

310-2202 โครงสร้างของระบบคอมพิวเตอร์ (Computer Organization)

Topic 2: Bit, Data Types, and Operations

Damrongrit Setsirichok

Topic

- How do we represent information in a computer?
- Integer Data Types
- 2' Complement Integers
- Conversion Between Binary and Decimal
- Operations on Bits: Arithmetic and Logical
- Other Representation

What kinds of information do we need to represent?

- **Kinds of Information**

- **Numbers** – natural number, integers, positive/negative integers, integers/decimals, real, complex, rational, irrational, signed, unsigned, floating point, ...
- **Text** – characters, strings, ...
- **Logical** – true, false
- **Images** – pixels, colors, shapes, ...
- **Sound** – sound of talk, sound of sing, ...
- **Video** – a series of images
- **Instructions** – plus(+), minus(-), times(*), divided by(/) , ...
- ...

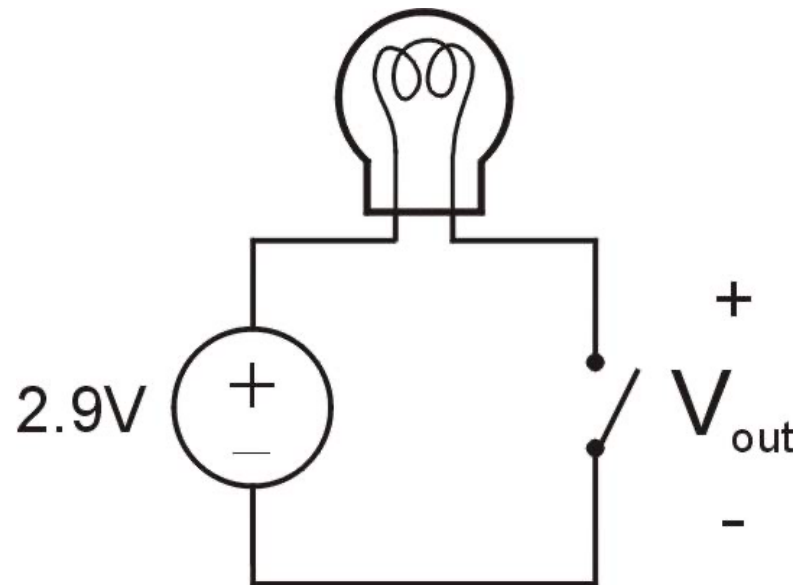
- Data type: representation and operations within the computer

We'll start with numbers...

How do we represent data in a computer?

- At the lowest level, a computer is an electronic machine.
 - works by controlling the flow of electrons
- Easy to recognize two conditions:
 - presence of a voltage – we'll call this state “1”
 - absence of a voltage – we'll call this state “0”
- Could base state on **value of voltage**, but control and detection circuits more complex.
 - compare turning on a light switch to measuring or regulating voltage
- We'll see examples of these circuits in the next chapter.

Simple Switch Circuit



Switch-based circuits can easily represent two states:
on/off, open/closed, voltage/no voltage.

- Switch open:
 - No current through circuit
 - Light is **off**
 - V_{out} is +2.9 V
- Switch closed:
 - Short circuit across switch
 - Current flows
 - Light is **on**
 - V_{out} is 0 V

Computer is a binary digital system.



Basic unit of information is the binary digit, or bit. Values with more than two states require multiple bits.

- A collection of **two** bits has **four** possible states:
00, 01, 10, 11
- A collection of **three** bits has **eight** possible states:
000, 001, 010, 011, 100, 101, 110, 111
- A collection of n bits has 2^n possible states.

