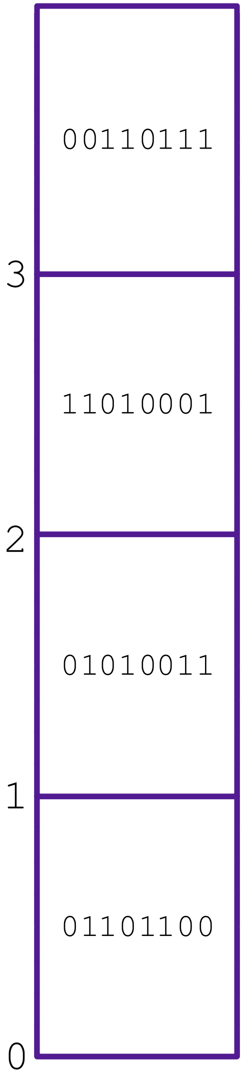
Lecture 2: Representing Integers

CS 105

Fall 2024

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



00110111

3

11010001

2

01010011

1

01101100

0

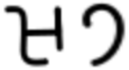
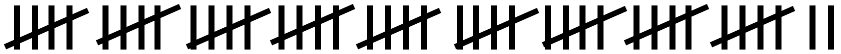
Review: Bits Require Interpretation

10001100 00001100 10101100 00000000

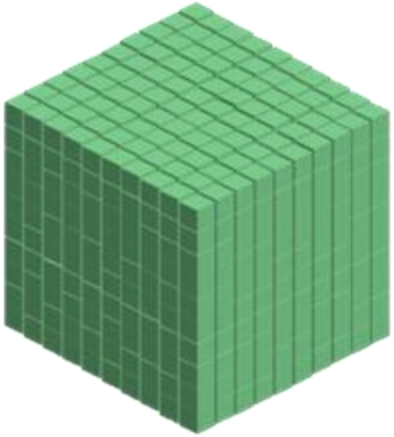
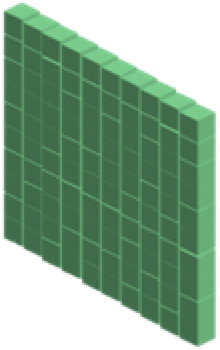
might be interpreted as

- 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

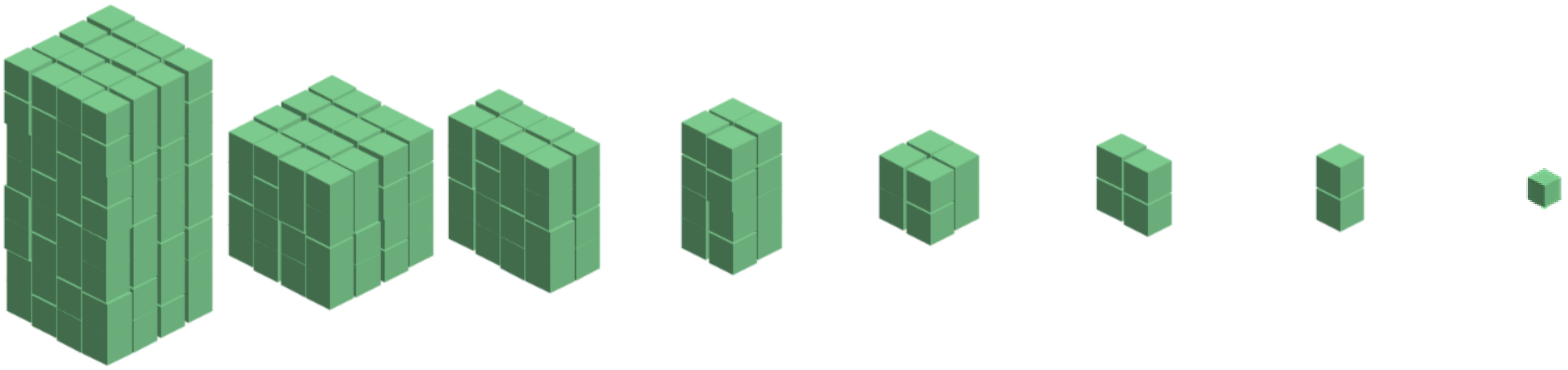
Representing Integers

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

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)



