Lecture 2: Representing Integers

CS 105

Fall 2026

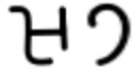
Review: Bits Require Interpretation

10001100 00001100 10101100 00000000

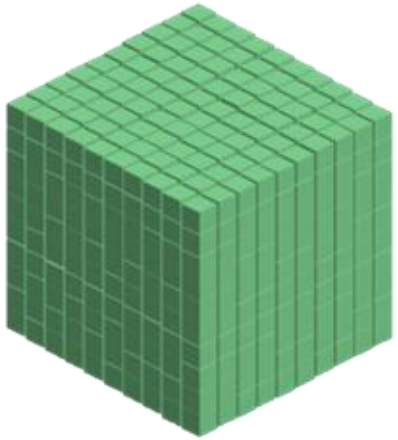
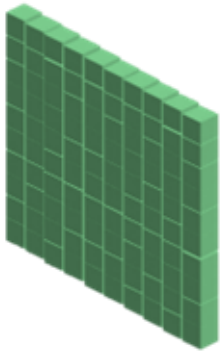
might be interpreted as

- The integer 3,485,745
- A floating point number close to 4.884569×10^{-39}
- The string “105”
- A portion of an image or video
- An address in memory

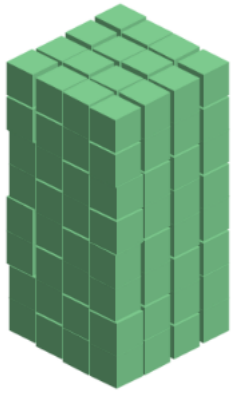
Representing Integers

- Arabic Numerals: 47
- Roman Numerals: XLVII
- Brahmi Numerals: 
- Tally Marks: 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 𝍸 II

Base-10 Integers

1000 (10^3)	100 (10^2)	10 (10^1)	1 (10^0)
			
0	0	0	5
0	0	4	7
1	8	8	7

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

Binary Numbers

- Decimal (Base-10):

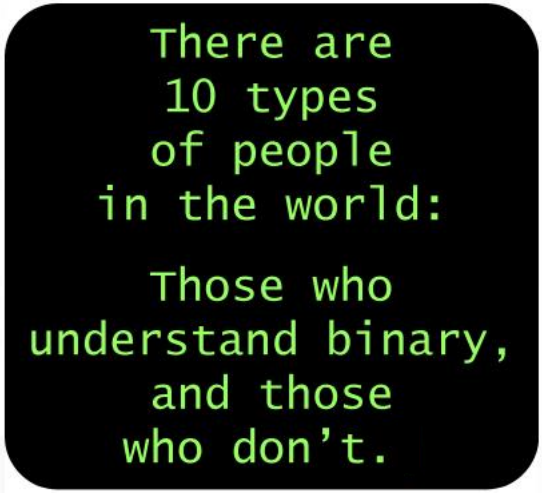
$$1011$$

$$= 1 \cdot 10^3 + 0 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0$$
$$= 1011$$

- Binary (Base-2):

$$1011$$

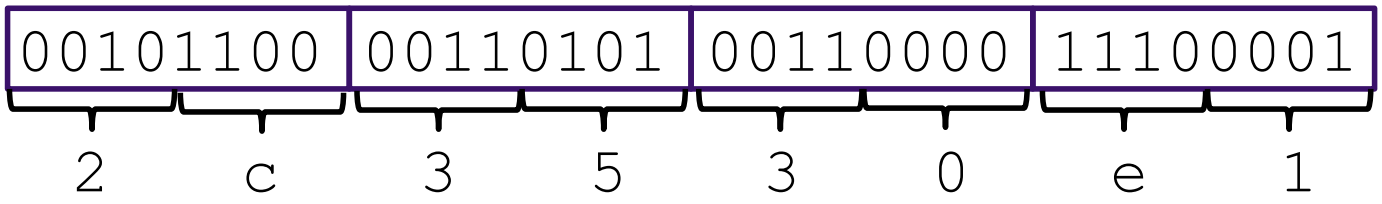
$$= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$
$$= 11$$



Exercise 1: Binary Numbers

- Consider the following four-bit binary values. What is the (base-10) integer interpretation of these values?
  1. `0001`
  2. `1010`
  3. `0111`
  4. `1111`

Hexidecimal Numbers



0x2c3530e1

Dec	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f

Exercise 2: Hexidecimal Numbers

- Consider the following hexidecimal values. What is the representation of each value in binary?
  1. `0x0a`
  2. `0x11`
  3. `0x2f`

Addition Example

- Compute 5 + 6 assuming all ints are stored as eight-bit (1 byte) unsigned values

```
      1
  00000101
+ 00000110
----------
  00001011    = 11 (Base-10)
```

Like you learned in grade school, only binary!

