Lecture 3: Representing Signed Integers

CS 105 Fall 2020

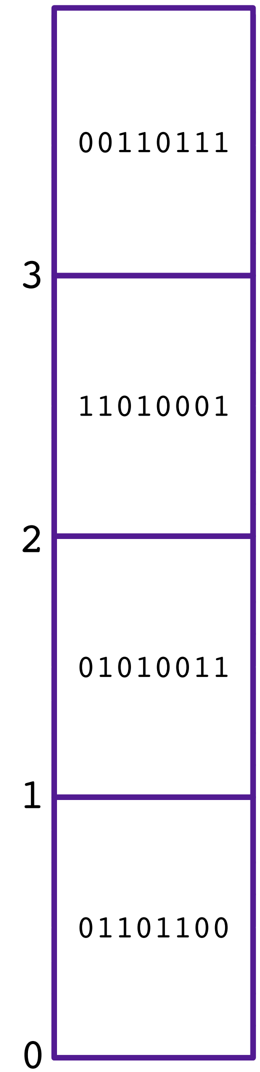
Memory: A (very large) array of bytes

- **Memory** is an array of ~~bits~~ bytes

- A **byte** is a unit of eight bits

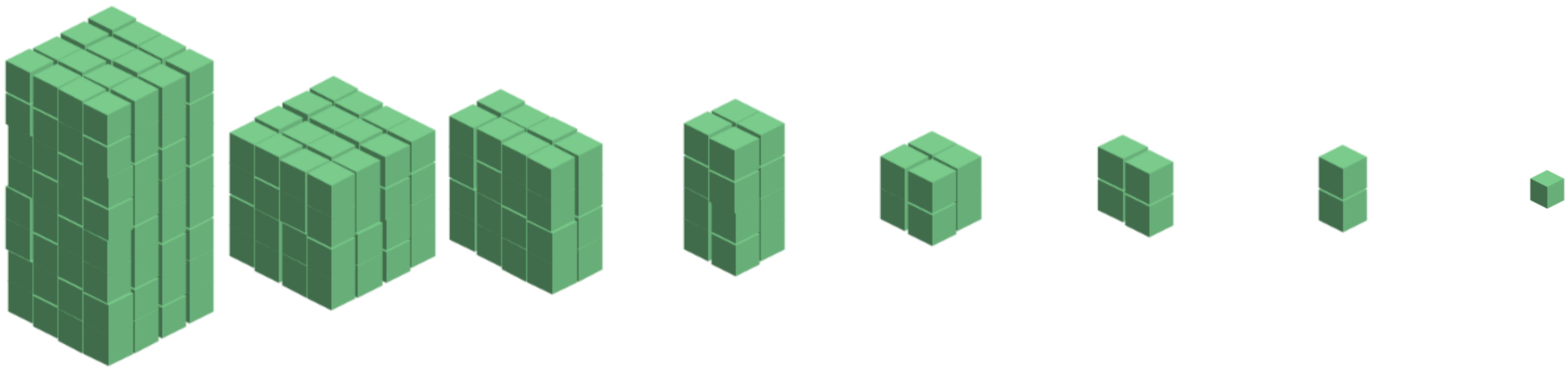
- An index into the array is an **address**, **location**, or **pointer**
  - Often expressed in hexadecimal

- 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)

