

Signed Arithmetic

- In the “real world” of mathematics, computers must represent both positive and negative binary numbers.
- For example, even when dealing with positive arguments, mathematical operations may produce a negative result:
 - **Example: $124 - 237 = -113$.**
- Thus needs to be a consistent method of representing negative numbers in binary computer arithmetic operations.
- There are various approaches, but they all involve using one of the digits of the binary number to represent the sign of the number.
- **Two methods are the sign/magnitude representation and the one's complement method of representation.**

Binary Sign Representations

- **Sign-magnitude:** The left bit is the sign (0 for + numbers and 1 for – numbers).

 All bits to right are the number magnitude

 Left bit is the sign bit

Advantages to sign-magnitude:

- Simple to implement.
- Useful for floating point representation.

(Big) Disadvantage of sign-magnitude:

- Sign bit independent of magnitude; can be both + 0 and – 0! (Makes math hard to do).

- **One's complement:** The negative number of the same magnitude as any given positive number is its one's complement.
 - If $m = 01001100$, then m complement (or $\bar{m}$) = 10110011
 - The most significant bit is the sign, and is 0 for + binary numbers and – for negative numbers.
 - Note the problem: If $m = 00000000$, then $\bar{m} = 11111111$; there are two zeros in this method as well!

Two's Complement Negative Binary Numbers

- Due to the problems with sign/magnitude and 1's complement, another approach has become the standard for representing the sign of a fixed-point binary number in computer circuits.
- Consider the following definition: **The “two's complement” of a binary integer is the 1's complement of the number plus 1.**
 - Thus if m is the 2's complement of n , then: **$m = \bar{n} + 1$**
- Examples:
 - $n = 0101\ 0100$, then $m = 1010\ 1011 + 1 = 1010\ 1100$
 - $n = 0101\ 1111$, then $m = 1010\ 0000 + 1 = 1010\ 0001$
 - $n = 0111\ 1111$, then $m = 1000\ 0000 + 1 = 1000\ 0001$
 - $n = 0000\ 0001$, then $m = 1111\ 1110 + 1 = 1111\ 1111$
- **Note that to properly represent 2's complement binary numbers, the full group of numbers showing the range of representation must be retained! That means that leading zeros must be kept.**

Two's Complement Negative Binary Numbers (2)

- For integer 2's complement representation **addition and subtraction of both + and – numbers always work out correctly (within the range of representation), and there is only one 0.**
- Furthermore, the left-most bit is always 1 for a negative number, always 0 for a positive number.
- This means that an n-bit binary number in two's complement can represent a magnitude range of $\pm 2^{n-1} - 1$.
- In an n-bit representation, there are no extra bits! If adding 2 n-bit numbers results in n+1 bits, the left most bit is discarded! Thus:
 - Let $n = 0000\ 0000$. Then $m = \bar{n} + 1$, or $m = 1111\ 1111 + 1 = (1)\ 0000\ 0000 = 0000\ 0000$ (the 1 is discarded).
 - Since we have an 8-bit number (byte), the leading (9th-column) 1 is thrown away, and the resulting number is still 0000 0000.
 - We see that the 2's complement of 0 is 0.

Finding Two's Complements: Examples

- In the following, remember that for any n-bit computing system, there are no extra bit positions.
- To convert a negative decimal number to 2's complement binary:
 - Convert the decimal number to a positive binary number.
 - Take the 1's complement of that binary number and add 1.
- Converting negative numbers (still using a single 8-bit byte length):
 - 50: $50 = 0011\ 0010$; 1's C. = $1100\ 1101$; 2's C. = $1100\ 1110$.
 - 127: $127 = 0111\ 1111$; 1's C. = $1000\ 0000$; 2's C. = $1000\ 0001$.
 - 1: $1 = 0000\ 0001$; 1's C. = $1111\ 1110$; 2's C. = $1111\ 1111$.
- But: **Positive decimal numbers are converted simply to positive binary numbers as before (no 2's complement).**

Example: +67 (using method of successive div.) $\rightarrow$ 0100 0011

Two's Complement Binary to Decimal

- Converting the “other direction” (2's complement to decimal) is also simple. Simply do the following:
 - Check the sign bit (left-most bit).
 - If the sign bit is 0 (positive number), simply convert the number directly to a positive decimal number as we learned previously.
 - If the sign bit is 1, the number is a 2's complement negative number. To convert this number to decimal:
 - Take the 2's complement of the negative binary number.
 - Convert the resulting + number to decimal and add a negative sign.