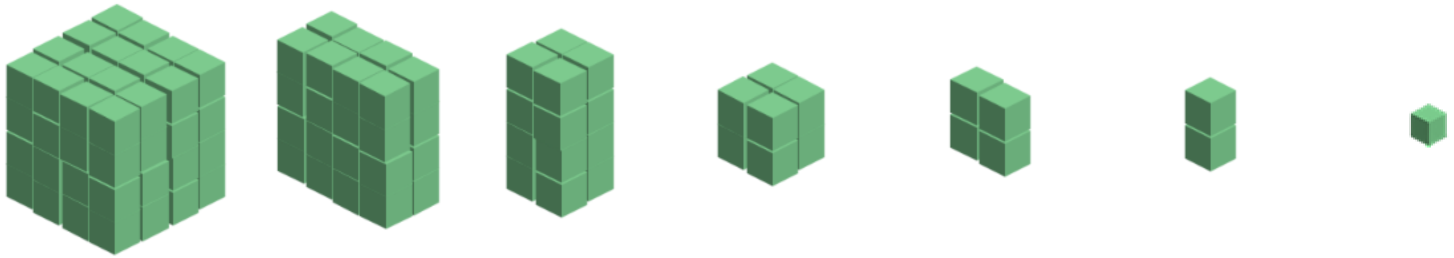
128 (2^7)	64 (2^6)	32 (2^5)	16 (2^4)	8 (2^3)	4 (2^2)	2 (2^1)	1 (2^0)
0	0	0	0	0	1	0	1
0	0	1	0	1	1	1	1
1	1	1	1	1	1	1	1

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`

Representing Signed Integers

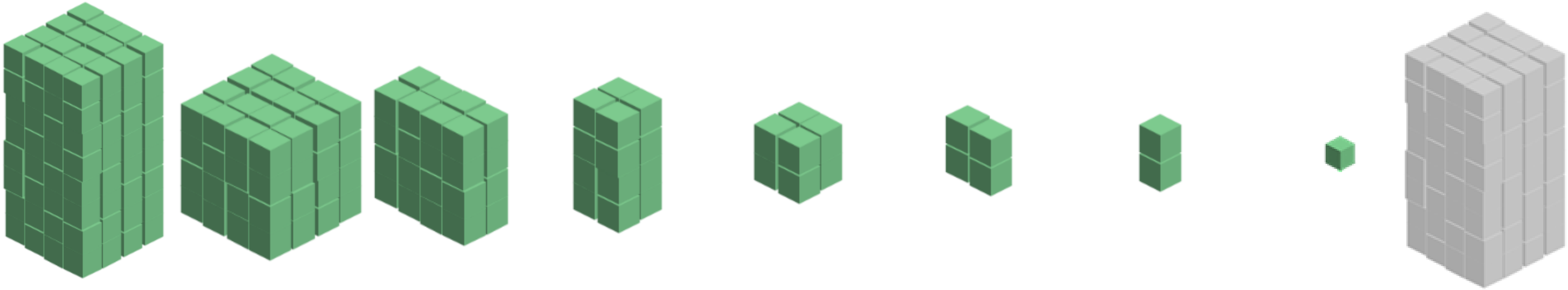
- Option 1: sign-magnitude
  - One bit for sign; interpret rest as magnitude
  - $Signed(x) = (-1)^{x_{w-1}} \cdot \sum_{i=0}^{w-2} x_i \cdot 2^i$



