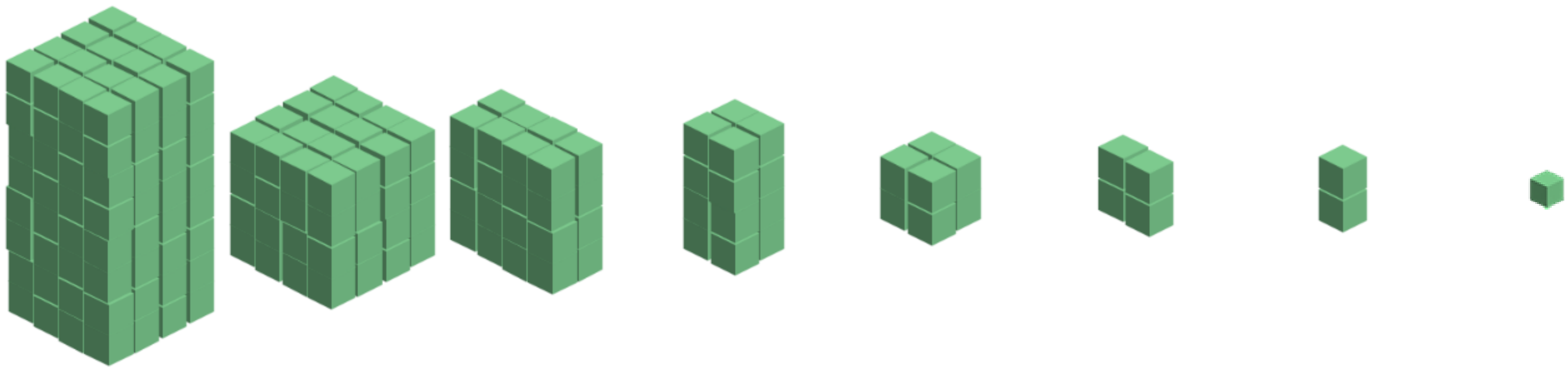
Lecture 3: Representing Signed Integers

CS 105 Spring 2026

Review: Binary Numbers

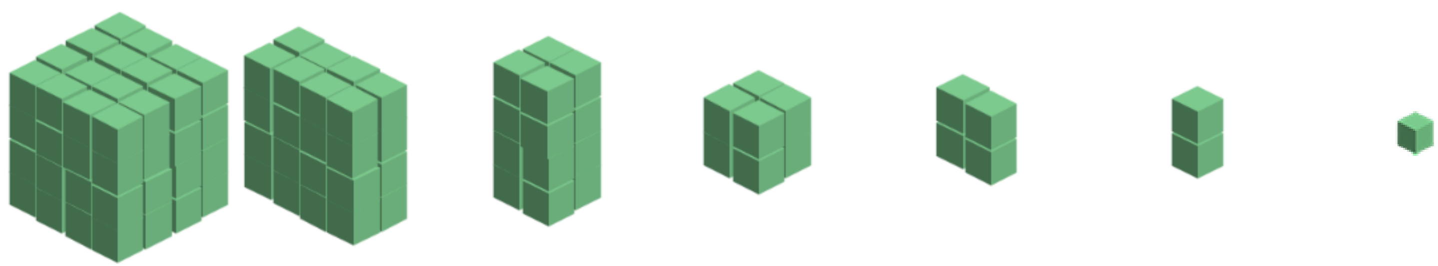


128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
0	0	0	0	0	1	0	1
0	0	1	0	1	1	1	1
1	1	1	1	1	1	1	1

Representing Signed Integers

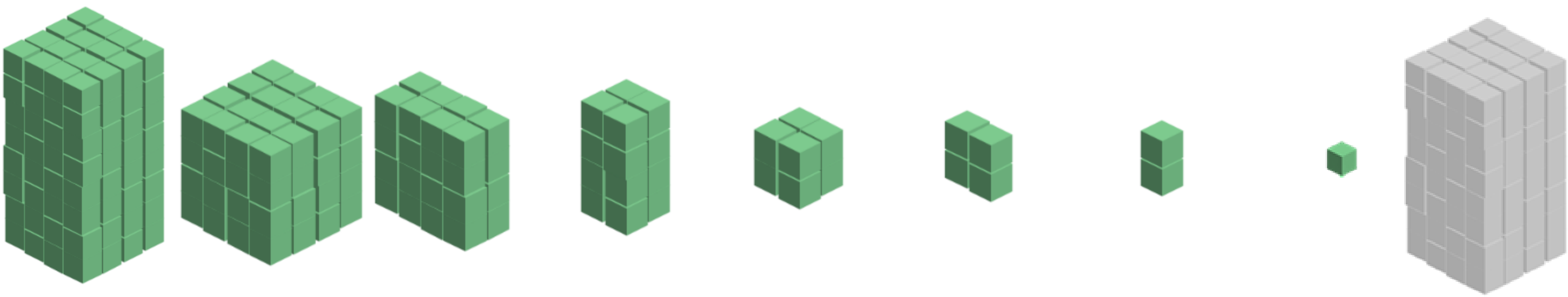
- Option 1: sign-magnitude
  - One bit for sign; interpret rest as magnitude
  - $Signed(x) = (-1)^{x_{w-1}} \cdot \sum_{i=0}^{w-2} x_i \cdot 2^i$

