

Lecture 2: Representing Integers

CS 105

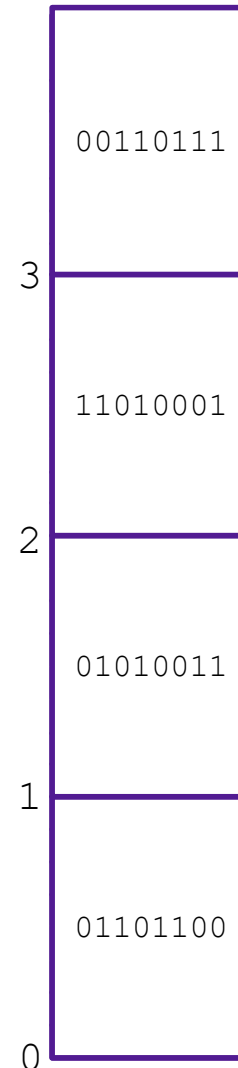
Spring 2026

Review: Abstraction



Review: Memory

- **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




Review: Bits Require Interpretation

10001100 00001100 10101100 00000000

might be interpreted as

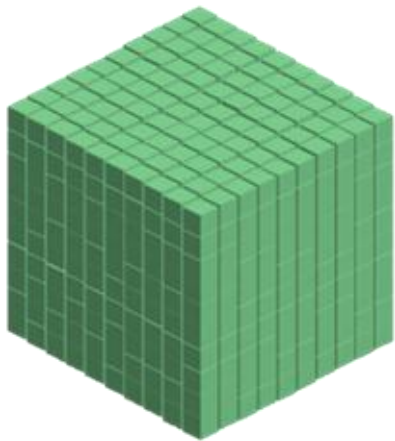
- The integer 3,485,745
- A floating point number close to 4.884569×10^{-39}
- The string "105"
- A portion of an image or video
- An address in memory

Representing Integers

- Arabic Numerals: 47
- Roman Numerals: XLVII
- Brahmi Numerals: 𑌔𑌗
- Tally Marks: 

Base-10 Integers

1000 (10^3)

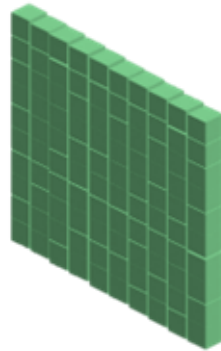


0

0

1

100 (10^2)



0

0

8

10 (10^1)



0

4

8

1 (10^0)