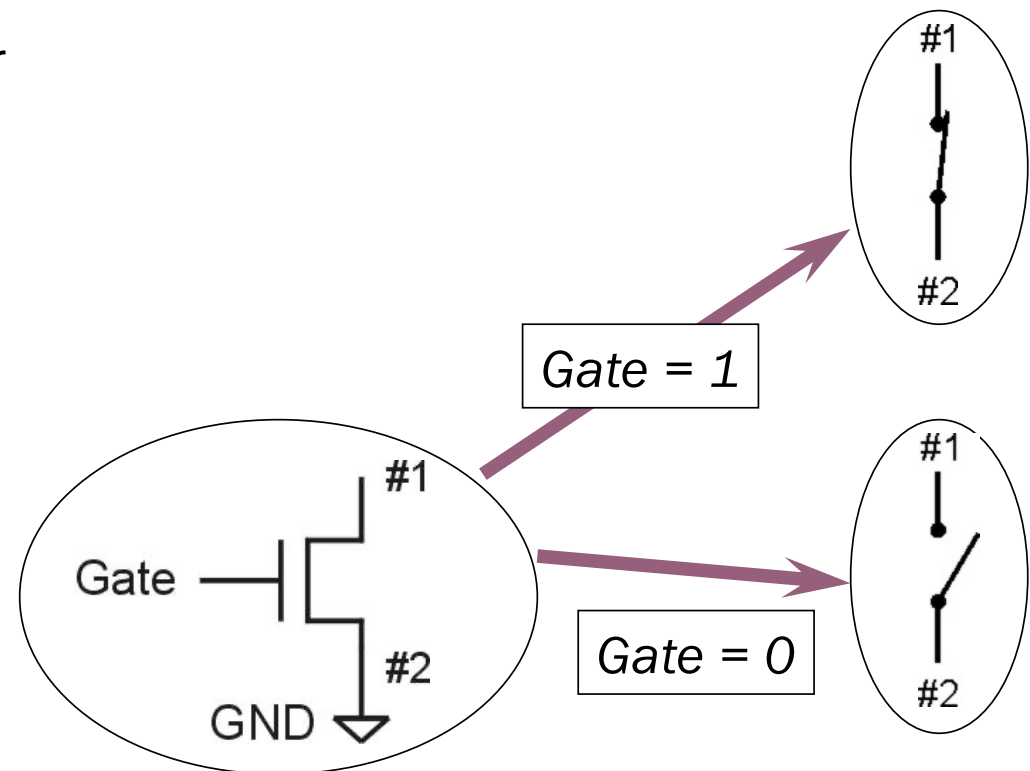
N-type MOS Transistor

- MOS: Metal–Oxide–Semiconductor

- two types: N-type and P-type

- **N-type**

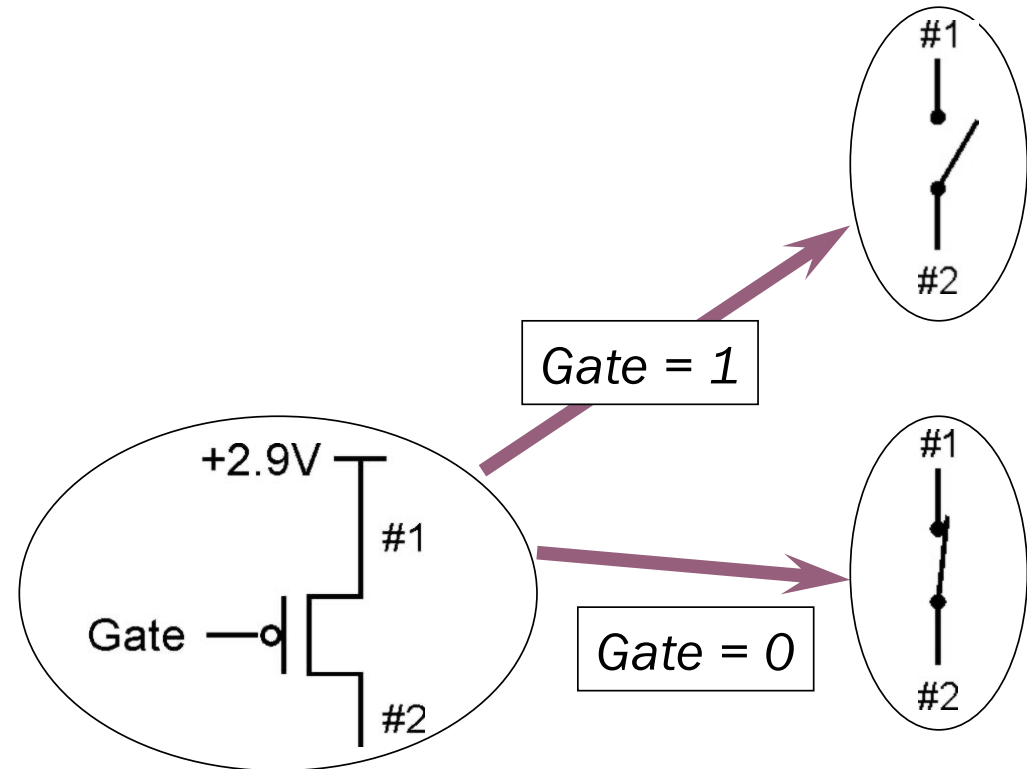
- when Gate has **positive** voltage, short circuit between #1 and #2 (switch **closed**)
 - when Gate has **zero** voltage, open circuit between #1 and #2 (switch **open**)



Terminal #2 must be connected to GND (0V).

P-type MOS Transistor

- **P-type** is complementary to N-type
 - when Gate has **positive** voltage, open circuit between #1 and #2 (switch **open**)
 - when Gate has **zero** voltage, short circuit between #1 and #2 (switch **closed**)



Terminal #1 must be connected to +2.9V.