Two's Complement Binary to Decimal (2)

- **Binary 2's complement-to-decimal examples, negative numbers:**

1111 1111 $\rightarrow$ **0000 0000+1 = 0000 0001 = 1;** $\rightarrow - 1$.

1010 0011 $\rightarrow$ **0101 1100+1 = 0101 1101 = 93;** $\rightarrow - 93$.

1000 1111 $\rightarrow$ **0111 0000+1 = 0111 0001 = 113;** $\rightarrow - 113$.

1000 0010 $\rightarrow$ **0111 1101+1 = 0111 1110 = 126;** $\rightarrow - 126$.

- **But for a positive binary number:**

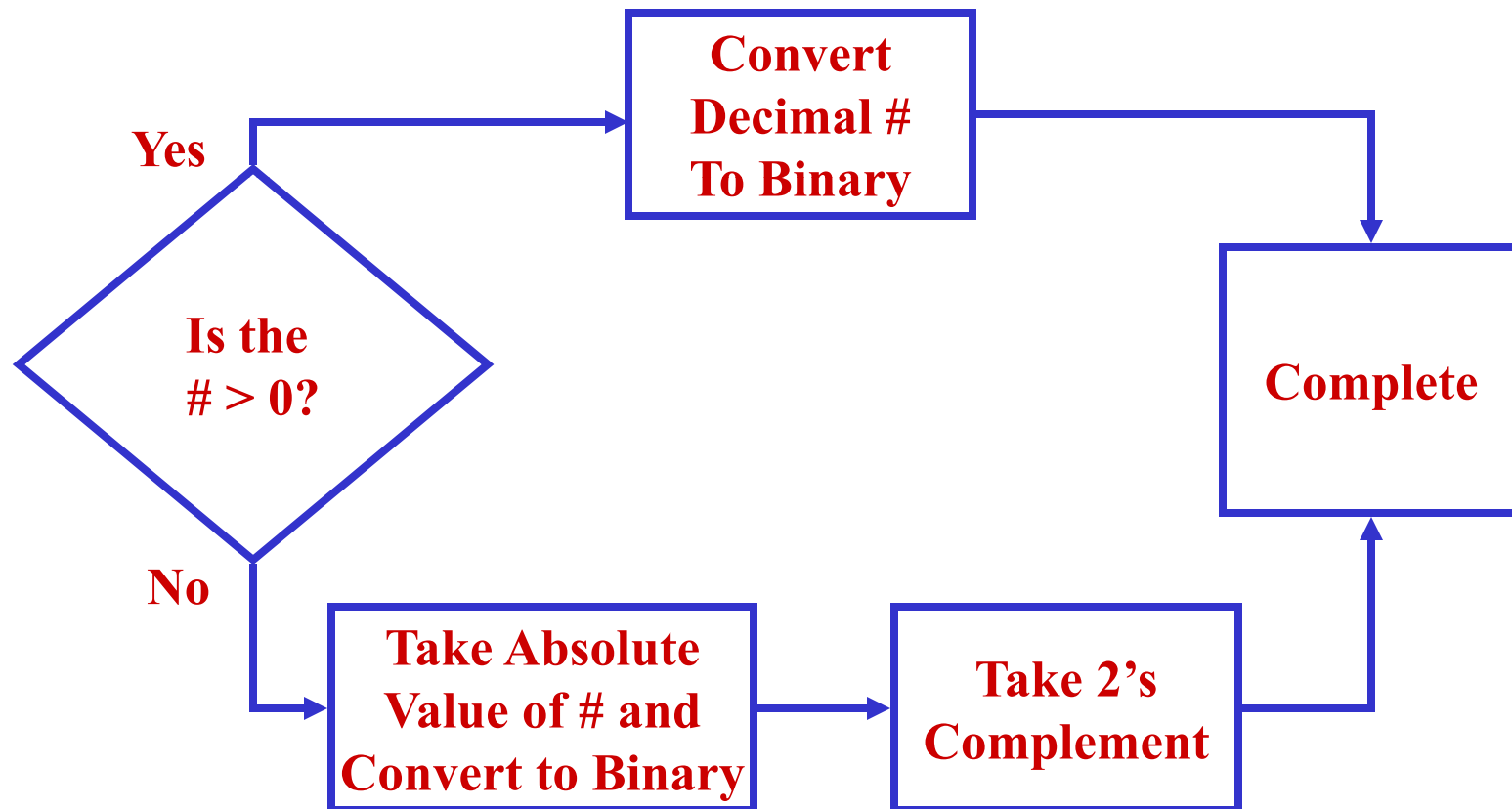
0000 0001 $\rightarrow$ **Not a negative number $\rightarrow 1$.**