+/-	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
1	0	0	0	0	1	0	1
1	1	1	1	1	1	1	1



Representing Signed Integers

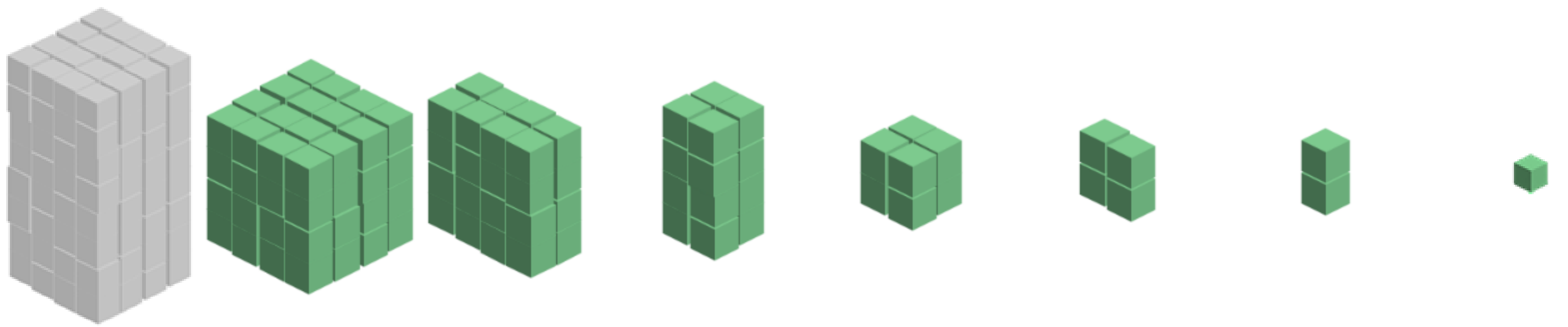
- Option 2: excess-K
  - Choose a positive K in the middle of the unsigned range
  - $Signed(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i - 2^{w-1}$



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

Representing Signed Integers

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



-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
1	0	0	0	0	1	0	1
1	1	1	1	1	1	1	1

Important Signed Integers

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

Exercise 1: (Signed) Binary Numbers

- Consider the following one-byte binary values. What is the (signed) integer interpretation of these values?
  1. `01101001`
  2. `10010111`
  3. `01000010`
  4. `10111110`
- What is the one-byte binary representation of the following integers?
  - 47
  - -47

```
-x == -x+1-1 == -1-x+1 == 111…1-x+1 = ~x+1
```

Integers in C

C Data Type	Size (bytes)
unsigned char	1
unsigned short	2
unsigned int	4
unsigned long	8

C Data Type	Size (bytes)
char	1
short	2
int	4
long	8

Casting between Numeric Types

- Casting from shorter to longer types preserves the value
- Casting from longer to shorter types drops the high-order bits
- 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 int) x		
(int) sy		
(short) x		

When to Use Unsigned

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

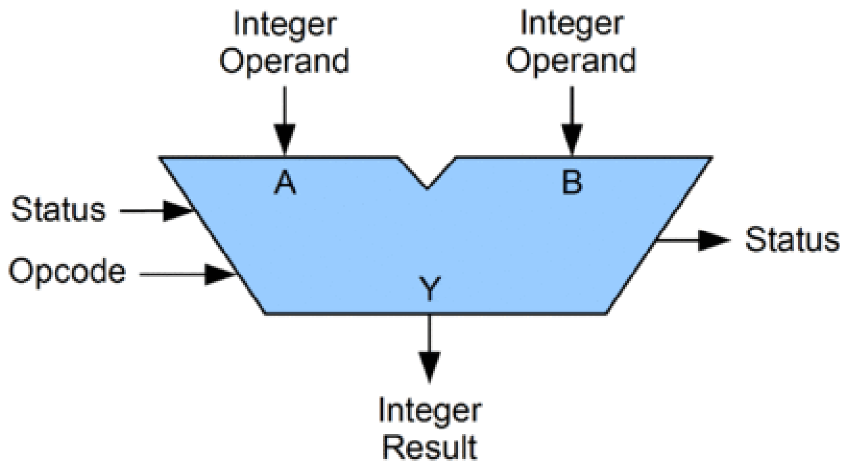
```
int example(){
  int a[5] = {1,2,3,4,5};

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

  return a[i]
}
```

Arithmetic Logic Unit (ALU)

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



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
  - Early 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

- What is the binary representation of each of the following expressions? Assume signed char data type (one byte).