Logic Gates

- Use switch behavior of MOS transistors to implement logical functions: AND, OR, NOT.
- Digital symbols:
 - recall that we assign a range of analog voltages to each digital (logic) symbol

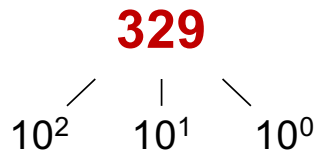


** Typical values for "1": +5V, +3.3V, +2.9V, +1.1V

Number Notation

- Weighted positional notation

- decimal numbers(denary numbers): “329”
- “3” is worth 300, because of its position (with place value 100),
- while “9” is only worth 9, because of its position (with place value 1)



$$3 \times 100 + 2 \times 10 + 9 \times 1 = 329$$

Denary numbers

Base is 10,

Place value according its position

Base-10 (Denary/ Decimal)

- $(4567)_{10}$

4567

Available digit	0, 1, 2, 3, 4, 5, 6, 7, 8, 9			
Place value	$10^3=1000$	$10^2=100$	$10^1=10$	$10^0=1$
Digit	4	5	6	7
Product of digit and place value	$4 \times 1000 = 4000$	$5 \times 100 = 500$	$6 \times 10 = 60$	$7 \times 1 = 7$

Base-8 (Octonary/ Octal)

- $(4567)_8$

4567

Available digit	0, 1, 2, 3, 4, 5, 6, 7			
Place value	$8^3=512$	$8^2=64$	$8^1=8$	$8^0=1$
Digit	4	5	6	7
Product of digit and place value	4×512	5×64	6×8	7×1

$$(4567)_8 = 4 \times 512 + 5 \times 64 + 6 \times 8 + 7 \times 1 = (2423)_{10}$$

$$(75)_8 = 7 \times 8 + 5 \times 1 = (61)_{10}$$

$$(31276)_8 = 3 \times 4096 + 1 \times 512 + 2 \times 64 + 7 \times 8 + 6 \times 1 = (12990)_{10}$$

Base-2 (Binary)

- $(101110)_2$

10 1110

Available digit	0, 1					
Place value	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
Digit	1	0	1	1	1	0
Product of digit and place value	32	0	8	4	2	0

$$(101110)_2 = 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = (46)_{10}$$

$$(11110100)_2 = 1 \times 128 + 1 \times 64 + 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 0 \times 1 = (244)_{10}$$

$$(4567)_{10} = (?)_2$$

Integer Data Types

Unsigned Integers

- An n -bit unsigned integer represents 2^n values: from 0 to 2^n-1 .

2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Unsigned Binary Arithmetic

- Base-2 addition – like base-10

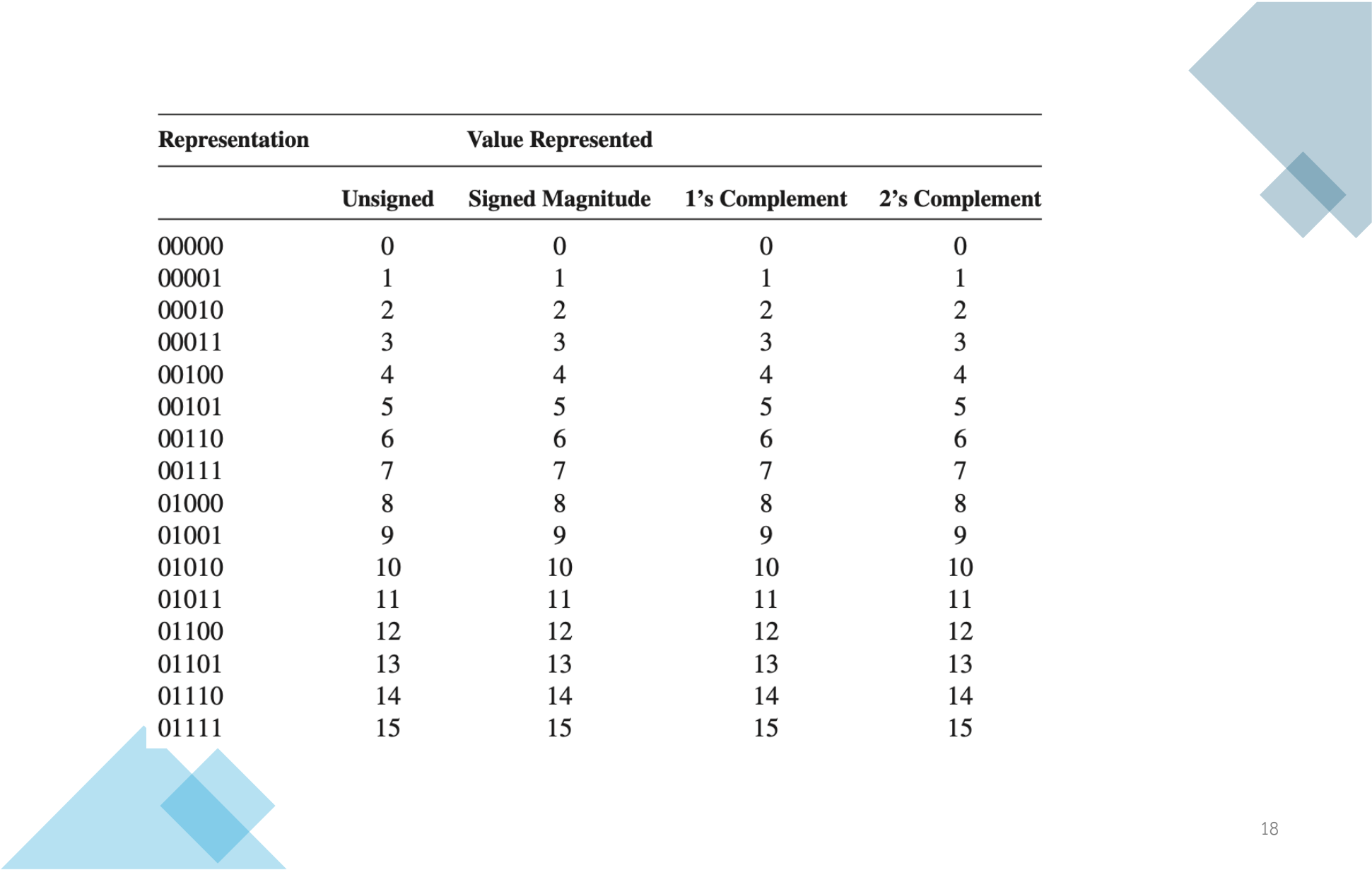
$$\begin{array}{r} 10010 \\ + \underline{1001} \\ 11011 \end{array}$$
$$\begin{array}{r} 10010 \\ + \underline{1011} \\ 11101 \end{array}$$
$$\begin{array}{r} 1111 \\ + \underline{1} \\ 10000 \end{array}$$
$$\begin{array}{r} 10111 \\ + \underline{111} \end{array}$$