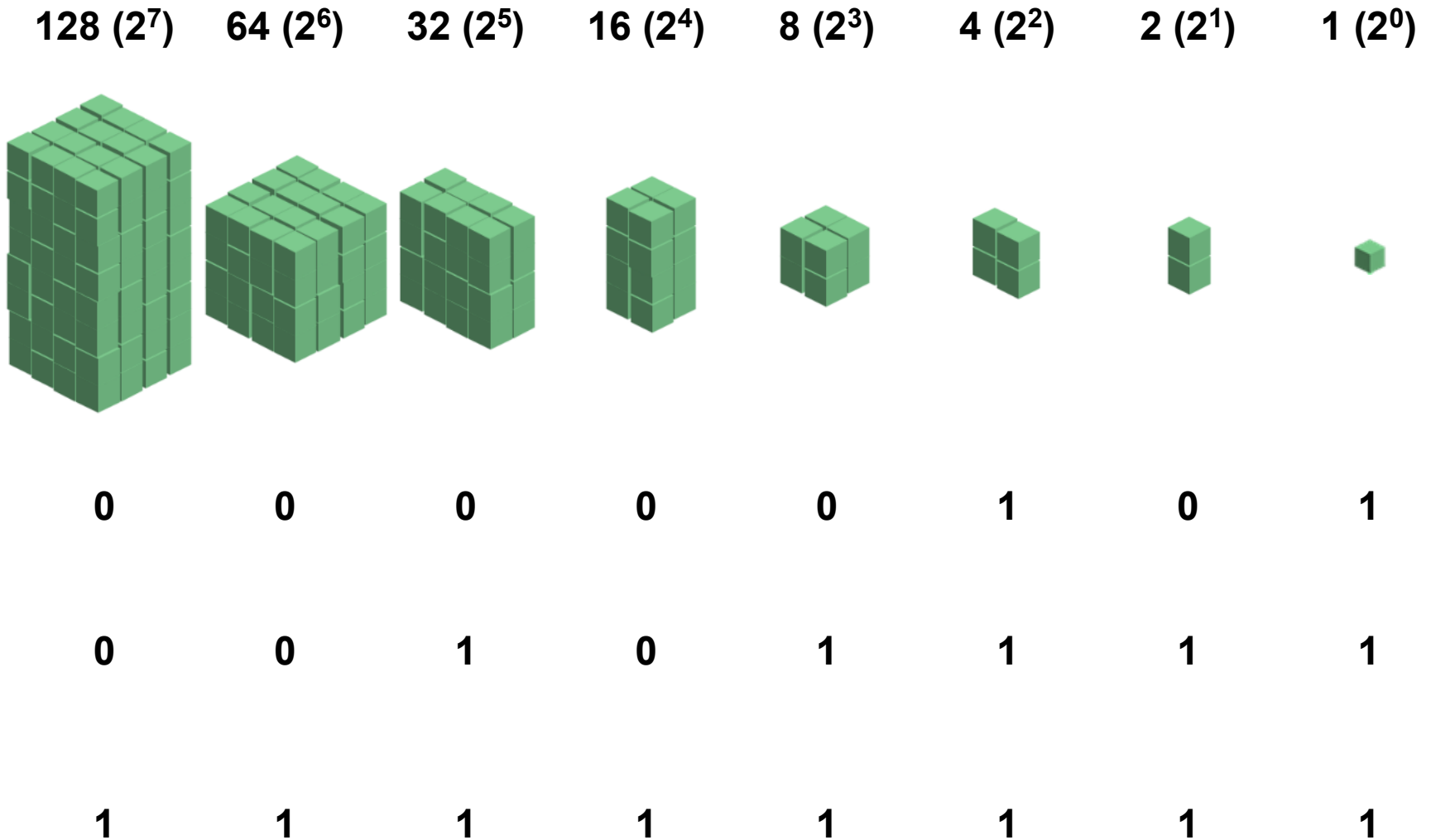


5

7

7

Base-2 Integers (aka Binary Numbers)



Binary Numbers

- Decimal (Base-10):

1011

$$\begin{aligned} &= 1 \cdot 10^3 + 0 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0 \\ &= 1011 \end{aligned}$$

- Binary (Base-2):

1011

$$\begin{aligned} &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 11 \end{aligned}$$

There are
10 types
of people
in the world:

Those who
understand binary,
and those
who don't.

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

Exercise 2: Binary Number Range

- What are the max number and min number that can be represented by a w -bit binary number?
 1. $w = 3$
 2. $w = 4$
 3. $w = 8$