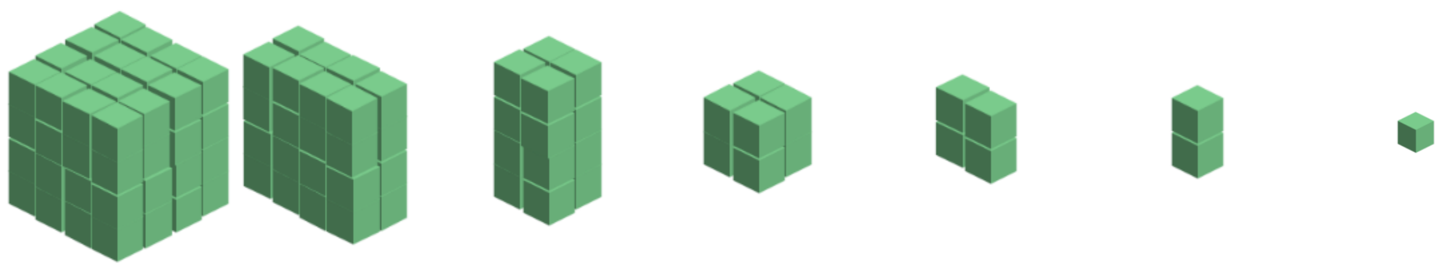


0	0	0	0	0	1	0	1
0	0	1	0	1	1	1	1
1	1	1	1	1	1	1	1

Representing Signed Integers

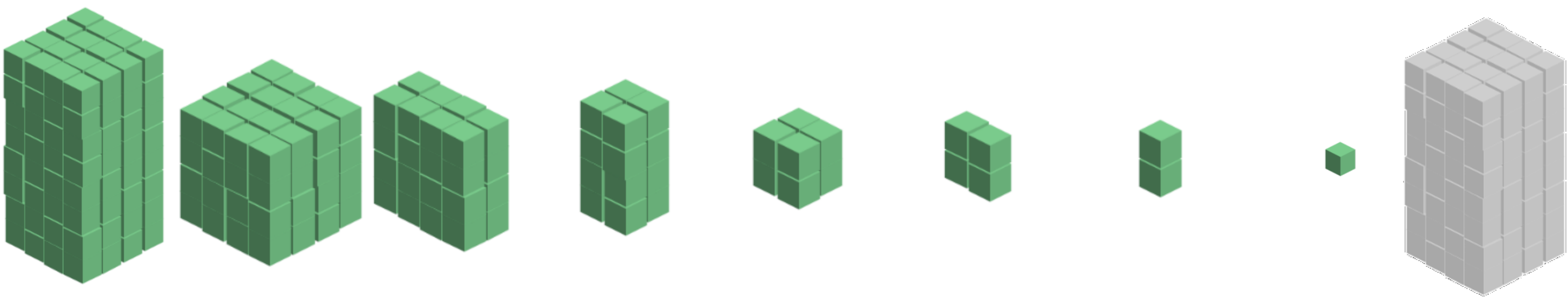
- Option 1: sign-magnitude
  - One bit for sign; interpret rest as magnitude

+/-	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



Representing Signed Integers

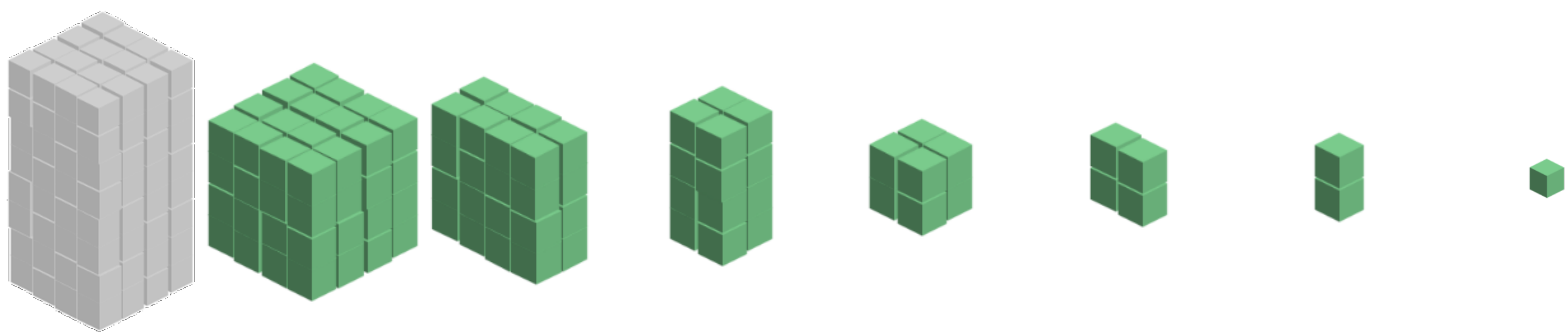
- Option 2: excess-K
  - Choose a positive K in the middle of the unsigned range
  - SignedValue(w) = UnsignedValue(w) – K



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	

Representing Signed Integers

- Option 3: two's complement
  - Most commonly used
  - 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 (-2^6)	**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	**1**	**1**	**1**	**1**	**0**	**1**	**1**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**

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 $11\ldots1$, UMax.
- The signed value for -1 is always $11\ldots1$.
- Signed values range between `TMin` and TMax.