How about **Subtraction, multiplication, division, ... ?**

Signed Integers

- With n bits, we have 2^n distinct values.
 - Positive integers \rightarrow 1 through $2^{n-1}-1$
 - Negative integers \rightarrow $-(2^{n-1}-1)$ through -1
 - that leaves two values: one for 0, and one extra
- Positive integers
 - just like unsigned – zero in Most Significant Bit: MSB
 $00101 = 5$
- Negative integers
 - sign-magnitude – set top bit to show negative, other bits are the same as unsigned
 $10101 = -5$
 - one's complement – flip every bit to represent negative
 $11010 = -5$
 - in either case, **MSB** indicates sign: 0=positive, 1=negative

MSB stands for “Most Significant Bit”, while LSB is “Least Significant Bit”



Representation	Value Represented			
	Unsigned	Signed Magnitude	1's Complement	2's Complement
00000	0	0	0	0
00001	1	1	1	1
00010	2	2	2	2
00011	3	3	3	3
00100	4	4	4	4
00101	5	5	5	5
00110	6	6	6	6
00111	7	7	7	7
01000	8	8	8	8
01001	9	9	9	9
01010	10	10	10	10
01011	11	11	11	11
01100	12	12	12	12
01101	13	13	13	13
01110	14	14	14	14
01111	15	15	15	15

Representation	Value Represented			
	Unsigned	Signed Magnitude	1's Complement	2's Complement
10000	16	−0	−15	−16
10001	17	−1	−14	−15
10010	18	−2	−13	−14
10011	19	−3	−12	−13
10100	20	−4	−11	−12
10101	21	−5	−10	−11
10110	22	−6	−9	−10
10111	23	−7	−8	−9
11000	24	−8	−7	−8
11001	25	−9	−6	−7
11010	26	−10	−5	−6
11011	27	−11	−4	−5
11100	28	−12	−3	−4
11101	29	−13	−2	−3
11110	30	−14	−1	−2
11111	31	−15	−0	−1

2' Complement Integers

Arithmetic in ALU



- For example, if the ALU processed five-bit input patterns, and the two inputs were **00110** and **00101**, the result (output of the ALU) would be **01011**. The addition is as follows:

$$\begin{array}{r} 00110 + \\ \underline{00101} \\ 01011 \end{array}$$

- if the inputs non-zero integers A and -A, the output of the ALU should be **00000**
→ To accomplish that, make it's **2's complement**

2's Complement Representation

- If number is positive or zero,
 - normal binary representation, zeroes in upper bit (s)
- If number is negative,
 - start with positive number
 - flip every bit (one's complement)
 - then add one

	00101	(5)			01001	(9)
	11010	(1's comp)			10110	(1's comp)
+	<u>1</u>			+	<u>1</u>	
	11011	(-5)			10111	(-9)

2's Complement

- Problems with sign-magnitude and 1's complement
 - two representations of zero (+0 and -0)
 - arithmetic circuits are complex

Example 2.1

What is the 2's complement representation for -13?

1. Let A be +13. Then the representation for A is 01101 since $13 = 8+4+1$.
2. The complement of A is 10010.
3. Adding 1 to 10010 gives us 10011, the 2's complement representation for -13.




