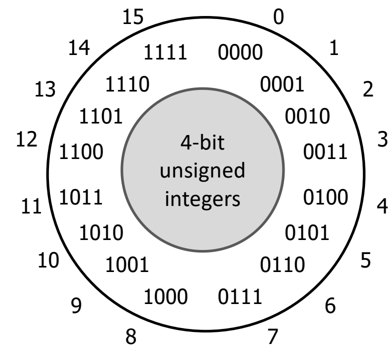
Wellesley CS 240

Integer Representation

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

modular arithmetic, overflow

11	1011
+ 2	+ 0010



13	1101
+ 5	+ 0101

`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-negative 1 means negative

Remaining bits are an unsigned magnitude

8-bit sign-magnitude:

00000000 represents _____

01111111 represents _____

10000101 represents _____

10000000 represents _____

Anything weird here?

Arithmetic?

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

00000100
+10000011

Zero?

ex

(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 =

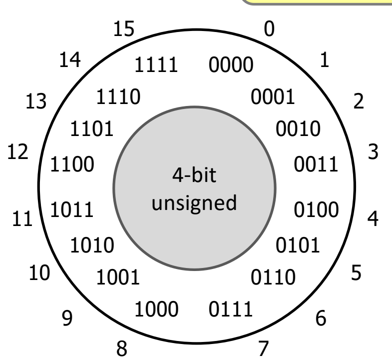
4-bit **unsigned** vs. 4-bit **two's complement**

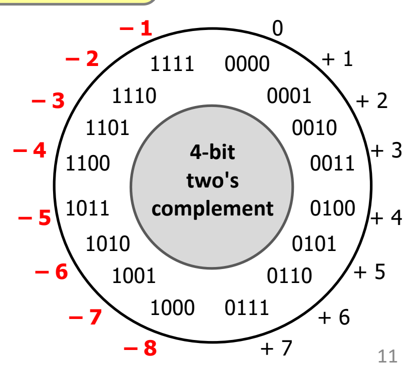
1 0 1 1

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

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

11 ← **difference = ___ = 2—** → -5





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…

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;
}
```