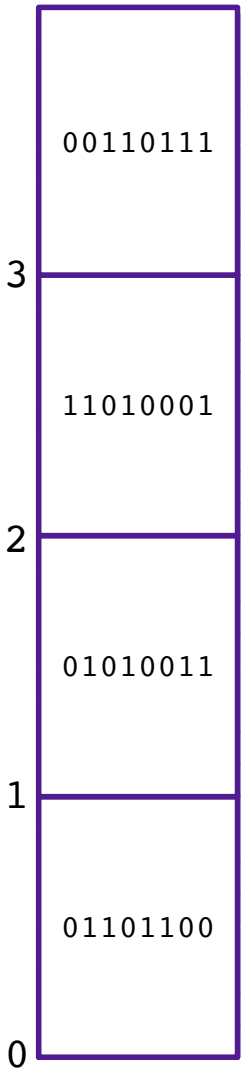
Lecture 3: Representing Signed Integers

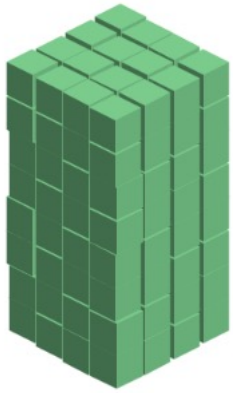
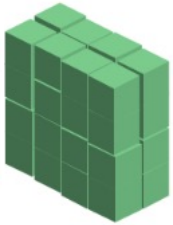
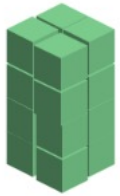
CS 105

Memory: A (very large) array of bytes

- **Memory** is an array of ~~bits~~ bytes
- A **byte** is a unit of eight bits
- An index into the array is an **address**, **location**, or **pointer**
  - Often expressed in hexadecimal
- We speak of the *value* in memory at an address
  - The value may be a single byte …
  - … or a multi-byte quantity starting at that address



Base-2 Integers (aka Binary Numbers)

128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
							
0	0	0	0	0	1	0	1
0	0	1	0	1	1	1	1
1	1	1	1	1	1	1	1

Representing **Signed** Integers

- Option 1: sign-magnitude
  - One bit for sign; interpret rest as magnitude

Representing **Signed** Integers

- Option 2: one's-complement
  - Complement all bits

Representing Signed Integers

- Option 3: two's complement
  - Most used, like one's complement, but with one zero value
  - Like unsigned, except the high-order contribution is *negative*
  - $Signed(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$

Two's Complement Signed Integers

- "Signed" does not mean "negative"
- High order bit is the *sign bit*
  - To negate, complement all the bits and add 1
- Arithmetic is the same as unsigned—same circuitry
- (Error conditions and comparisons are different)

Example: Three-bit integers

unsigned		signed
111	7	
110	6	
101	5	
100	4	
011	3	011
010	2	010
001	1	001
000	0	000
	−1	111
	−2	110
	−3	101
	−4	100

Example: Three-bit integers

unsigned		signed
111	7	
110	6	
101	5	
100	4	
011	3	011
010	2	010
001	1	001
000	0	000
	−1	111
	−2	110
	−3	101
	−4	100

- The high-order bit is the *sign bit.*
- The largest unsigned value is `11...1`, `UMax`.
- The signed value for `−1` is always `11...1`.
- Signed values range between `TMin` and `TMax`.

This representation of signed values is called *two's complement.*

Important Signed Numbers

w	8	16	32	64
Max	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
Min	0x80	0x8000	0x80000000	0x8000000000000000
0	0x00	0x0000	0x00000000	0x0000000000000000
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF

Exercise 1: Signed Integers

Assume an 8 bit (1 byte) signed integer representation using two's complement:

- What is the binary representation for 47? **00101111**
- What is the binary representation for -47? **11010001**
- What is the number represented by 10000110? **-122**
- What is the number represented by 00100101? **37**

Casting between Numeric Types

- Casting from **shorter** to **longer** types preserves the value
- Casting from **longer** to **shorter** types drops the high-order bits (modulus)
- Casting between signed/unsigned types preserves the bits (it just changes the interpretation)
- Implicit casting occurs in assignments and parameter lists. In mixed expressions, signed values are implicitly cast to unsigned
  - Source of many errors!

Exercise 2: Casting

- Assume you have a machine with 6-bit integers/3-bit shorts
- Assume variables: `int x = -17; short sy = -3;`
- Complete the following table

Expression	Decimal	Binary
x	-17	
sy	-3	
(unsigned) x		
(int) sy		
(short) x		

Exercise 2: Casting

- Assume you have a machine with 6-bit integers/3-bit shorts
- Assume variables: `int x = -17; short sy = -3;`
- Complete the following table

Expression	Decimal	Binary
x	-17	**101111**
sy	-3	**101**
(unsigned) x	**47**	**101111**
(int) sy	**-3**	**111101**
(short) x	**-1**	**111**

When to Use Unsigned

- Rarely
- When doing multi-precision arithmetic, or when you need an extra bit of range … but be careful!

