CS 240 Spring 2020
Foundations of Computer Systems
Ben Wood

WELLESLEY

Integer Representation

Representation of integers: unsigned and signed
Modular arithmetic and overflow
Sign extension
Shifting and arithmetic
Multiplication
Casting

https://cs.wellesley.edu/~cs240/s20/

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**

(4-bit) **unsigned integer representation**

1	0	1	1
8	4	2	1
2^3	2^2	2^1	2^0
3	2	1	0

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

weight

position

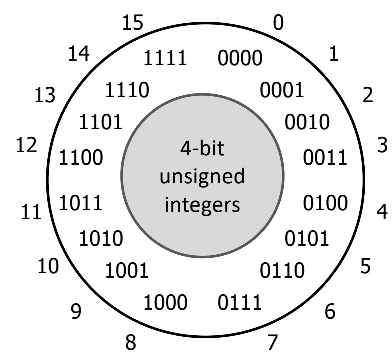
n-bit unsigned integers:

minimum =

maximum =

modular arithmetic, overflow

$$\begin{array}{rr} 11 & 1011 \\ \underline{+\ 2} & \underline{+\ 0010} \end{array}$$



$$\begin{array}{rr} 13 & 1101 \\ \underline{+\ 5} & \underline{+\ 0101} \end{array}$$

`x+y` in n-bit unsigned arithmetic is in math

unsigned overflow =

=

Unsigned addition *overflows* if and only if

sign-magnitude

!!!

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

0 means non-negative1 means negative

Remaining bits are an unsigned magnitude

8-bit sign-magnitude:

00000000 represents _____

01111111 represents _____

10000101 represents _____

10000000 represents _____

ex

Anything weird here?

Arithmetic?

Example:
4 - 3 != 4 + (-3)

```
 00000100
+10000011
```

Zero?

(4-bit) **two's complement signed integer representation**

compare to unsigned

1	0	1	1
-2^3	2^2	2^1	2^0

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

4-bit two's complement integers:

minimum =

maximum =

two's complement vs. unsigned

_	_	… _	_	_
2^{n-1}	2^{n-2}	… 2^2	2^1	2^0
-2^{n-1}	2^{n-2}	… 2^2	2^1	2^0

unsigned places

two's complement places

What's the difference?

n-bit unsigned numbers:

minimum =

maximum =

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 =

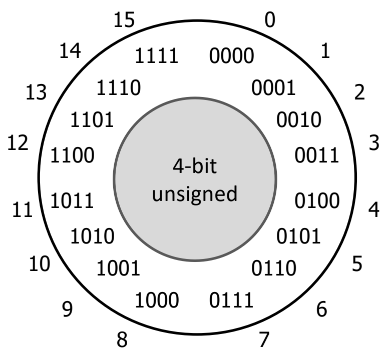
maximum =

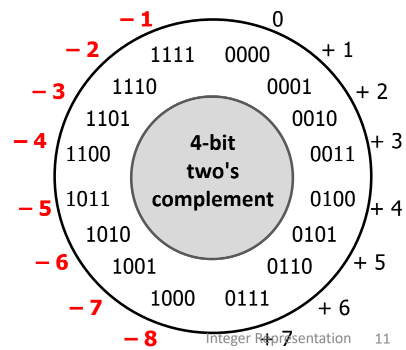
4-bit **unsigned** vs. 4-bit **two's complement**

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

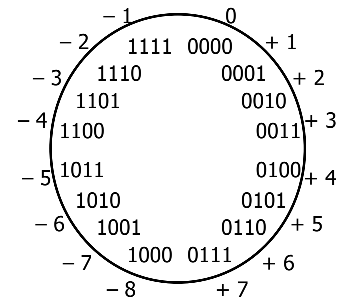




two's complement **addition**

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

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



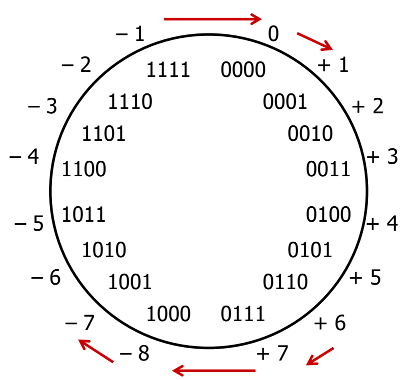
Modular Arithmetic

two's complement ***overflow***

Addition *overflows*
if and only if
if and only if

-1	1111
+ 2	+ 0010

6	0110
+ 3	+ 0011



Modular Arithmetic

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

Addition, subtraction, hardware

Sign

Negative one

Complement rules

Another derivation

ex

How should we represent 8-bit negatives?

- For all positive integers ***x***,
  we want the representations of ***x*** and ***–x*** to sum to zero.
- We want to use the standard addition algorithm.

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

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

Convert/cast *signed number to* ***larger*** *type.*

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

Rule/name?

unsigned **shifting** and **arithmetic**

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

logical shift right

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

two's complement **shifting** and **arithmetic**

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

arithmetic shift right

1 1 1 0 1 1 0 1

1 1 1 1 1 0 1 1 0 1

signed

x = -19;

y = x >> 2;

y == -5

shift**-and-**add

ex

Available operations

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

x + y

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

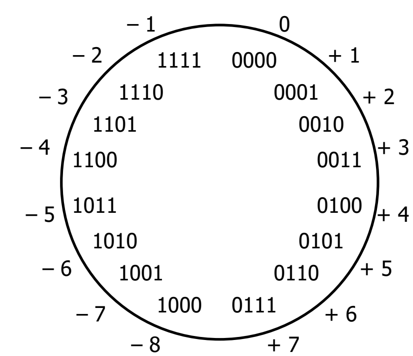
What does this function compute?

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;
}
```


multiplication

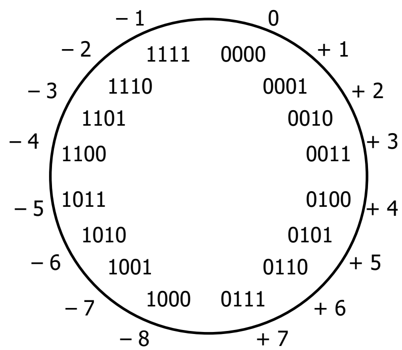
2	0010
x 3	x 0011
6	00000110
-2	1110
x 2	x 0010
-4	11111100



Modular Arithmetic

multiplication

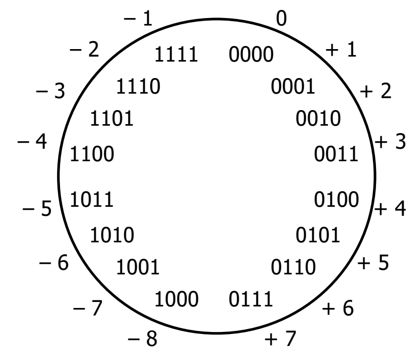
5	0101
x 4	x 0100
~~20~~	00010100
4	
-3	1101
x 7	x 0111
~~-21~~	11101011
-5	



Modular Arithmetic

multiplication

5	0101
x 5	x 0101
~~25~~	00011001
-7	
-2	1110
x 6	x 0110
~~-12~~	11110100
4	



Modular Arithmetic

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 unsigned**.*

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	<	-2147483647-1		
2147483647U	<	-2147483647-1		
-1	<	-2		
(unsigned)-1	<	-2		
2147483647	<	2147483648U		
2147483647	<	(int)2147483648U		

Note: T_{min} = -2,147,483,648 T_{max} = 2,147,483,647
T_{min} must be written as `-2147483647-1` (see pg. 77 of CSAPP for details)