0000 1111 $\rightarrow$ **Not a negative number $\rightarrow 15$.**

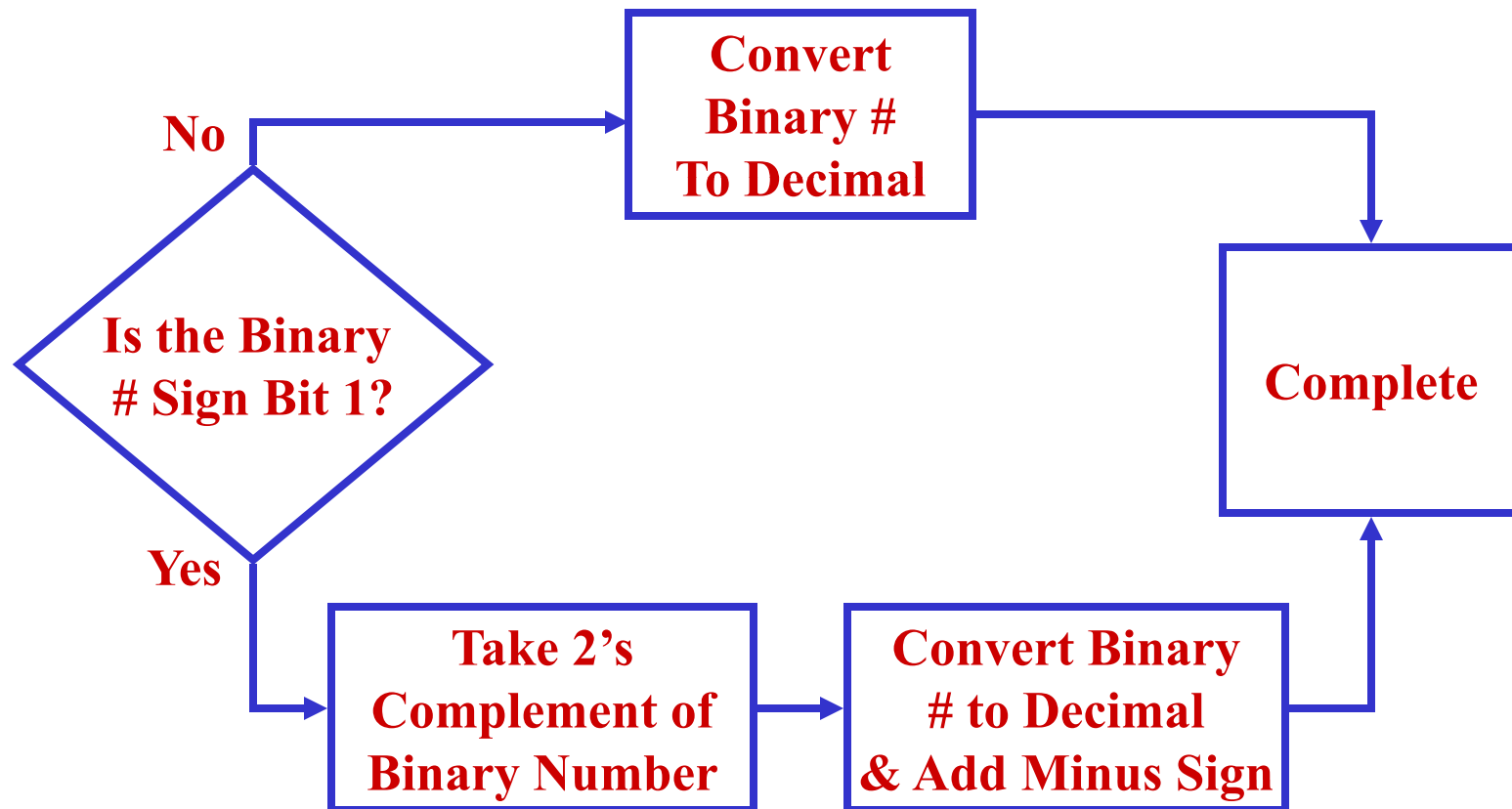
0110 1100 $\rightarrow$ **Not a negative number $\rightarrow 108$.**