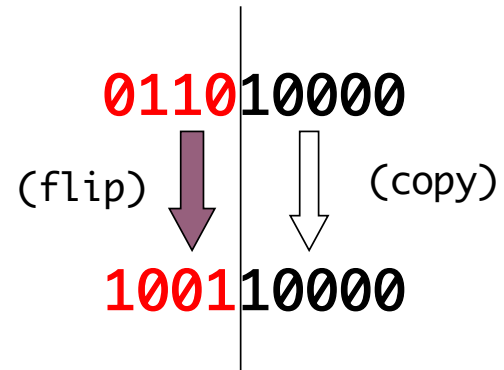
We can verify our result by adding the representations for A and $-A$,

$$\begin{array}{r} 01101 \\ 10011 \\ \hline 00000 \end{array}$$

2's Complement

- Some **shortcut** to take the 2's complement of a number
 - copy bits from right to left until (and including) the first “1”
 - flip remaining bits to the left


$$\begin{array}{r} 011010000 \\ 100101111 \text{ (1's comp)} \\ + \quad \quad \quad 1 \\ \hline 100110000 \end{array}$$


$$\begin{array}{c} 011010000 \\ \text{(flip)} \downarrow \quad \downarrow \text{(copy)} \\ 100110000 \end{array}$$

Conversion Between Binary and Decimal

Converting Binary (2's C) to Decimal

1. If leading bit is one, take 2's complement to get a positive number.
2. Add powers of 2 that have "1" in the corresponding bit positions.
3. If original number was negative, add a minus sign.

$$\begin{aligned} X &= 01101000_2 \\ &= 2^6 + 2^5 + 2^3 = 64 + 32 + 8 \\ &= 104_{10} \end{aligned}$$

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$$X = 00100111_2 = 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1 = 39_{10}$$

$$X = 11100110_2$$

$$-X = 00011010 = 2^4 + 2^3 + 2^1 = 16 + 8 + 2 = 26_{10}$$

$$X = -26_{10}$$

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Convert the 2's complement integer 11000111 to a decimal integer value.

1. Since the leading binary digit is a 1, the number is negative. We must first find the 2's complement representation of the positive number of the same magnitude. This is 00111001.
2. The magnitude can be represented as

$$0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

or,

$$32 + 16 + 8 + 1.$$

3. The decimal integer value corresponding to 11000111 is -57 .

Converting Decimal to Binary (2's C)

First Method: **Division**

1. Divide by two – remainder is **least significant bit: LSB**.
2. Keep dividing by two until answer is zero, writing remainders from **right to left**.
3. Append a zero as the **MSB**,
if original number negative, take two's complement.

$X = 104_{10}$	$104/2 = 52$	r 0	<i>bit 0</i>
	$52/2 = 26$	r 0	<i>bit 1</i>
	$26/2 = 13$	r 0	<i>bit 2</i>
	$13/2 = 6$	r 1	<i>bit 3</i>
	$6/2 = 3$	r 0	<i>bit 4</i>
	$3/2 = 1$	r 1	<i>bit 5</i>
$X = \mathbf{01101000}_2$	$1/2 = 0$	r 1	<i>bit 6</i>

Converting Decimal to Binary (2's C)

First Method: **Subtract Powers of Two**

1. Change to positive decimal number.
2. Subtract largest power of two less than or equal to number.
3. Put a one in the corresponding bit position.
4. Keep subtracting until result is zero.
5. Append a zero as **MSB**,
if original was negative, take two's complement.

$$X = 104_{10}$$

$$104 - 64 = 40 \quad \text{bit 6}$$

$$40 - 32 = 8 \quad \text{bit 5}$$

$$8 - 8 = 0 \quad \text{bit 3}$$

$$X = 01101000_2$$

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Wrap-up

- Everything in a computer is a number, only “0” and “1”.
- Negative numbers are represented with **2’s complement**
- **MSB / LSB**
- **Overflows** can be detected utilizing the **carry bit**

Operations on Bits: Arithmetic and Logical

Operations: Arithmetic and Logical

- Recall: a data type includes **representation** and **operations**. We now have a good representation for **signed integers**
 - Addition
 - Subtraction
 - Sign Extension
- Overflow conditions for addition. Multiplication, division, etc., can be built from these basic operations.
- Logical operations are also useful:
 - AND
 - OR
 - NOT

Addition

- Assume all integers have the same number of bits
- **Ignore carry out**

Using our five-bit notation, what is $11 + 3$?

The decimal value 11 is represented as 01011

The decimal value 3 is represented as 00011

The sum, which is the value 14, is 01110

Example 2.3

Assuming 5-bit 2's complement numbers.

Subtraction

Example 2.4

What is $14 - 9$?

The decimal value 14 is represented as 01110

The decimal value 9 is represented as 01001

First we form the negative, that is, -9: 10111

Adding 14 to -9, we get

01110
10111

which results in the value 5. 00101

Note again that the carry out is ignored.

Assuming 5-bit 2's complement numbers.