… and with a finite number of digits

Addition Example with Overflow

- Compute 200 + 100 assuming all ints are stored as eight-bit (1 byte) unsigned values

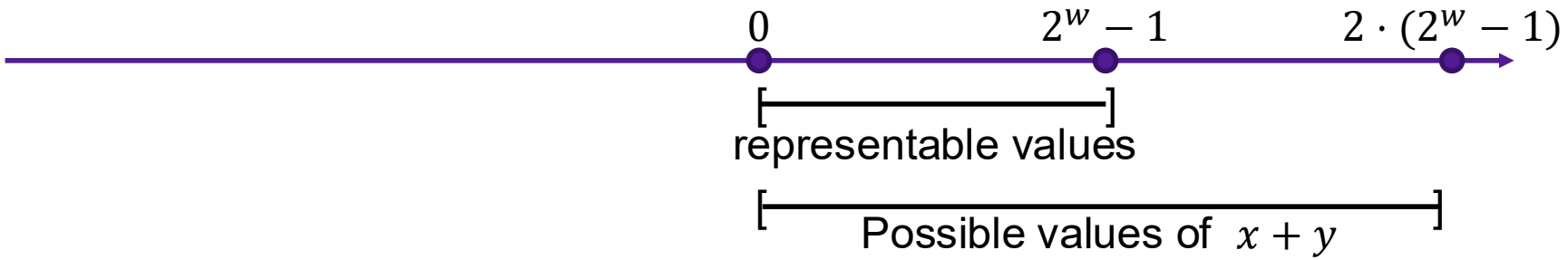
```
  1 1
  1 1 0 0 1 0 0 0
+ 0 1 1 0 0 1 0 0
-----------------
  0 0 1 0 1 1 0 0     = 44 (Base-10)
```

Like you learned in grade school, only binary!

… and with a finite number of digits

Error Cases (Unsigned Ints)

- Assume w-bit unsigned values



- $x + y = \begin{cases} x + y & (\text{normal}) \\ x + y - 2^w & (\text{overflow}) \end{cases}$ } modular addition

- overflow has occurred iff $x + y < x$

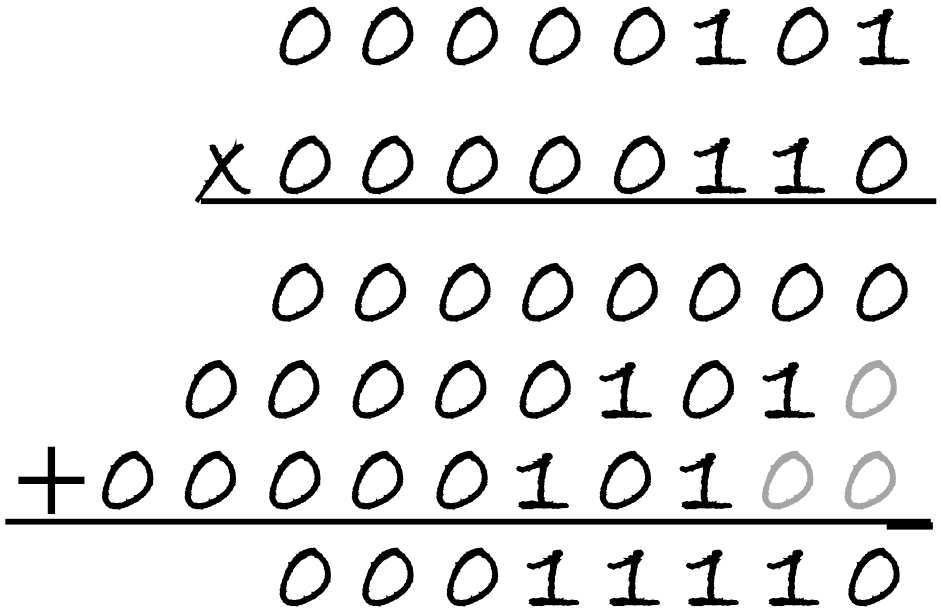
Exercise 3: Binary Addition

- Given the following 5-bit unsigned 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 5 x 6 assuming all ints are stored as eight-bit (1 byte) unsigned values



= 30 (Base-10)

Like you learned in grade school, only binary!