0111 1111 $\rightarrow$ **Not a negative number $\rightarrow 127$.**

Decimal $\rightarrow$ 2's Complement Binary



2's Complement Binary → Decimal



Exercise 1

- Convert the following:

78 -----

– 106 -----

11101011 -----

11000010 -----

01010101 -----

Two's Complement Binary Math

- If we represent binary numbers in 2's complement form, simple addition and subtraction is easy.
- To subtract binary number **b** from **a**, simply take the 2's complement of **b**, and add to **a**. That is:
$$\mathbf{a - b = a + (2's\ comp.\ of\ b) = a + (\bar{b} + 1) = a + \bar{b} + 1}$$
- Adding numbers is the same as decimal addition. To add a positive and negative number, simply perform the addition and the minus sign (i.e., the left-most bit in the number) will take care of itself (assuming the result is within the range of representation).

Two's Complement Math

- Subtract 0111 0101 from 0111 1100:

The 2's complement of 0111 0101 is $1000\ 1010 + 1 = 1000\ 1011$.

Adding:	0111 1100	Check:	124
	+ 1000 1011		<u>-117</u>
	(1)0000 0111		007

“Thrown away” (with red arrow pointing to the carry-out)

- Add 1100 0001 + 0110 1110:

Note that the 2's complement of 1100 0001 is 0011 1111, so the first number is equivalent to -63 decimal.

Adding:	1100 0001	Check:	-63
	+0110 1110		<u>+110</u>
	(1)0010 1111		+ 47

“Thrown away” (with blue arrow pointing to the carry-out)

Two's Complement Binary Math (2)

- Subtract 1101 1101 from 0101 1100 (note we are subtracting a negative number):

Adding: 0101 1100
 + 0010 0011
 0111 1111

2's complement
of 1101 1101. →

Check: 92 92
 -(-35) → + 35
 + 127

- Add 1000 0001 + 0111 0010:

Adding: 0111 0010
 + 1000 0001
 1111 0011

Check: 114
 + (-127)
 - 13

Check!

Check: 2's C of 1111 0011 = 0000 1101 = 13, so the number = - 13.

“Outside the Range of Representation”

- A number is “**outside the range of representation,**” when it cannot be represented by the bits available.
- For 8-bit, 2’s complement numbers (since left-most bit is the sign bit):
 - **Biggest + number to be represented = 0111 1111 (+127).**
 - **Biggest negative number possible is 1000 0001 (= − 127, 2’s C of 0111 1111).**
- **Numbers outside the range of representation of n bits may only be represented by adding more bits.**

Examples:

$$\begin{array}{rcl}
 0111\ 0011 & & 115 \\
 +\ 0011\ 1111 & + & \underline{63} \\
 \hline
 1011\ 0010 & & 178
 \end{array}$$

→ = − 78!!!

$$\begin{array}{rcl}
 1001\ 0000 & & -112 \\
 1110\ 0000 & - & \underline{32} \\
 \hline
 0111\ 0000 & & -144
 \end{array}$$

→ = + 112!!!

