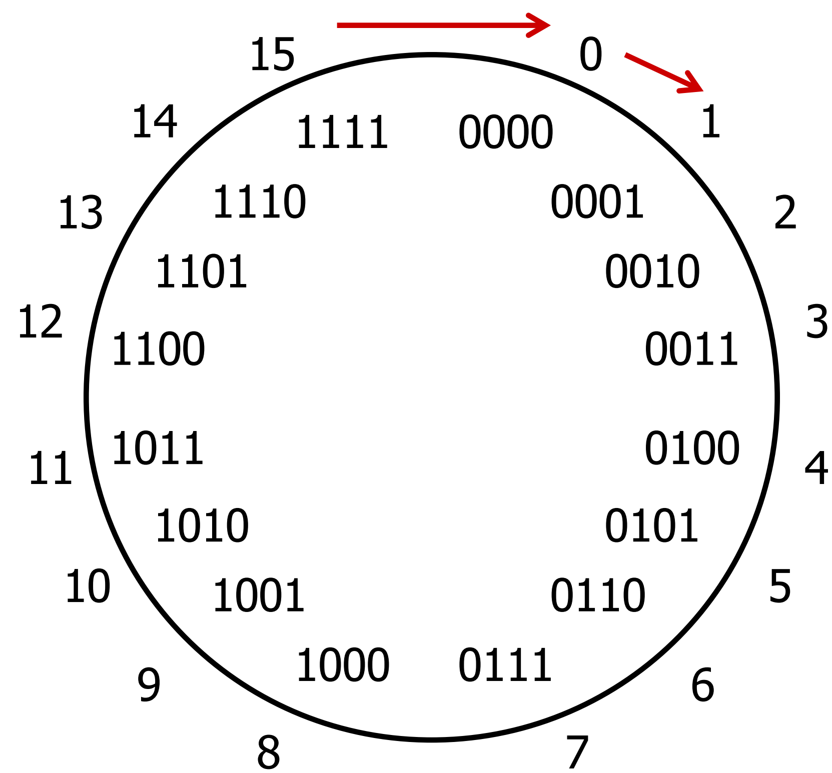
unsigned overflow = *"wrong" answer = wrap-around*

= carry 1 out of MSB = math answer too big to fit

Unsigned **overflow**

Addition *overflows* if and only if **a carry bit is dropped.**

$$\begin{array}{r} 15 \\ +\ 2 \\ \hline \end{array} \qquad \begin{array}{r} 1111 \\ +\ 0010 \\ \hline \end{array}$$



Modular Arithmetic

Sign-Magnitude representation?

!!!

Most-significant bit (MSB) is *sign bit*

0 means non-negative 1 means negative

Remaining bits are an unsigned magnitude

8-bit sign-and-magnitude:

ex

0x00 = 00000000 represents _____________

0x7F = 01111111 represents _____________

0x85 = 10000101 represents _____________

0x80 = 10000000 represents _____________

Max and min for n bits? **Anything weird here?**

Sign-Magnitude Negatives

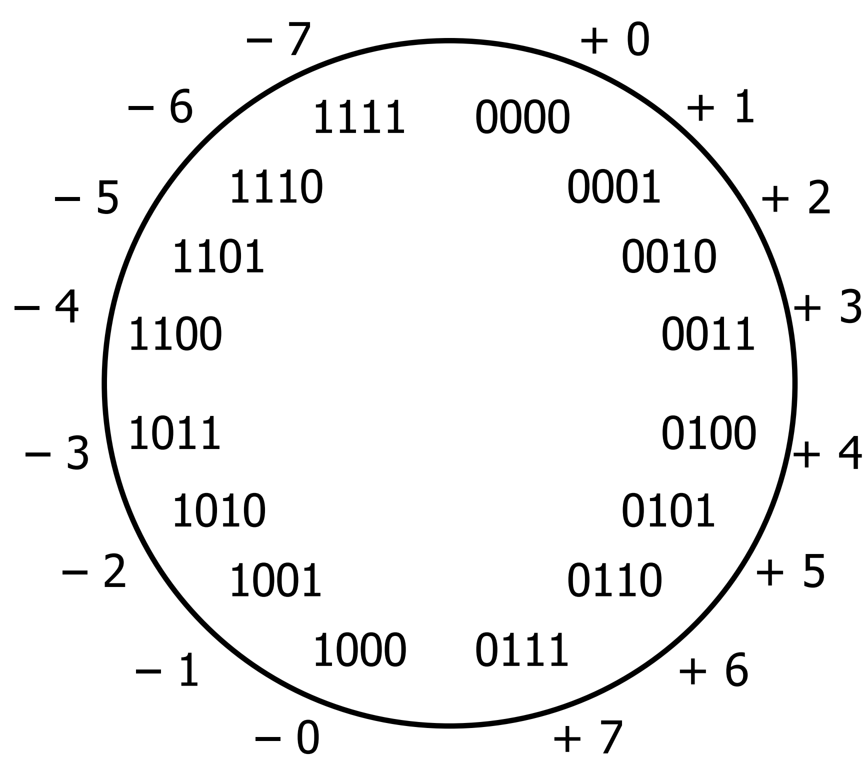
Cumbersome arithmetic.

