Integers II

Integers

- **Binary representation of integers**
  - **Unsigned and signed**
  - **Arithmetic operations**
- Consequences of finite width representations
  - Overflow
- Shifting operations

Values to Remember

Unsigned

- **UMin** = 0
  - 0b00…00
- **UMax** = 2^w-1
  - 0b11…11

Signed (2's Complement)

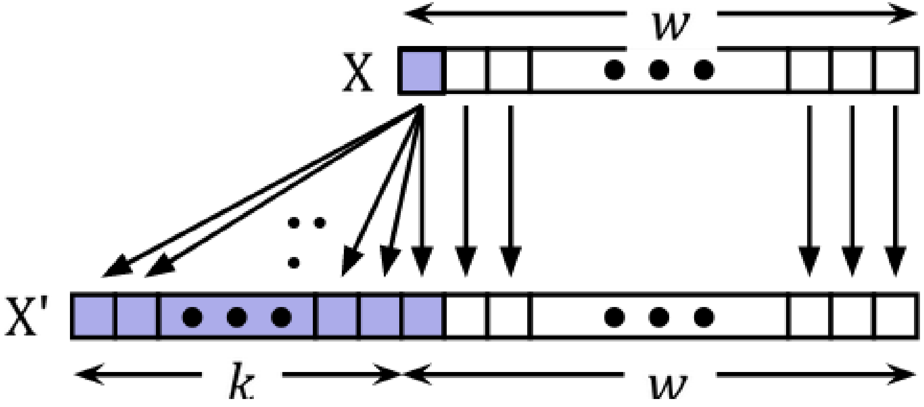
- **TMin** = -2^{w-1}
  - 0b10…00
- **TMax** = 2^{w-1} - 1
  - 0b01…11

<u>Example:</u> if w = 64

	Hex	Decimal
UMax	FF FF FF FF FF FF FF FF	18,446,744,073,709,551,615
TMax	7F FF FF FF FF FF FF FF	9,223,372,036,854,775,807
UMin	00 00 00 00 00 00 00 00	0
TMin	80 00 00 00 00 00 00 00	-9,223,372,036,854,775,808

Sign Extension

- Given a *w*-bit integer, how can we extend it to a (*w*+*k*)-bit integer while keeping the value the same?
  - Unsigned - pad with 0s
    - Ex: 0b1000 = 0b00001000 = 8
  - Signed - pad with the **most significant bit**
    - Ex: 0b1000 = 0b11111000 = -8



Two’s Complement Arithmetic

- Same as unsigned!
  - Simplifies hardware, no special algorithm needed
  - Just add as normal, then discard the highest carry bit
    - **Modular addition**: result = sum modulo 2^w

Example:

0011	=	3
+0001	=	1
0100	=	4

carry

1111		
1101	=	-3
+1111	=	-1
~~1~~1100	=	-4

Why Does Two's Complement Work?

- For all representable numbers *x*, we theoretically want *additive inverse*:
  - i.e. (bit representation of *x*) + (bit representation of *-x*) = 0
- What are the 8-bit negative encodings for the following?

```
  00000001        00000010        11000011
+ ????????      + ????????      + ????????
  --------        --------        --------
  00000000        00000000        00000000
```

Why Does Two's Complement Work? (pt 2)

- For all representable numbers *x*, we theoretically want *additive inverse*:
  - i.e. (bit representation of *x*) + (bit representation of *-x*) = 0
- What are the 8-bit negative encodings for the following?

00000001	00000010	11000011
+ 11111111	+ 11111110	+00111101
00000000	00000000	00000000

These are the bitwise complement plus 1!

`-x == ~x + 1`

Integers

- Binary representation of integers
    - Unsigned and signed
    - Arithmetic operations
- **Consequences of finite width representations**
    - **Overflow**
- Shifting operations

Arithmetic Overflow

- What happens if a calculation produces a result that *can't* be represented in the current encoding scheme? **Overflow!**
  - Remember: fixed width integers can't represent every possible number
  - Occurs in both signed and unsigned
  - Can occur in both positive *and* negative directions
- Both C and Java ignore overflow exceptions
  - You end up with a bad value in your program and no indication/warning

Overflow: Unsigned

- Addition: drop carry bit (result is 2^w too small)

$$\begin{array}{rcr} 1111 & = & 15 \\ +\ 0001 & = & 1 \\ \hline \cancel{1}0000 & = & 16 \end{array}$$