Summary: 2's Complement Binary Math

- For integral mathematics in all modern computers, 2's complement arithmetic is the customary approach.
 - No ambiguous values crop up in CPU operations.
 - A binary adder can subtract with minor modifications.
 - The 2's complement binary math unit is simple.
- For 2's complement subtraction, the algorithm is very simple:
 - Take the 2's complement of the number being subtracted.
 - Add the two numbers together.
 - Note that the sign takes care of itself (“assuming the answer is within the range of representation”).

Exercise 2

- Two 2's comp. problems:

$$\begin{array}{r} 0101\ 0101 \\ + \underline{1011\ 0110} \end{array}$$

$$\begin{array}{r} 1111\ 1100 \\ - \underline{0101\ 0001} \end{array}$$

Binary Codes

- **Computers sometimes use binary numbers to represent non-numeric information, such as text or graphics.**
- **Many such binary representations, for example, of text, (letters, textual numbers, punctuation symbols, etc.) are called codes.**
- **In a binary code, the binary number does not represent an actual number, but a value or symbol.**
- **We will discuss mainly text codes, and give a brief example of a numerical representation code. **These are weighted binary codes (BCD) and alphanumeric codes, primarily ASCII.****

Binary Codes (2)

- **Characteristics of binary codes:**
 - **The significance of the code is not the numerical value.**
 - **The code is a positive binary number which stands for or symbolizes the quantity represented by the code.**
 - **The quantity normally cannot be “operated on” in the usual fashion – mathematical, logical, etc. That is, one cannot usually add up, for example, two binary codes. It would be like attempting to add A and G!**
 - **Codes are usually stored in tables for reference.**
- **In modern computers, binary codes are used to symbolize alphanumeric characters, font types, font sizes, alternate language alphabets, pixels on a screen, and so forth.**

A Numeric Code: BCD

- Binary→decimal conversion is clumsy, so many years ago, the binary-coded-decimal (BCD) system was developed.
- BCD is an 8-4-2-1 code that **symbolizes decimal numbers directly**.
- A BCD table is shown at right. BCD codes extend only from 0000 to 1001.
- **Thus, the binary numbers 10-15 (1010-1111) have no meaning in BCD.**
- Converting from decimal to BCD and vice-versa is easy:
 - $150 = (\text{BCD } 1) - (\text{BCD } 5) - (\text{BCD } 0) = 0001\ 0101\ 0000$
 - $2403 = 0010\ 0100\ 0000\ 0011$
 - $0001\ 1001\ 0101\ 0010 = 1952$
 - $1001\ 0101\ 0110\ 0011\ 0001 = 95631$

BCD Table	
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Alphanumeric Codes

- **Binary codes can also represent text characters, including punctuation and printer codes such as carriage return.**
- **The two most common alphanumeric codes are:**
 - **The American Standard Code for Information Interchange (or ASCII, pronounced “ask-ee”).**
 - **The Extended Binary-Coded Decimal Interchange Code (or EBCDIC, pronounced “ebb-see-dik”). Older than ASCII; developed for, and still used in, IBM mainframe computers.**
- **Tokheim also mentions other codes such as the 7-bit Selectric (IBM typewriter) and 12-bit Hollerith (IBM punched-card), which we will not discuss.**

The ASCII Alphanumeric Code

- **ASCII code represents alphanumeric data in most computers.**
 - **Data on this transparency is coded in ASCII.**
 - **ASCII codes are used for virtually all printers today.**
 - **In the basic ASCII code that we will study, a single byte is used for each character. The least significant 7 bits represent the character. The eighth bit (the most significant bit, or MSB) may be used for error checking.**
- **Special “super ASCII” codes can use all 8 bits for special control characters, function keys, etc., and even more elaborate codes, with more bits, can represent other alphabets and character sets (Greek, Katakana, etc.).**

ASCII Code (2)

- There are 128 basic ASCII characters, $0-127_{10}$, or $0-7f_{16}$ (0000 0000 to 0111 1111 binary).
- Each ASCII code is unique, for example:
 - $M = 0100\ 1101 = 77_{10} = 4D_{16}$.
 - $m = 0110\ 1101 = 109_{10} = 6D_{16}$.
 - Note that the small letters are exactly 32_{10} (20 hex) larger in numerical value than the capital letters.
- ASCII characters are stored as **bytes** in the computer.
- ASCII characters are normally represented as pairs of hex numbers (since 1 byte = 2 nibbles = 2 hex numbers).