Example:

4 - 3 != 4 + (-3)

$$\begin{array}{r} 0100 \\ +\underline{1011} \end{array}$$

What about zero?



Sign-magnitude is not such a good idea...

Two's complement representation

for signed integers

n-bit representation

_	…	_	…	_	_	_	_
$-2^{(n-1)}$	…	2^i	…	2^3	2^2	2^1	2^0
n-1	…	i	…	3	2	1	0

weight

position

Positional representation, ***but***

most significant position has *negative weight*.

8-bit representations

ex

0 0 0 0 1 0 0 1

1 0 0 0 0 0 0 1

1 1 1 1 1 1 1 1

0 0 1 0 0 1 1 1

n-bit two's complement numbers:

minimum =

maximum =

4-bit unsigned vs. 4-bit two's complement

1 0 1 1

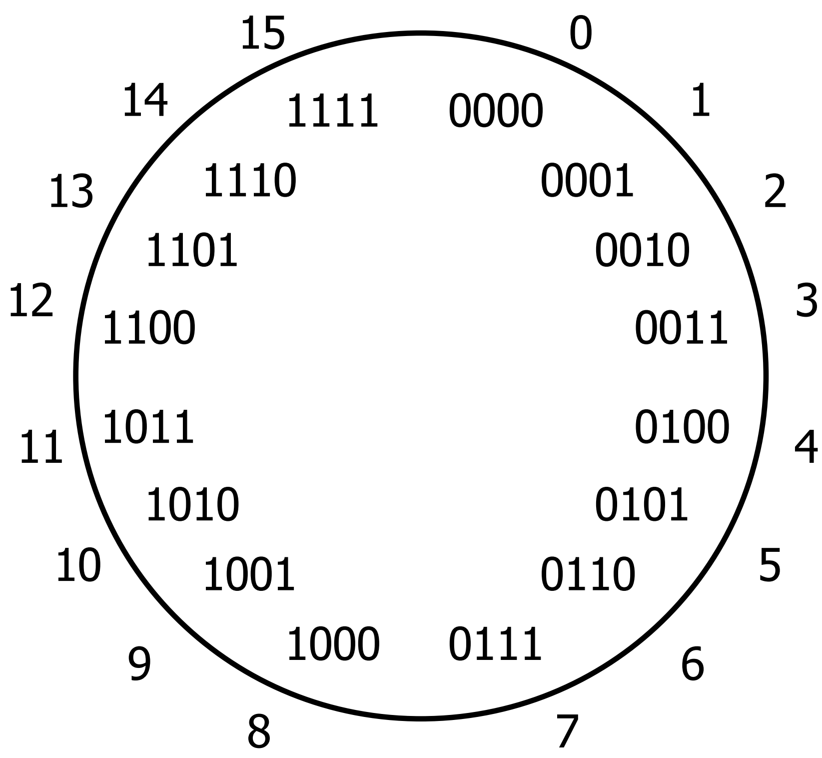
$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