Sign Extension

- To add two numbers, we must represent them with the **same number of bits**.
- Pad with zeroes on the left:

<u>4-bit</u>	<u>8-bit</u>	
0100 (4)	00000100	(still 4)
1100 (-4)	00001100	(12, not -4)

- Instead, replicate the MS bit -- the sign bit:

<u>4-bit</u>	<u>8-bit</u>	
0100 (4)	00000100	(still 4)
1100 (-4)	11111100	(still -4)

What if too big/small?

- Integer and floating-point operations can lead to results too big/small to store within their representations: **overflow/underflow**
- Binary bit patterns are simply representatives of numbers. “**numerals**”
- Numerals really have an ∞ number of digits
- Finite length of register, memory unit \rightarrow limited number of bits
- If result of arithmetic **cannot** be represented by these finite bits in a computer, we say **overflow** occurred

Overflow

- If operands are too big, then sum **cannot** be represented as an n -bit 2's comp number.

$\begin{array}{r} 01000 \\ + 01001 \\ \hline 10001 \end{array}$	$\begin{array}{r} 11000 \\ + 10111 \\ \hline 01111 \end{array}$
(8) (9) (-15)	(-8) (-9) $(+15)$

- We have overflow if:
 - signs of both operands are the same, and sign of sum is different.
- Another test -- easy for hardware:
 - carry into MS bit does not equal carry out

Overflow

- 8-bit Unsigned Integer

$$\begin{array}{r} 1001\ 0110\ (150) \\ +\ \underline{0111\ 1000}\ (120) \\ \hline 1\ 0000\ 1110\ (270) \end{array}$$

Overflow

- 8-bit Signed Integer

0110 0100 (A = 100)
+ 0011 0010 (B = 50)

1001 0110 (Signed Int = -126)

MSB Changed from 0 to 1 (Overflow)

Logical Operations

- Operations on logical TRUE or FALSE
 - Two states -- takes one bit to represent: TRUE=1, FALSE=0
- View n -bit number as a collection of n logical values
 - Operation **applied to each bit independently**

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

Logical Operations

AND

- useful for clearing bits
 - AND with zero = 0
 - AND with one = no change

$$\begin{array}{r} \text{AND} \quad 11000101 \\ \quad \underline{00001111} \\ \quad 00000101 \end{array}$$

OR

- useful for setting bits
 - OR with zero = no change
 - OR with one = 1

$$\begin{array}{r} \text{OR} \quad 11000101 \\ \quad \underline{00001111} \\ \quad 11001111 \end{array}$$

NOT

- unary operation -- one argument
- flips every bit

$$\begin{array}{r} \text{NOT} \quad 11000101 \\ \quad \underline{} \\ \quad 00111010 \end{array}$$

Exclusive-OR Function

If a and b are 16-bit patterns as before, then c (shown here) is the XOR of a and b .

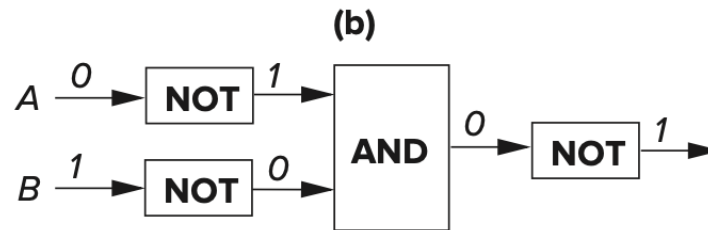
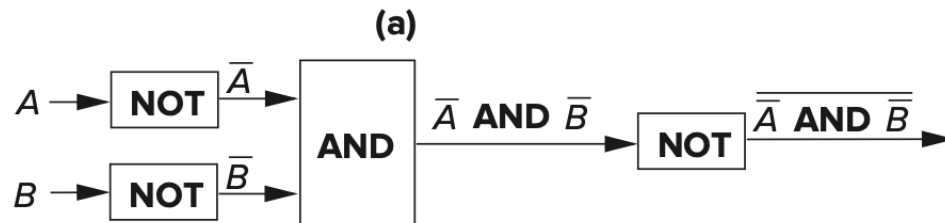
```
a: 0011101001101001
b: 0101100100100001
c: 0110001101001000
```

Note the distinction between the truth table for XOR shown here and the truth table for OR shown earlier. In the case of exclusive-OR, if both source operands are 1, the output is 0. That is, the output is 1 if the first operand is 1 but the second operand is not 1 or if the second operand is 1 but the first operand is not 1. The term *exclusive* is used because the output is 1 if *only* one of the two sources is 1. The OR function, on the other hand, produces an output 1 if only one of the two sources is 1, or if both sources are 1. Ergo, the name *inclusive-OR*.