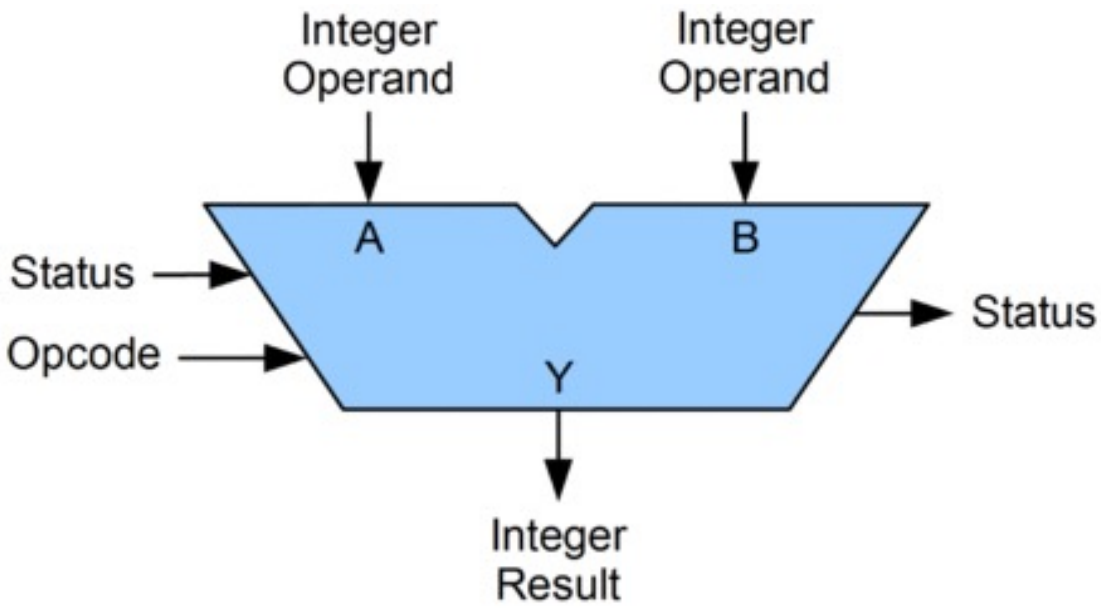
```
unsigned i;
for (i = cnt-2; i >= 0; i--){
    a[i] += a[i+1];
}
```

Arithmetic Logic Unit (ALU)

- A circuit that performs bitwise operations and arithmetic on integer binary types

Status examples:
- Carry-out
- Zero
- Negative
- Overflow
- parity

Opcode examples:
- Add, Subtract
- Increment, Decrement
- AND, OR, XOR
- Shift, Rotate



Bitwise vs Logical Operations in C

- Bitwise Operators `&, |, ~, ^`
  - View arguments as bit vectors
  - operations applied bit-wise in parallel
- Logical Operators `&&, ||, !`
  - View 0 as “False”
  - View anything nonzero as “True”
  - Always return 0 or 1
  - Short-circuit termination
- Shift operators `<<, >>`
  - Left shift fills with zeros
  - For signed integers, right shift is arithmetic (fills with high-order bit)

Exercise 3: Bitwise vs Logical Operations

- Assume signed char data type (one byte)

```
~(-30)      = ~11100010 = 00011101 = 29
!(-30)      = !11100010 = 00000000 = 0

120 &  85   = 01111000 &  01010101 = 01010000 = 80
120 |  85   = 01111000 |  01010101 = 01111101 = 125
120 && 85   = 01111000 && 01010101 = 00000001 = 1
120 || 85   = 01111000 || 01010101 = 00000001 = 1

-106 << 4   = 10010110 << 4 = 01100000 = 96
-106 << 2   = 10010110 << 2 = 01011000 = 88
-106 >> 4   = 10010110 >> 4 = 11111001 = -7
-106 >> 2   = 10010110 >> 2 = 11100101 = -27
```

Addition/Subtraction Example

- Compute 5 + -3 assuming all ints are stored as **four-bit signed** values

```
  1  1
  0 1 0 1
+ 1 1 0 1
---------
  0 0 1 0     = 2 (Base-10)
```

Exactly the same as unsigned numbers!