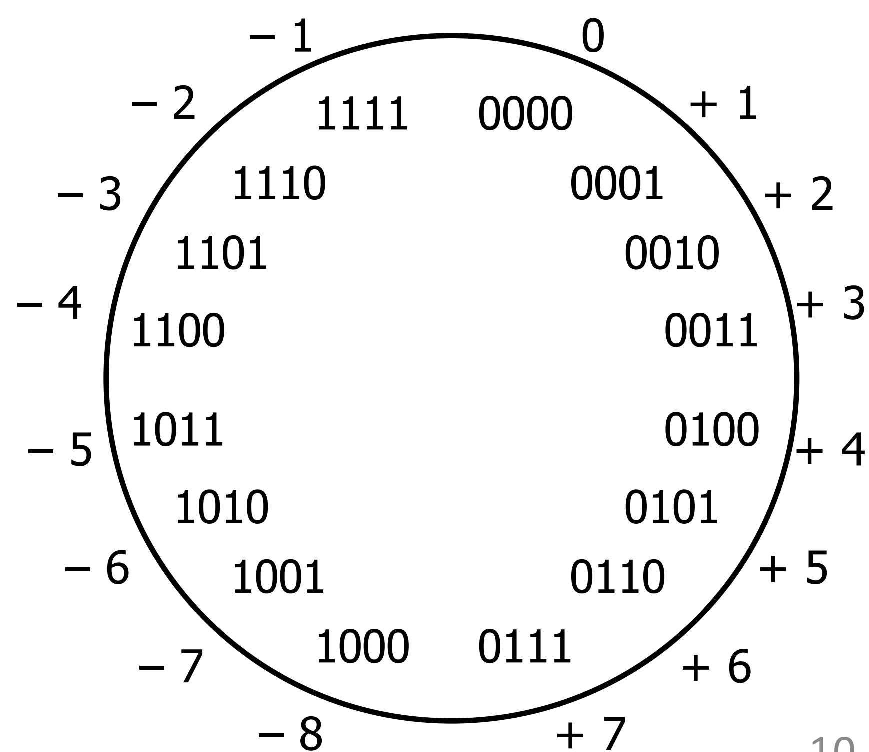
$1 \times -2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