A Word on Code Conventions

- Although ASCII text codes always mean the same thing (i.e., an “m” is always an “m”), **ASCII control characters may mean different things on different computer systems.**
- As shown below, for example, some operating systems interpret control characters for line feed and carriage return differently.
- **This is a good example showing that the computer hardware/software person must know his/her system and how it is designed.**

Operating System	End of line (ASCII)	Character	Hexadecimal
Unix	LF	\n	0A
Mac OS	CR	\r	0D
VMS	CR LF	\r\n	0D 0A
Windows (X86)	CR LF	\r\n	0D 0A

Summary of Codes

- **Binary codes are numbers that have a different meaning than their simple numerical value.**
- **ASCII is the default text code in most computer systems.**
- **BCD and ASCII are the primary binary codes in use today (although EBCDIC is still used by IBM). BCD is mainly used in number displays.**
- **You will not be tested on ASCII or BCD codes, but you need to be familiar with ASCII, in particular.**
- **The basic ASCII code set is shown at the end of these notes.**

Exercise 3

- **Decode the following ASCII message:**

47-6f-2c-20-55-54-44-20-43-6f-6d-65-74-73-21

All numbers are ASCII characters in hex. Hint: We might say this if we had a football team.

Boolean Algebra

- **Boolean Algebra** (named for its developer, George Boole), is the algebra of **digital logic circuits** that all computers use.
- It is a symbolic representation of logic principles that date back to Greek logic studies (Aristotle, ~384-322 BC).
- In the Victorian Age, mathematicians began to apply formal principles to Aristotelian logic, which led to the field of **symbolic logic**.
- Technically, “Boolean Algebra” refers to a family of logic systems, but it is generally used to refer to one system -- and that is how we will use it.
- George Boole (1815-1864) was a mathematician and teacher, the self-educated son of a cobbler, who married Mary Everest (niece of Sir George Everest of mountain fame).*
- Boole’s thinking was typical of Victorians: All processes (even thinking!) could be represented by formal rules. His intent was to formalize the “rules of thinking” with logic equations – a well-meant but totally impossible task.
- Scientists soon began to apply “switching algebra” to **logic problems**.

* His key work was “An Investigation of the Laws of Thought on Which are Founded the Mathematical Theories of Logic and Probabilities.”

Boolean Algebra (2)

- Later contributors improved and adapted Boole's work, slowly evolving it towards computer applications, including **William S. Jerons (1869 – “the logical piano”), Paul Ehrenfest (1910 – prototype electrical telephone office switch), and Claude Shannon (1930's – a great early data transmission expert, who died in 2001; he invented digital data transmission [“it's all bits”])**.
- Despite many contributors Boolean Algebra is mostly unchanged.
- Boolean Algebra is “1-0 logic.”
 - 0 = “off,” “not present,” “false,” “de-asserted.”
 - 1 = “on,” “present,” “true,” “asserted.”
- All computer circuits solve problems by performing Boolean transformations on binary (0-1) variables. It could be said that Boolean algebra is the algebra of binary systems.

Very important statement!!!!

Boolean Algebra in the Computer

- In a computer, logic levels 1 and 0 correspond to voltages:
 - “**Positive logic**” uses a + voltage (e.g., 5 V) for 1 and 0 V for 0.
 - Sometimes, “**negative logic**” (1 = 0V, 0 = +V [e.g., 5V]) is used.
- There are two kinds of digital or computer logic:
 - **Combinational logic** – output depends only on the inputs.
 - **Sequential logic** – output depends on the inputs, the internal state of the logic, and possibly a clock or timing mechanism (studied later).
- We will only cover circuit behavior in terms the manipulation of 1’s and 0’s. You will cover electronics of digital circuits later.
- A logic circuit (“gate”), is an electronic device that performs a Boolean function* on one or more inputs, and provides at least one output.
- There are three basic Boolean functions: **AND OR NOT**

* A function is an action that can be performed on a value, state, or number, resulting in a new value, state, or number.