… but with different error cases

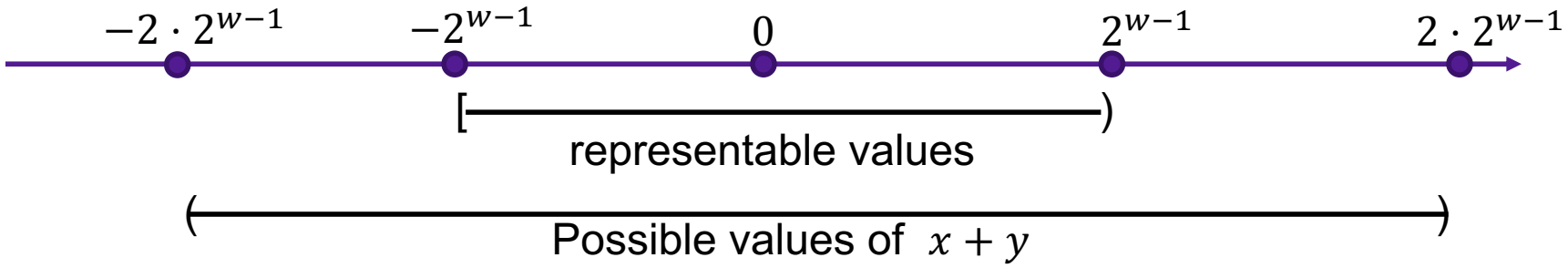
Addition/Subtraction with Overflow

- Compute 5 + 3 assuming all ints are stored as **four-bit signed** values

```
  1 1 1
  0 1 0 1
+ 0 0 1 1
-------
  1 0 0 0     = -8 (Base-10)
```

Error Cases

- Assume w-bit signed values



- $x +_w^t y = \begin{cases} x + y - 2^w & \text{(positive overflow)} \\ x + y & \text{(normal)} \\ x + y + 2^w & \text{(negative overflow)} \end{cases}$

- overflow has occurred iff $x > 0$ and $y > 0$ and $x +_w^t y < 0$
  or $x < 0$ and $y < 0$ and $x +_w^t y > 0$

Exercise 4: Binary Addition

- Given the following 5-bit signed values, compute their sum and indicate whether an overflow occurred

x	y	x+y	overflow?
00010	00101		
01100	00100		
10100	10001		

Exercise 4: Binary Addition

- Given the following 5-bit signed values, compute their sum and indicate whether an overflow occurred

x	y	x+y	overflow?
00010	00101	**00111**	**no**
01100	00100	**10000**	**yes**
10100	10001	**00101**	**yes**

Multiplication Example

- Compute 3 x 2 assuming all ints are stored as four-bit signed values

```
  0011
x 0010
------
  0000
+ 0011 0
--------
  0110   = 6 (Base-10)
```

Exactly like unsigned multiplication!

… except with different error cases

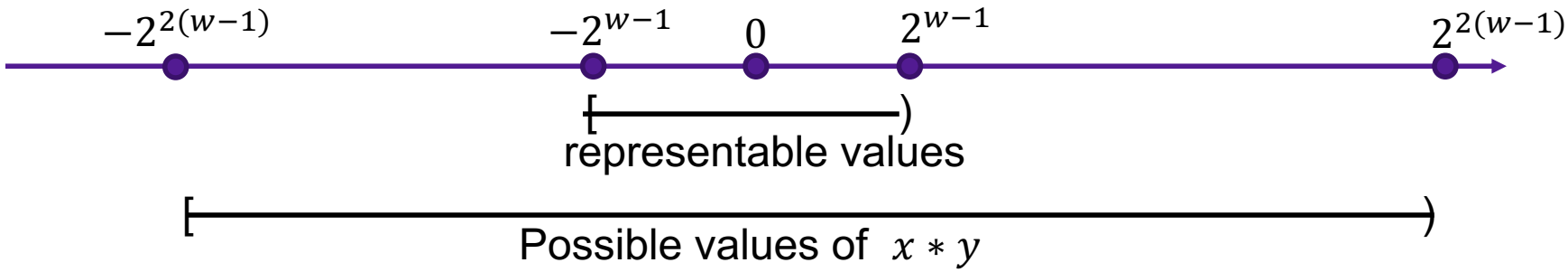
Multiplication Example

- Compute 5 x 2 assuming all ints are stored as four-bit signed values

```
  0101
 x0010
 ----
  0000
+0101 0
 ------
  1010   = -6 (Base-10)
```

Error Cases

- Assume w-bit unsigned values



- $x *_w^t y = U2T((x \cdot y) \bmod 2^w)$

Exercise 5: Binary Multiplication

- Given the following 3-bit signed values, compute their product and indicate whether an overflow occurred

x	y	x*y	overflow?
100	101		
010	011		
111	010		

Exercise 5: Binary Multiplication

- Given the following 3-bit signed values, compute their product and indicate whether an overflow occurred

x	y	x*y	overflow?
100	101	**100**	**yes**
010	011	**110**	**yes**
111	010	**110**	**no**