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

Important Signed Numbers

	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 Integers

Assume an 8 bit (1 byte) signed integer representation:

- What is the binary representation for 47?
- What is the hex representation for 47?
- What is the binary representation for -47?
- What is the hex representation for -47

Exercise 1: Signed Integers

Assume an 8 bit (1 byte) signed integer representation:

- What is the binary representation for 47? **00101111**
- What is the hex representation for 47? **0x2F**
- What is the binary representation for -47? **11010001**
- What is the hex representation for -47 **0xD1**

Casting between Numeric Types

- Casting from shorter to longer types preserves the value
- Casting from longer to shorter types evaluates to $U2T_k(x \bmod 2^k)$
- 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
	-6	
		101010
(unsigned int) x		
(int) sy		
TMax		
TMin		

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
	-6	**111010**
	-22	101010
(unsigned int) x	**47**	**101111**
(int) sy	**-3**	**111101**
TMax	**31**	**011111**
TMin	**-32**	**100000**

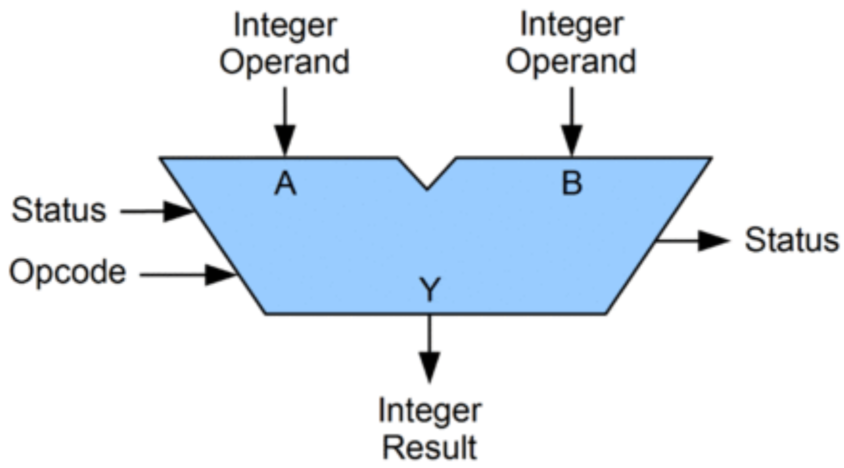
When to Use Unsigned

- 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)

- 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

- Assume signed one-byte integer values

  - `~0xe2`
  - `!0xe2`

  - `0x78 &  0x55`
  - `0x78 |  0x55`
  - `0x78 && 0x55`
  - `0x78 || 0x55`

  - `0x96 << 4`
  - `0x96 << 2`
  - `0x96 >> 4`
  - `0x96 >> 2`

Exercise 3: Bitwise vs Logical Operations

- Assume signed char data type (one byte)
  - `~0xe2` = `~11100010 = 00011101 = 0x1d`
  - `!0xe2` = `!11100010 = 00000000 = 0x00`

  - `0x78 & 0x55` = `01111000 & 01010101 = 01010000 = 0x50`
  - `0x78 | 0x55` = `01111000 | 01010101 = 01111101 = 0x7d`
  - `0x78 && 0x55` = `01111000 && 01010101 = 00000001 = 0x01`
  - `0x78 || 0x55` = `01111000 || 01010101 = 00000001 = 0x01`

  - `0x96 << 4` = `10010110 << 4 = 01100000 = 0x60`
  - `0x96 << 2` = `10010110 << 2 = 01011000 = 0x58`
  - `0x96 >> 4` = `10010110 >> 4 = 00001001 = 0x09`
  - `0x96 >> 2` = `10010110 >> 2 = 00100101 = 0x25`

Addition Example

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

$$
\begin{array}{r}
1\ \ \ 1\ \ \ \ \\
0\,1\,0\,1 \\
+\ 1\,1\,0\,1 \\
\hline
0\,0\,1\,0
\end{array}
\quad = 2 \text{ (Base-10)}
$$

Exactly the same as unsigned numbers!