Boolean Functions: Logical Negation

- **NOT** is the simplest logical function: 1 input and 1 output.
- **NOT** is defined as follows: “The output f of **NOT**, given an input a , is the complement or opposite of the input.” Or : $f = \bar{a}$
- Since **NOT** can have only a 0 or 1 input, the output of **NOT** is the reverse, or complement, of the input.
 - If the input of **NOT** is 1, the output is 0.
 - If the input of **NOT** is 0, the output is 1.
- The **NOT** function is called inversion, and the digital circuit which inverts is an inverter. The electronic circuit symbol for **NOT** is:



Truth Tables

- The input/output relationship is easy to show for an inverter (see last slide).
- For more complex functions, a **table of outputs versus inputs** is helpful.
- Such a table is called a **truth table**, since the table indicates the 1 (or “true”) outputs, although the table normally shows outputs for **all** input combinations.
- **We will use truth tables to demonstrate many Boolean functions.**

Boolean Function Truth Table

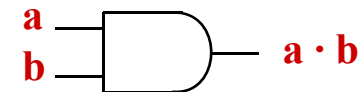
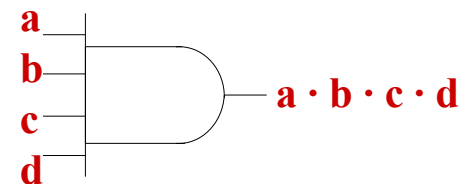
Input x	Input y	Output f
0	0	0
0	1	1
1	0	1
1	1	0

Logical AND

- **AND** has two or more inputs.
- The truth table for a two-input **AND** with **a** and **b** inputs is shown in the chart.
- **AND** is defined as follows: **a AND b** = 1 if and only if (iff) **a** = 1 and **b** = 1.
- Mathematically, we represent “**a AND b**” as **a · b** (an unfortunate choice).
- **AND** may have more than two inputs, i. e.: **a AND b AND c AND d**.
- The electronic circuit symbols for 2- and 4-input **ANDs** are shown at the right.
- **Regardless of the number of inputs, the output of AND is 1 iff all inputs are 1**.

AND Truth Table

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

2-Input AND**4-Input AND**

Logical OR

- **OR** has two or more inputs.
- The truth table for two inputs **a**, **b** is shown in the adjacent chart.
- **OR** is defined as follows: **a OR b** = 1 if either **a** or **b** or both **a** and **b** = 1.
- Mathematically, we represent “**a OR b**” as **a + b** (another bad choice).
- **OR** may have more than two inputs, i. e.: **a OR b OR c OR d**.
- The electronic circuit symbols for 2- and 4- input **ORs** are shown at the right.
- **Regardless of the number of inputs, the output of OR is 0 iff all inputs are 0.**