+/-	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

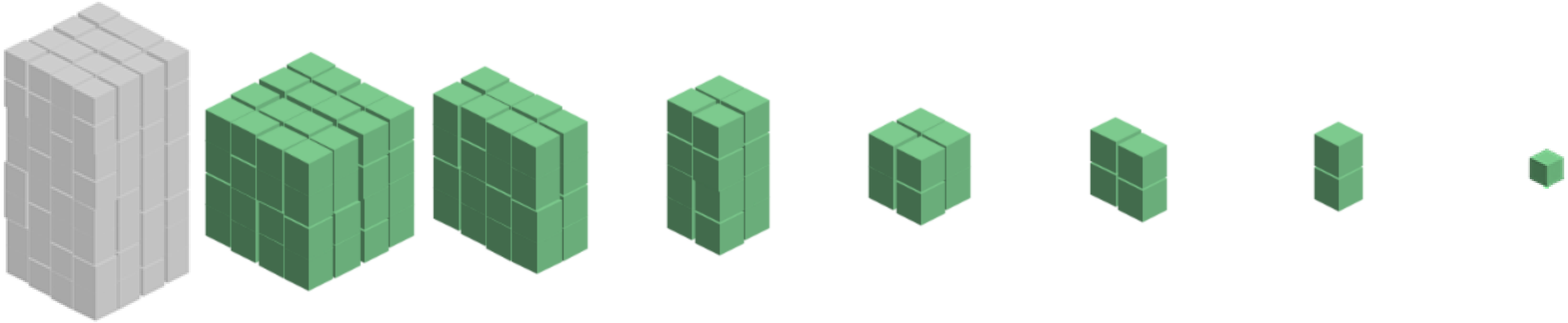
- Option 2: excess-K
  - Choose a positive K in the middle of the unsigned range
  - $Signed(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i \ - 2^{w-1}$



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

Exercise 2: (Signed) Binary Numbers

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

Signed Integer Trivia

Base-10	unsigned	signed
7	`111`	
6	`110`	
5	`101`	
4	`100`	
3	`011`	`011`
2	`010`	`010`
1	`001`	`001`
0	`000`	`000`
-1		`111`
-2		`110`
-3		`101`
-4		`100`

- For signed ints:
  - high-order (left-most) bit is 0 for pos values, 1 for neg
  - 000…0 is 0
  - 111…1 is -1
  - same representation as unsigned for numbers that can be represented with both
  - `~x+1 == -1*x`

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

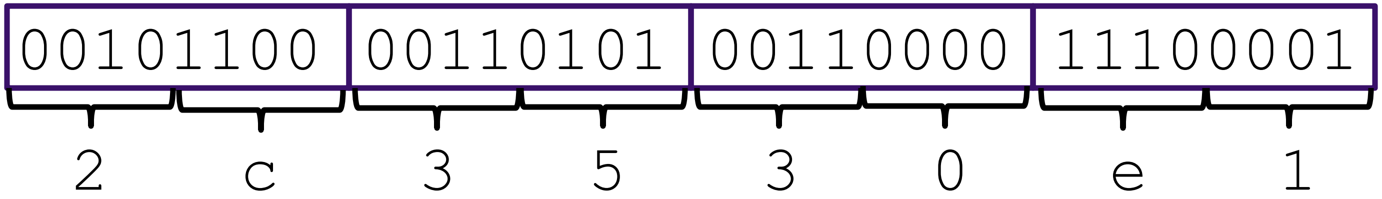
ASCII characters

Char	Dec	Binary	Char	Dec	Binary	Char	Dec	Binary	Char	Dec	Binary	Char	Dec	Bi
!	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	0111

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!