11

difference = ___ = 2—

-5

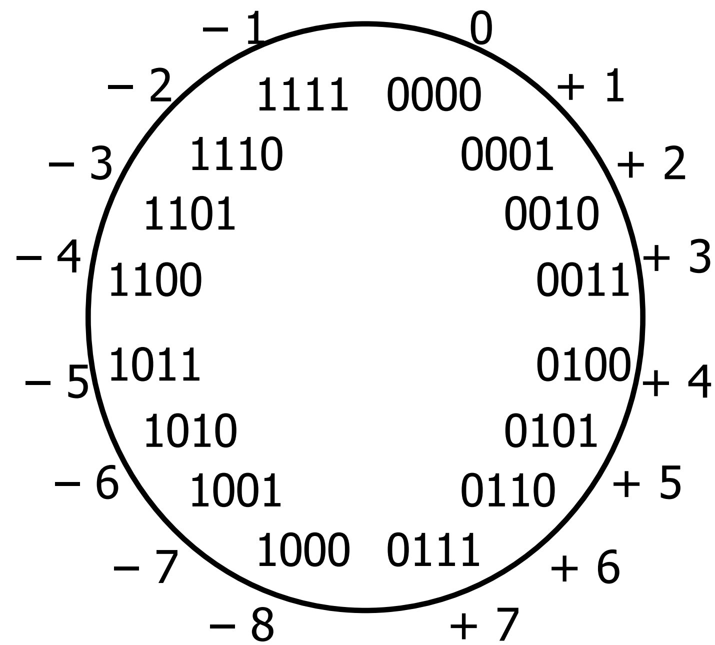




Two’s complement **addition *Just Works***

2	0010	-2	1110
+ 3	+ 0011	+ -3	+ 1101

-2	1110	2	0010
+ 3	+ 0011	+ -3	+ 1101



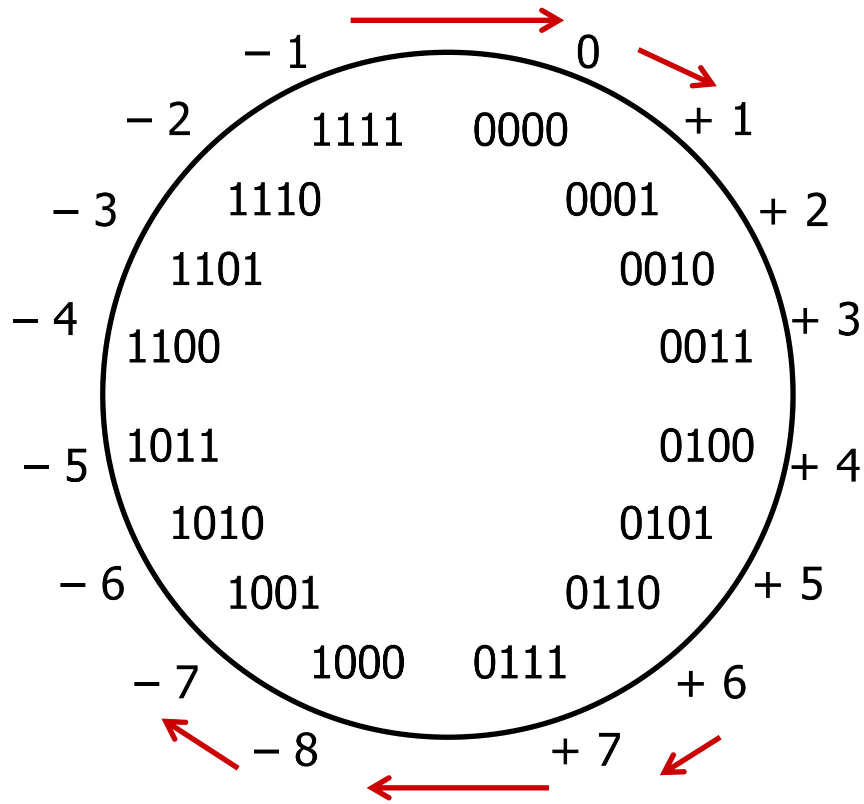
Modular Arithmetic

Two’s complement ***overflow***

Addition ***overflows***

if and only if the **arguments have the same sign** but the **result does not.**
if and only if the **carry in and out** of the **sign bit differ.**

-1	1111
+ 2	+ 0010
6	0110
+ 3	+ 0011



Modular Arithmetic

Some CPUs/languages raise exceptions on overflow.
C and Java cruise along silently... Oops?

Reliability

Ariane 5 Rocket, 1996

Exploded due to **cast** of 64-bit floating-point number to 16-bit signed number. **Overflow.**

Boeing 787, 2015

"... a **Model 787 airplane** … can lose all alternating current (AC) electrical power … caused by a **software counter** internal to the GCUs that will **overflow** after **248 days** of continuous power. We are issuing this AD to prevent loss of all AC electrical power, which could result in **loss of control of the airplane**."
--FAA, April 2015

A few reasons two's complement is awesome

Same exact addition algorithm as for unsigned numbers.

Easy: `x + -x == 0`

Subtraction is just addition: `4 - 3 == 4 + (-3)`

Simple hardware!

MSB is sign: negatives start with 1, non-negatives start with 0

Negative one is `111…11`.

Complement rules:

```
x + ~x == -1
~x + 1 == -x
```

Another derivation

ex

How should we represent 8-bit negatives?

- For all positive integers ***x***,
  the representations of ***x*** and ***–x*** must sum to zero.
- Use the standard addition algorithm.

```
  00000001        00000010        00000011
+               +               +
  --------        --------        --------
  00000000        00000000        00000000
```

- Find a rule to represent –x where that works…

Convert small *two's complement representation to a* ***larger*** *representation?*

	0 0 0 0 0 0 1 0	8-bit 2
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	16-bit 2
How should these bits be filled?	1 1 1 1 1 1 0 0	8-bit -4
	? ? ? ? ? ? ? ? 1 1 1 1 1 1 0 0	16-bit -4

ex

Sign extension

Fill new bits with copies of the sign bit.

0 0 0 0 0 0 1 0	8-bit 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	16-bit 2
1 1 1 1 1 1 0 0	8-bit -4
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0	16-bit -4

Casting from smaller to larger signed type does sign extension.

How are **shifting** and **arithmetic** related? ex

unsigned

x = 27;

y = x << 2;

y == 108

0 0 0 1 1 0 1 1

0 0 0 1 1 0 1 1 0 0

logical shift left:

shift in zeros from right

logical shift right:

shift in zeros from left

1 1 1 0 1 1 0 1

0 0 1 1 1 0 1 1 0 1

unsigned

x = 237;

y = x >> 2;

y == 59

How are **shifting** and **arithmetic** related? ex

signed

x = -101;

y = x << 2;

y == 108

1 0 0 1 1 0 1 1

1 0 0 1 1 0 1 1 0 0

logical shift left:

shift in zeros from the right

signed

x = -19;

y = x >> 2;

y == -5

1 1 1 0 1 1 0 1

1 1 1 1 1 0 1 1 0 1

arithmetic shift right:

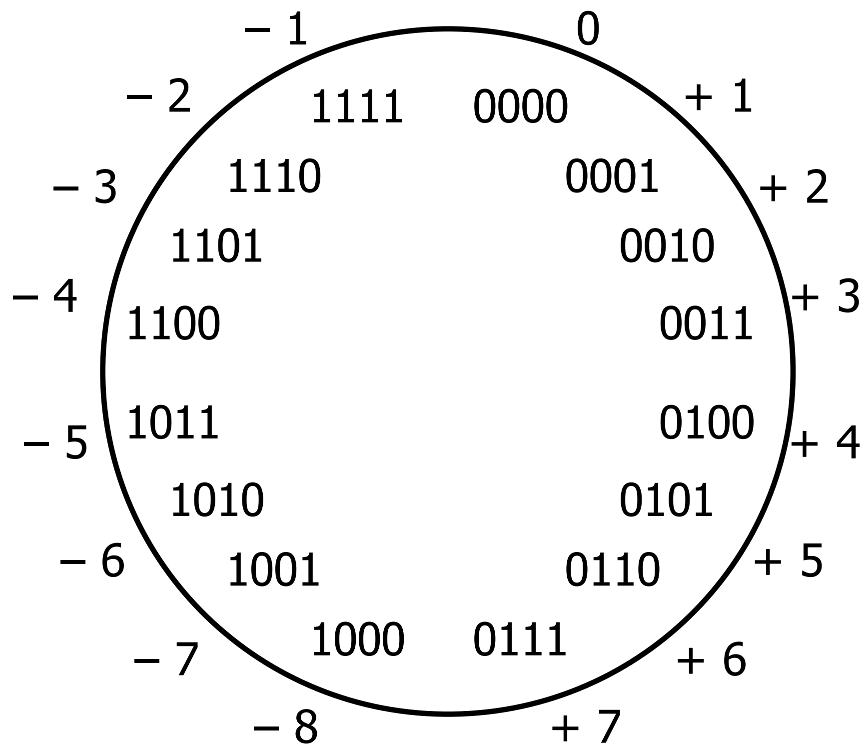
shift in copies of MSB from left

Multiplication

Compute answer in **2x bits**. Most languages drop high order half.

2	0010
x 3	x 0011
6	00000100

-2	1110
x 2	x 0010
-4	11111100



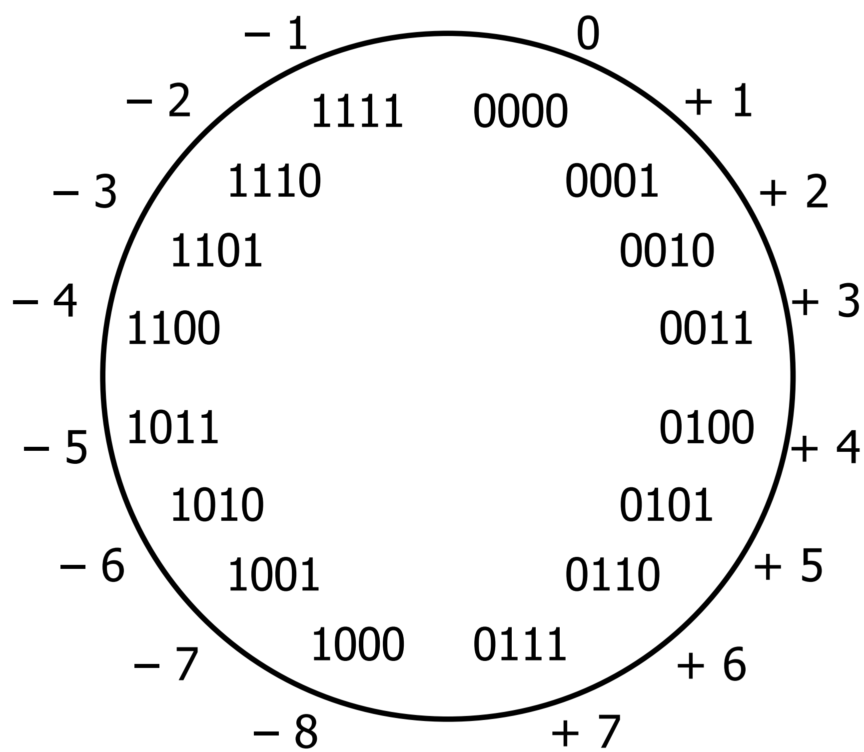
Modular Arithmetic

Multiplication

Compute answer in **2x bits**. Most languages drop high order half.