… and with a finite number of digits

Multiplication Example

- Compute 200 x 3 assuming all ints are stored as eight-bit (1 byte) unsigned values

```
  11001000
x 00000011
----------
  11001000
+ 11001000 0
----------
 100101100 0
```

= 88 (Base-10)

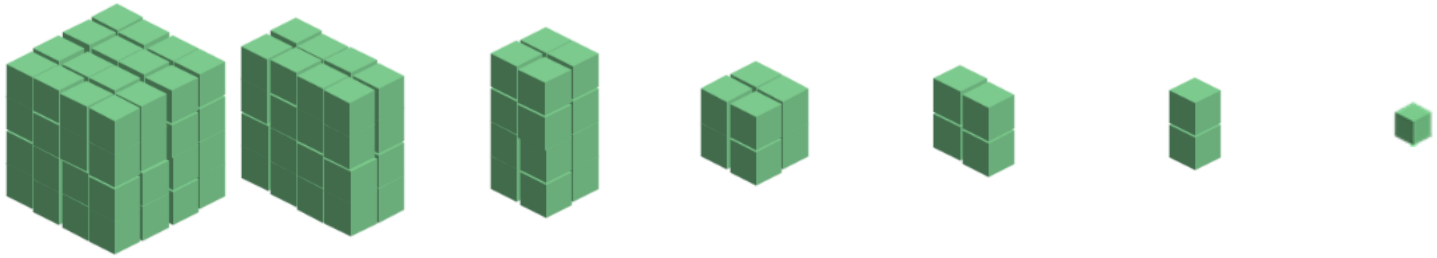
Like you learned in grade school, only binary!

… and with a finite number of digits

What about positive/negative integers?

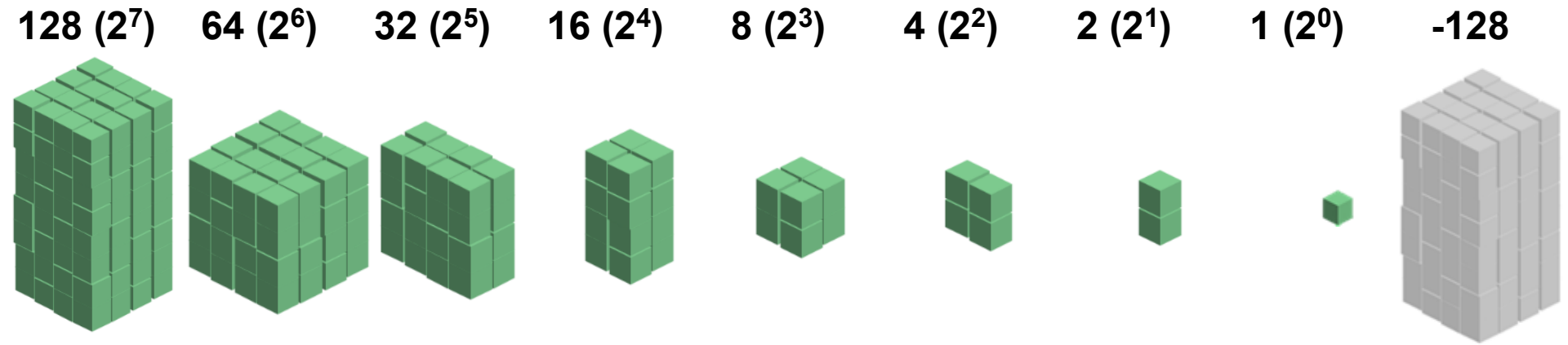
Representing Signed Integers

- 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

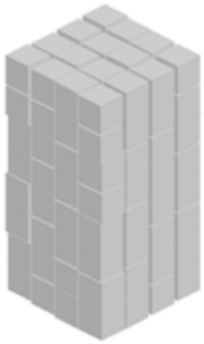
- Option 2: excess-K
  - Choose a positive K in the middle of the unsigned range
  - $Signed(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i - 2^{w-1}$



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 4: (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

Addition Example

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

```
  1  1
  0101
+ 1101
------
  0010
```

= 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)
```

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
x 0010
------
  0000
+ 0101 0
--------
  1010
```

= -6 (Base-10)