Hexidecimal Numbers



00101100 00110101 00110000 11100001

2 c 3 5 3 0 e 1

0x2c3530e1

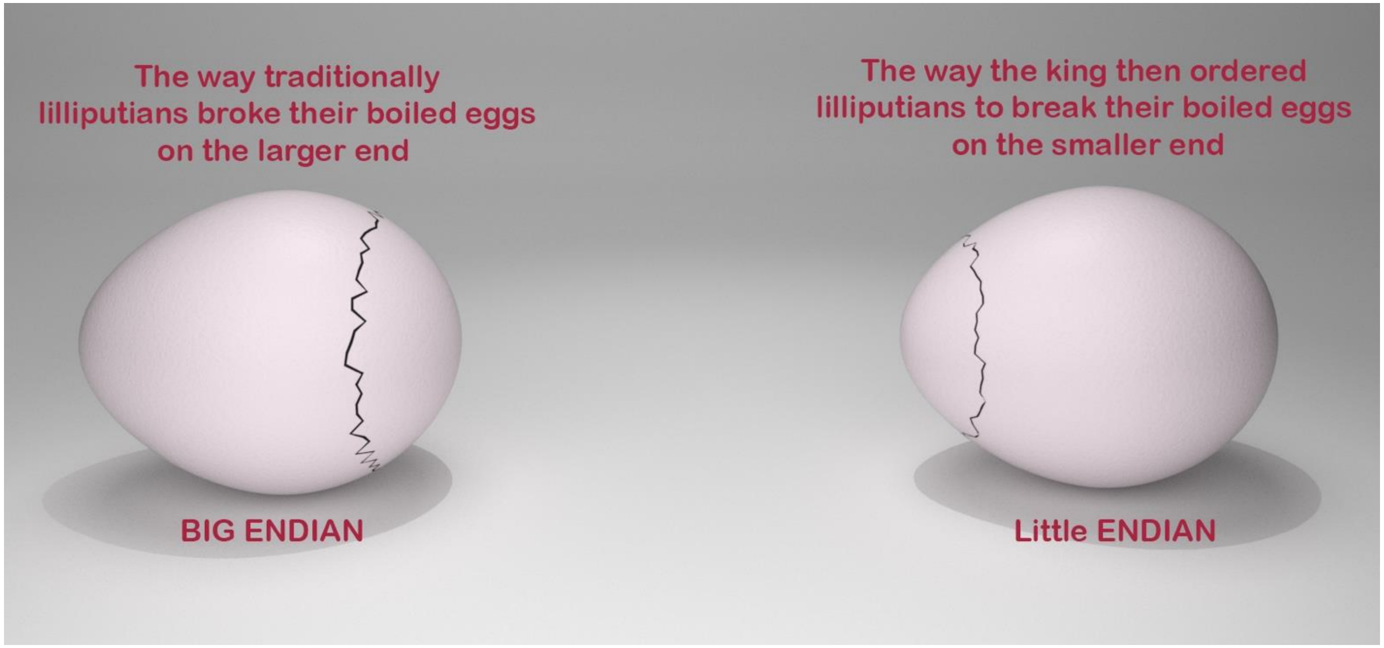
Dec	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	a
11	b
12	c
13	d
14	e
15	f

Exercise 3: Hexidecimal Numbers

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

Endianness

47 vs 74



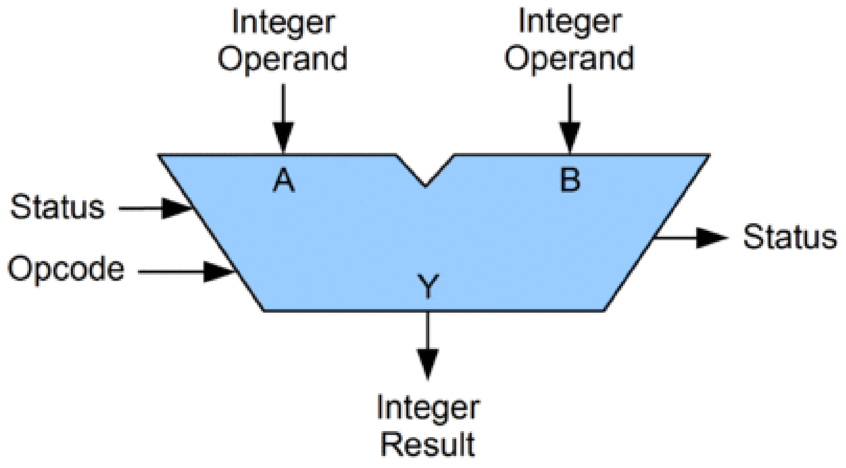
Endianness

- **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!

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)
  - For signed integers, right shift is arithmetic (fills with high-order bit)

Exercise 4: 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/division, and is often faster
  - x * 8 = x << 3
  - x * 10 = x << 3 + x << 1

- Most compilers will automatically replace multiplications with shifts where possible

Arithmetic Operations (in C)

- Basic Math Operators `+, -, *, /`
  - division is integer division (rounds towards zero)

- Modulus Operator %

- Increment/Decrement operators **`++, --`**
  - x++ is the same as x = x+1 or x += 1
  - x-- is the same as x = x-1 or x -= 1

Addition Example

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

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

Like you learned in grade school, only binary!