OR Truth Table

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

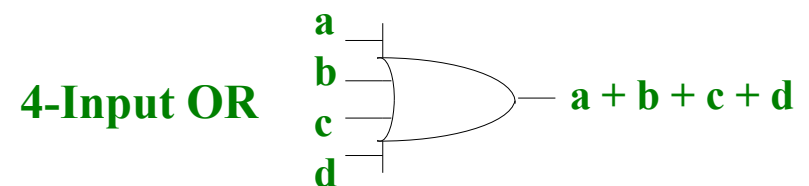
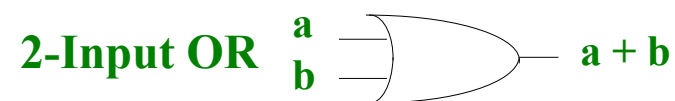
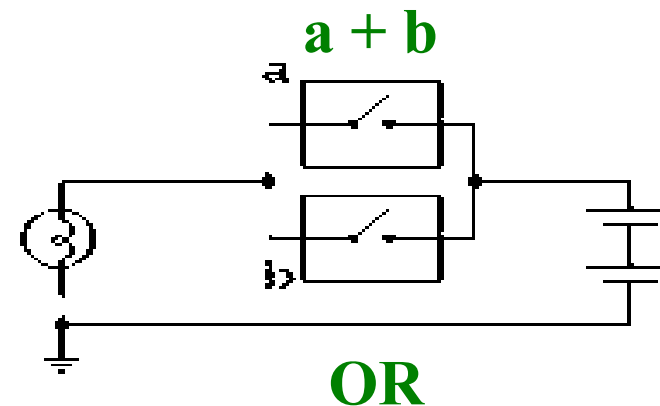
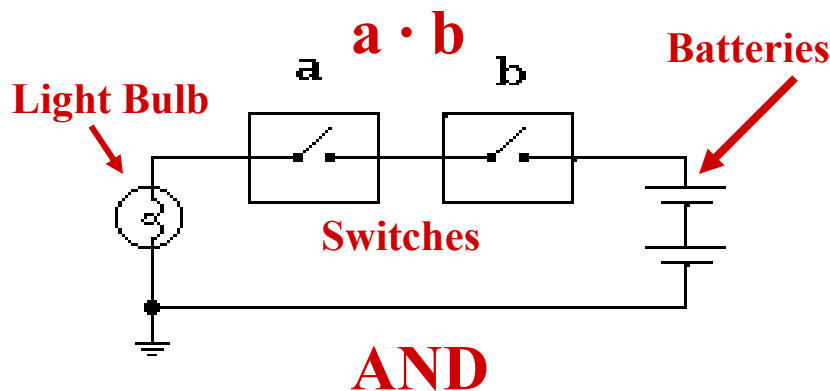


Illustration of **AND** and **OR** with “Goofy Flashlights”

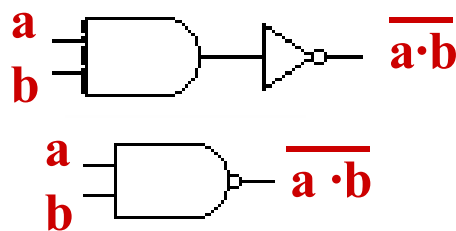


- The **AND** “goofy flashlight” is a circuit connecting batteries to a light bulb via **series** switches, **a** & **b**.
- The bulb will be lighted **only if both switches are closed**.
- Thus we say that the “**AND** light bulb function” = **a AND b = a · b**.
- The **OR** “goofy flashlight” connects batteries to a light bulb via **parallel** switches, **a** & **b**.
- The bulb will be lighted **if either switch is closed**.
- Thus we say that the “**OR** light bulb function” = **a OR b = a + b**.

NAND and NOR

- Both **NAND** and **NOR** are composite functions, but we discuss them here, since they are commercially available.
- The **NAND** function is inverted AND; **NOR** is inverted OR.
- Mathematically, we use a horizontal bar to indicate the inversion, i.e.: **a NAND b** is written as $\overline{a \cdot b}$, and **a NOR b** is written as $\overline{a + b}$
- As **NAND** is the reverse of **AND**, the **NAND** output is 1 unless all inputs are 1. Likewise, **NOR** outputs 0 unless all inputs are 0.

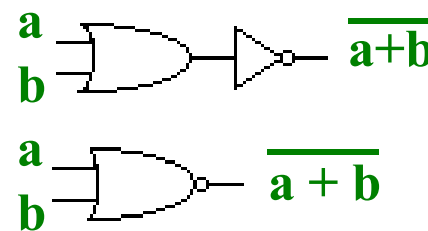
NAND



NAND Truth Table

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

NOR



NOR Truth Table

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

Homework

- **Write down the two or three most important things you learned today. Add these to the list you started last time.**
- **Write down two or three things you did not clearly understand. After finishing the assigned reading, if you still have questions, see me during office hours.**
- **Patterson and Hennessy, 3.1-3.3.**
- **Read or review Pervin, chapter 1-1.5.**
- **Read or review Tokheim, chapters 1-2.**
- **Complete homework #1.**

ASCII Codes

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□