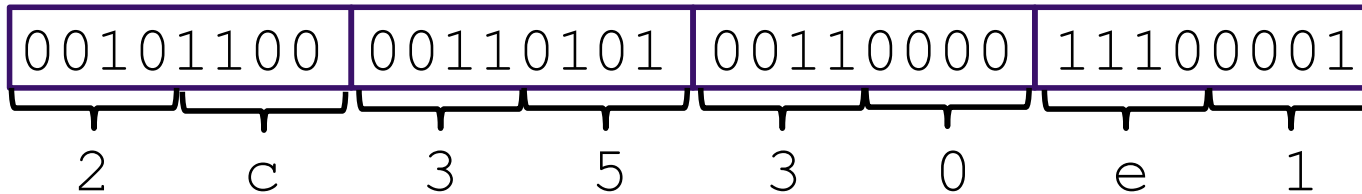
Unsigned Integers in C

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

ASCII characters

Char	Dec	Binary	Char	Dec	Binary	Char	Dec	Binary	Char	Dec	Binary	Char	Dec	Bin
!	33	00100001	1	49	00110001	A	65	01000001	Q	81	01010001	a	97	0110
"	34	00100010	2	50	00110010	B	66	01000010	R	82	01010010	b	98	0110
#	35	00100011	3	51	00110011	C	67	01000011	S	83	01010011	c	99	0110
\$	36	00100100	4	52	00110100	D	68	01000100	T	84	01010100	d	100	0110
%	37	00100101	5	53	00110101	E	69	01000101	U	85	01010101	e	101	0110
&	38	00100110	6	54	00110110	F	70	01000110	V	86	01010110	f	102	0110
'	39	00100111	7	55	00110111	G	71	01000111	W	87	01010111	g	103	0110
(40	00101000	8	56	00111000	H	72	01001000	X	88	01011000	h	104	0110
)	41	00101001	9	57	00111001	I	73	01001001	Y	89	01011001	i	105	0110
*	42	00101010	:	58	00111010	J	74	01001010	Z	90	01011010	j	106	0110
+	43	00101011	;	59	00111011	K	75	01001011	[91	01011011	k	107	0110
,	44	00101100	<	60	00111100	L	76	01001100	\	92	01011100	l	108	0110
-	45	00101101	=	61	00111101	M	77	01001101]	93	01011101	m	109	0110
.	46	00101110	>	62	00111110	N	78	01001110	^	94	01011110	n	110	0110
/	47	00101111	?	63	00111111	O	79	01001111	_	95	01011111	o	111	0110
0	48	00110000	@	64	01000000	P	80	01010000	`	96	01100000	p	112	0110

Hexadecimal 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

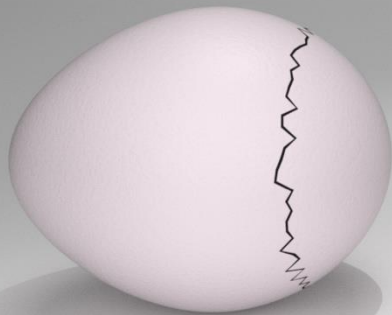
Exercise 3: Hexidecimal Numbers

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

Endianness

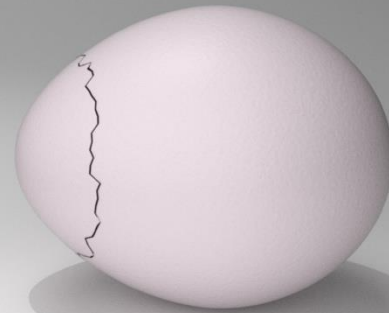
47 vs 74

The way traditionally
lilliputians broke their boiled eggs
on the larger end



BIG ENDIAN

The way the king then ordered
lilliputians to break their boiled eggs
on the smaller end



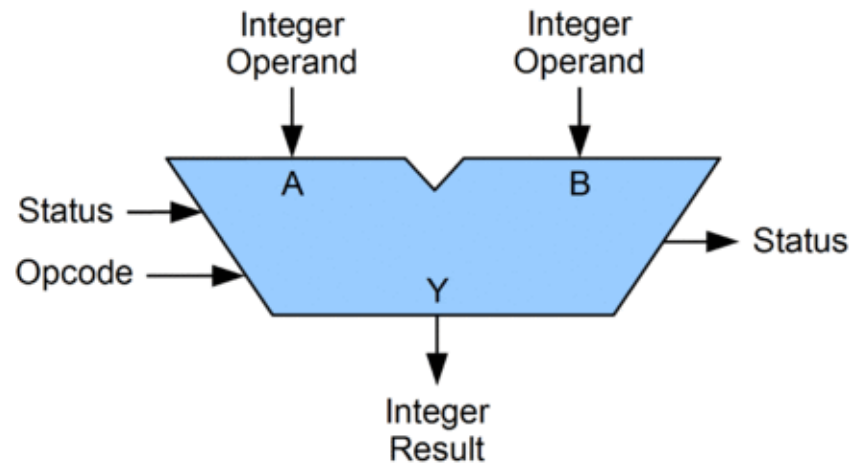
Little ENDIAN

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!

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

Exercise 4: Bitwise vs Logical Operations

Assume unsigned char data type (one byte). What do each of the following expressions evaluate to (interpreted as unsigned integers and expressed base-10)?