… and with a finite number of digits

Addition/Subtraction with Overflow

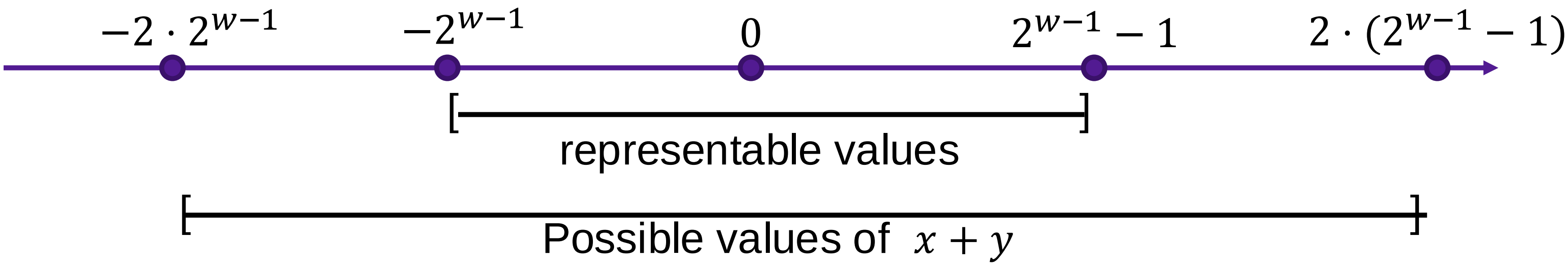
- Compute 5 + 6 assuming all ints are stored as four-bit signed values

$$\begin{array}{ccccc} & 1 & & & \\ & 0 & 1 & 0 & 1 \\ + & 0 & 1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 1 \end{array}$$

= -5 (Base-10)

Error Cases

- Assume w-bit signed values



$-2 \cdot 2^{w-1}$

-2^{w-1}

0

$2^{w-1} - 1$

$2 \cdot (2^{w-1} - 1)$

representable values

Possible values of $x + y$

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

- 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 5: 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)
```

Like you learned in grade school, only binary!

… and with a finite number of digits

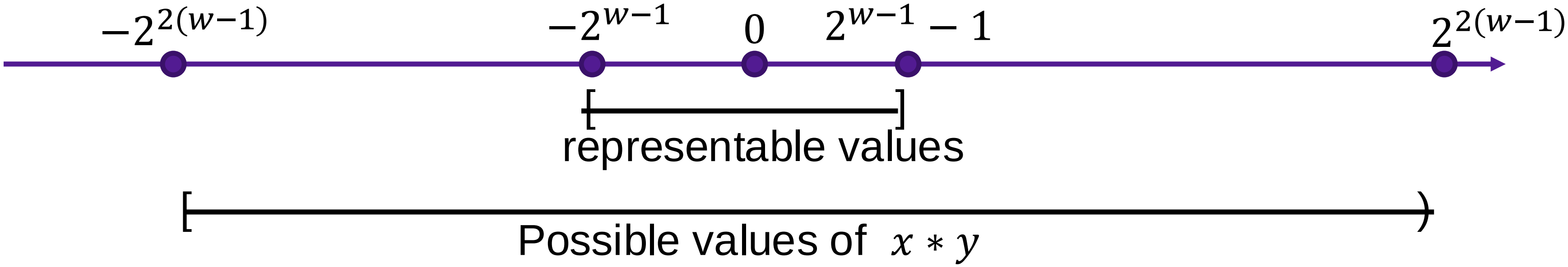
Multiplication Example

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

$$\begin{array}{r} 0101 \\ \times\,0010 \\ \hline 0000 \\ +\,01010 \\ \hline 1010 \end{array} \quad = -6 \text{ (Base-10)}$$

Error Cases

- Assume w-bit unsigned values



$-2^{2(w-1)}$ $\quad -2^{w-1}$ $\quad 0$ $\quad 2^{w-1}-1$ $\quad 2^{2(w-1)}$

representable values

Possible values of $x * y$

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

Exercise 6: 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		