Example 2.9

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

DeMorgan's Laws



(c)

A	B	\bar{A}	\bar{B}	$\bar{A} \text{ AND } \bar{B}$	$\overline{\bar{A} \text{ AND } \bar{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1



$$\overline{\bar{A} \text{ AND } \bar{B}} = A \text{ OR } B$$

Other Representations

Floating Point Data Type (Greater Range, Less Precision)

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

- How can we represent **fractions**?
 - Use a “binary point” to separate positive from negative powers of two – just like “decimal point.”
 - 2’s comp addition and subtraction still work.
 - if binary points are aligned

$$\begin{array}{r} 00101000.101 \\ + 11111110.110 \\ \hline 00100111.011 \end{array}$$

$2^{-1} = 0.5$
 $2^{-2} = 0.25$
 $2^{-3} = 0.125$

Fractions: Fixed-Point

- Example: 5-bit fraction

fraction			Integer		
010.10	2.5	(10/2 ²)	01010	10	
<u>+101.11</u>	-2.25	(-9/2 ²)	+ <u>10111</u>	-9	
000.01	0.25	(1/4)	00001	1	

Very Large and Very Small Data

- The LC-3 use the 16 bit 2's complement data type,
- One bit to identify positive or negative, 15 bits to represent the magnitude of the value. We can express values:

$$-2^{15} \text{ through } 2^{15} - 1$$

$$(-32768 \text{ through } 32767)$$

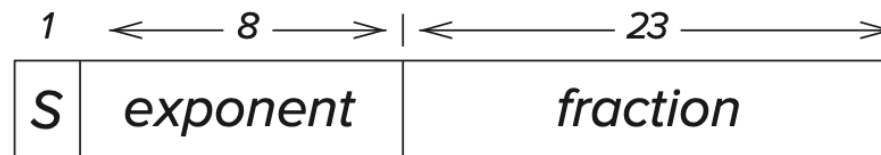
- How can we represent very large and very small data?

Very Large and Very Small Data

- Large values: $6.023 \times 10^{23} \rightarrow$ requires 79 bits
- Small values: $6.626 \times 10^{-34} \rightarrow$ requires >110 bits

Use equivalent of “scientific notation”: $F \times 2^E$

Need to represent F (*fraction*), E (*exponent*), and S (sign).

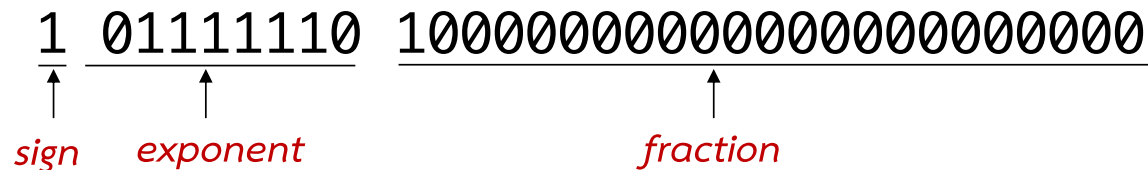


*IEEE 754 Floating-Point
Standard (32-bits)*

Normalized Form: $N = (-1)^S \times 1.\textit{fraction} \times 2^{\textit{exponent}-127}, 1 \leq \textit{exponent} \leq 254$

Floating Point Example

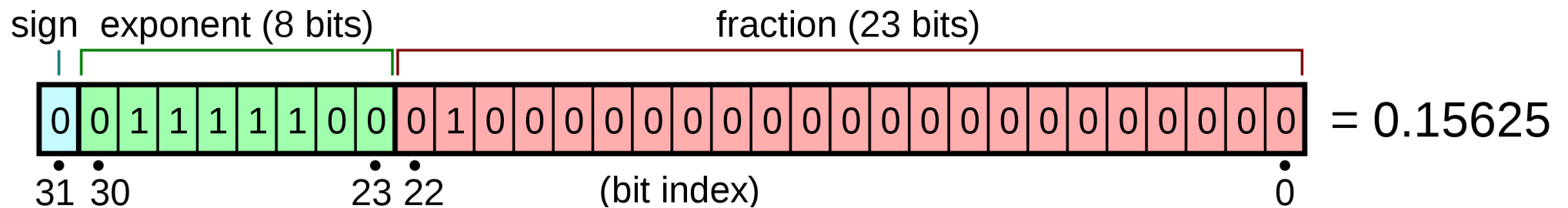
Single-precision IEEE floating point number:



- Sign is 1 – number is negative.
- **Exponent** field is 01111110 = 126 (decimal).
- **Fraction** is 0.100000000000... = 0.5 (decimal).

$$\text{Value} = -1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$$

Floating Point Example2



Wikipedia

- Sign is 0 – number is positive.
- **Exponent** field is 01111100 = 124 (decimal).
- **Fraction** is 0.010000000000... = 0.25 (decimal).

$$\text{Value} = 1.25 \times 2^{(124-127)} = 1.25 \times 2^{-3} = 0.15625$$

What does the floating point data type

00111101100000000000000000000000

represent?

The leading bit is a 0. This signifies a positive number. The next eight bits represent the unsigned number 123. If we subtract 127, we get the actual exponent -4 . The last 23 bits are all 0. Therefore, the number being represented is $+1.00000000000000000000000 \cdot 2^{-4}$, which is $\frac{1}{16}$.