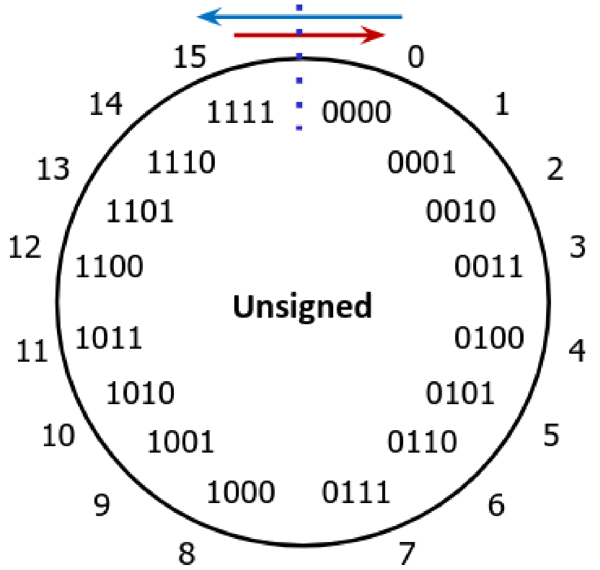
0

- Subtraction: “borrow” extra bit (result is 2^w too large)

$$\begin{array}{rcr} \mathbf{1}0001 & = & 1 \\ -\ 0010 & = & 2 \\ \hline \mathbf{1111} & = & -1 \end{array}$$

15

Occurs when result is *less than* both operands for addition, or *greater than* for subtraction



Overflow: Signed

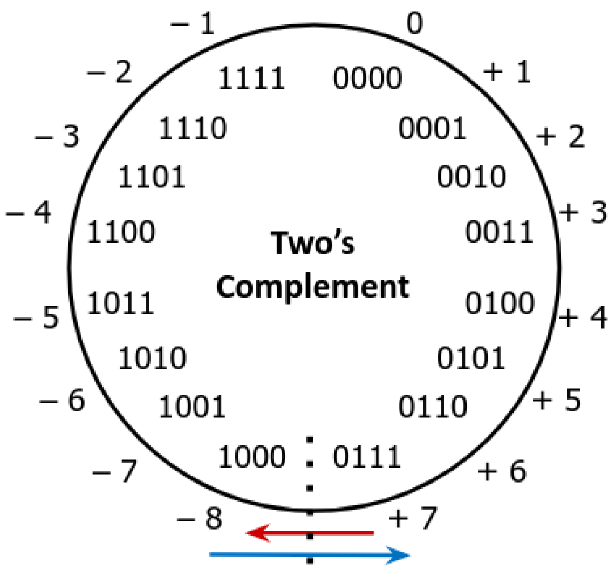
Occurs when both operands for an *addition* have the same sign, and result doesn't match

- Positive addition: (+) + (+) = (-)

```
   0110   =   6
+  0011   =   3
   1001   =  -7???
```

- Negative addition (i.e. subtraction): (-) + (-) = (+)

```
   1001   =  -7
-  0011   =   3
   0110   =   6???
```



Why does this matter?

- **1985:** Therac-25 radiation therapy machine
  - Overdoses of radiation due to arithmetic overflow on 1-byte safety flag
- **2000:** Y2K problem
  - Limited representation (2-digit decimal year)
  - Similar issue will occur with Unix time in 2038!

- **2013:** Deep impact spacecraft lost
  - Suspected integer overflow from storing time as tenth-seconds in unsigned int
    - Lost on 8/11/13, 00:38:49.6

Integers

- Binary representation of integers
  - Unsigned and signed
  - Arithmetic operations
- Consequences of finite width representations
  - Overflow
- **Shifting operations**

Shift Operations

- Move all bits left or right, extra bits “fall off” the end
- Left shift by *n* positions (x << n)
    - Lose the most-significant *n* bits, fill in the least-significant *n* bits with 0s
- Right shift by *n* positions (x >> n)
    - Lose the least-significant *n* bits
    - Unsigned, use **logical**: fill with most-significant *n* bits with 0s
    - Signed, use **arithmetic**: replicate the previous most-significant bit

Ex: 0x22

x	0010 0010
x << 3	0001 0000
(logical) x >> 2	0000 1000
(arithmetic) x>> 2	0000 1000