1. ~(-30)
2. -30 & 22
3. -30 && 22
4. 22 << 1
5. 22 >> 1
6. -30 >> 1

Addition 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

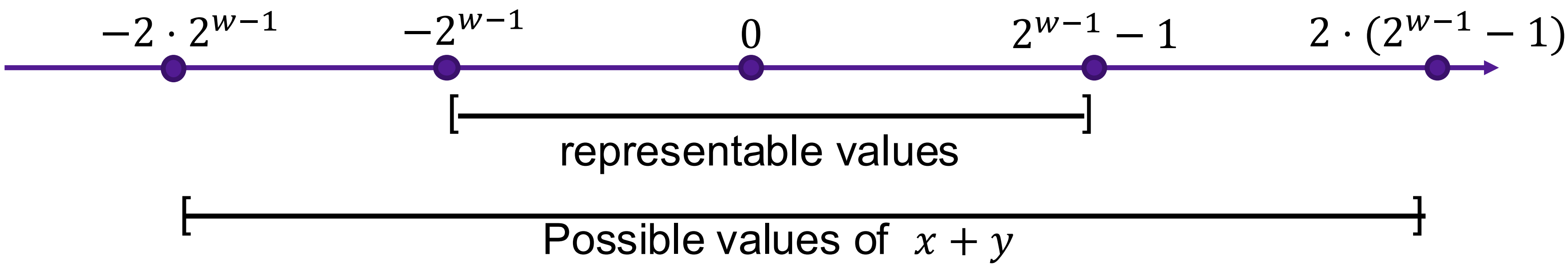
Addition/Subtraction with Overflow

- Compute 5 + 6 assuming all ints are stored as four-bit signed values

```
    1
    0101
  + 0110
  ------
    1011     = -5 (Base-10)
```

Error Cases

- Assume w-bit signed values



- $x +_w y = \begin{cases} x + y \; - 2^{w-1} & \text{(positive overflow)} \\ x + y. & \text{(normal)} \\ x + y + 2^{w-1} & \text{(negative overflow)} \end{cases}$

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

Exercise 4: Binary Addition

- Given the following 5-bit signed values, compute their sum and indicate whether or not an overflow occurred

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

Multiplication Example

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

```
    0011
  x 0010
  ------
    0000
+ 00110
  ------
    0110
```

= 6 (Base-10)

Exactly like unsigned multiplication!

… except with different error cases

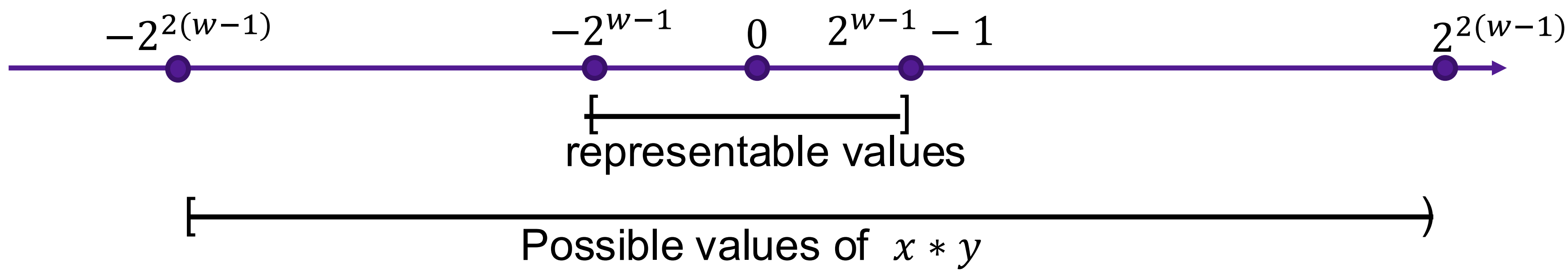
Multiplication Example

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

```
    0101
  x 0010
  ------
    0000
 + 01010
 -------
    1010   = -6 (Base-10)
```

Error Cases

- Assume w-bit unsigned values



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

Exercise 5: Binary Multiplication

- Given the following 3-bit signed values, compute their product and indicate whether or not an overflow occurred

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