5	0101
x 4	x 0100
~~20~~	00010100
4	

-3	1101
x 7	x 0111
~~-21~~	11101011
-2	

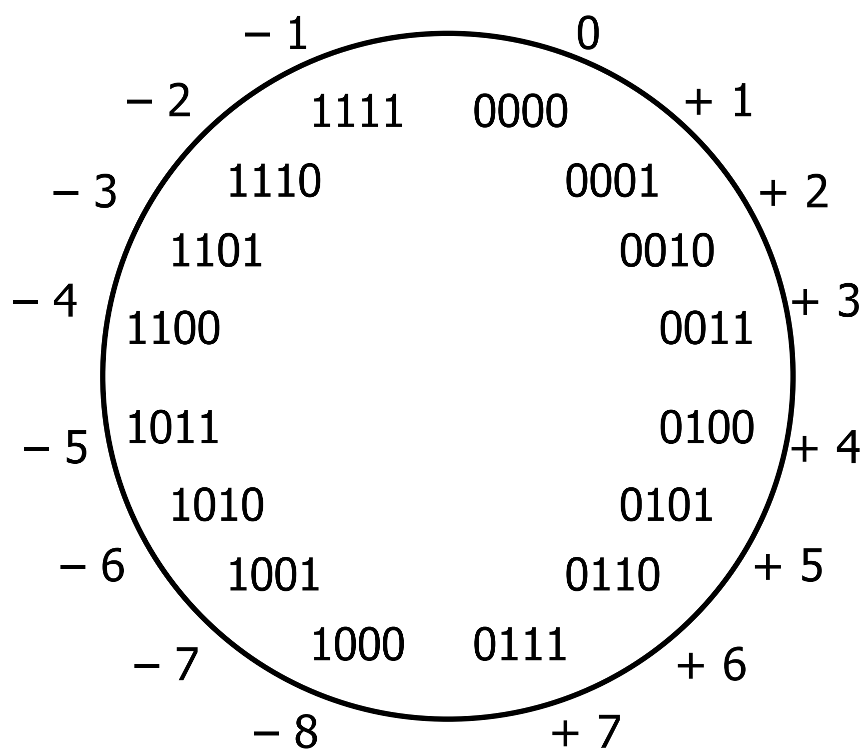


Modular Arithmetic

Multiplication

Compute answer in **2x bits**. Most languages drop high order half.

5	0101
x 5	x 0101
$\cancel{25}$	00011001
-7	
-2	1110
x 6	x 0110
$\cancel{-12}$	11110100
4	



Modular Arithmetic

Multiplication by ***shift**-and-**add***

ex

Available operations

`x << k` implements `x *` 2^k

`x + y`

Implement `y = x * 24` using only <<, +, and integer literals

What does this function compute?

ex

```
unsigned puzzle(unsigned x, unsigned y) {
  unsigned result = 0;
  for (unsigned i = 0; i < 32; i++){
    if (y & (1 << i)) {
      result = result + (x << i);
    }
  }
  return result;
}
```

Casting Integers in C

!!!

Number literals: `37` is signed, `37U` is unsigned

Integer Casting: *bits unchanged, just reinterpreted.*

Explicit casting:

```
int tx = (int) 73U;              // still 73
unsigned uy = (unsigned) -4;    // big positive #
```

Implicit casting: **Actually does**

```
tx = ux;                // tx = (int)ux;
uy = ty;                // uy = (unsigned)ty;
void foo(int z) { ... }
foo(ux);                // foo((int)ux);
if (tx < ux) ...        // if ((unsigned)tx < ux) ...
```

More Implicit Casting in C

!!!

If you **mix unsigned** and **signed** in a single expression, then
signed** values are **implicitly cast to <u>unsigned</u>.

Includes comparisons (<, >, ==, <=, >=)

How are the argument bits interpreted?

Argument$_1$	**Op**	**Argument$_2$**	**Type**	**Result**
`0`	`==`	`0U`	unsigned	1
`-1`	`<`	`0`	signed	1
`-1`	`<`	`0U`	unsigned	**0**
`2147483647`	`<`	`-2147483648`		
`2147483647U`	`<`	`-2147483648`		
`-1`	`<`	`-2`		
`(unsigned)-1`	`<`	`-2`		
`2147483647`	`<`	`2147483648U`		
`2147483647`	`<`	`(int)2147483648U`		

Note: T_{min} = -2,147,483,648 T_{max} = 2,147,483,647