Ex: 0xA2

x	1010 0010
x << 3	0001 0000
(logical) x >> 2	0010 1000
(arithmetic) x>> 2	1110 1000

Shift Operations (pt 2)

- Arithmetic
  - Left shift (x << n) == multiply by 2^n
  - Right shift (x >> n) == divide by 2^n
    - For signed values, logical right shift preserves the sign
  - **Fun fact:** Shifting is often *faster* than the general multiply and divide operations!
- Notes:
  - Shifts by less than 0 or more than *w* (width of the variable) are **undefined**
    - i.e. we don't know what will happen!
  - In Java, arithmetic shift is >>, logical is >>>

Left Shifting, 8-bit Example

- Shifting can cause overflow!
- In theory x << n should be $x*2^n$

Signed overflow

Code	Binary	Signed	Unsigned	Theoretical Value
x = 25	00011001	25	25	25
L1 = x << 2	00 01100100	100	100	100
L2 = x << 3	000 11001000	-56	200	200
L3 = x << 4	0001 10010000	-112	114	400

Unsigned overflow

Right Shifting, 8-bit Example

- Unsigned = logical shift
- In theory, x >> n should be $x \div 2^n$

Code	Binary	Unsigned	Theoretical Value
x = 240u	11110000	240	240
R1 = x >> 3	00011110 000	30	30
R2 = x >> 5	00000111 10000	7	7.5?

Right Shifting, 8-bit Example (pt 2)

- Signed = arithmetic shift
- In theory, x >> n should be $x \div 2^n$

Code	Binary	Unsigned	Theoretical Value
x = -16	11110000	-16	-16
R1 = x >> 3	11111110 000	-2	-2
R2 = x >> 5	11111111 10000	-1	-0.5?

Summary

- We can represent a limited number of values in *w* bits
    - When we exceed the limit (in either direction), we get **overflow**
- **Shifting** is a useful bitwise behavior
    - Can be used to remove certain bits (similar to masking), or in place of multiplication
    - Right shift can be **logical** or **arithmetic**
        - Logical pads with 0s, used for unsigned
        - Arithmetic pads with MSB, used for signed

BONUS SLIDES

Some examples of using shift operators in combination with bitmasks.

- Extract the 2nd most significant byte of an `int`
- Extract the sign bit of a signed `int`
- Conditionals as Boolean expressions

Practice Question 1

- Assuming 8-bit data (*i.e.*, bit position 7 is the MSB), what will the following expression evaluate to?
  - UMin = 0, UMax = 255, TMin = -128, TMax = 127

Practice Questions 2

- For the following additions, did signed and/or unsigned overflow occur?
  - `0x27 + 0x81`
  - `0x7F + 0xD9`
- Helpful values (assuming 8-bit integers):
  - **`0x27`** = 39 (signed) = 39 (unsigned)
  - **`0xD9`** = -39 (signed) = 217 (unsigned)
  - **`0x7F`** = 127 (signed) = 127 (unsigned)
  - **`0x81`** = -127 (signed) = 129 (unsigned)

Using Shifts and Masks

- Extract the 2nd most significant *byte* of an `int`:
  - First shift, then mask: (x>>16) & 0xFF

x	00000001 00000010 00000011 00000100
x>>16	00000000 00000000 00000001 00000010
0xFF	00000000 00000000 00000000 11111111
(x>>16) & 0xFF	00000000 00000000 00000000 00000010

  - Or first mask, then shift: `(x & 0xFF0000)>>16`

x	00000001 00000010 00000011 00000100
0xFF0000	00000000 11111111 00000000 00000000
X & 0xFF0000	00000000 00000010 00000000 00000000
(x & 0xFF)>>16	00000000 00000000 00000000 00000010

Using Shifts and Masks (pt 2)

- Extract the *sign bit* of a signed `int`:
  - First shift, then mask: `(x>>31) & 0x1`
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear `1`s possibly shifted in

x	00000001	00000010	00000011	00000100
x>>31	00000000	00000000	00000000	00000000
0x1	00000000	00000000	00000000	00000001
(x>>31) & 0x1	00000000	00000000	00000000	00000000

x	10000001	00000010	00000011	00000100
x>>31	11111111	11111111	11111111	11111111
0x1	00000000	00000000	00000000	00000001
(x>>31) & 0x1	00000000	00000000	00000000	00000001