… but with different error cases

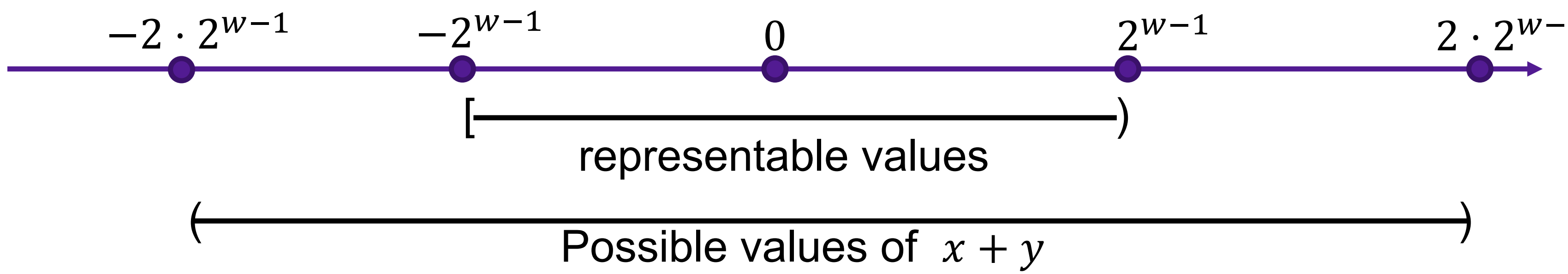
Addition/Subtraction with Overflow

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

$$
\begin{array}{r}
{}^{1\,1\,1} \\
0101 \\
+\,0011 \\
\hline
1000
\end{array}
\quad = -8 \text{ (Base-10)}
$$

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 or not 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 or not 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
 + 00110
 --------
     0110   = 6 (Base-10)
```

Exactly like unsigned multiplication!

… except with different error cases

Multiplication Example

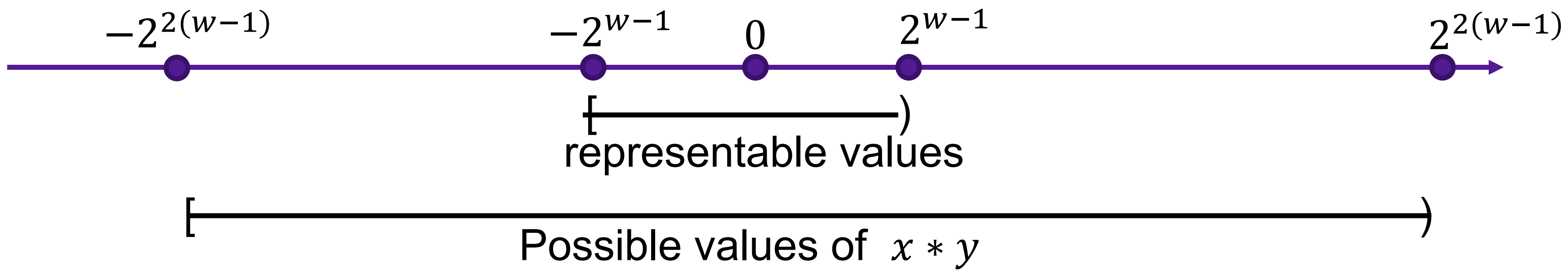
- Compute 5 x 2 assuming all ints are stored as four-bit signed values

$$
\begin{array}{r}
0101 \\
\times\ 0010 \\
\hline
0000 \\
+\ 01010 \\
\hline
1010
\end{array}
$$

= -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		

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`	**`100`**	**yes**
`010`	`011`	**`110`**	**yes**
`111`	`010`	**`110`**	**yes**

Multiplying with Shifts

- Multiplication is slow
- Bit shifting is kind of like multiplication, and is often faster
  - x * 8 = x << 3
  - x * 10 = x << 3 + x << 1

- Most compilers will automatically replace multiplications with shifts where possible

Signed Division by a Power of 2

- `x >> k` computes `x / 2`^{k} (rounded towards $-\infty$)
  - `-12 >> 2 = 11110100 >> 2 = 11111101 = -3`
- C on Intel processors rounds towards 0
  - `-11 >> 2 == -3`, but `-11/4 == -2`
- Solution: If x < 0, add 2^k-1 before shifting

```
if (x < 0){
   x += (1 << k) - 1;
}
return x >> k;
```

Exercise 6: Feedback

1. Rate how well you think this recorded lecture worked
   1. Better than an in-person class
   2. About as well as an in-person class
   3. Less well than an in-person class, but you still learned something
   4. Total waste of time, you didn't learn anything
2. How much time did you spend on this video lecture (including time spent on exercises)?
3. Do you have any comments or feedback?