1. `~150`
2. `!150`

3. `150 & 51`
4. `150 | 51`
5. `150 && 51`
6. `150 || 51`

7. `150 << 2`
8. `150 >> 2`

Addition Example

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

$$\begin{array}{r} 1 \\ 00000101 \\ + 00000110 \\ \hline 00001011 \end{array} = 11 \text{ (Base-10)}$$

Like you learned in grade school, only binary!
... and with a finite number of digits

Addition Example with Overflow

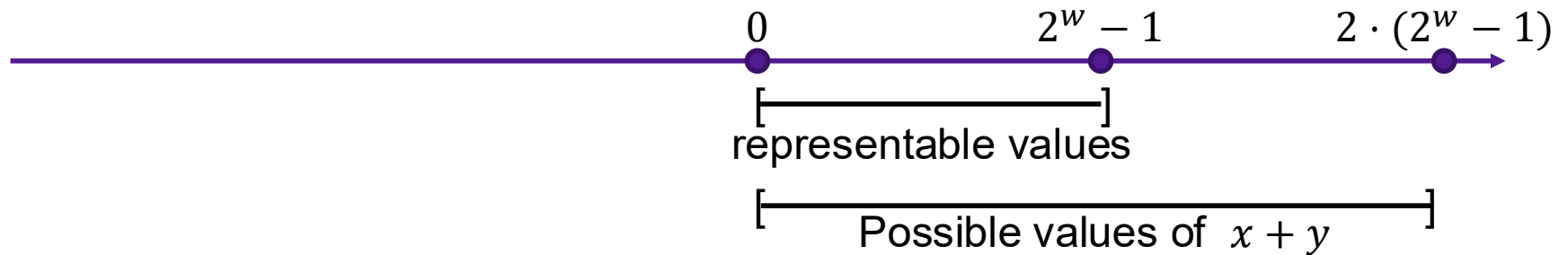
- Compute $200 + 100$ assuming all ints are stored as eight-bit (1 byte) unsigned values

$$\begin{array}{r} 11 \\ 11001000 \\ + 01100100 \\ \hline 00101100 \end{array} = 44 \text{ (Base-10)}$$

Like you learned in grade school, only binary!
... and with a finite number of digits

Error Cases (Unsigned Ints)

- Assume w -bit unsigned values



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

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

Exercise 5: 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 5×6 assuming all ints are stored as eight-bit (1 byte) unsigned values

$$\begin{array}{r} 00000101 \\ \times 00000110 \\ \hline 00000000 \\ 000001010 \\ + 0000010100 \\ \hline 00011110 \end{array} = 30 \text{ (Base-10)}$$

Like you learned in grade school, only binary!
... and with a finite number of digits

Multiplication Example

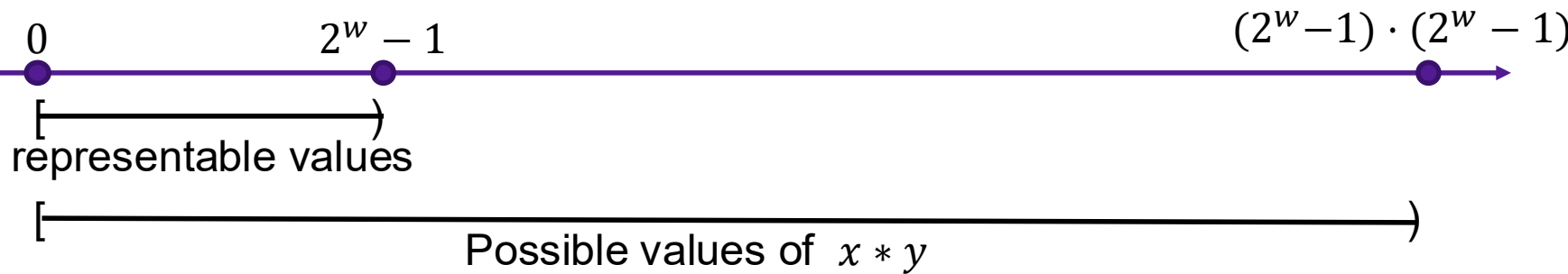
- Compute 200×3 assuming all ints are stored as eight-bit (1 byte) unsigned values

$$\begin{array}{r} 11001000 \\ \times 00000011 \\ \hline 11001000 \\ + 110010000 \\ \hline 1001011000 \end{array} = 88 \text{ (Base-10)}$$

Like you learned in grade school, only binary!
... and with a finite number of digits

Error Cases

- Assume w -bit unsigned values



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

Exercise 6: Binary Multiplication

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

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

Multiplying with Shifts

- Multiplication is slow
- Bit shifting is kind of like multiplication, and is often faster
 - $x * 8 = x \ll 3$
 - $x * 10 = x \ll 3 + x \ll 1$
- Most compilers will automatically replace multiplications with shifts where possible