Example 2.12

How is the number $-6\frac{5}{8}$ represented in the floating point data type?

First, we express $-6\frac{5}{8}$ as a binary number: -110.101 .

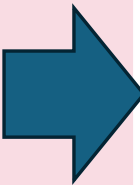
$$-(1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3})$$

Then we normalize the value, yielding $-1.10101 \cdot 2^2$.

The sign bit is 1, reflecting the fact that $-6\frac{5}{8}$ is a negative number. The exponent field contains 10000001, the unsigned number 129, reflecting the fact that the real exponent is $+2$ ($129 - 127 = +2$). The fraction is the 23 bits of precision, after removing the leading 1. That is, the fraction is 10101000000000000000000. The result is the number $-6\frac{5}{8}$, expressed as a floating point number:

1 10000001 10101000000000000000000

Example 2.13


$$\begin{array}{l} 6 \div 2 = 3 \rightarrow 0 \\ 3 \div 2 = 1 \rightarrow 1 \\ 1 \div 2 = 0 \rightarrow 1 \end{array}$$

$$\begin{array}{l} 0.625 \times 2 = 1.25 \rightarrow 1 \\ 0.25 \times 2 = 0.5 \rightarrow 0 \\ 0.5 \times 2 = 1.0 \rightarrow 1 \end{array}$$

$$\begin{aligned} 6.625 &= (110.101)_2 \\ &= 1.10101 \times 2^2 \\ &= 1.10101 \times 2^{129-127} \end{aligned}$$

Floating-Point Operations

- $6.625 + 0.025 = 6.650000095367431640625$ (Why ?)
- Will regular 2's complement arithmetic work for Floating Point numbers?

Text: ASCII Characters

- ASCII: Maps 128 characters to 7-bit code.
 - both printable and non-printable (ESC, DEL, ...) characters

8-bit Hex	➡	00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
		01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
		02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
		03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
		04	eot	14	dc4	24	\$	34	4	44	D	54	T	64	d	74	t
		05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
		06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
		07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
		08	bs	18	can	28	(38	8	48	H	58	X	68	h	78	x
		09	ht	19	em	29)	39	9	49	I	59	Y	69	i	79	y
		0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
		0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[6b	k	7b	{
		0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	
		0d	cr	1d	gs	2d	-	3d	=	4d	M	5d]	6d	m	7d	}
		0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
		0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

ASCII :

American Standard Code for Information Interchange

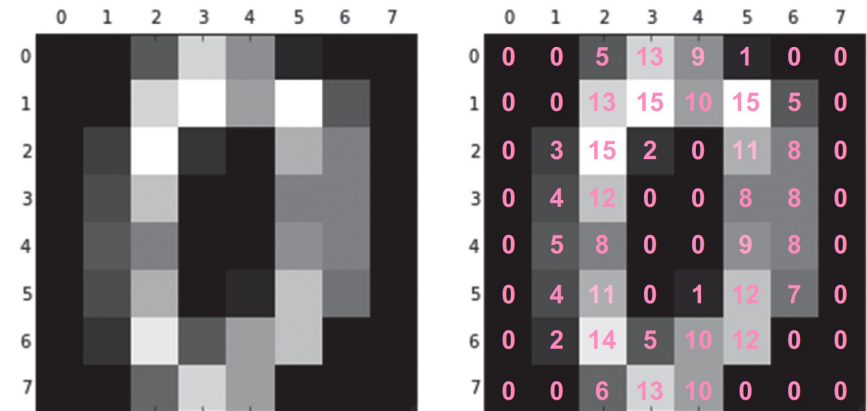
Are 128 characters enough?

(<http://www.unicode.org/>)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	32	20	(space)	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	\$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(72	48	H	104	68	h
9	9	HT	41	29)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	SLE	48	30	0	80	50	P	112	70	p
17	11	CS1	49	31	1	81	51	Q	113	72	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SIB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Computer: Everything is a Number

- Text strings
 - sequence of characters, terminated with NULL(0)
 - typically, no hardware support
- Image: Array of pixels
 - monochrome: one bit (1/0 = black/white)
 - color: red, green, blue (RGB) components (e.g., 8 bits each)
 - other properties: transparency
- Sound: Sequence of fixed-point numbers



```
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

LC-3 Data Types

- Some data types are supported directly by the instruction set architecture.
- For **LC-3**, there is only one supported data type:
 - 16-bit 2's complement signed integer
 - Operations: ADD, AND, NOT
- Other data types are supported by interpreting 16-bit values as logical, text, fixed-point, etc., in the software

Wrap-up

- Representation of floating point
- Hexadecimal Notation
- ASCII Code
- n -bit data type

Homework

- In-class: Explain the solution in binary format and IEEE 754 Floating-Point Standard (32-bits):
 - 3.5 – 3.75
 - 4.4 – 4.5
- Exercises:
2.2, 2.10, 2.11, 2.15, 2.18,
2.30, 2.39, 2.41, 2.43, 2.47, 2.54