Exercise 5: (Signed) Binary Multiplication

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

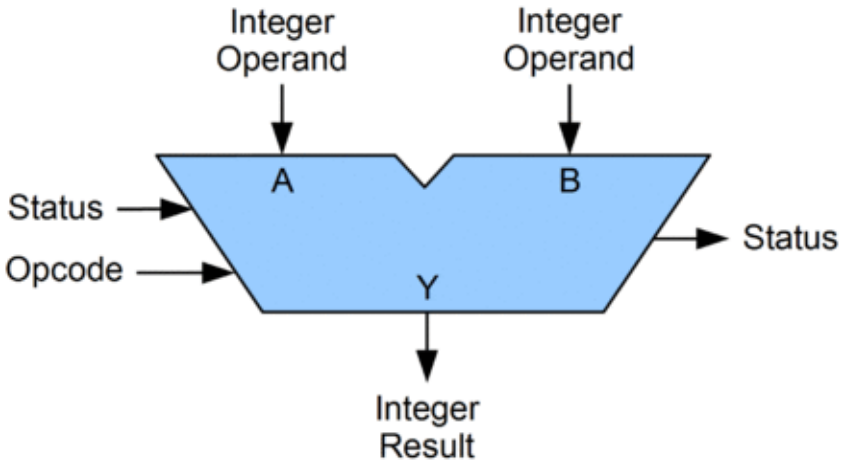
x	y	x*y	overflow?
100	101		
010	011		
111	010		

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!

Arithmetic Logic Unit (ALU)

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



Review: Bitwise vs Logical Operations

- 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 unsigned integers, right shift is logical (fills with zeros)

Exercise 6: 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

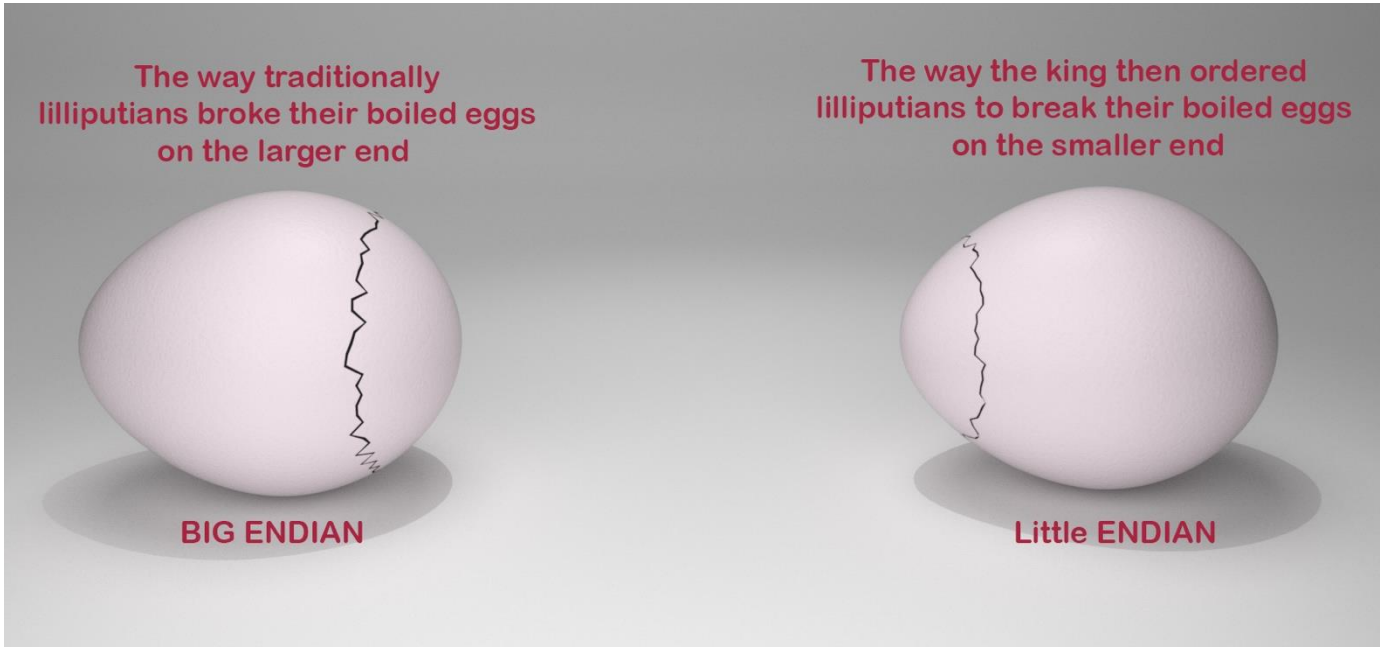
Multiplying with Shifts

- 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

Endianness

47 vs 74



Endianness

- **Big Endian:** low-order bits go on the right (47)
  - I tend to think in big endian numbers, so examples in class will generally use this representation
  - Networks generally use big endian (aka network byte order)
- **Little Endian:** low-order bits go on the left (74)
  - Most modern machines use this representation

- I will try to always be clear about whether I'm using a big endian or little endian representation
- When in doubt, ask!